Data Structuring

Textbook reading

Lecture Notes
Overview

Design principles:
■ Use data structures to do the job
■ Plane sweeping
■ Augment data structures to support new operations

Data structures:
■ \((a, b)\)-trees
■ Order statistics trees
■ Priority search trees
■ Range trees

Problems:
■ Line segment intersection reporting and counting
■ Range searching and counting
The Dictionary ADT

**Goal:** Store a set $S$ of elements in a structure $T$ that supports the following operations:

- **Insert**($T, x$) Update $T$ so that it represents the set $S \cup \{x\}$
- **Delete**($T, x$) Update $T$ so that it represents the set $S \setminus \{x\}$
- **Find**($T, x$) Decide whether $x \in S$ and, if so, report all information associated with $x$ in $T$
Ordered Dictionaries

If the elements in $S$ come from a total order, we may also want to support:

- **Range-Query** $(T, a, b)$: Report the set $\{x \in S \mid a \leq x \leq b\}$
- **Predecessor** $(T, x)$: If $x \in S$, report the next smaller element in $S$
- **Successor** $(T, x)$: If $x \in S$, report the next greater element in $S$
- **Minimum** $(T, x)$: Report $\min S$
- **Maximum** $(T, x)$: Report $\max S$
Examples of Dictionaries
Examples of Dictionaries

Simple dictionaries:
Examples of Dictionaries

*Simple dictionaries:*

- (Sorted) arrays
- (Sorted) doubly-linked lists
- Hash tables
Examples of Dictionaries

Simple dictionaries:

- (Sorted) arrays
- (Sorted) doubly-linked lists
- Hash tables

Efficient dictionaries:
Examples of Dictionaries

**Simple dictionaries:**
- (Sorted) arrays
- (Sorted) doubly-linked lists
- Hash tables

**Efficient dictionaries:**
- Hash tables
- Balanced binary trees (AVL, red-black, BB[\(\alpha\)], AA, \ldots)
- \((a, b)\)-trees
(a, b)-Trees

- All leaves are at the same depth
- All data elements are stored at the leaves
- The root has degree between 2 and b
- Any other non-leaf node has degree between a and b

\[2 \leq a \text{ and } 2a - 1 \leq b\]
For a leaf $v$ storing element $x$, $\text{key}(v) = x$

For an internal node $v$ with children $w_1, w_2, \ldots, w_k$,
$\text{key}(v) = \min\{\text{key}(w_i) \mid 1 \leq i \leq k\}$.
The Height and Size of an \((a, b)\)-Tree
The Height and Size of an $(a, b)$-Tree

Lemma: An $(a, b)$-tree storing $n$ items has height
The Height and Size of an \((a, b)\)-Tree

Lemma: An \((a, b)\)-tree storing \(n\) items has height \(\mathcal{O}(\log_a n) = \mathcal{O}(\lg n)\).
**The Height and Size of an \((a, b)\)-Tree**

**Lemma:** An \((a, b)\)-tree storing \(n\) items has height \(O(\log_a n) = O(\lg n)\).

**Lemma:** An \((a, b)\)-tree storing \(n\) items has less than \(\ldots\) nodes.
The Height and Size of an $$(a, b)$$-Tree

Lemma: An $$(a, b)$$-tree storing $$n$$ items has height $$O(\log_a n) = O(\lg n)$$.

Lemma: An $$(a, b)$$-tree storing $$n$$ items has less than $$2n$$ nodes.
Representing an \((a, b)\)-Tree

*Problem:* Storing \(b\) child pointers per node seems wasteful.
Representing an \((a, b)\)-Tree

**Problem:** Storing \(b\) child pointers per node seems wasteful.

**Better representation:**
- Key of \(v\)
- Degree of \(v\)
- Pointer to \(v\)’s leftmost child
- Pointer to \(v\)’s parent
- Pointers to \(v\)’s left and right sibling
Lemma: An \((a, b)\)-tree storing \(n\) items uses \(O(n)\) space.
The Find Operation
The Find Operation

While \( v \) is not a leaf:

- Locate the leftmost child \( w \) of \( v \) such that
  - \( w \) has no right sibling or
  - The key of \( w \)'s right sibling is greater than \( x \).

- Let \( v \) be this child.

When \( v \) is a leaf, report \( v \) if \( \text{key}(v) = x \) and \( \text{nil} \) otherwise.
Lemma: Operation Find is correct.
Lemma: Operation Find is correct.

Lemma: Operation Find takes
Lemma: Operation Find is correct.

Lemma: Operation Find takes $O(\lg n)$ time.
The Minimum Operation

- While \( v \) is not a leaf, proceed to \( v \)'s left child.
- When \( v \) is a leaf, report \( v \).
While \( v \) is not a leaf, proceed to \( v \)'s left child.

When \( v \) is a leaf, report \( v \).

**Lemma:** Procedure Minimum takes
The Minimum Operation

While $v$ is not a leaf, proceed to $v$'s left child.

When $v$ is a leaf, report $v$.

**Lemma:** Procedure Minimum takes $O(\lg n)$ time.
The Predecessor Operation
The Predecessor Operation

- While \( v \) has no left sibling and is not the root, proceed to \( v \)'s parent.
- When \( v \) is the root, report \( \text{nil} \).
- When \( v \) has a left sibling, \( u \), report Maximum(\( u \)).
The Predecessor Operation

- While \( v \) has no left sibling and is not the root, proceed to \( v \)'s parent.
- When \( v \) is the root, report \( \text{nil} \).
- When \( v \) has a left sibling, \( u \), report \( \text{Maximum}(u) \).

**Lemma:** Procedure Predecessor takes
The Predecessor Operation

- While $v$ has no left sibling and is not the root, proceed to $v$’s parent.
- When $v$ is the root, report nil.
- When $v$ has a left sibling, $u$, report Maximum($u$).

Lemma: Procedure Predecessor takes $\mathcal{O}(\lg n)$ time.
The Insert Operation
Use the Find procedure to locate the rightmost leaf \( v \) storing an element no greater than \( x \).

Create a new leaf, store \( x \) at this leaf, and make it the right sibling of \( v \).
Use the Find procedure to locate the rightmost leaf $v$ storing an element no greater than $x$.

Create a new leaf, store $x$ at this leaf, and make it the right sibling of $v$.

Is the result still an $(a, b)$-tree? No.
The Insert Operation

- Use the Find procedure to locate the rightmost leaf \( v \) storing an element no greater than \( x \).
- Create a new leaf, store \( x \) at this leaf, and make it the right sibling of \( v \).

Is the result still an \((a, b)\)-tree? No.

How do we rebalance?
Node Splitting

```plaintext
1
/  \
1  34
/  \
1 15
/  \
34 41
/  \
43
/  \
43 66
/  \
76
/  \
90
```
Node Splitting

A diagram showing a tree structure with nodes containing numbers. The tree is split at various points to accommodate more data points.
Node Splitting

CSci 3110 • Data Structuring • 15/81
A node split takes $O(b) = O(1)$ time.
A node split takes $\mathcal{O}(b) = \mathcal{O}(1)$ time.

There is at most one node split per level.
A node split takes $O(b) = O(1)$ time.

There is at most one node split per level.

Lemma: An insertion into an $(a, b)$-tree takes
A node split takes $O(b) = O(1)$ time.

There is at most one node split per level.

**Lemma:** An insertion into an $(a, b)$-tree takes $O(\lg n)$ time.
What do we do when we split the root?
What do we do when we split the root?
Splitting the Root

What do we do when we split the root?
What do we do when we split the root?

Note: This is the situation why we have to allow the root to have degree less than \( a \).
The Delete Operation

Diagram of a binary search tree showing the deletion of node 80.
The Delete Operation
The Delete Operation

The diagram illustrates a process where a value (81) is deleted from a tree structure. The tree contains various values, and the deletion process is shown by removing the value 81 from the root of the tree.
The Delete Operation

- Remove the leaf storing $x$ from its parent's child list.
- Update the keys of all its ancestors.
The Delete Operation

- Remove the leaf storing \( x \) from its parent’s child list.
- Update the keys of all its ancestors.
- Rebalance using node fusions.
Node Fusing
Node Fusing
A node fusion takes $O(b) = O(1)$ time.
A node fusion takes $O(b) = O(1)$ time.

There is at most one node fusion per level.
A node fusion takes $O(b) = O(1)$ time.
There is at most one node fusion per level.

**Lemma:** A deletion from an $(a, b)$-tree takes
A node fusion takes $O(b) = O(1)$ time.

There is at most one node fusion per level.

**Lemma:** A deletion from an $(a, b)$-tree takes $O(\lg n)$ time.
Fusing Children of the Root

What do we do when the root degree becomes 1?
What do we do when the root degree becomes 1?

- We remove the root.
Fusing Children of the Root

What do we do when the root degree becomes 1?

- We remove the root.
What if node $v$ and its sibling together have more than $b$ children?
Node Sharing

What if node $v$ and its sibling together have more than $b$ children?

- We fuse
Node Sharing

What if node $v$ and its sibling together have more than $b$ children?

- We fuse
What if node $v$ and its sibling together have more than $b$ children?

- We fuse and then split (essentially borrowing children from $v$’s sibling).
What if node $v$ and its sibling together have more than $b$ children?

- We fuse and then split (essentially borrowing children from $v$’s sibling).
What if node $v$ and its sibling together have more than $b$ children?
- We fuse and then split (essentially borrowing children from $v$’s sibling).

**Lemma:** After a fusion followed by a split, the tree is an $(a, b)$-tree again.
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than \( b \) and
    - whose right sibling does not exist or has a key no less than \( a \).
  - At every leaf, report the stored point if it is in the query range.

Query range: \([35, 76]\)
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than \( b \) and
    - whose right sibling does not exist or has a key no less than \( a \).
  - At every leaf, report the stored point if it is in the query range.

Query range: \([35, 76]\)
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than $b$ and
    - whose right sibling does not exist or has a key no less than $a$.
  - At every leaf, report the stored point if it is in the query range.

Query range: $[35, 76]$
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than $b$ and
    - whose right sibling does not exist or has a key no less than $a$.
  - At every leaf, report the stored point if it is in the query range.

Query range: $[35, 76]$
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than $b$ and
    - whose right sibling does not exist or has a key no less than $a$.
  - At every leaf, report the stored point if it is in the query range.

Query range: [35, 76]
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than \( b \) and
    - whose right sibling does not exist or has a key no less than \( a \).
  - At every leaf, report the stored point if it is in the query range.

Query range: \([35, 76]\)
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than \( b \) and
    - whose right sibling does not exist or has a key no less than \( a \).
  - At every leaf, report the stored point if it is in the query range.

Query range: \([35, 76]\)
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than $b$ and
    - whose right sibling does not exist or has a key no less than $a$.
  - At every leaf, report the stored point if it is in the query range.

Query range: $[35, 76]$
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than \( b \) and
    - whose right sibling does not exist or has a key no less than \( a \).
  - At every leaf, report the stored point if it is in the query range.

Query range: \([35, 76]\)
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than \( b \) and
    - whose right sibling does not exist or has a key no less than \( a \).
  - At every leaf, report the stored point if it is in the query range.

Query range: \([35, 76]\)
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than $b$ and
    - whose right sibling does not exist or has a key no less than $a$.
  - At every leaf, report the stored point if it is in the query range.

Query range: [35, 76]
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than \( b \) and
    - whose right sibling does not exist or has a key no less than \( a \).
  - At every leaf, report the stored point if it is in the query range.

Query range: \([35, 76]\)
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than $b$ and
    - whose right sibling does not exist or has a key no less than $a$.
  - At every leaf, report the stored point if it is in the query range.

Query range: [35, 76]
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children whose key is no greater than \( b \) and whose right sibling does not exist or has a key no less than \( a \).
  - At every leaf, report the stored point if it is in the query range.

Query range: \([35, 76]\)
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than $b$ and
    - whose right sibling does not exist or has a key no less than $a$.
  - At every leaf, report the stored point if it is in the query range.

Query range: $[35, 76]$
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than \( b \) and
    - whose right sibling does not exist or has a key no less than \( a \).
  - At every leaf, report the stored point if it is in the query range.

Query range: [35, 76]
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than $b$ and
    - whose right sibling does not exist or has a key no less than $a$.
  - At every leaf, report the stored point if it is in the query range.

Query range: $[35, 76]$
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than $b$ and
    - whose right sibling does not exist or has a key no less than $a$.
  - At every leaf, report the stored point if it is in the query range.

Query range: $[35, 76]$
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than $b$ and
    - whose right sibling does not exist or has a key no less than $a$.
  - At every leaf, report the stored point if it is in the query range.

Query range: $[35, 76]$
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than $b$ and
    - whose right sibling does not exist or has a key no less than $a$.
  - At every leaf, report the stored point if it is in the query range.

Query range: $[35, 76]$
The Range-Query Operation

- Perform a depth-first traversal of the tree:
  - At every internal node, recursively visit all children
    - whose key is no greater than \( b \) and
    - whose right sibling does not exist or has a key no less than \( a \).
  - At every leaf, report the stored point if it is in the query range.

Query range: \([35, 76]\)
Lemma: Procedure Range-Query reports all points in the query range (and only those).
**Lemma:** Procedure Range-Query reports all points in the query range (and only those).

**Lemma:** Procedure Range-Query takes
Lemma: Procedure Range-Query reports all points in the query range (and only those).

Lemma: Procedure Range-Query takes $O(\lg n + t)$ time, where $t$ is the number of points in the query range.
Lemma: Procedure Range-Query reports all points in the query range (and only those).

Lemma: Procedure Range-Query takes $O(\lg n + t)$ time, where $t$ is the number of points in the query range.

- Visiting every brown or green node takes constant time.
Lemma: Procedure Range-Query reports all points in the query range (and only those).

Lemma: Procedure Range-Query takes $O(\lg n + t)$ time, where $t$ is the number of points in the query range.

- Visiting every brown or green node takes constant time.
- There are at most two brown nodes per level $\Rightarrow O(\lg n)$ time.
Disk-based Dictionaries

Disk storage:
Disk storage:

- Disks are slow.
Disk-Based Dictionaries

Disk storage:

- Disks are slow.
- Main factor: seek time.
Disk-Based Dictionaries

**Disk storage:**

- Disks are slow.
- Main factor: seek time.

∴ Amortization of seek time: access more than one item at a time.
Disk-Based Dictionaries

Disk storage:

- Disks are slow.
- Main factor: seek time.
  - Amortization of seek time: access more than one item at a time.
  - Disks are divided into blocks.
Disk-Based Dictionaries

Disk storage:

- Disks are slow.
- Main factor: seek time.
  - Amortization of seek time: access more than one item at a time.
  - Disks are divided into blocks.
- One disk access = read or write one block.
Disk-Based Dictionaries

**Disk storage:**

- Disks are slow.
- Main factor: seek time.
  - Amortization of seek time: access more than one item at a time.
  - Disks are divided into blocks.
- One disk access = read or write one block.

**Goal:** Develop an efficient ordered dictionary for storing data on disk.
Disk-Based Dictionaries

Disk storage:

- Disks are slow.
- Main factor: seek time.
  - Amortization of seek time: access more than one item at a time.
  - Disks are divided into blocks.
- One disk access = read or write one block.

Goal: Develop an efficient ordered dictionary for storing data on disk.

- Every computation step takes $O(1)$ disk accesses.
Disk-Based Dictionaries

**Disk storage:**
- Disks are slow.
- Main factor: seek time.
  - Amortization of seek time: access more than one item at a time.
  - Disks are divided into blocks.
- One disk access = read or write one block.

**Goal:** Develop an efficient ordered dictionary for storing data on disk.

- Every computation step takes $O(1)$ disk accesses.
  - Balanced search trees require $O(lg n)$ disk accesses per operation.
B-Trees are a variant of \((a, b)\)-trees:

- \(B\) = number of items/pointers that can be stored in a disk block.
- \(a = B/4\) and \(b = B/2\).
- Do not store parent pointers.
- Every node stores direct pointers to its children.
- Every node stores the keys of its children.
- Every leaf stores between \(B/2\) and \(B\) items.
Lemma: B-Trees support operations Insert, Delete, Find, Predecessor, Successor, Minimum, and Maximum in $O(\log_B n)$ disk accesses. Procedure Range-Search takes $O(\log_B n + t/B)$ disk accesses.
**(a, b)-Trees: Summary**

**(a, b)-Trees are efficient ordered dictionaries:**

- Operations Insert, Delete, Find, Predecessor, Successor, Minimum, and Maximum take $O(\lg n)$ time.
- Operation Range-Query takes $O(\lg n + t)$ time.

*B-trees, a variant of (a, b)-trees, are efficient ordered dictionaries for storage on disk:*

- Operations Insert, Delete, Find, Predecessor, Successor, Minimum, and Maximum take $O(\log_B n)$ disk accesses.
- Operation Range-Query takes $O(\log_B n + t/B)$ disk accesses.
The data structuring paradigm:

Delegate the non-trivial work to one or more data structures.

Most common application:

- Model the computation as a sequence of transformations.
- Use data structures to represent the current state.
- Update data structure to reflect the effect of applied transformations.
Graph exploration algorithms use data structures to maintain the unexplored vertices adjacent to explored ones. (BFS, DFS, Dijkstra’s algorithm, Prim’s algorithm)

Kruskal’s algorithm uses a union-find structure to maintain the set of spanning forests.

Huffman’s algorithm maintains the current set of characters in a priority queue.
Core of the problem: Find the intersection points.
**Special case:** Find all intersections between

- a set, $V$, of $n$ vertical segments, $v_1, v_2, \ldots, v_n$, and
- a set, $H$, of $m$ horizontal segments, $h_1, h_2, \ldots, h_m$. 
Output-Sensitivity

How many intersections are there in the worst case?
Output-Sensitivity

How many intersections are there in the worst case?

So the trivial algorithm of testing every pair of segments is optimal.
Output-Sensitivity

How many intersections are there in the worst case?

So the trivial algorithm of testing every pair of segments is optimal.

Can we still do better?
Output-Sensitivity

How many intersections are there in the worst case?

So the trivial algorithm of testing every pair of segments is optimal.

Can we still do better?

■ Yes, we can aim to spend little time unless the output is big.
■ This is called output-sensitivity.
The Sweep-Line Paradigm

Idea:

- Sweep a horizontal *sweep line* across the plane, bottom-up.
- Use *sweep-line structure* to maintain the interaction between scene and sweep line.
**Event Points**

*Discretization of sweep-line paradigm:*

- Update sweep-line structure only at certain *event points*.
- Solve problem by asking queries on sweep-line structure at other event points.
**Sweep-line status:** 

$((a, b))$-tree $T$ storing intersections between sweep-line and vertical segments, sorted from left to right.
**Sweep-line status:** $(a, b)$-tree $T$ storing intersections between sweep-line and vertical segments, sorted from left to right.

**Event points:**

- Bottom endpoint of vertical segment $v_i$:

- Top endpoint of vertical segment $v_i$:

- Horizontal segment $h_j$:
Sweep-line status: \((a, b)\)-tree \(T\) storing intersections between sweep-line and vertical segments, sorted from left to right.

Event points:

- Bottom endpoint of vertical segment \(v_i\):
  - Sweep line starts to intersect \(v_i\).
  \[
  \therefore \text{ Insert } v_i \text{ into } T. 
  \]

- Top endpoint of vertical segment \(v_i\):

- Horizontal segment \(h_j\):
**Sweep-line status:** \((a, b)\)-tree \(T\) storing intersections between sweep-line and vertical segments, sorted from left to right.

**Event points:**

- **Bottom endpoint of vertical segment** \(v_i\):
  - Sweep line starts to intersect \(v_i\).
  - \(\therefore\) Insert \(v_i\) into \(T\).

- **Top endpoint of vertical segment** \(v_i\):
  - Sweep line stops intersecting \(v_i\).
  - \(\therefore\) Delete \(v_i\) from \(T\).

- **Horizontal segment** \(h_j\):
**Sweep-line status:** \((a, b)\)-tree \(T\) storing intersections between sweep-line and vertical segments, sorted from left to right.

**Event points:**

- **Bottom endpoint of vertical segment** \(v_i\):
  - Sweep line starts to intersect \(v_i\).
  - \(\therefore\) Insert \(v_i\) into \(T\).

- **Top endpoint of vertical segment** \(v_i\):
  - Sweep line stops intersecting \(v_i\).
  - \(\therefore\) Delete \(v_i\) from \(T\).

- **Horizontal segment** \(h_j\):
  - Sweep line intersects all vertical segments whose \(y\)-range includes the \(y\)-coordinate of \(h_j\).
  - These are the segments in \(T\).
  - \(\therefore\) Find intersections with \(h_j\) by answering a range query on \(T\).
Orthogonal Line-Segment Intersection: Analysis

Event points:
Orthogonal Line-Segment Intersection: Analysis

**Event points:**

- $n$ bottom endpoints of vertical segments $\implies n$ insertions into $T$. 
Orthogonal Line-Segment Intersection: Analysis

Event points:

- $n$ bottom endpoints of vertical segments $\Rightarrow n$ insertions into $T$.
- $n$ top endpoints of vertical segments $\Rightarrow n$ deletions from $T$. 
Orthogonal Line-Segment Intersection: Analysis

Event points:

- $n$ bottom endpoints of vertical segments $\Rightarrow$ $n$ insertions into $T$.
- $n$ top endpoints of vertical segments $\Rightarrow$ $n$ deletions from $T$.
- $m$ horizontal segments $\Rightarrow$ $m$ range queries on $T$. 
Orthogonal Line-Segment Intersection: Analysis

Event points:

- $n$ bottom endpoints of vertical segments $\Rightarrow n$ insertions into $T$.
- $n$ top endpoints of vertical segments $\Rightarrow n$ deletions from $T$.
- $m$ horizontal segments $\Rightarrow m$ range queries on $T$.

Cost per insertion or deletion $= \mathcal{O}(\lg n)$. 
Orthogonal Line-Segment Intersection: Analysis

Event points:
- $n$ bottom endpoints of vertical segments ⇒ $n$ insertions into $T$.
- $n$ top endpoints of vertical segments ⇒ $n$ deletions from $T$.
- $m$ horizontal segments ⇒ $m$ range queries on $T$.

- Cost per insertion or deletion = $O(\log n)$.
- Cost per range query with segment $h_j$ is $O(\log n + t_j)$, where $t_j$ is the number of segments intersecting $h_j$. 
Orthogonal Line-Segment Intersection: Analysis

Event points:
- $n$ bottom endpoints of vertical segments $\Rightarrow n$ insertions into $T$.
- $n$ top endpoints of vertical segments $\Rightarrow n$ deletions from $T$.
- $m$ horizontal segments $\Rightarrow m$ range queries on $T$.

Cost per insertion or deletion $= \mathcal{O}(\lg n)$.
Cost per range query with segment $h_j$ is $\mathcal{O}(\lg n + t_j)$, where $t_j$ is the number of segments intersecting $h_j$.

Total cost:

$$\mathcal{O}((n + m) \lg n) + \sum_{j=1}^{m} \mathcal{O}(t_j) = \mathcal{O}((n + m) \lg n + t)$$
Orthogonal Line-Segment Intersection: Analysis

Event points:
- $n$ bottom endpoints of vertical segments $\Rightarrow$ $n$ insertions into $T$.
- $n$ top endpoints of vertical segments $\Rightarrow$ $n$ deletions from $T$.
- $m$ horizontal segments $\Rightarrow$ $m$ range queries on $T$.

- Cost per insertion or deletion $= \mathcal{O}(\lg n)$.
- Cost per range query with segment $h_j$ is $\mathcal{O}(\lg n + t_j)$, where $t_j$ is the number of segments intersecting $h_j$.

Total cost:

$$\mathcal{O}((n + m) \lg n) + \sum_{j=1}^{m} \mathcal{O}(t_j) = \mathcal{O}((n + m) \lg n + t)$$

Theorem: The orthogonal line-segment intersection problem can be solved in $\mathcal{O}((n + m) \lg n + t)$ time.
Line-Segment Intersection
Questions:
- What is the sweep-line status?
Line-Segment Intersection

Questions:
- What is the sweep-line status?
  All segments intersecting the sweep line.
- How do we order the segments?
Questions:

- What is the sweep-line status?
  All segments intersecting the sweep line.

- How do we order the segments?
  By the $x$-coordinates of their intersections with the sweep line.

- Where does the sweep-line status change?
Questions:
- What is the sweep-line status?
  All segments intersecting the sweep line.
- How do we order the segments?
  By the $x$-coordinates of their intersections with the sweep line.
- Where does the sweep-line status change?
  At segment endpoints and intersection points!
The Event Schedule

**Apparent problem:** We want to compute intersections points, but they are part of the event schedule.
The Event Schedule

**Apparent problem:** We want to compute intersections points, but they are part of the event schedule.

**Consequence:** We cannot generate all event points before we start the sweep.
**Apparent problem:** We want to compute intersections points, but they are part of the event schedule.

**Consequence:** We cannot generate all event points before we start the sweep.

**Solution:**

- Maintain set of event points in a priority queue $Q$, sorted by $y$-coordinates.
- Initially, $Q$ contains all segment endpoints.
- As we detect intersections, we insert them into $Q$. 
Observation: If two segments $s_1$ and $s_2$ intersect, the sweep-line must intersect $s_1$ and $s_2$ simultaneously at some point.
Detecting Intersection Points: First Attempt

**Observation:** If two segments $s_1$ and $s_2$ intersect, the sweep-line must intersect $s_1$ and $s_2$ simultaneously at some point.

**Idea:**
- As in the orthogonal case, insert and delete segments into and from $T$ when the sweep line passes their endpoints.
- When inserting a segment into $T$, test for intersections with all other segments in $T$. 
Problem: We may perform a quadratic number of intersection tests only to discover that there are no intersections.
Observation: Two segments $s_1$ and $s_2$ that intersect are adjacent in $T$ immediately before they intersect.
**Bottom endpoint:**

- Insert $s$ into $T$ and test for intersections with its two neighbours, $s_1$ and $s_2$.
- If there are intersections, insert them into event schedule.
- If $s_1$ and $s_2$ intersect after the current $y$-coordinate, remove the intersection point from event schedule.
**Top endpoint:**

- Delete $s$ from $T$.
- Test for intersection between the two segments that become adjacent in $T$.
- If they intersect, insert intersection point into event schedule.
**Intersection point:**

- Report the intersection.
- Swap the order of the two intersecting segments.
- Remove intersections with their old neighbours from the event schedule.
- Test for intersections with their new neighbours and insert them into the event schedule if necessary.
2n + t event points:

- n bottom endpoints
- n top endpoints
- t intersection points

Each incurs $O(1)$ updates and queries on $T$ and $Q$.

∴ Cost per event point is $O(lg \, n)$.

**Theorem:** The line-segment intersection problem can be solved in $O((n + t) \, lg \, n)$ time.
**Dynamic Order Statistics**

**Problem:** Maintain a set $S$ of numbers under insertion and deletion and support the following two types of queries:

- **Rank** $(S, x)$: Report $\text{rank}_S(x) = 1 + |\{ y \in S \mid y < x \}|$.
- **Select** $(S, k)$: Report the $k$-th order statistic of $S$.

---

**Examples:**

- Insert(18)
  - $S = \{8, 1, 5, 12, 34, 3\}$
  - $\text{rank}(29) = 7$
  - $\text{select}(5) = 12$

- Delete(8)
  - $S = \{5, 12, 34, 3\}$
  - $\text{rank}(29) = 6$
  - $\text{select}(5) = 27$

- Insert(18)
  - $S = \{18, 5, 12, 34, 3\}$
  - $\text{rank}(29) = 7$
  - $\text{select}(5) = 18$

---

**CSci 3110 • Data Structuring • 45/81**
Orthogonal Line-Segment Intersection Counting

**Problem:** Instead of reporting all intersections between a set of vertical and a set of horizontal segments, only count how many there are.
Orthogonal Line-Segment Intersection Counting

**Problem:** Instead of reporting all intersections between a set of vertical and a set of horizontal segments, only count how many there are.

Segment $h$ intersects $\text{rank}(b) - \text{rank}(a) = 3$ vertical segments.
Orthogonal Line-Segment Intersection Counting

**Problem:** Instead of reporting all intersections between a set of vertical and a set of horizontal segments, only count how many there are.

- Instead of asking a range query for every horizontal segment, ask two rank queries.

- Segment $h$ intersects $\text{rank}(b) - \text{rank}(a) = 3$ vertical segments.
Orthogonal Line-Segment Intersection Counting

**Problem:** Instead of reporting all intersections between a set of vertical and a set of horizontal segments, only count how many there are.

Instead of asking a range query for every horizontal segment, ask two rank queries.

**Lemma:** If `Insert`, `Delete`, and `Rank` operations can be supported in $O(\log n)$ time, the orthogonal line-segment intersection counting problem can be solved in $O(n \log n)$ time.
Observation: The rank of an element is one more than the number of leaves to the left of the path to the corresponding leaf.
In addition to the normal information, store at every node $v$ the number of leaves in $v$’s subtree.
Lemma: Rank queries can be answered in $O(\lg n)$ time using the leaf counts.

rank(77) = 17 = 1 + (5 + 5 + 3 + 2 + 1)
Lemma: Select queries can be answered in $O(\log n)$ time using the leaf counts.

rank(77) = 17 = 1 + (5 + 5 + 3 + 2 + 1)
Insertions

After an insertion of a new leaf $v$, which leaf counts need to be updated?
After an insertion of a new leaf $v$, which leaf counts need to be updated? Those of $v$’s ancestors must be increased by one.
After an insertion of a new leaf $v$, which leaf counts need to be updated?

Those of $v$'s ancestors must be increased by one.
Deletions

After a deletion of a leaf $v$, which leaf counts need to be updated?
Deletions

After a deletion of a leaf $v$, which leaf counts need to be updated?

Those of $v$’s ancestors must be decreased by one.
After a deletion of a leaf \( v \), which leaf counts need to be updated?

Those of \( v \)'s ancestors must be decreased by one.
Node Splits
Node Splits

- The leaf counts of nodes $v_1$ and $v_2$ are the sums of the leaf counts of their children.
Node Splits

- The leaf counts of nodes $v_1$ and $v_2$ are the sums of the leaf counts of their children.
- All other leaf counts remain unchanged.
Node Splits

- The leaf counts of nodes $v_1$ and $v_2$ are the sums of the leaf counts of their children.
- All other leaf counts remain unchanged.

Lemma: A node split takes $\mathcal{O}(1)$ time, including the time to recompute leaf counts.
The leaf counts of nodes $v_1$ and $v_2$ are the sums of the leaf counts of their children.

All other leaf counts remain unchanged.

**Lemma:** A node split takes $O(1)$ time, including the time to recompute leaf counts.

**Corollary:** An insertion into a dynamic order statistics tree takes $O(\lg n)$ time.
Node Fusions

- The leaf counts of the fused node $v$ is the sum of the leaf counts of its children.
- All other leaf counts remain unchanged.

**Lemma:** A node fusion takes $\mathcal{O}(1)$ time, including the time to recompute leaf counts.

**Corollary:** A deletion from a dynamic order statistics tree takes $\mathcal{O}(\lg n)$ time.
Theorem: There exists a data structure that can maintain a set, $S$, of numbers under insertions and deletions and supports Rank and Select queries. Each update or query takes $O(\lg n)$ time.
**Problem:** Maintain a set, \( S \), of points in the plane under insertions and deletions and support three-sided range queries, that is, queries of the type:

*Report all points whose \( x \)-coordinates are between \( x_l \) and \( x_r \) and whose \( y \)-coordinates are at least \( y_b \).*
3-Sided Range Searching and \((a, b)\)-Trees

**First attempt:**

- Store points in an \((a, b)\)-tree based on their \(x\)-coordinates.
- To answer a 3-sided range query, answer a standard range query, but output only those points whose \(y\)-coordinates are above the bottom boundary of the query.
3-Sided Range Searching and \((a, b)-Trees\)

**First attempt:**
- Store points in an \((a, b)-tree\) based on their \(x\)-coordinates.
- To answer a 3-sided range query, answer a standard range query, but output only those points whose \(y\)-coordinates are above the bottom boundary of the query.

**Problem:**
We inspect too many nodes without reporting points.
First attempt:

- Store points in an \((a, b)\)-tree based on their \(x\)-coordinates.
- To answer a 3-sided range query, answer a standard range query, but output only those points whose \(y\)-coordinates are above the bottom boundary of the query.

Problem:

We inspect too many nodes without reporting points.
Priority search tree:

- Start by building an \((a, b)\)-tree on the \(x\)-coordinates.
Priority search tree:

- Start by building an \((a, b)\)-tree on the \(x\)-coordinates.
- Then propagate points up the tree to establish max-heap property on \(y\)-coordinates.
Priority search tree:

- Start by building an \((a, b)\)-tree on the \(x\)-coordinates.
- Then propagate points up the tree to establish max-heap property on \(y\)-coordinates.
Priority search tree:

- Start by building an \((a, b)\)-tree on the \(x\)-coordinates.
- Then propagate points up the tree to establish max-heap property on \(y\)-coordinates.
Priority search tree:

- Start by building an \((a, b)\)-tree on the \(x\)-coordinates.
- Then propagate points up the tree to establish max-heap property on \(y\)-coordinates.
Priority Search Trees

Priority search tree:

- Start by building an \((a, b)\)-tree on the \(x\)-coordinates.
- Then propagate points up the tree to establish max-heap property on \(y\)-coordinates.
Priority Search Trees

Priority search tree:

- Start by building an \((a, b)\)-tree on the \(x\)-coordinates.
- Then propagate points up the tree to establish max-heap property on \(y\)-coordinates.
Priority search tree:

- Start by building an \((a, b)\)-tree on the \(x\)-coordinates.
- Then propagate points up the tree to establish max-heap property on \(y\)-coordinates.
Priority search tree:

- Start by building an \((a, b)\)-tree on the \(x\)-coordinates.
- Then propagate points up the tree to establish max-heap property on \(y\)-coordinates.
Priority search tree:

- Start by building an \((a, b)\)-tree on the \(x\)-coordinates.
- Then propagate points up the tree to establish max-heap property on \(y\)-coordinates.
Three-Sided Range Queries

- As in a standard range query, traverse the two paths to leftmost and rightmost leaves in the $x$-range of the query.
- For every node on these two paths, inspect the point stored at the node and report it if it is in the query range.
For every node node $v$ between the extreme paths:

- Inspect the point stored at $v$.
- If there is no point or its $y$-coordinate is below the bottom boundary of the query, stop.
- Otherwise, report the point and recursively inspect $v$’s children.
Three-Sided Range Queries: Analysis

- We spend constant time per node on the two extreme paths.
- There are $O(\lg n)$ such nodes, at most two per level.
∴ Traversing the two extreme paths takes $O(\lg n)$ time.
Three-Sided Range Queries: Analysis

- We spend constant time per node on the two extreme paths.
- There are $O(\lg n)$ such nodes, at most two per level.
  \[\therefore \text{Traversing the two extreme paths takes } O(\lg n) \text{ time.}\]
- For every node we visit between the two extreme paths, we spend constant time.
  - \textit{We visit at most } $b(\lg n + t) = O(\lg n + t) \text{ such nodes.}$
Three-Sided Range Queries: Analysis

- We spend constant time per node on the two extreme paths.
- There are $\mathcal{O}(\lg n)$ such nodes, at most two per level.
  \[ \therefore \text{Traversing the two extreme paths takes } \mathcal{O}(\lg n) \text{ time.} \]

- For every node we visit between the two extreme paths, we spend constant time.
- **We visit at most** $b(\lg n + t) = \mathcal{O}(\lg n + t)$ such nodes.
  \[ \therefore \text{Visiting nodes between the two extreme paths takes } \mathcal{O}(\lg n + t) \text{ time.} \]
Three-Sided Range Queries: Analysis

- We spend constant time per node on the two extreme paths.
- There are $O(lg \ n)$ such nodes, at most two per level.
- \[ \therefore \text{Traversing the two extreme paths takes } O(lg \ n) \text{ time.} \]

- For every node we visit between the two extreme paths, we spend constant time.
- \[ \textbf{We visit at most } b(lg \ n + t) = O(lg \ n + t) \text{ such nodes.} \]
- \[ \therefore \text{Visiting nodes between the two extreme paths takes } O(lg \ n + t) \text{ time.} \]

\textbf{Lemma:} A priority search tree supports three-sided range queries in $O(lg \ n + t)$ time.
Insert $p$ as into a standard $(a, b)$-tree, based on $x$-coordinates.
Insertions

- Insert $p$ as into a standard $(a, b)$-tree, based on $x$-coordinates.
- Perform a Heapify-Up operation starting at the new leaf.
Insertions

- Insert $p$ as into a standard $(a, b)$-tree, based on $x$-coordinates.
- Perform a Heapify-Up operation starting at the new leaf.
Insertions

- Insert \( p \) as into a standard \((a, b)\)-tree, based on \( x \)-coordinates.

- Locate the lowest ancestor \( v \) whose parent does not store a point \( q \) with \( y_q < y_p \).
Insert $p$ as into a standard $(a, b)$-tree, based on $x$-coordinates.

Locate the lowest ancestor $v$ whose parent does not store a point $q$ with $y_q < y_p$.

While $p \neq \text{nil}$:

- Replace point $q$ at current node $v$ with $p$
- $p \leftarrow q$
- Proceed to $v$'s child that is an ancestor of $p$'s leaf.
Insertions

- Insert $p$ as into a standard $(a, b)$-tree, based on $x$-coordinates.

- Locate the lowest ancestor $v$ whose parent does not store a point $q$ with $y_q < y_p$.

- While $p \neq \text{nil}$:
  - Replace point $q$ at current node $v$ with $p$
  - $p \leftarrow q$
  - Proceed to $v$'s child that is an ancestor of $p$'s leaf.
Insert $p$ as into a standard $(a, b)$-tree, based on $x$-coordinates.

- Locate the lowest ancestor $v$ whose parent does not store a point $q$ with $y_q < y_p$.

- While $p \neq \text{nil}$:
  - Replace point $q$ at current node $v$ with $p$
  - $p \leftarrow q$
  - Proceed to $v$’s child that is an ancestor of $p$’s leaf.
Insert $p$ as into a standard $(a, b)$-tree, based on $x$-coordinates.

Locate the lowest ancestor $v$ whose parent does not store a point $q$ with $y_q < y_p$.

While $p \neq \text{nil}$:
- Replace point $q$ at current node $v$ with $p$
- $p \leftarrow q$
- Proceed to $v$'s child that is an ancestor of $p$'s leaf.
Insertions

- Insert \( p \) as into a standard \((a, b)\)-tree, based on \( x \)-coordinates.

- Locate the lowest ancestor \( v \) whose parent does not store a point \( q \) with \( y_q < y_p \).

- While \( p \neq \text{nil} \):
  - Replace point \( q \) at current node \( v \) with \( p \)
  - \( p \leftarrow q \)
  - Proceed to \( v \)'s child that is an ancestor of \( p \)'s leaf.
Deletions

A diagram showing a tree with nodes labeled from $p_1$ to $p_{14}$, illustrating the concept of deletions in data structures.
Delete the leaf corresponding to $p$ as from a standard $(a, b)$-tree.
Delete the leaf corresponding to $p$ as from a standard $(a, b)$-tree.
Deletions

- Delete the leaf corresponding to $p$ as from a standard $(a, b)$-tree.
- Delete $p$ from the node where it is stored.
Deletions

- Delete the leaf corresponding to $p$ as from a standard $(a, b)$-tree.
- Delete $p$ from the node where it is stored.
Delete the leaf corresponding to $p$ as from a standard $(a, b)$-tree.

Delete $p$ from the node where it is stored.

While the current node $v$ has a child that stores a point:

- Choose the child $w$ whose point $p$ has maximum $y$-coordinate
- Store $p$ at $v$
- $v \leftarrow w$
Deletions

- Delete the leaf corresponding to \( p \) as from a standard \((a, b)\)-tree.
- Delete \( p \) from the node where it is stored.
- While the current node \( v \) has a child that stores a point:
  - Choose the child \( w \) whose point \( p \) has maximum \( y \)-coordinate
  - Store \( p \) at \( v \)
  - \( v \leftarrow w \)
Deletions

- Delete the leaf corresponding to $p$ as from a standard $(a, b)$-tree.
- Delete $p$ from the node where it is stored.
- While the current node $v$ has a child that stores a point:
  - Choose the child $w$ whose point $p$ has maximum $y$-coordinate
  - Store $p$ at $v$
  - $v \leftarrow w$
Node Splits
Node Splits

- Where do we store $p$?
- What do we store at the two nodes resulting from the split?
Node Splits

- Where do we store \( p \)?
- What do we store at the two nodes resulting from the split?
- The one that is an ancestor of \( p \)'s leaf stores \( p \).
- Where do we store $p$?
- What do we store at the two nodes resulting from the split?
- The one that is an ancestor of $p$'s leaf stores $p$. 
Node Splits

- Where do we store \( p \)?
- What do we store at the two nodes resulting from the split?
  - The one that is an ancestor of \( p \)'s leaf stores \( p \).
  - For the other one, we apply the same bubble-up operation as after a deletion.
Node Fusions

Diagram of node fusions showing nodes p and q.
Which of the two points do we store at the merged node?
Node Fusions

Which of the two points do we store at the merged node?
- The one with greater $y$-coordinate.
Node Fusions

- Which of the two points do we store at the merged node?
- The one with greater \( y \)-coordinate.
Node Fusions

Which of the two points do we store at the merged node?

- The one with greater $y$-coordinate.
- The other one is pushed down as after an insertion.
Node Fusions

Which of the two points do we store at the merged node?

- The one with greater $y$-coordinate.
- The other one is pushed down as after an insertion.
Priority Search Tree Updates: Analysis

Insertion:
- Insertion as into an \((a, b)\)-tree \(\Rightarrow O(lg \, n)\)
- Locate ancestor where to store \(p\) \(\Rightarrow O(lg \, n)\)
- Trickle-down operation \(\Rightarrow O(lg \, n)\)
- \(O(lg \, n)\) node splits

Deletion:
- Deletion as from an \((a, b)\)-tree \(\Rightarrow O(lg \, n)\)
- Locate ancestor where \(p\) is stored and delete \(p\) \(\Rightarrow O(lg \, n)\)
- Bubble-up operation \(\Rightarrow O(lg \, n)\)
- \(O(lg \, n)\) node fusions and at most one node split
**Node split:**
- Regular split $\Rightarrow \mathcal{O}(1)$
- Bubble-up operation $\Rightarrow \mathcal{O}(\lg n)$

**Node fusion:**
- Regular fusion $\Rightarrow \mathcal{O}(1)$
- Trickle-down operation $\Rightarrow \mathcal{O}(\lg n)$

**Lemma:** A priority search tree supports insertions and deletions in $\mathcal{O}(\lg^2 n)$ time.
Weight-Balanced \((a, b)\)-Trees
Weight-Balanced \((a, b)\)-Trees

Weight of a node = number of leaf descendants

**New balancing condition:** (Weight balancing)

- The root has at least two children.
- Any non-leaf node at height \(h\) has weight at most \(\beta \gamma^h\).
- Any non-leaf, non-root node at height \(h\) has weight at least \(\alpha \gamma^h\).

\[
\alpha = \sqrt{a}, \quad \beta = \sqrt{b}, \quad \text{and} \quad \gamma = \alpha \beta.
\]
Lemma: In a weight-balanced $(a, b)$-tree, the root has degree between 2 and $b$. Any other internal node has degree between $a$ and $b$. 
Lemma: In a weight-balanced \((a, b)\)-tree, the root has degree between 2 and \(b\). Any other internal node has degree between \(a\) and \(b\).

Lower bound:
Lemma: In a weight-balanced \((a, b)\)-tree, the root has degree between 2 and \(b\). Any other internal node has degree between \(a\) and \(b\).

Lower bound:

- For the root, obvious.
Lemma: In a weight-balanced \((a, b)\)-tree, the root has degree between 2 and \(b\). Any other internal node has degree between \(a\) and \(b\).

Lower bound:

■ For the root, obvious.
■ For any other node \(v\):
Lemma: In a weight-balanced \((a, b)\)-tree, the root has degree between 2 and \(b\). Any other internal node has degree between \(a\) and \(b\).

**Lower bound:**

- For the root, obvious.
- For any other node \(v\):
  - \(\text{weight}(v) \geq \alpha \gamma^h\)
Lemma: In a weight-balanced \((a, b)\)-tree, the root has degree between 2 and \(b\). Any other internal node has degree between \(a\) and \(b\).

Lower bound:

- For the root, obvious.
- For any other node \(v\):
  - \(\text{weight}(v) \geq \alpha \gamma^h\)
  - \(\text{weight}(w) \leq \beta \gamma^{h-1}\), for every child, \(w\), of \(v\)
Lemma: In a weight-balanced \((a, b)\)-tree, the root has degree between 2 and \(b\). Any other internal node has degree between \(a\) and \(b\).

**Lower bound:**

- For the root, obvious.
- For any other node \(v\):
  - \(\text{weight}(v) \geq \alpha \gamma^h\)
  - \(\text{weight}(w) \leq \beta \gamma^{h-1}\), for every child, \(w\), of \(v\)

\[
\therefore \deg(v) \geq \frac{\alpha \gamma^h}{\beta \gamma^{h-1}} = \frac{\alpha \gamma}{\beta} = \frac{\alpha^2 \beta}{\beta} = a.
\]
Weight Balance Implies Degree Balance

Lemma: In a weight-balanced \((a, b)\)-tree, the root has degree between 2 and \(b\). Any other internal node has degree between \(a\) and \(b\).

Lower bound:
- For the root, obvious.
- For any other node \(v\):
  - weight\((v)\) \(\geq\) \(\alpha\gamma^h\)
  - weight\((w)\) \(\leq\) \(\beta\gamma^{h-1}\), for every child, \(w\), of \(v\)

\[\therefore \deg(v) \geq \frac{\alpha\gamma^h}{\beta\gamma^{h-1}} = \frac{\alpha\gamma}{\beta} = \frac{\alpha^2\beta}{\beta} = a.\]
Lemma: In a weight-balanced \((a, b)\)-tree, the root has degree between \(2\) and \(b\). Any other internal node has degree between \(a\) and \(b\).

**Lower bound:**
- For the root, obvious.
- For any other node \(v\):
  - \(\text{weight}(v) \geq \alpha \gamma^h\)
  - \(\text{weight}(w) \leq \beta \gamma^{h-1}\), for every child, \(w\), of \(v\)

\[
\therefore \text{deg}(v) \geq \alpha \gamma^h \div \beta \gamma^{h-1} = \frac{\alpha \gamma}{\beta}
\]

\[
= \frac{\alpha^2 \beta}{\beta} = a.
\]

**Upper bound:**
- \(\text{weight}(v) \leq \beta \gamma^h\)
- \(\text{weight}(w) \geq \alpha \gamma^{h-1}\), for every child, \(w\), of \(v\)
Lemma: In a weight-balanced $(a, b)$-tree, the root has degree between 2 and $b$. Any other internal node has degree between $a$ and $b$.

**Lower bound:**
- For the root, obvious.
- For any other node $v$:
  - $\text{weight}(v) \geq \alpha \gamma^h$
  - $\text{weight}(w) \leq \beta \gamma^{h-1}$, for every child, $w$, of $v$

  $\therefore \deg(v) \geq \frac{\alpha \gamma^h}{\beta \gamma^{h-1}} = \frac{\alpha \gamma}{\beta} = \frac{\alpha^2 \beta}{\beta} = a.$

**Upper bound:**
- $\text{weight}(v) \leq \beta \gamma^h$
- $\text{weight}(w) \geq \alpha \gamma^{h-1}$, for every child, $w$, of $v$

  $\therefore \deg(v) \leq \frac{\beta \gamma^h}{\alpha \gamma^{h-1}} = \frac{\beta \gamma}{\alpha} = \frac{\beta^2 \alpha}{\alpha} = b.$
Lemma: The nodes produced by a split or node fusion at height $h$ in a weight-balanced $(a, b)$-tree have weight at most $\frac{3}{4} \beta \gamma^h$. If $h$ is not the root level, these nodes also have weight at least $\frac{3}{2} \alpha \gamma^h$. 
Rebalancing Weight-Balanced \((a, b)\)-Trees

**Lemma:** The nodes produced by a split or node fusion at height \(h\) in a weight-balanced \((a, b)\)-tree have weight at most \(\frac{3}{4} \beta \gamma^h\). If \(h\) is not the root level, these nodes also have weight at least \(\frac{3}{2} \alpha \gamma^h\).

**Proof:** (Only splits)
Lemma: The nodes produced by a split or node fusion at height $h$ in a weight-balanced $(a, b)$-tree have weight at most $\frac{3}{4} \beta \gamma^h$. If $h$ is not the root level, these nodes also have weight at least $\frac{3}{2} \alpha \gamma^h$.

Proof: (Only splits)

- Splitting to distribute the leaves 50-50 may require splitting a child of $v$. 

50-50 split
Rebalancing Weight-Balanced \((a, b)\)-Trees

**Lemma:** The nodes produced by a split or node fusion at height \(h\) in a weight-balanced \((a, b)\)-tree have weight at most \(\frac{3}{4} \beta \gamma^h\). If \(h\) is not the root level, these nodes also have weight at least \(\frac{3}{2} \alpha \gamma^h\).

**Proof:** (Only splits)

- Splitting to distribute the leaves 50-50 may require splitting a child of \(v\).
- Make this child a child of the left node.
For \( u \in \{v', v''\} \), \( \frac{\beta \gamma^h + 1}{2} - \beta \gamma^{h-1} \leq \text{weight}(u) \leq \frac{\beta \gamma^h + 1}{2} + \beta \gamma^{h-1} \).
For $u \in \{v', v''\}$, $\frac{\beta \gamma^h + 1}{2} - \beta \gamma^{h-1} \leq \text{weight}(u) \leq \frac{\beta \gamma^h + 1}{2} + \beta \gamma^{h-1}$.

Lower bound:

$$\text{weight}(u) \geq \frac{\beta \gamma^h + 1}{2} - \beta \gamma^{h-1}$$

$$\geq \beta \gamma^h \left( \frac{1}{2} - \frac{1}{\gamma} \right)$$

$$\geq 4 \alpha \gamma^h \left( \frac{1}{2} - \frac{1}{8} \right)$$

$$= \frac{3}{2} \alpha \gamma^h.$$
For $u \in \{v', v''\}$, \(\frac{\beta \gamma^h + 1}{2} - \beta \gamma^{h-1} \leq \text{weight}(u) \leq \frac{\beta \gamma^h + 1}{2} + \beta \gamma^{h-1}\).

**Lower bound:**

\[
\text{weight}(u) \geq \frac{\beta \gamma^h + 1}{2} - \beta \gamma^{h-1} \\
\geq \beta \gamma^h \left(\frac{1}{2} - \frac{1}{\gamma}\right) \\
\geq 4\alpha \gamma^h \left(\frac{1}{2} - \frac{1}{8}\right) \\
= \frac{3}{2} \alpha \gamma^h.
\]

**Upper bound:**

\[
\text{weight}(u) \leq \frac{\beta \gamma^h + 1}{2} + \beta \gamma^{h-1} \\
\leq \beta \gamma^h \left(\frac{1}{2} + \frac{1}{\gamma}\right) + \frac{1}{2} \\
\leq \beta \gamma^h \left(\frac{1}{2} + \frac{1}{8}\right) + \frac{1}{2} \\
= \frac{5}{8} \beta \gamma^h + \frac{1}{2} \\
\leq \frac{3}{4} \beta \gamma^h.
\]
Amortization

Given:

- A type of data structure $\mathcal{T}$ that supports operations $o_1, o_2, \ldots, o_k$ (e.g., Insert, Delete, Find, ...).
Amortization

**Given:**

- A type of data structure $\mathcal{T}$ that supports operations $o_1, o_2, \ldots, o_k$ (e.g., Insert, Delete, Find, \ldots)

We say that operation $o_i$ has **amortized cost** $T_i(n)$, for $1 \leq i \leq n$, if any sequence of $n$ operations on an initially empty data structure of type $\mathcal{T}$ takes at most

$$\sum_{i=1}^{k} n_i T_i(n)$$

time, where $n_i$ is the number of operations of type $o_i$ in the sequence.
Amortization

Given:

- A type of data structure $\mathcal{T}$ that supports operations $o_1, o_2, \ldots, o_k$ (e.g., Insert, Delete, Find, \ldots)

We say that operation $o_i$ has \textbf{amortized cost} $T_i(n)$, for $1 \leq i \leq n$, if any sequence of $n$ operations on an initially empty data structure of type $\mathcal{T}$ takes at most

$$\sum_{i=1}^{k} n_i T_i(n)$$

time, where $n_i$ is the number of operations of type $o_i$ in the sequence.

What we get is a \textbf{worst-case bound}, but not for individual operations, but for the cost of any \textbf{sequence} of operations on the data structure.
Credit Method

Idea:

- Associate a credit with the nodes of the data structure.
- An operation can pay credit to certain nodes, in addition to paying for its own cost.
- An operation can cover all or part of its actual cost by taking from the credit of some nodes.
Credit Method

Idea:
- Associate a credit with the nodes of the data structure.
- An operation can pay credit to certain nodes, in addition to paying for its own cost.
- An operation can cover all or part of its actual cost by taking from the credit of some nodes.

Formally:
- Define the amortized cost of an operation as its actual cost plus the total change in credit.
Credit Method

Idea:

■ Associate a credit with the nodes of the data structure.
■ An operation can pay credit to certain nodes, in addition to paying for its own cost.
■ An operation can cover all or part of its actual cost by taking from the credit of some nodes.

Formally:

■ Define the amortized cost of an operation as its actual cost plus the total change in credit.

**Observation:** If the credit is initially zero and never becomes negative, the amortized cost of any sequence of operations is an upper bound on its actual cost.
Credit Method

Idea:

- Associate a credit with the nodes of the data structure.
- An operation can pay credit to certain nodes, in addition to paying for its own cost.
- An operation can cover all or part of its actual cost by taking from the credit of some nodes.

Formally:

- Define the amortized cost of an operation as its actual cost plus the total change in credit.

Observation: If the credit is initially zero and never becomes negative, the amortized cost of any sequence of operations is an upper bound on its actual cost.

Note: The algorithm does not actually maintain credits. They are an analysis technique.
Lemma: A weight-balanced priority-search tree supports insertions and deletions in $O(\lg n)$ amortized time.
Lemma: A weight-balanced priority-search tree supports insertions and deletions in $O(\lg n)$ amortized time.

Proof:
Lemma: A weight-balanced priority-search tree supports insertions and deletions in $O(\lg n)$ amortized time.

Proof:

- Make every insertion or deletion pay one credit to every ancestor of the affected leaf.

∴ the amortized cost per insertion or deletion, excluding rebalancing costs, is $O(\lg n)$. 
Lemma: A weight-balanced priority-search tree supports insertions and deletions in $O(\lg n)$ amortized time.

Proof:

- Make every insertion or deletion pay one credit to every ancestor of the affected leaf.

  $\therefore$ the amortized cost per insertion or deletion, excluding rebalancing costs, is $O(\lg n)$.

- A node split or node fusion triggered by a node $v$ at height $h$ costs $O(h) = O(\beta \gamma^h) = O(\text{weight}(v))$ time.
An Amortized Update Bound

**Lemma:** A weight-balanced priority-search tree supports insertions and deletions in $O(lg n)$ amortized time.

**Proof:**

- Make every insertion or deletion pay one credit to every ancestor of the affected leaf.
  \[ \therefore \text{the amortized cost per insertion or deletion, excluding rebalancing costs, is } O(lg n). \]

- A node split or node fusion triggered by a node $v$ at height $h$ costs $O(h) = O(\beta \gamma^h) = O(\text{weight}(v))$ time.

- After the creation of $v$, $\Omega(\text{weight}(v))$ insertions or deletions below $v$ are required to make $v$ split or fuse.
**Lemma:** A weight-balanced priority-search tree supports insertions and deletions in $\mathcal{O}(\lg n)$ amortized time.

**Proof:**

- Make every insertion or deletion pay one credit to every ancestor of the affected leaf.

  \[ \therefore \] the amortized cost per insertion or deletion, excluding rebalancing costs, is $\mathcal{O}(\lg n)$.

- A node split or node fusion triggered by a node $v$ at height $h$ costs $\mathcal{O}(h) = \mathcal{O}(\beta\gamma^h) = \mathcal{O}(\text{weight}(v))$ time.

- After the creation of $v$, $\Omega(\text{weight}(v))$ insertions or deletions below $v$ are required to make $v$ split or fuse.

- Each insertion or deletion below $v$ has paid one credit to $v$. 
Lemma: A weight-balanced priority-search tree supports insertions and deletions in $O(\lg n)$ amortized time.

Proof:

- Make every insertion or deletion pay one credit to every ancestor of the affected leaf. 
  $\therefore$ the amortized cost per insertion or deletion, excluding rebalancing costs, is $O(\lg n)$.

- A node split or node fusion triggered by a node $v$ at height $h$ costs $O(h) = O(\beta \gamma^h) = O(\text{weight}(v))$ time.

- After the creation of $v$, $\Omega(\text{weight}(v))$ insertions or deletions below $v$ are required to make $v$ split or fuse.

- Each insertion or deletion below $v$ has paid one credit to $v$. 
  $\therefore$ $v$ can pay for the split or fusion with its credits.
  $\therefore$ the amortized cost per node split or fusion is 0! (We have already paid for it by charging earlier insertions and deletions.)
**Theorem:** Priority search trees can be used to solve the dynamic three-sided range search problem. Queries take $O(lg n + t)$ time in the worst case. Updates take $O(lg^2 n)$ time in the worst case and $O(lg n)$ amortized time.

**Note:** By using more advanced techniques or by building priority search trees on top of red-black trees, the update bound can be made $O(lg n)$ in the worst case.
Higher-Dimensional Range Searching

Goal:

- Build a *static* data structure over a point set $S$ in $\mathbb{R}^d$ that allows to report all the points in $S$ that fall in a given ($d$-dimensional) query rectangle.
- Queries should be fast.
- Data structure should be small.
- Data structure should be fast to build.
One-Dimensional Range Searching
One-Dimensional Range Searching

**Straightforward solution:**

- Data structure = search tree on \( x \)-coordinates
- Search down paths to leftmost and rightmost nodes in \( x \)-range
- Report all the points stored at the leaves between these two paths.

**Query time:**
One-Dimensional Range Searching

**Straightforward solution:**

- Data structure = search tree on $x$-coordinates
- Search down paths to leftmost and rightmost nodes in $x$-range
- Report all the points stored at the leaves between these two paths.

**Query time:** $O(\lg n + k)$

**Space:**
One-Dimensional Range Searching

**Straightforward solution:**

- Data structure = search tree on $x$-coordinates
- Search down paths to leftmost and rightmost nodes in $x$-range
- Report all the points stored at the leaves between these two paths.

**Query time:** $O(\lg n + k)$

**Space:** $O(n)$

**Build time:**
One-Dimensional Range Searching

**Straightforward solution:**
- Data structure = search tree on $x$-coordinates
- Search down paths to leftmost and rightmost nodes in $x$-range
- Report all the points stored at the leaves between these two paths.

**Query time:** $O(\lg n + k)$

**Space:** $O(n)$

**Build time:** $O(n \lg n)$
- Sort points by $x$-coordinates
- Build the tree bottom-up in *linear time*. 
Two-Dimensional Range Searching (1)

What if the query range is a rectangle instead of a slab?
Two-Dimensional Range Searching (1)

What if the query range is a rectangle instead of a slab?

*Observations:*
What if the query range is a rectangle instead of a slab?

**Observations:**

- Easy to check whether leftmost and rightmost leaves are in query range.
What if the query range is a rectangle instead of a slab?

**Observations:**

- Easy to check whether leftmost and rightmost leaves are in query range.
- Again, expensive to inspect subtrees between paths.
Two-Dimensional Range Searching (1)

What if the query range is a rectangle instead of a slab?

**Observations:**

- Easy to check whether leftmost and rightmost leaves are in query range.
- Again, expensive to inspect subtrees between paths.
- Points between paths are guaranteed to be in $x$-range.
What if the query range is a rectangle instead of a slab?

**Observations:**

- Easy to check whether leftmost and rightmost leaves are in query range.
- Again, expensive to inspect subtrees between paths.
- Points between paths are guaranteed to be in $x$-range.
- $y$-coordinates are distinguishing factor.
Two-Dimensional Range Searching (2)

Solution:
Solution:

- Every node stores a search tree over the points corresponding to its leaf descendants, sorted by $y$-coordinate.
Solution:

- Every node stores a search tree over the points corresponding to its leaf descendants, sorted by \( y \)-coordinate.

Query time:
Two-Dimensional Range Searching (2)

Solution:

- Every node stores a search tree over the points corresponding to its leaf descendants, sorted by $y$-coordinate.

Query time: $O(lg^2 n + k)$

Space:
Two-Dimensional Range Searching (2)

Solution:

- Every node stores a search tree over the points corresponding to its leaf descendants, sorted by $y$-coordinate.

Query time: $\mathcal{O}(\lg^2 n + k)$

Space: $\mathcal{O}(n \lg n)$

Build time:
Two-Dimensional Range Searching (2)

Solution:

- Every node stores a search tree over the points corresponding to its leaf descendants, sorted by $y$-coordinate.

Query time: $O(\lg^2 n + k)$

Space: $O(n \lg n)$

Build time: $O(n \lg n)$

- Sort points by $x$-coordinates
- Build the tree bottom-up in $O(n \lg n)$ time.
What about $d$-dimensional range searching?
What about $d$-dimensional range searching?

**Solution:**

- Every node stores a $(d-1)$-dimensional search tree over the points corresponding to its leaf descendants.
What about $d$-dimensional range searching?

Solution:

- Every node stores a $(d-1)$-dimensional search tree over the points corresponding to its leaf descendants.

Query time:
**d-Dimensional Range Searching**

What about \(d\)-dimensional range searching?

**Solution:**

- Every node stores a \((d - 1)\)-dimensional search tree over the points corresponding to its leaf descendants.

**Query time:** \(O(\lg^d n + k)\)

**Space:**
What about $d$-dimensional range searching?

**Solution:**

- Every node stores a $(d - 1)$-dimensional search tree over the points corresponding to its leaf descendants.

**Query time:** $O(\lg^d n + k)$

**Space:** $O(n \lg^{d-1} n)$

**Build time:**
What about \( d \)-dimensional range searching?

**Solution:**

- Every node stores a \((d - 1)\)-dimensional search tree over the points corresponding to its leaf descendants.

**Query time:** \( O(\lg^d n + k) \)

**Space:** \( O(n \lg^{d-1} n) \)

**Build time:** \( O(n \lg^{d-1} n) \)
**Theorem:** $d$-dimensional range trees can be used to solve the $d$-dimensional range searching problem in $O(lg^d n + t)$ time per query. For $d \geq 2$, a $d$-dimensional range tree uses $O(n lg^{d-1} n)$ space and can be built in $O(n lg^{d-1} n)$ time.

**Notes:**

- Using a weight-balanced $(a, b)$-tree as the primary tree, updates can be supported in $O(lg^d n)$ amortized time.
- Using an advanced technique known as **fractional cascading**, the query time can be reduced to $O(lg^{d-1} n)$. Fractional cascading as such is static; but a dynamic version with slightly worse performance has also been developed.
Data Structuring: Summary

Data structuring is a very useful paradigm that can be seen as the algorithmic equivalent of two important strategies in software engineering: modularization and code reuse.

- **Modularization**: By using the data structuring paradigm, we can often split the problem into manageable units. The nasty details are hidden inside each data structure.
- **Code reuse**: Once we have a powerful data structure, it can often be used to solve more than one problem.

To build a new data structure often does not mean to start from scratch, but to augment an existing one:

- Store additional information at each node (e.g., order statistics).
- Change the rules where data items are stored (e.g., priority search trees).
- Replicate data and define recursive structures (e.g., range trees).