1 Overview

In the last lecture we continued looking at ARBORALLY SATISFIED SETS and their equivalence to binary search trees. Using the concept of POSITIVELY SATISFIED SETS of points, we proved that the optimal BST solution is bounded from below by the maximum number of independent, positive rectangles in the geometric view.

In this lecture we diverge from previous lectures with a guest talk by visiting professor Riko Jacob. His talk focuses on the I/O model [AV88] and tree structures in that context. In this lecture he provides a background for many concepts such as the I/O model, (2,3) and (2,4) trees, and other fundamental data structures concepts.

2 Motivation for Binary Search Trees

When considering data structures and their efficiency, we look at the operations needed to perform and the corresponding amortized work needed. The most fundamental operations we first consider are Search and Update(insert). For these basic operations, the amortized cost of a self-balancing and update-efficient search tree, like an AVL tree is $O(\log n)$. However, there are data structures that can improve on this, such as a hash map, which is capable of performing these operations in constant expected time. While hash maps provide constant time searching and updating for certain cases, they are unable to provide efficient PREDECESSOR QUERY and RANGE QUERY operations. BSTs, however, are able to handle these operations.

2.1 Predecessor Query

The PREDECESSOR QUERY operation returns the largest element that is smaller than the target $x$ in set $S$. Formally, it is defined as:

Given target $x$ and sorted set $S$, find the largest $y \in S$ s.t. $y \leq x$.

With a BST, we can easily perform a predecessor search by simply searching for the target $x$ and, if found, traverse back up the tree until a left branch can be taken, then the right-most path is taken to the leaf to find the predecessor of $x$.

Note that there are data structures that can perform predecessor search in $O(\log \log U)$, where $U$ is the size of the universe of numbers we draw from. However, for any irreducible set of numbers using a comparison-based search, $O(\log n)$ is optimal for performing a predecessor search.
2.2 Range Query

**Range Query** is the operation defined as:

Given search range \([x_1, x_2]\), s.t. \(x_1 < x_2\), and set \(S\), find all elements \(y \in S\) s.t. \(x_1 \leq y \leq x_2\).

This can be accomplished with a BST by the following steps.

- Identify the lowest common ancestor, \(x'\), of \(x_1\) and \(x_2\),
- follow the two paths from \(x'\) to \(x_1\) and \(x_2\), marking all subtrees falling between them, and
- report the leaves of all marked subtrees as the matches of range query.

Note that the cost of performing a range query on a BST is \(O(\log n + k)\), where \(k\) is the number of elements matching the range query. Figure 1 illustrates an example of a range query on a BST.

3 I/O Model

The I/O model [AV88] is a computational model that describes the complexity of an algorithm in terms of the number of input and output operations to external memory. Given a large input set \(A\), of size \(N\), and an internal memory of size \(M < N\), we determine the execution time of an algorithm in the number of accesses to \(A\). Note that all internal computations are not included in runtime and are considered ‘free’. The model further goes to consider **blocked access**, where \(B\) consecutive elements are loaded from \(A\) to internal memory in a single access. Figure 2 illustrates the I/O model and how memory is loaded from input to internal memory.
3.1 BSTs in the I/O Model

To be I/O efficient, data structures must be constructed to take advantage of data locality and efficiently use all $B$ elements of each I/O operation. Typical BST structures, however, do not exhibit this type of data locality. Let us consider loading $B$ elements per I/O operation and attempting to query a standard, balanced BST, stored in memory in a level-by-level (breadth-first search) order. The first I/O will load the first $B$ elements of the BST, which corresponds to all of the elements in the first $\log B$ levels of the tree. After that, all data accesses will be to arbitrary memory locations and we will have to perform a separate I/O for each level of the tree. This gives us a total number of I/O to query a BST of:

$$Q(T) = O(\log N - \log B) = O(\log \frac{N}{B})$$

Clearly this is not optimal, since we only utilize all $B$ elements of the first I/O operation and waste data from all subsequent I/Os. To improve the data access pattern, we consider a search tree where $B$ consecutive elements can be searched at once, such as a B-tree. To introduce the B-tree, we first give a background on (2,3) and (2,4) trees.

4 (2,3) and (2,4) trees

A (2,3) tree is a type of search tree where each node can have either 2 or 3 children. A node with 2 children will contain 1 key value, and a node with 3 children will have 2 key values. A key feature of this type of tree is that it can easily be re-balanced from inserts and deletes by the node
operations: SPLIT and MERGE. A SPLIT operation occurs when a full node (i.e., a node with the maximum number of keys) has another key inserted into it. When this happens, the node is split into 2 smaller nodes. In the case of a (2,3) tree, a node with 2 keys (and 3 children) becomes 2 nodes with 1 key (and 2 children) each. The new key can then be inserted into one of the new nodes.

The MERGE operation is the inverse of SPLIT and can occur when deleting a key from the tree. If we delete a key from a node with minimum keys (1 for a (2,3) tree), we merge the node with its neighbor to create a node with more keys. We can then delete the key from this new node.

For the case of (2,3) trees, the amortized cost of inserting or deleting a key is $O(\log n)$. To illustrate why our amortized cost is $O(\log n)$, we look at the case where a SPLIT operation will cause $O(\log n)$ split operations to occur. Figure 3 illustrates how the SPLIT operation can propagate from the leaf to the root. Note that, after this occurs, all the nodes that were just split now have the minimum number of keys. Therefore, deleting a key from any node in this path will cause MERGE operations to propagate all the way to the root. This switch from all nodes having the maximum number of keys to all having the minimum number of keys causes the updating operation to have an amortized cost of $O(\log n)$.

To reduce the cost of updating, we consider instead the (2,4) tree. (2,4) trees ‘relax’ the limits on the number of keys per node by allowing nodes to have up to 4 children each (and 3 keys). This reduces the amortized cost of inserting or deleting by causing a split or merge to leave the new node(s) in a state that allows additional inserts or deletes to be done without causing another split or merge. Figure 4 shows how splitting a full node in a (2,4) tree allows additional delete operations to be performed without having to merge. Similarly, after a merge, we can insert an additional element before having to split. This gives us an amortized cost of insert/delete of $O(1)$ for (2,4) trees (unlike (2,3) trees that have $O(\log n)$ amortized cost).

Figure 3: Illustration of $O(\log n)$ amortized update cost for (2,3) trees.
B-trees are a generalization of (2,4) trees. Formally, B-trees are defined as a \(((\frac{1}{2} - \epsilon)B, B)\) tree. That is, a B-tree is a tree where nodes can have anywhere from \((\frac{1}{2} - \epsilon) \cdot B\) to \(B\) children \(((\frac{1}{2} - \epsilon) \cdot B) - 1\) to \(B - 1\) keys). As with (2,4) trees, B-trees have a ‘relaxed’ range on the number of keys per node, giving us constant amortized time updates (excluding time to search for the update node).

Note that the lower-bound on the number of children per node must be \(< \frac{1}{2}\) to obtain this \(O(1)\) amortized update performance. Since each node has \(O(B)\) children, the amortized number of nodes that need to be accessed per operation is:

- Search: \(O(\log_B N)\)
- Predecessor: \(O(\log_B N)\)
- Range Query: \(O(\log_B N + \frac{K}{B})\)
- Update: \(O(1)\) (once the update location is found)

And at each node \(O(B)\) elements need to be accessed and evaluated. However, if we let \(B\) of our B-tree be the \(B\) parameter from the I/O model (i.e., we can load \(B\) elements per I/O), we can load an entire node with 1 I/O operation. Therefore, the above amortized costs for each operation becomes the amortized cost for our I/O model.

### 6 Buffered B-tree

In the previous section we saw that a B-tree is optimal for searching in the I/O model. Furthermore, B-trees can perform amortized constant time updates (excluding search time required to find the update location), thanks to a relaxed lower bound on the number of children per node. However, \(O(1)\) update time per update is not optimal, since only 1 element is updated, while each I/O is of size \(B\). If we are able to batch updates together in shared regions, we can group updates into a single I/O and, potentially, obtain an amortized update cost of \(\frac{1}{B}\) per update.

The Buffered B-tree [BF03] is a data structure that groups updates for improved amortized efficiency. A buffered B-tree has \(\Theta(B)\) elements per leaf and each internal node has degree \(\Theta(\frac{\delta}{\log N})\).
This $\delta$ parameter acts as a trade-off between amortized update and search performance. Each node has a buffer of $B$ elements to store delayed updates and allow updates to be performed in a batch. Similar to the Buffer Tree [Arg95], when a buffer becomes full, the updates are propagated down the tree. However, unlike buffer trees, buffered B-trees propagate only a portion of this full buffer to whichever child that can take the most elements. This allows a single I/O to remove the maximum number of elements from the full buffer. Once a leaf node buffer becomes full, the updates are applied and the tree is re-balanced like an ordinary buffer tree. Using this technique of delaying updates, we are able to obtain improved amortized update performance at the cost of reduced search performance.

If we let $\delta = B^c \log N$, the resulting tree will have internal nodes of degree $O(B^c)$, giving the tree a height of $O(\frac{1}{\epsilon} \log_B N)$. Since we are able to store the first $O(\log_B M)$ levels of the tree in memory, searching will take $O(\frac{1}{\epsilon} \log_B \frac{N}{M})$ I/Os, which is within a constant factor of the standard B-tree. However, if we consider that, for $N$ updates, there are at most $O(\frac{N}{(B \log N)/\delta})$ buffer overflows per level, we get an amortized I/O cost of $B$ updates of:

$$O \left( \frac{B \cdot \delta}{B \log N} \cdot \frac{1}{\epsilon} \log_B \frac{N}{M} \right) = O \left( \frac{\delta}{\log N} \cdot \frac{1}{\epsilon} \log_B \frac{N}{M} \right)$$

$$= O \left( \frac{B^c \log N}{\log N} \cdot \frac{1}{\epsilon} \log_B \frac{N}{M} \right)$$

$$= O \left( \frac{B^c}{\epsilon} \cdot \log_B \frac{N}{M} \right)$$

Note that this is an improvement over the amortized update bounds of performing $B$ updates on a standard B-tree of $O(B \log_B \frac{N}{M})$ (when we include the cost of searching for the update node).

References

