Yesterday

• Fan-out $\Theta(B^{1/c})$ B-tree ($c \geq 1$)
  – Degree balanced tree with each node/leaf in $O(1)$ blocks
  – $O(N/B)$ space
  – $O(\log_B N + T_B)$ I/O query
  – $O(\log_B N)$ I/O update

• Persistent B-tree
  – Update current version, query all previous versions
  – B-tree bounds with $N$ number of operations performed

• Buffer tree technique
  – Lazy update/queries using buffers attached to each node
  – $O(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B})$ amortized bounds
  – E.g. used to construct structures in $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os
Simplifying Assumption

- **Model**
  - $N$ : Elements in structure
  - $B$ : Elements per block
  - $M$ : Elements in main memory
  - $T$ : Output size in searching problems

- **Assumption**
  - Today (and tomorrow) assume that $M > B^2$
  - Assumption not crucial but simplify expressions a lot, e.g.:
    $$O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right) = O\left(\frac{N}{B} \log B \ N\right)$$
Today

• “Dimension 1.5” problems:
  – More complicated problems: Interval stabbing and point location
  – Looking for same bounds:
    * $O(N/B)$ space
    * $O(\log_B N + T/B)$ query
    * $O(\log_B N)$ update
    * $O(\frac{N}{B} \log_{M/B} \frac{N}{B}) = O(\frac{N}{B} \log_B N)$ construction

• Use of tools/techniques discussed yesterday as well as
  – Logarithmic method
  – Weight-balanced B-trees
  – Global rebuilding
Interval Management

• Problem:
  – Maintain $N$ intervals with unique endpoints dynamically such that stabbing query with point $x$ can be answered efficiently

  

• As in (one-dimensional) B-tree case we are interested in
  – $O(N/B)$ space
  – $O(\log_B N)$ update
  – $O(\log_B N + T/B)$ query
Interval Management: Static Solution

- **Sweep** from left to right maintaining persistent B-tree
  - Insert interval when left endpoint is reached
  - Delete interval when right endpoint is reached
- Query \( x \) answered by reporting all intervals in B-tree at “time” \( x \)
  - \( O\left(\frac{N}{B}\right) \) space
  - \( O\left(\log_B N + \frac{T_B}{B}\right) \) query
  - \( O\left(\frac{N}{B} \log_B N\right) \) construction using buffer technique
- Dynamic with \( O\left(\log_B^2 N\right) \) insert bound using logarithmic method
Internal Memory Logarithmic Method Idea

• Given (semi-dynamic) structure $D$ on set $V$
  – $O(\log N)$ query, $O(\log N)$ delete, $O(N \log N)$ construction
• Logarithmic method:
  – Partition $V$ into subsets $V_0, V_1, \ldots V_{\log N}$, $|V_i| = 2^i$ or $|V_i| = 0$
  – Build $D_i$ on $V_i$

* Delete: $O(\log N)$

* Query: Query each $D_i \Rightarrow O(\log^2 N)$

* Insert: Find first empty $D_i$ and construct $D_i$ out of
  \[ 1 + \sum_{j=0}^{i-1} 2^j = 2^i \text{ elements in } V_0, V_1, \ldots V_{i-1} \]
  – $O(2^i \log 2^i)$ construction $\Rightarrow O(\log N)$ per moved element
  – Element moved $O(\log N)$ times $\Rightarrow O(\log^2 N)$ amortized
External Logarithmic Method Idea

- Decrease number of subsets $V_i$ to $\log_B N$ to get $O(\log_B^2 N)$ query.

- **Problem**: Since $1 + \sum_{j=0}^{i-1} B^j < B^i$ there are not enough elements in $V_0, V_1, \ldots V_{i-1}$ to build $V_i$.

- **Solution**: We allow $V_i$ to contain any number of elements $\leq B^i$.
  - **Insert**: Find first $D_i$ such that $\sum_{j=0}^{i} |V_j| < B^i$ and construct new $D_i$ from elements in $V_0, V_1, \ldots V_i$.
    - We move $\sum_{j=0}^{i-1} |V_j| \geq B^{i-1}$ elements.
    - If $D_i$ constructed in $O((|V_i|/B) \log_B |V_i|)$ = $O(B^{i-1}\log_B N)$ I/Os.
    - Every moved element charged $O(\log_B N)$ I/Os.
    - Element moved $O(\log_B N)$ times $\Rightarrow$ $O(\log_B^2 N)$ amortized.
External Logarithmic Method Idea

• Given (semi-dynamic) linear space external data structure with
  – $O(\log_B N + \frac{T}{B})$ I/O query
  – $O(\frac{N}{B} \log_B N)$ I/O construction
  (– $O(\log_B N)$ I/O delete)

↓

• Linear space **dynamic** data structure with
  – $O(\log_B^2 N + \frac{T}{B})$ I/O query
  – $O(\log_B^2 N)$ I/O insert amortized
  (– $O(\log_B N)$ I/O delete)

• Dynamic interval management
  – $O(\log_B^2 N + \frac{T}{B})$ I/O query
  – $O(\log_B^2 N)$ I/O insert amortized
- Base tree on endpoints – “slab” $X_v$ associated with each node $v$
- Interval stored in highest node $v$ where it contains midpoint of $X_v$
- Intervals $I_v$ associated with $v$ stored in
  - **Left slab list** sorted by left endpoint (search tree)
  - **Right slab list** sorted by right endpoint (search tree)

$\Rightarrow$ Linear space and $O(\log N)$ update (assuming fixed endpoint set)
• **Query** with $x$ on left side of midpoint of $X_{\text{root}}$
  – Search **left slab list** left-right until finding non-stabbed interval
  – Recurse in left child

$\Rightarrow O(\log N + T)$ query bound
External memory data structures

**Externalizing Interval Tree**

- **Natural idea:**
  - Block tree
  - Use B-tree for **slab lists**

- Number of stabbed intervals in large slab list may be small (or zero)
  - We can be forced to do I/O in each of \( O(\log N) \) nodes
Externalizing Interval Tree

- Idea:
  - Decrease fan-out to $\Theta(\sqrt{B}) \Rightarrow$ height remains $O(\log_B N)$
  - $\Theta(\sqrt{B})$ slabs define $\Theta(B)$ multislabs
  - Interval stored in two slab lists (as before) and one multislab list
  - Intervals in small multislab lists collected in underflow structure
  - Query answered in $\nu$ by looking at 2 slab lists and not $O(\log N)$
External Interval Tree

• Base tree: Fan-out $\Theta(\sqrt{B})$ B-tree on endpoints
  – Interval stored in highest node $v$ where it contains slab boundary

• Each internal node $v$ contains:
  – **Left slab list** for each of $\Theta(\sqrt{B})$ slabs
  – **Right slab lists** for each of $\Theta(\sqrt{B})$ slabs
  – $\Theta(B)$ multislab lists
  – Underflow structure

• Interval in set $I_v$ of intervals associated with $v$ stored in
  – **Left slab list** of slab containing left endpoint
  – **Right slab list** of slab containing right endpoint
  – Widest multislab list it spans

• If $< B$ intervals in multislab list they are instead stored in underflow structure ($\Rightarrow$ contains $\leq B^2$ intervals)
External Interval tree

- Each leaf contains $O(B)$ intervals (unique endpoint assumption)
  - Stored in one $O(1)$ block
- Slab lists implemented using B-trees
  - $O(1 + \frac{T_v}{B})$ query
  - Linear space
    * We may “wasted” a block for each of the $\Theta(\sqrt{B})$ lists in node
    * But only $\Theta(\frac{N}{B\sqrt{B}})$ internal nodes
- Underflow structure implemented using static structure
  - $O(\log_B B^2 + \frac{T_v}{B}) = O(1 + \frac{T_v}{B})$ query
  - Linear space
- Linear space
External Interval Tree

• Query with $x$
  – Search down tree for $x$ while in node $\nu$
    reporting all intervals in $I_\nu$ stabbed by $x$

• In node $\nu$
  – Query two slab lists
  – Report all intervals in relevant multislab lists
  – Query underflow structure

• Analysis:
  – Visit $O(\log B \ N)$ nodes
  – Query slab lists
  – Query multislab lists $O(1+\frac{T_\nu}{B})$
  – Query underflow structure $O(1+\frac{T_\nu}{B})$

$\Rightarrow O(\log B \ N + \frac{T}{B})$
External Interval Tree

- **Update** (assuming fixed endpoint set – static base tree):
  - Search for relevant node
  - Update two slab lists
  - Update multislab list or underflow structure

\[ O(\log_B N) \]

- Update of **underflow structure** in \( O(I) \) I/Os amortized
  - Maintain update block with \( \leq B \) updates
  - Check of update block adds \( O(I) \) I/Os to query bound
  - Rebuild structure when \( B \) updates have been collected using
    \[ O\left(\frac{B^2 \log_B B^2}{B} \right) = O(B) \] I/Os (**Global rebuilding**)↓

Update in \( O(\log_B N) \) I/Os amortized
External Interval Tree

• Note:
  – Insert may increase number of intervals in underflow structure for same multislab to $B$
  – Delete may decrease number of intervals in multislab to $B$

\[ \downarrow \]

Need to move $B$ intervals to/from multislab/underflow structure

• We only move
  – intervals from multislab list when decreasing to size $B/2$
  – Intervals to multislab list when increasing to size $B$

\[ \downarrow \]

$O(1)$ I/Os amortized used to move intervals
Removing Fixed Endpoint Assumption

- We need to use dynamic base tree
  - Natural choice is B-tree

- Insertion:
  - Insert new endpoints and rebalance base tree (using splits)
  - Insert interval as previously in $O(\log_B N)$ I/Os amortized

- Split: Boundary in $v$ becomes boundary in $\text{parent}(v)$
Splitting Interval Tree Node

- When $v$ splits we may need to move $O(w(v))$ intervals
  - Intervals in $v$ containing boundary
  - Intervals in $\text{parent}(v)$ with endpoints in $X_v$ containing boundary
- Intervals move to two new slab and multislab lists in $\text{parent}(v)$
• Moving intervals in $v$ in $O(w(v))$ I/Os
  – Collected in left order (and remove) by scanning left slab lists
  – Collected in right order (and remove) by scanning right slab lists
  – Removed multislab lists containing boundary
  – Remove from underflow structure by rebuilding it
  – Construct lists and underflow structure for $v'$ and $v''$ similarly
Splitting Interval Tree Node

- Moving intervals in \textit{parent}(v) in \(O(w(v))\) I/Os
  - Collect in left order by scanning left slab list
  - Collect in right order by scanning right slab list
  - Merge with intervals collected in \(v\) \(\Rightarrow\) two new slab lists
  - Construct new multislab lists by splitting relevant multislab list
  - Insert intervals in small multislab lists in underflow structure
Removing Fixed Endpoint Assumption

• Split of node \( v \) use \( O(w(v)) \) I/Os
  - If \( \Omega(w(v)) \) inserts have to be made below \( v \)
    \( \Rightarrow O(I) \) amortized split bound
    \( \Rightarrow O(\log_B N) \) amortized insert bound

• Nodes in standard B-tree do not have this property

(2,4)–tree
**BB[α]-tree**

- In internal memory BB[α]-trees have the desired property
- Defined using **weight-constraints**
  - Ratio between weight of left child and weight of right child of a node \( v \) is between \( α \) and \( 1-α \)
  
  \[ \downarrow \]

  Height \( O(\log N) \)

- If \( \frac{1}{11} < α < 1 - \frac{1}{2}\sqrt{2} \) rebalancing can be performed using rotations

\[ \xymatrix{ x \ar[d] & } \quad \xymatrix{ x \ar[d] & y \ar[l] } \]

- Seems hard to implement BB[α]-trees I/O-efficiently
**Weight-balanced B-tree**

- **Idea:** Combination of B-tree and BB[α]-tree
  - Weight constraint on nodes instead of degree constraint
  - Rebalancing performed using split/fuse as in B-tree

- **Weight-balanced B-tree** with parameters $a$ and $k$ ($a > 4$, $k > 0$)
  - All leaves on same level and contain between $k$ and $2k - 1$ elements
  - Internal node $v$ at level $l$ has $w(v) < 2a^l k$
  - Except for the root, internal node $v$ at level $l$ have $w(v) > \frac{1}{2}a^l k$
  - The root has more than one child
Weight-balanced B-tree

• Every internal node has degree between
  \( \frac{1}{2} a^l k / 2a^{l-1} k = \frac{1}{4} a \) and \( 2a^l k / \frac{1}{2} a^{l-1} k = 4a \)

\[ \downarrow \]

Height \( O(\log_a \frac{N}{k}) \)

• External memory:
  – Choose \( 4a=B \) (or even \( B^c \) for \( 0 < c \leq l \))
  – \( 2k=B \)

\[ \downarrow \]

\( O(N/B) \) space, \( O(\log_B N) \) query
Weight-balanced B-tree

• **Insert:**
  
  - Search and insert element in leaf $v$
  - If $w(v)=2k$ then split $v$
  - For each node $v$ on path to root
    if $w(v)>2a^lk$ then
      split $v$ into two nodes with weight $<2a^lk-2a^{l-1}k<\frac{3}{2}a^lk$
      insert element (ref) in $\text{parent}(v)$

  \[
  \begin{array}{c}
  \frac{1}{4}a^l k...2a^l k \\
  \frac{1}{4}a^{l-1}k...2a^{l-1}k
  \end{array}
  \text{level } l
  \begin{array}{c}
  \frac{1}{4}a^{l-1}k...2a^{l-1}k
  \end{array}
  \text{level } l-1
  \]

• Number of splits after insert is $O(\log_a \frac{N}{k})$
• A split level $l$ node will not split for next $\frac{1}{2}a^lk$ inserts below it
  \[\downarrow\]

**Desired property:** $\Omega(w(v))$ inserts below $v$ between splits
External Interval Tree

- Use weight-balanced B-tree with $4a = \sqrt{B}$ and $2k=B$ as base structure
  - Space: $O(N/B)$
  - Query: $O(\log_B N + T/B)$
  - Insert: $O(\log_B N)$ I/Os amortized

- Deletes in $O(\log_B N)$ I/Os amortized using global rebuilding:
  - Delete interval as previously using $O(\log_B N)$ I/Os
  - Mark relevant endpoint as deleted
  - Rebuild structure in $O(N \log_B N)$ after $N/2$ deletes

- Note: Deletes can also be handled using fuse operations
External Interval Tree

• External interval tree
  – Space: $O(N/B)$
  – Query: $O(\log_B N + T/B)$
  – Updates: $O(\log_B N)$ I/Os amortized

• Removing amortization:
  – Moving intervals to/from underflow structure
  – Delete global rebuilding
  – Underflow structure update
  – Base node tree splits

Perform operations/construction lazily
Move lazily – complicated:
• Interference
• Queries
Other Applications

• Examples of applications of external interval tree:
  – Practical visualization applications
  – Point location
  – External segment tree

• Examples of applications of weight-balance B-tree
  – Base tree of external data structures
  – Remove amortization from internal structures (alternative to BB[α]-tree)
  – Cache-oblivious structures
Summary: Interval Management

- Interval management corresponds to simple form of 2d range search
  - Diagonal corner queries
- We obtained the same bounds as for the 1d case
  - Space: $O(N/B)$
  - Query: $O(\log B N + \frac{T}{B})$
  - Updates: $O(\log B N)$ I/Os
Summary: Interval Management

- Main problem in designing structure:
  - Binary $\rightarrow$ large fan-out
- Large fan-out resulted in the need for
  - Multislabs and multislab lists
  - Underflow structure to avoid $O(B)$-cost in each node

- General solution techniques:
  - Filtering: Charge part of query cost to output
  - Bootstrapping:
    * Use $O(B^2)$ size structure in each internal node
    * Constructed using persistence
    * Dynamic using global rebuilding
  - Weight-balanced B-tree: Split/fuse in amortized $O(1)$
Planar Point Location

- **Static problem:**
  - Store planar subdivision with $N$ segments on disk such that region containing query point $q$ can be found I/O-efficiently

- **We concentrate on** vertical ray shooting query
  - Segments can store regions it bounds
  - Segments do not have to form subdivision

- **Dynamic problem:**
  - Insert/delete segments
**Static Solution**

- Vertical line imposes **above-below** order on intersected segments

- **Sweep** from left to right maintaining persistent B-tree on above-below order
  - Left endpoint: Insert segment
  - Right endpoint: Delete segment

- Query $q$ answered by successor query on B-tree at time $q_x$
  - $O(N/B)$ space
  - $O(\log_B N + T_B)$ query
Static Solution

- **Note**: Not all segments comparable!
  - Have to be careful about what we compare

- **Problem**: Routing elements in internal nodes of leaf oriented B-trees
  - Luckily we can modify persistent B-tree to use regular elements as routing elements

- However, buffer technique construction cannot be used

- Only $O(N \log_B N)$ I/O construction algorithm
- Cannot be made dynamic using logarithmic method
Dynamic Point Location

- Structure similar to external interval tree
  - Built on $x$-projection of segments
- Fan-out $\Theta(\sqrt{B})$ base B-tree on $x$-coordinates
  - Interval stored in highest node $v$ where it contains slab boundary

\[ \Theta(\sqrt{B}) \]
Dynamic Point Location

• Linear space in node $v \Rightarrow$ linear space
• Query idea:
  – Search for $q_x$
  – Answer query in each node $v$ encountered
  – Result is globally closest segment

$O(\log_B N)$ query in each node $\Rightarrow O(\log_B^2 N)$ I/O query
Dynamic Point Location

• Secondary structures:
  – For each slab:
    * **Left slab structure** on segments with left endpoint in slab
    * **Right slab structure** on segments with right endpoint in slab
  – **Multislab structure** on part of segments completely spanning slab
Dynamic Point Location

- To answer query we query
  - One left slab structure
  - One right slab structure
  - Multislab structure
  and return globally closest segment

- We need to answer query on each secondary structure in $O(\log_B N)$ I/Os
Left (right) slab Structure

- B-tree on segments sorted by $y$-coordinate of right endpoint
- Each internal node $v$ augmented with $\Theta(B)$ segments
  - For each child $c_v$:
    - The segment in leaves below $c_v$ with minimal left $x$-coordinate
    $\downarrow$
    $O(N/B)$ space (each node fits in block)

- Construction:
  - Sort segments
  - Build level-by-level bottom up
  $\downarrow$
  $O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os
Left (right) slab Structure

- **Invariant**: Search top-down such that $i^{th}$ step visit nodes $v_u$ and $v_d$
  - $v_u$ contains answer to **upward** query among segments on level $i$
  - $v_d$ contains answer to **downward** query among segments on level $i$
  \[ \Rightarrow v_u \text{ contains query result when reaching leaf level} \]

- **Algorithm**: At level $i$
  - Consider two children of $v_u$ and $v_d$ containing two segments hit on level $i$
  - Update $v_u$ and $v_d$ to relevant of these nodes base on their segments

- **Analysis**: $O(1)$ I/Os on each of $O(\log_B N)$ levels
Multislab Structure

- Segments crossing a slab are ordered by *above-below order*
  - But *not* all segments are comparable!
- B-tree in each of $\Theta(\sqrt{B})$ slabs on segments crossing the slab
  $\Rightarrow$ query answered in $O(\log_B N)$ I/Os
- **Problem:** Each segment stored in many structures
- **Key idea:**
  - Use *total order* consistent with above-below order in each slab
  - Build one structure on *total order*
Multislab Structure

- Fan-out $\Theta(\sqrt{B})$ B-tree on total order
- Node $v$ augmented with $\Theta(\sqrt{B})$ segments for each of $\Theta(\sqrt{B})$ children
  - For child $v_i$ and each slab $s_i$:
    
    Maximal segment below $v_i$ crossing $s_i$
    
    $\Rightarrow O(N/B)$ space (each node $v$ fits in one block)

- $O(\log_B N)$ query as in normal B-tree
  - Only $\Theta(\sqrt{B})$ segments crossing $s_i$ considered in $v$
Multislab Structure Construction

- Multislab structure constructed in $O(N/B)$ I/Os bottom-up
  - after total order computed

- **Sorting:**
  - Distribute segments to a list for each multislab
  - Sort lists individually
  - Merge sorted lists: Repeatedly consider top segment all lists and select/output (any) segment not below any of the other segments

- **Correctness:**
  - Selected top segment cannot be below any unprocessed segment

- **Analysis:**
  - Distribute/Merge in $O(N/B)$, sort in $O(N/B \log_{M/B} N)$ I/Os
Dynamic Point Location

- Static point location structure:
  - $O(N/B)$ space
  - $O(\frac{N}{B} \log_B \frac{N}{B})$ I/O construction
  - $O(\log_B^2 N)$ I/O query

- Updates involve:
  - Updating (and rebalance) base tree
  - Updating two slab structures
  - Updating one multislab structure

- Base tree update as in interval tree case using weight-balanced B-tree
  - Inserts: Node split in $O(w(v))$ I/Os
  - Deletes: Global rebuilding
Updating Left (right) Slab Structures

- Recall that each internal node augmented with minimal left $x$-coordinate segment below each child

  **Insert:**
  - Insert in leaf $l$ and (B-tree) rebalance
  - Insert segment in relevant nodes on root-$l$ path

  **Delete:**
  - Delete from leaf $l$ and rebalance as in B-tree
  - Find new minimal $x$-coordinate segment in $l$
  - Replace deleted segment in relevant nodes on root-$l$ path

$O(\log_B N)$ update
Updating Multislab Structure

- **Problem:** Insertion of segment may change total order completely

  - Seems hard to control changes

  \[ \downarrow \]

  Need to rebuild multislab structure completely!

- **Segment deletion** does not change order \( \Rightarrow O(\log_B N) \) I/O delete
Updating Multislab Structure

• Recall that each node in multislab structure is augmented with maximal segment for each child and each slab
  – Deleted segment may be stored in nodes on one root-leaf path
  – Stored segment may correspond to several slabs

• **Delete** in $O(\log_B N)$ I/Os amortized:
  – Search leaf-root path and replace segment with segment above in relevant slab
  – Relevant replacement segments found in leaf or on path
  – Use global rebuilding to delete from leaf
Dynamic Point Location

• Semi-dynamic point location structure:
  – $O(N/B)$ space
  – $O\left(\frac{N}{B} \log B \frac{N}{B}\right)$ I/O construction
  – $O\left(\log^2 B N\right)$ I/O query
  – $O\left(\log B N\right)$ I/O amortized delete

• Using external logarithmic method we get:
  – Space: $O(N/B)$
  – Insert: $O\left(\log^2 B N\right)$ amortized
  – Deletes: $O\left(\log B N\right)$ amortized
  – Query: $O\left(\log^3 B N\right)$
    * Improved to $O\left(\log^2 B N\right)$ (complicated – fractional cascading)
Summary: Dynamic Point Location

• Maintain planar subdivision with $N$ segments such that region containing query point $q$ can be found efficiently

• We did not quite obtain desired ($1d$) bounds
  – Space: $O(N/B)$
  – Query: $O(\log_B^2 N)$
  – Insert: $O(\log_B^2 N)$ amortized
  – Deletes: $O(\log_B N)$ amortized

• Structure based on interval tree with use of several techniques, e.g.
  – Weight-balancing, logarithmic method, and global rebuilding
  – Segment sorting and augmented B-trees
Summary

• **Today** we discussed “dimension 1.5” problems:
  – **Interval stabbing** and point location
  – We obtained linear space structures with update and query bounds similar to the ones for 1d structures

• We developed a number of
  – Logarithmic method
  – Weight-balanced B-trees
  – Global rebuilding

• We also used techniques from yesterday:
  – Persistent B-trees
  – Construction using buffer technique
Summary

- **Tomorrow** we will consider two dimensional problems
  - 3-sided queries
  - Full (4-sided) queries