This and the next few lectures will be devoted to range searching, which is another classic problem in computer science. We will study the problem in 2d space. Let $P$ be a set of $N$ points in $\mathbb{R}^2$. Given an axis-parallel rectangle $q$, a range query reports all the points of $P$ covered by $q$, namely, $P \cap q$. In the static version of the problem, we want to pre-process $P$ into a structure so that a range query can be answered efficiently. In its dynamic version, we want to maintain the structure along with the updates on $P$ while still ensuring good query performance.

This lecture will focus on a special version of the problem called 3-sided range searching, where the query rectangle $q$ has the form $[x_1, x_2] \times [y, \infty)$, i.e., it is a 3-sided rectangle with its top open. We will introduce the external priority search tree, which is a dynamic structure that solves this problem optimally. Our real objective of discussing this structure, however, is to learn an important technique called filtering search whose main idea is as follows. Assume that, in general, we want to achieve the query time of $O(\log_B N + K/B)$ where $N$ is the dataset size and $K$ is the number of elements reported. It is deceptively clear that the terms $\log_B N$ and $K/B$ account for the cost of searching and reporting (the results), respectively, which is indeed true in many problems we have seen so far (e.g., range reporting from a B-tree). In filtering search, we challenge this conventional wisdom. Instead, we will make the search cost as large as $O(\log_B N + K/B)$, namely, charging some searching time over the elements reported. This, together with the non-escapable $O(K/B)$ output time, gives the total cost of $O(\log_B N + 2K/B) = O(\log_B N + K/B)$.

The following result is a direct corollary from the persistent B-tree:

**Lemma 1.** There is a persistent B-tree that consumes $O(N/B)$ spaces and answers any 3-sided range query in $O(\log_B N + K/B)$ I/Os, where $K$ is the number of points reported.

**Proof.** We can convert the 3-sided range searching problem as follows. From each point $p \in P$, create a vertical ray shooting downwards from $p$. Let $R$ be the set of all such rays. Then, $p$ falls in a 3-sided rectangle $q = [x_1, x_2] \times [y, \infty)$ if and only if its ray intersects the horizontal segment $[x_1, x_2] \times y$. Hence, we can instead find all the rays in $R$ intersecting $[x_1, x_2] \times y$, a problem that can be solved by a persistent B-tree with the performance claimed. \qed

The persistent B-tree is static. The external priority search tree we will introduce is fully dynamic. As in the previous lecture, we will assume that $M \geq B^2$, and that $P$ is in general position (such that no two points have the same x- or y-coordinate).

## 1 Priority search tree in RAM

As usual, we start by seeing how 3-sided range searching can be solved in RAM. The structure to be described is called the priority search tree, which as will be clear shortly can be regarded as the
introduction of a balanced binary tree and a priority queue. Build a binary tree $T$ on $X$ where all the leaf nodes are at the bottom level and all the points are stored at the leaf nodes. Given a node $u$ in $T$, denote by $sub(u)$ the subtree rooted at $u$. We say that a point is stored in $sub(u)$ if its x-coordinate is in $sub(u)$. Each node $u$ in $T$ corresponds to a vertical slab $\sigma(u)$ in $\mathbb{R}^2$, which is the range of the x-coordinates of points that should be inserted in $sub(u)$.

We associate each node $u$ with a point, which is the prior point of $u$, and is denoted as $prior(u)$. If $u$ is the root of $T$, $prior(u)$ is the highest point in $P$ (i.e., the one with the maximum y-coordinate). In general, $prior(u)$ is the highest point among all the points stored in $sub(u)$, excluding the prior points of the proper ancestors of $u$. It is clear that every point in $P$ is a prior point of exactly one node. The overall space consumption is therefore $O(N)$.

Given a query rectangle $q = [x_1, x_2] \times [y, \infty)$, let $\Pi_1$ and $\Pi_2$ be the root-to-leaf paths of $T$ leading to the leaf nodes whose slabs contain $x_1$ and $x_2$, respectively. Let $\ell$ be the horizontal line intersecting the y-axis at $y$. Starting from the root $T$, we will ensure that only nodes whose slabs intersect $[x_1, x_2]$ will be searched. In general, if we are currently at a node $u$ in $T$, check whether $prior(u)$ is below $\ell$. If yes, all the points in $sub(u)$ are either below $\ell$ or have already been inspected as the prior points of some ancestors of $u$. Hence, $sub(u)$ can be safely pruned. If no, we report $prior(u)$ if it is in $q$, and then, descend to the child node(s) whose slabs intersect $[x_1, x_2]$.

A bit of analysis will show that the query time is $O(\log N + K)$. There are two types of non-root nodes we accessed: (i) those whose parents are on $\Pi_1$ or $\Pi_2$, and (ii) those whose parents have slabs completely inside $[x_1, x_2]$. Clearly, there are only $O(\log N)$ nodes of the first type. As for a node $u$ of the second type, notice that a point must have been reported at the parent of $u$. Hence, the number of such nodes is at most $O(K)$.

Remark. Observe that we followed a “pay-as-you-go” strategy in the query algorithm to justify the constant cost of accessing a node. The justification is straightforward if the node’s parent is on $\Pi_1$ or $\Pi_2$ because there are not many such cases (i.e., only $O(\log N)$). We may also, however, need to visit many nodes whose parents are not in $\Pi_1$ or $\Pi_2$. Fortunately, the cost of processing such a node has already been paid when we were at its parent.

2 External priority search tree

Structure. The external priority search tree implements the idea of the previous section in external memory. The base tree is a weight balanced B-tree $T$ on $X$ where each leaf node has a capacity $B$ and each internal node can have at most $B$ child nodes. Each node $u$ in $T$ naturally corresponds to a vertical slab $\sigma(u)$ in $\mathbb{R}^2$. As before, let $sub(u)$ be the subtree of $u$.

Each node $u$ is associated with a prior set denoted as $prior(u)$. To explain its definition, first assume that $u$ is an internal node with $f$ child nodes $u_1, \ldots, u_f$. Let $prior(u, u_i)$ be the $B$ highest points in $sub(u_i)$ excluding the prior points of the proper ancestors of $u$ (if $u$ is the root, then no point is excluded). If less than $B$ points satisfy the condition, $prior(u, i)$ includes all of them. Then, $prior(u)$ simply unions $prior(u, u_1), \ldots, prior(u, u_f)$. If $u$ is a leaf node, $prior(u)$ is the set of points in $u$ that have not been included as a prior point in any proper ancestor of $u$.

If $u$ is an internal node, we associate $u$ with a persistent B-tree $T(u)$ built on $prior(u)$ (which has at most $B^2$ points). To facilitate updates, we also use a B-tree $T'(u)$ to index the points of $prior(u)$ by their y-coordinates. If $u$ is a leaf node, it is associated with an extra block to store $prior(u)$. A leaf node requires $O(1)$ blocks, whereas an internal node consumes $O(B)$ blocks. As
the total number of internal nodes in $T$ is $O(N/B^2)$, the overall space consumption is $O(N/B)$.

**Query.** Given a query rectangle $q = [x_1, x_2] \times [y, \infty)$, as before, define $\Pi_1$ ($\Pi_2$) as the root-to-leaf path of $T$ to the leaf node whose slab contains $x_1$ ($x_2$). The query algorithm will report points only from the prior sets. The root of $T$ is the first node accessed. In general, at an internal node $u$, search $T(u)$ to report all the points there that fall in $q$. Let $u_1, \ldots, u_f$ be the child nodes of $f$. For each $i \in [1, f]$ such that $B$ points have been reported from $T(u_i)$, we access $u_i$. Furthermore, if a child node is on $\Pi_1$ or $\Pi_2$, it is also accessed. Note that if a child node, say $u_j$, is on neither $\Pi_1$ nor $\Pi_2$, and yet, less than $B$ points from $\text{prior}(u, u_j)$ have been reported from $T(u)$, we can claim that all the qualifying points in $\text{sub}(u_j)$ have already been output. Finally, at a leaf node $u$, we simply examine all the points in $\text{prior}(u)$.

The analysis we presented earlier in RAM can be adapted to prove that the cost of the above algorithm is $O(\log_B N + K/B)$. At each node $u$ accessed, we spend $O(1+K_u/B)$ I/Os (see Lemma 1), where $K_u$ is the number of points reported from $T(u)$. Refer to the term “1” as the search cost at $u$. All the nodes accessed can be divided into two groups: (i) those on $\Pi$ and $\Pi$, and (ii) those whose slabs are covered completely by $[x_1, x_2]$. We concentrate on the second type. For each node $u$ of this type, $\Omega(B)$ points from $\text{sub}(u)$ must have been reported at the parent of $u$. Hence, we charge the search cost of $u$ on those points. In this way, each point reported bears $O(1/B)$ additional I/Os. The overall query cost is therefore $O(\log_B N + K/B)$.

### 3 Performing updates

Next, we show how to make the external priority search tree dynamic. First of all, recall that each node $u$ has a secondary structure $T(u)$, which is a persistent B-tree indexing at most $B^2$ points. From Lemma 1 of the previous lecture, we have:

**Lemma 2.** Under the tall-cache assumption, $T(u)$ can be updated in $O(1)$ I/Os per insertion and deletion.

Given a point $p$ and a node $u$, sometimes we need to perform a demotion to push $p$ into the prior sets of the nodes in $\text{sub}(u)$. If $u$ is a leaf node, we simply put $p$ into the block storing $\text{prior}(u)$. Otherwise, let $u'$ be the child node of $u$ such that $\sigma(u')$ contains $p$. If $\text{prior}(u, u')$ currently has less than $B$ points, we finish by inserting $p$ in $T(u)$ and $T'(u)$. Otherwise, we first find the lowest point, say $p'$, in $\text{prior}(u, u')$, which can be done in $O(1)$ I/Os by simply retrieving all the $B$ points of $\text{prior}(u, u')$ by performing a special 3-sided range query using $\sigma(u')$ as the search region on $T(u)$. Then, remove (insert) $p'$ ($p$) in $T(u)$ and $T'(u)$. After this, we recursively perform a demotion of $p'$ on $u'$. In general, if $u$ is at level $l$, in the worst case we perform constant I/Os at each node along a single path from $u$ to a leaf node. Hence, a demotion finishes in $O(l+1)$ I/Os.

Conversely, given a node $u$, occasionally we need to perform a promotion to remove the highest point $p$ from the prior sets of the nodes in $\text{sub}(u)$, if such a point exists. If $u$ is a leaf node, this is trivial. Otherwise, $p$ is exactly the highest point in $\text{prior}(u)$, which can be obtained from $T'(u)$ in constant I/Os. Then, we remove $p$ from $\text{prior}(u)$, recursively promote a point, say $p'$, from the prior sets of the nodes in $\text{sub}(u')$, where $u'$ is the child node of $u$ having $p'$ in its subtree, and finally add $p'$ to $\text{prior}(u)$. In this process, the secondary structures of the nodes along a single path from $u$ to a leaf node need to be updated by reversing the steps in a demotion. If $u$ is at level $l$, the promotion takes $O(l+1)$ I/Os.

3
3.1 Insertion

Assume that $p$ is the point being inserted. We first insert the x-coordinate of $p$ in $T$, without handling the overflows that may have happened. Let $\Pi$ be the root-to-leaf path we just followed. Next, we determine the node whose prior set $p$ belongs to. Towards this purpose, we examine the nodes on $\Pi$ in the top-down order. Let $u$ be an internal node being examined, and let $u'$ be its child node on $\Pi$. We retrieve all the (at most) $B$ points in $\text{prior}(u, u')$ in $O(1 + B/B) = O(1)$ I/Os (see a similar operation explained in promotion). If $p$ is higher than the lowest point in $\text{prior}(u, u')$, we know that $p$ should be added to $\text{prior}(u, u')$ (and hence, $\text{prior}(u)$); otherwise, we descend to inspect $u'$. If eventually we have come to the leaf node on $\Pi$, $p$ should be added to its prior set. In any case, adding $p$ to the prior set of a node can be achieved by simply performing a demotion of $p$ on that node. The cost so far is $O(\log_B N)$.

Finally, we handle in the bottom-up order the nodes that may have overflowed during the insertion of $p$ in $T$. Let $u$ be such a node and $\hat{u}$ its parent node. Split $u$ into $u_1, u_2$. Rebuild the secondary structures of $u_1$ and $u_2$ respectively in $O(B)$ I/Os (notice that all the points indexed by each of those structures can be loaded in memory). The split has divided $\text{prior}(\hat{u}, u)$ into $\text{prior}(\hat{u}, u_1)$ and $\text{prior}(\hat{u}, u_2)$. Now $\text{prior}(\hat{u}, u_1)$ may have less than $B$ points. Hence, we perform up to $B$ promotions to fill up $\text{prior}(\hat{u}, u_1)$. Repeat the same for $\text{prior}(\hat{u}, u_2)$. At this time, rebuild the secondary structure of $\hat{u}$ in $O(B)$ I/Os.

Assume that $u$ is at level $l$. If $l = 0$, the overflow handling finishes in constant I/Os. Otherwise, the cost is $O(lB)$. As $T$ is a weight-balanced B-tree, the weight of $u$ is $\Theta(B^{l+1})$, meaning that $\Omega(B^{l+1})$ updates have been performed in $\text{sub}(u)$ since the creation of $u$. Hence, we can amortize the overflow handling cost over those updates, such that each of them bears $O(lB/B^{l+1}) = O(1)$. As each update can bear such a cost at most $O(\log_B N)$ times, each insertion can be performed in $O(\log_B N)$ I/Os amortized.

3.2 Deletion

Recall that, in answering a query, we report points only from secondary structures. This suggests that we can handle deletions by applying global rebuilding, in a way similar to what we did in the external interval tree. Leaving this as an exercise, we conclude:

**Theorem 1.** Under the tall-cached assumption, there exists a structure on a set of $N$ points that uses $O(N/B)$ space, answers a 3-sided range query in $O(\log_B N + K/B)$, and can be updated in $O(\log_B N)$ amortized I/Os per insertion and deletion.

Bibliography

The priority search tree in internal memory was developed by McCreight [3]. Its external version is due to Arge, Samoladas and Vitter [1]. They also showed that Theorem 1 still holds even without the tall-cache assumption, and that the update cost can be made worst-case. The filtering search technique was proposed by Chazelle [2].

References
