In this lecture, we will continue our discussion on the range searching problem. Recall that the input set \( P \) consists of \( N \) points in \( \mathbb{R}^2 \). Given an axis-parallel rectangle \( q \), a range query reports all the points of \( P \cap q \). We want to maintain a fully dynamic structure on \( P \) to answer range queries efficiently.

We will focus on non-replicating structures [2, 3]. Specifically, consider that each point in \( P \) has an information field (e.g., the menu of a restaurant) of \( L \) words, where \( L = o(B) \). Given a query, an algorithm must report the information fields of all the points that fall in the query window. A non-replicating structure is allowed to use \( O(N/B) + NL/B \) space. Note that the term \( NL/B \) is outside the big-\( O \). In other words, the structure can store each information field exactly once, and on top of that, consume \( O(N/B) \) extra space. The external range tree we discussed previous is not non-replicating (think: why?).

It is known [3] that the best query time of a non-replicating structure is \( O(\sqrt{NL/B} + KL/B) \) I/Os. We will introduce two structures that are able to guarantee this cost. The first one, called the kd-tree [1], is very simple but unfortunately is difficult to update. Then, we will see how to utilize the kd-tree to design another structure called the O-tree [2], which retains the same query performance as the kd-tree, and supports an update in \( O(\log_B N) \) I/Os amortized.

For convenience, we will assume \( L = O(1) \), namely, each information field requires constant words to store. Extensions to general \( L = o(B) \) are straightforward.

1 Kd-Tree

Structure. The kd-tree is a binary tree \( T \). Let \( splitdim \) be a variable whose value equals either the x- or y-axis. \( T \) is built by a function \( \text{build}(P, splitdim) \) which returns the root of \( T \). If \( P \) has at most \( B \) points, the function returns a single node containing all those points. Otherwise, it finds a line \( \ell \) perpendicular to axis \( splitdim \) that divides \( P \) into \( P_1 \) and \( P_2 \) of equal size. This can be done in \( O(|P|/B) \) I/Os using a “k-selection” algorithm. The function then creates a node \( r \) storing \( \ell \) (which is called the split line of \( r \)), and sets the left and right children of \( r \) to the nodes returned by recursively invoking \( \text{build}(P_1, alterdim) \) and \( \text{build}(P_2, alterdim) \), respectively, where \( alterdim \) equals the x-axis if \( splitdim \) is the y-axis, and vice versa. The function terminates by returning \( r \).

Figure 1 shows an example assuming \( B = 1 \). It is easy to see that every leaf node has at least \( B/2 \) points (think: why?). Hence, \( T \) has \( O(\log(N/B)) \) levels and can be constructed in \( O((N/B) \log(N/B)) \) I/Os.

Query. Observe that each node \( u \) of \( T \) corresponds to a bounding rectangle \( \text{rec}(u) \) which is the intersection of all the half-planes implied by the root-to-\( u \) path. For example, in Figure 1, the rectangle of node \( \ell_3 \) is the half-plane on the right of \( \ell_1 \), whereas that of node \( h \) is bounded by \( \ell_1, \ell_3, \ell_6 \) and the x-axis. Given a range query with search region \( q \), we simply access all the nodes \( u \) such that \( \text{rec}(u) \) intersects \( q \), and report the points covered by \( q \) stored in the leaf nodes visited.
Analysis. We will show that the query cost is $O(\sqrt{N/B} + K/B)$. Clearly, the nodes accessed can be divided into two categories: nodes whose bounding rectangles:

1. intersect at least one edge of $q$;

2. are enclosed by $q$.

For a node of Category 2, its entire subtree must be visited, with all of its leaf nodes having to be reported. Hence, the number of nodes of this category is $O(K/B)$. Next, we focus on the nodes of Category 1.

We prove actually a stronger result:

**Lemma 1.** The number of nodes whose bounding rectangles intersect any vertical (or horizontal) line $\ell$ is at most $O(\sqrt{N/B})$.

**Proof.** Let $f(N)$ be the maximum number of nodes whose bounding rectangles intersect $\ell$ among all the kd-trees with $N$ nodes. Let $u_1$ be the root of any such kd-tree. Assume without loss of generality that the split line of $u_1$ is perpendicular to the x-axis. Again, without loss of generality, assume that $\ell$ is on the right of the split line $\ell_1$ of $u_1$. Let the right child of $u_1$ be $u_3$ having split line $\ell_3$. Let the left and right children of $u_3$ be $u_4$ and $u_5$, respectively. See Figure 2.

Clearly, $\ell$ intersects $\text{rec}(u_1)$ and $\text{rec}(u_3)$, and does not intersect the bounding rectangle of any node in the left subtree of $u_1$. The subtree of $u_4$ ($u_5$) is a kd-tree with $N/4$ nodes with the split line
of the root being perpendicular to the x-axis. Hence, the number of nodes in that kd-tree whose bounding rectangles intersect ℓ is at most \( f(N/4) \). It follows that

\[
    f(N) = 2 + 2f(N/4)
\]

with \( f(N) = 1 \) if \( N \leq B \). Solving the recurrence gives \( f(N) = O(\sqrt{N/B}) \).

We thus conclude that there are \( 4 \cdot O(\sqrt{N/B}) = O(\sqrt{N/B}) \) nodes of Category 1.

**Theorem 1.** A kd-tree on a set of \( N \) points in \( \mathbb{R}^2 \) occupies \( O(N/B) \) space, answers a range query in \( O(\sqrt{N/B} + K/B) \) I/Os, and can be constructed in \( O(\frac{N}{B \log^2 B} \) I/Os.

The following follows immediately:

**Corollary 1.** For some integer \( N \), the kd-tree on a dataset of size \( O(B \log^2 N) \) consumes \( O(\log^2 N) \) space, answers a query in \( O(\log_B N + K/B) \) I/Os, and can be updated in \( O(\log^2 N \cdot \log_B N) \) I/Os per insertion and deletion, by re-constructing the tree from scratch after every update.

## 2 O-Tree

Next, we will leverage Corollary 1 to design the next structure O-tree. We will learn a technique called bootstrapping, which utilizes an inefficient structure (such as the kd-tree) to build an efficient structure.

### 2.1 Structure

Let \( N_0 \) be an integer that equals \( \Theta(N) \), where \( N \) is the number of points in the underlying dataset \( P \). The O-tree takes \( N_0 \) as a parameter. You may wonder at this point what happens if \( N \) has grown (or shrunk) sufficiently such that \( N_0 = \Theta(N) \) no longer holds. We will see that this can be dealt with using global rebuilding. Until then, we will assume that \( N_0 = \Theta(N) \) always holds.

Let \( V \) be a set of \( s \) vertical slabs that partition \( P \) into \( P_1, ..., P_s \) of roughly the same size. Specifically, we will make sure each \( P_i \) (\( 1 \leq i \leq s \)) has between \( \frac{1}{4} \sqrt{N_0 B} \cdot \log_B N_0 \) and \( \sqrt{N_0 B} \cdot \log_B N_0 \) points. In other words, \( s = \Theta(\sqrt{N_0} \sqrt{\frac{N_0}{B \log_B N_0}}) \). We use a B-tree \( V \) to index the (total order of the) slabs in \( V \). Number those slabs as \( 1, ..., s \) from left to right.

Next let us focus on one particular \( P_i \). We use a set \( H_i \) of \( h_i \) horizontal slabs to partition it into \( P_i[1], ..., P_i[h_i] \) of roughly the same size. Specifically, each \( P_i[j] \) (\( 1 \leq j \leq h_i \)) has between \( \frac{1}{4} B \log_B^2 N_0 \) and \( B \log_B^2 N_0 \) points, namely, \( h_i = \Theta(\sqrt{N_0} \sqrt{\frac{N_0}{B \log_B N_0}}) \). The slabs in \( H_i \) are indexed by a B-tree \( H_i \). Number them as \( 1, ..., h_i \) from bottom to top.

We refer to each set \( P_i[j] \) of points as a cell, and manage them with a kd-tree of Corollary 1. Note that each cell is naturally associated with a rectangle, which is the intersection of the \( i \)-th cell of \( V \) and the \( h_i \)-th cell of \( H_i \).

This completes the description of the O-tree. Since the information field of each point is stored in only one kd-tree, the O-tree is non-replicating. As for the space consumption, all the kd-trees occupy \( O(N/B) \) space in total. \( V, H_1, ..., H_s \) together use \( O(\frac{N_0}{B^2 \log_B N_0}) = o(N/B) \) space. The total space is therefore linear.
2.2 Query

Given a range query with search region \( q = [x_1, x_2] \times [y_1, y_2] \), we first identify \( \alpha_1 \) (\( \alpha_2 \)) such that \( x_1 \) (\( x_2 \)) is covered by the \( \alpha_1 \)-th (\( \alpha_2 \)-th) slab of \( V \). Then, for each \( i \in [\alpha_1, \alpha_2] \), identify \( \beta_i[1] \) (\( \beta_i[2] \)) such that \( y_1 \) (\( y_2 \)) is covered by the \( y_1 \)-th (\( y_2 \)-th) slab of \( H_i \). We then simply search the kd-trees of all \( P_i[j] \) where \( \alpha_1 \leq i \leq \alpha_2 \) and \( \beta_i[1] \leq j \leq \beta_i[2] \).

Using the relevant B-trees, \( \alpha_1, \alpha_2 \), and the \( \beta_i[1], \beta_i[2] \) of all \( i \) can be found in \( O(s \log_B N) = O(\log_B N \cdot \sqrt{N}) = O(N/B) \) I/Os. Regarding the cost on kd-trees, first notice that all the points in cell \( P_i[j] \) where \( \alpha_1 < i < \alpha_2 \) and \( \beta_i[1] < j < \beta_i[2] \) must be covered by \( q \). Therefore, the time of accessing the kd-trees on those cells is \( O(K/B) \). The rest of the query cost comes from the kd-trees on the “boundary cells” whose rectangles intersect an edge of \( q \). Clearly, there can be at most \( O(N/B) \) such kd-trees. By Corollary 1, querying each of them takes \( O(\log_B N) \) cost (plus the linear output cost). Thus, the overall query overhead is \( O(\sqrt{N/B} + K/B) \).

2.3 Update

**Insertion.** To insert a point \( p \), we first identify the cell \( P_i[j] \) whose rectangle covers it in \( O(\log_B N) \) I/Os. Then, we insert \( p \) in the kd-tree of that cell in \( O(\log_B N) \) I/Os.

If \( P_i[j] \) has more than \( \gamma_{cell} = B \log_B^2 N_0 \) points, a cell overflow occurs. In this case, we split the cell by a horizontal line into two cells of the same size, and rebuild their kd-trees in \( O(\gamma_{cell} \cdot \log_B \log_B N) \) I/Os. Note that a new cell has at most \( 1 + \gamma_{cell}/2 \) points. Accordingly, we update \( \mathcal{H}_i \) in \( O(\log_B N) \) I/Os.

If \( P_i \) (i.e., the \( i \)-th slab in \( V \)) has more than \( \gamma_{slab} = \sqrt{N_0 B} \cdot \log_B N_0 \) points, a slab overflow occurs. In this case, we split \( P_i \) into two slabs \( P_i, P_{i+1} \), and cut each of them horizontally into cells of size \( \gamma_{cell}/2 \) in \( \gamma_{slab}/B \) I/Os (think how to do so\(^1\)). Then, we rebuild the kd-trees of those cells, as well as \( \mathcal{H}_i \) and \( \mathcal{H}_{i+1} \), in \( O(\gamma_{slab} \cdot \log_B \log_B N) \) I/Os. Note that a new slab has at most \( 1 + \gamma_{slab}/2 \) points. Finally, \( \mathcal{V} \) is updated in \( O(\log_B N) \) I/Os.

**Deletion.** To delete a point \( p \), we first remove it from the cell \( P_i[j] \) it belongs to in \( O(\log_B^3 N) \) I/Os. If \( P_i[j] \) has less than \( \gamma_{cell}/4 \) points, a cell underflow occurs, in which case we merge it with the cell above it (or below it, whichever exists). If the resulting cell contains more than \( 3\gamma_{cell}/4 \) points, split it into two of equal size. In this way, we can ensure that a new cell has between \( 3\gamma_{cell}/8 \) and \( 3\gamma_{cell}/4 \) points. In any case, we rebuild the kd-trees of the new cells in \( O(\log_B^2 N \cdot \log_B \log_B N) \) I/Os, and modify \( \mathcal{H}_i \) in \( O(\log_B N) \) I/Os.

If \( P_i \) has less than \( \gamma_{slab}/4 \) points, a slab underflow occurs. In this case, we merge \( P_i \) with its left (or right) slab. If the resulting slab has more than \( 3\gamma_{slab}/4 \) points, split it into two of equal size, to guarantee that a new slab has between \( 3\gamma_{slab}/8 \) and \( 3\gamma_{slab}/4 \) points. In any case, we rebuild the kd-trees of the new cells in \( O(\gamma_{slab} \cdot \log_B \log_B N) \) I/Os, and modify \( \mathcal{V} \) in \( O(\log_B N) \) I/Os.

**Construction.** All the cells can be easily obtained in \( O(N/B \log_B N) \) I/Os by sorting. After that, each kd-tree can be constructed in \( O(\log_B^2 N \cdot \log_B \log_B N) \) I/Os, rendering the total overhead of \( O(N/B \log_B \log_B N \) of building all kd-trees.

**Cost.** Clearly, if no cell/slab overflow/underflow happens, an update finishes in \( O(\log_B^3 N) \) I/Os. A cell overflow/underflow, on the other hand, demands \( O(\gamma_{cell} \cdot \log_B \log_B N) \) I/Os. However, since a new cell requires at least \( \Omega(\gamma_{slab}) \) updates to incur the next overflow/underflow, each update

\(^1\)The last cell may have less than \( \gamma_{cell}/2 \) points. If it has at least \( 3\gamma_{cell}/8 \) points, we leave it there directly. Otherwise, we merge it with the cell below it to create a cell of size less than \( 7\gamma_{cell}/8 \).
accounts for only $O(\log_2 \log_B N)$ I/Os for a cell overflow/underflow. A similar argument shows that an update is amortized on $O(\log_2 \log_B N)$ I/Os for the cost of remedying a slab overflow/underflow.

We conclude:

**Lemma 2.** As long as the assumption $N_0 = \Theta(N)$ holds, there is a non-replicating structure that consumes linear space, answers a query in $O(\sqrt{N/B} + K/B)$ I/Os, and supports an update in $O(\log_B^3 N)$ I/Os amortized. The structure can be built in $O(\frac{N}{B} \log_2 \frac{N}{B})$ I/Os.

### 2.4 Global Rebuilding

The assumption $N_0 = \Theta(N)$ can be removed easily. Suppose that we have rebuilt a new O-tree by setting $N_0$ to the size $N$ of the current dataset. Then, we handle the next $N_0/2$ updates using the algorithms of the previous subsection, during which $N$ can range from $N_0/2$ to $3N_0/2$, and is therefore $\Theta(N_0)$. Right after finishing with $N_0/2$ updates, however, we destroy the O-tree, and construct a fresh one by performing $N$ insertions in $O(N \log_B^3 N)$ I/Os. By the standard analysis of global rebuilding, each update bears only $O(\log_B^3 N)$ I/Os amortized. So, now we can claim:

**Lemma 3.** There is a non-replicating structure that consumes linear space, answers a query in $O(\sqrt{N/B} + K/B)$ I/Os, and supports an update in $O(\log_B^3 N)$ I/Os amortized. The structure can be built in $O(\frac{N}{B} \log_2 \frac{N}{B})$ I/Os.

### 2.5 Bootstrapping

We have obtained a linear space structure with the optimal query performance which can be updated in poly-logarithmic I/Os. This is a significant improvement over the kd-tree. Remember that this is achieved by using the inferior structure of Corollary 1 to handle small datasets (of size at most $B \log_B^2 N$)—an idea known as bootstrapping.

Somewhat surprisingly, we can bootstrap again to achieve our desired logarithmic update bound, by doing (almost) nothing. Observe that Lemma 3 gives us a stronger version of Corollary 1:

**Corollary 2.** For some integer $N$, there is a non-replicating structure on a dataset of size $O(B \log_B^2 N)$ consumes $O(\log_B^3 N)$ space, answers a query in $O(\log_B^3 N + K/B)$ I/Os, and can be updated in $O(\log_B^3 \log_B N)$ I/Os amortized per insertion and deletion. The tree can be constructed in $O(\log_B^2 N \cdot \log_2 \log_B N)$ I/Os.

Now, let us implement every cell structure of the O-tree (which was a kd-tree before) as a structure of Corollary 2. Everything remains the same, except that now an update takes $O(\log_B N + \log_B^3 \log_B N) = O(\log_B N)$ I/Os if no cell/slab overflow/underflow occurs. Therefore, we have arrived at our ultimate structure:

**Theorem 2.** There is a non-replicating structure that consumes linear space, answers a query in $O(\sqrt{N/B} + K/B)$ I/Os, and supports an insertion and deletion in $O(\log_B N)$ I/Os amortized.

**Remarks.** It is natural to wonder whether we can apply it once more to lower the update time even further. The answer is negative because by utilizing the structure of Corollary 2 we have already conquered the bottleneck, which was the expensive update cost of the kd-tree in Corollary 1. The new bottleneck is the logarithmic cost of finding which cell to update, and cannot be improved any more.
References

