11

Multiway Tree Data Structures

In this chapter, we explore some important search-tree data structures in external memory. An advantage of search trees over hashing methods is that the data items in a tree are sorted, and thus the tree can be used readily for one-dimensional range search. The items in a range $[x, y]$ can be found by searching for x in the tree, and then performing an inorder traversal in the tree from x to y.

11.1 B-trees and Variants

Tree-based data structures arise naturally in the online setting, in which the data can be updated and queries must be processed immediately. Binary trees have a host of applications in the (internal memory) RAM model. One primary application is search. Some binary search trees, for example, support lookup queries for a dictionary of N items in $O(\log N)$ time, which is optimal in the comparison model of computation. The generalization to external memory, in which data are transferred in blocks of B items and B comparisons can be done per I/O, provides an $\Omega(\log_B N)$ I/O lower bound. These lower bounds depend heavily

upon the model; we saw in the last chapter that hashing can support dictionary lookup in a constant number of I/Os.

In order to exploit block transfer, trees in external memory generally represent each node by a block that can store $\Theta(B)$ pointers and data values and can thus achieve $\Theta(B)$-way branching. The well-known balanced multiway *B-tree* due to Bayer and McCreight [74, 114, 220], is the most widely used nontrivial EM data structure. The degree of each node in the B-tree (with the exception of the root) is required to be $\Theta(B)$, which guarantees that the height of a B-tree storing N items is roughly $\log_B N$. B-trees support dynamic dictionary operations and one-dimensional range search optimally in linear space using $O(\log_B N)$ I/Os per insert or delete and $O(\log_B N + z)$ I/Os per query, where $Z = zB$ is the number of items output. When a node overflows during an insertion, it splits into two half-full nodes, and if the splitting causes the parent node to have too many children and overflow, the parent node splits, and so on. Splittings can thus propagate up to the root, which is how the tree grows in height. Deletions are handled in a symmetric way by merging nodes. Franceschini et al. [167] show how to achieve the same I/O bounds without space for pointers.

In the *B^+-tree* variant, pictured in Figure 11.1, all the items are stored in the leaves, and the leaves are linked together in symmetric order to facilitate range queries and sequential access. The internal nodes store only key values and pointers and thus can have a higher branching factor. In the most popular variant of B^+-trees, called *B^*-trees*, splitting can usually be postponed when a node overflows, by

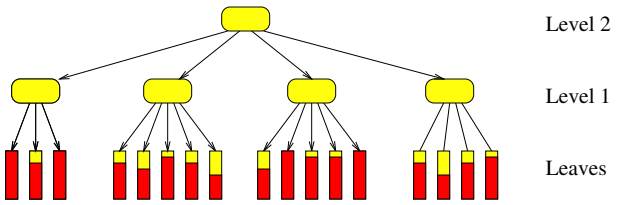

Fig. 11.1 B^+-tree multiway search tree. Each internal and leaf node corresponds to a disk block. All the items are stored in the leaves; the darker portion of each leaf block indicates its relative fullness. The internal nodes store only key values and pointers, $\Theta(B)$ of them per node. Although not indicated here, the leaf blocks are linked together sequentially.

"sharing" the node's data with one of its adjacent siblings. The node needs to be split only if the sibling is also full; when that happens, the node splits into two, and its data and those of its full sibling are evenly redistributed, making each of the three nodes about 2/3 full. This local optimization reduces the number of times new nodes must be created and thus increases the storage utilization. And since there are fewer nodes in the tree, search I/O costs are lower. When no sharing is done (as in B^+-trees), Yao [360] shows that nodes are roughly $\ln 2 \approx 69\%$ full on the average, assuming random insertions. With sharing (as in B*-trees), the average storage utilization increases to about $2\ln(3/2) \approx 81\%$ [63, 228]. Storage utilization can be increased further by sharing among several siblings, at the cost of more complicated insertions and deletions. Some helpful space-saving techniques borrowed from hashing are partial expansions [65] and use of overflow nodes [321].

A cross between B-trees and hashing, where each subtree rooted at a certain level of the B-tree is instead organized as an external hash table, was developed by Litwin and Lomet [236] and further studied in [64, 237]. O'Neil [275] proposed a B-tree variant called the SB-tree that clusters together on the disk symmetrically ordered nodes from the same level so as to optimize range queries and sequential access. Rao and Ross [290, 291] use similar ideas to exploit locality and optimize search tree performance in the RAM model. Reducing the number of pointers allows a higher branching factor and thus faster search.

Partially persistent versions of B-trees have been developed by Becker et al. [76], Varman and Verma [335], and Arge et al. [37]. By persistent data structure, we mean that searches can be done with respect to any time stamp y [142, 143]. In a partially persistent data structure, only the most recent version of the data structure can be updated. In a fully persistent data structure, any update done with time stamp y affects all future queries for any time after y. Batched versions of partially persistent B-trees provide an elegant solution to problems of point location and visibility discussed in Chapter 8.

An interesting open problem is whether B-trees can be made fully persistent. Salzberg and Tsotras [298] survey work done on persistent access methods and other techniques for time-evolving data. Lehman and Yao [231], Mohan [260], Lomet and Salzberg [239], and Bender

et al. [81] explore mechanisms to add concurrency and recovery to B-trees.

11.2 Weight-Balanced B-trees

Arge and Vitter [58] introduce a powerful variant of B-trees called *weight-balanced B-trees*, with the property that the weight of any subtree at level h (i.e., the number of nodes in the subtree rooted at a node of height h) is $\Theta(a^h)$, for some fixed parameter a of order B. By contrast, the sizes of subtrees at level h in a regular B-tree can differ by a multiplicative factor that is exponential in h. When a node on level h of a weight-balanced B-tree gets rebalanced, no further rebalancing is needed until its subtree is updated $\Omega(a^h)$ times. Weight-balanced B-trees support a wide array of applications in which the I/O cost to rebalance a node of weight w is $O(w)$; the rebalancings can be scheduled in an amortized (and often worst-case) way with only $O(1)$ I/Os. Such applications are very common when the the nodes have secondary structures, as in multidimensional search trees, or when rebuilding is expensive. Agarwal et al. [11] apply weight-balanced B-trees to convert partition trees such as kd-trees, BBD trees, and BAR trees, which were designed for internal memory, into efficient EM data structures.

Weight-balanced trees called BB[α]-trees [87, 269] have been designed for internal memory; they maintain balance via rotations, which is appropriate for binary trees, but not for the multiway trees needed for external memory. In contrast, weight-balanced B-trees maintain balance via splits and merges.

Weight-balanced B-trees were originally conceived as part of an optimal dynamic EM interval tree structure for stabbing queries and a related EM segment tree structure. We discuss their use for stabbing queries and other types of range queries in Sections 12.3–12.5. They also have applications in the (internal memory) RAM model [58, 187], where they offer a simpler alternative to BB[α]-trees. For example, by setting a to a constant in the EM interval tree based upon weight-balanced B-trees, we get a simple worst-case implementation of interval trees [144, 145] in the RAM model. Weight-balanced B-trees are also

preferable to BB[α]-trees for purposes of augmenting one-dimensional data structures with range restriction capabilities [354].

11.3 Parent Pointers and Level-Balanced B-trees

It is sometimes useful to augment B-trees with parent pointers. For example, if we represent a total order via the leaves in a B-tree, we can answer order queries such as "Is $x < y$ in the total order?" by walking upwards in the B-tree from the leaves for x and y until we reach their common ancestor. Order queries arise in online algorithms for planar point location and for determining reachability in monotone subdivisions [6]. If we augment a conventional B-tree with parent pointers, then each split operation costs $\Theta(B)$ I/Os to update parent pointers, although the I/O cost is only $O(1)$ when amortized over the updates to the node. However, this amortized bound does not apply if the B-tree needs to support cut and concatenate operations, in which case the B-tree is cut into contiguous pieces and the pieces are rearranged arbitrarily. For example, reachability queries in a monotone subdivision are processed by maintaining two total orders, called the leftist and rightist orders, each of which is represented by a B-tree. When an edge is inserted or deleted, the tree representing each order is cut into four consecutive pieces, and the four pieces are rearranged via concatenate operations into a new total order. Doing cuts and concatenation via conventional B-trees augmented with parent pointers will require $\Theta(B)$ I/Os per level in the worst case. Node splits can occur with each operation (unlike the case where there are only inserts and deletes), and thus there is no convenient amortization argument that can be applied.

Agarwal et al. [6] describe an interesting variant of B-trees called *level-balanced B-trees* for handling parent pointers and operations such as cut and concatenate. The balancing condition is "global": The data structure represents a forest of B-trees in which the number of nodes on level h in the forest is allowed to be at most $N_h = 2N/(b/3)^h$, where b is some fixed parameter in the range $4 < b < B/2$. It immediately follows that the total height of the forest is roughly $\log_b N$.

Unlike previous variants of B-trees, the degrees of individual nodes of level-balanced B-trees can be arbitrarily small, and for storage

purposes, nodes are packed together into disk blocks. Each node in the forest is stored as a node record (which points to the parent's node record) and a doubly linked list of child records (which point to the node records of the children). There are also pointers between the node record and the list of child records. Every disk block stores only node records or only child records, but all the child records for a given node must be stored in the same block (possibly with child records for other nodes). The advantage of this extra level of indirection is that cuts and concatenates can usually be done in only $O(1)$ I/Os per level of the forest. For example, during a cut, a node record gets split into two, and its list of child nodes is chopped into two separate lists. The parent node must therefore get a new child record to point to the new node. These updates require $O(1)$ I/Os except when there is not enough space in the disk block of the parent's child records, in which case the block must be split into two, and extra I/Os are needed to update the pointers to the moved child records. The amortized I/O cost, however, is only $O(1)$ per level, since each update creates at most one node record and child record at each level. The other dynamic update operations can be handled similarly.

All that remains is to reestablish the global level invariant when a level gets too many nodes as a result of an update. If level h is the lowest such level out of balance, then level h and all the levels above it are reconstructed via a postorder traversal in $O(N_h)$ I/Os so that the new nodes get degree $\Theta(b)$ and the invariant is restored. The final trick is to construct the new parent pointers that point from the $\Theta(N_{h-1}) = \Theta(bN_h)$ node records on level $h-1$ to the $\Theta(N_h)$ level-h nodes. The parent pointers can be accessed in a blocked manner with respect to the new ordering of the nodes on level h. By sorting, the pointers can be rearranged to correspond to the ordering of the nodes on level $h-1$, after which the parent pointer values can be written via a linear scan. The resulting I/O cost is $O\big(N_h + \mathit{Sort}(bN_h) + \mathit{Scan}(bN_h)\big)$, which can be amortized against the $\Theta(N_h)$ updates that have occurred since the last time the level-h invariant was violated, yielding an amortized update cost of $O\big(1 + (b/B)\log_m n\big)$ I/Os per level.

Order queries such as "Does leaf x precede leaf y in the total order represented by the tree?" can be answered using $O(\log_B N)$

I/Os by following parent pointers starting at x and y. The update operations insert, delete, cut, and concatenate can be done in $O\big(\big(1 + (b/B)\log_m n\big)\log_b N\big)$ I/Os amortized, for any $2 \le b \le B/2$, which is never worse than $O\big((\log_B N)^2\big)$ by appropriate choice of b.

Using the multislab decomposition we discuss in Section 12.3, Agarwal et al. [6] apply level-balanced B-trees in a data structure for point location in monotone subdivisions, which supports queries and (amortized) updates in $O\big((\log_B N)^2\big)$ I/Os. They also use it to dynamically maintain planar st-graphs using $O\big(1 + (b/B)(\log_m n)\log_b N\big)$ I/Os (amortized) per update, so that reachability queries can be answered in $O(\log_B N)$ I/Os (worst-case). (Planar st-graphs are planar directed acyclic graphs with a single source and a single sink.) An interesting open question is whether level-balanced B-trees can be implemented in $O(\log_B N)$ I/Os per update. Such an improvement would immediately give an optimal dynamic structure for reachability queries in planar st-graphs.

11.4 Buffer Trees

An important paradigm for constructing algorithms for batched problems in an internal memory setting is to use a dynamic data structure to process a sequence of updates. For example, we can sort N items by inserting them one by one into a priority queue, followed by a sequence of N *delete_min* operations. Similarly, many batched problems in computational geometry can be solved by dynamic plane sweep techniques. In Section 8.1, we showed how to compute orthogonal segment intersections by dynamically keeping track of the active vertical segments (i.e., those hit by the horizontal sweep line); we mentioned a similar algorithm for orthogonal rectangle intersections.

However, if we use this paradigm naively in an EM setting, with a B-tree as the dynamic data structure, the resulting I/O performance will be highly nonoptimal. For example, if we use a B-tree as the priority queue in sorting or to store the active vertical segments hit by the sweep line, each update and query operation will take $O(\log_B N)$ I/Os, resulting in a total of $O(N \log_B N)$ I/Os, which is larger than the optimal $Sort(N)$ bound (5.1) by a substantial factor of roughly B.

One solution suggested in [339] is to use a binary tree data structure in which items are pushed lazily down the tree in blocks of B items at a time. The binary nature of the tree results in a data structure of height $O(\log n)$, yielding a total I/O bound of $O(n \log n)$, which is still nonoptimal by a significant $\log m$ factor.

Arge [32] developed the elegant *buffer tree* data structure to support *batched dynamic* operations, as in the sweep line example, where the queries do not have to be answered right away or in any particular order. The buffer tree is a balanced multiway tree, but with degree $\Theta(m)$ rather than degree $\Theta(B)$, except possibly for the root. Its key distinguishing feature is that each node has a buffer that can store $\Theta(M)$ items (i.e., $\Theta(m)$ blocks of items). Items in a node are pushed down to the children when the buffer fills. Emptying a full buffer requires $\Theta(m)$ I/Os, which amortizes the cost of distributing the M items to the $\Theta(m)$ children. Each item thus incurs an amortized cost of $O(m/M) = O(1/B)$ I/Os per level, and the resulting cost for queries and updates is $O\big((1/B)\log_m n\big)$ I/Os amortized.

Buffer trees have an ever-expanding list of applications. They can be used as a subroutine in the standard sweep line algorithm in order to get an optimal EM algorithm for orthogonal segment intersection. Arge showed how to extend buffer trees to implement segment trees [82] in external memory in a batched dynamic setting by reducing the node degrees to $\Theta(\sqrt{m})$ and by introducing *multislabs* in each node, which were explained in Section 8.1 for the related batched problem of intersecting rectangles.

Buffer trees provide a natural amortized implementation of priority queues for *time-forward processing* applications such as discrete event simulation, sweeping, and list ranking [105]. Govindarajan et al. [182] use time-forward processing to construct a well-separated pair decomposition of N points in d dimensions in $O\big(Sort(N)\big)$ I/Os, and they apply it to the problems of finding the K nearest neighbors for each point and the K closest pairs. Brodal and Katajainen [92] provide a worst-case optimal priority queue, in the sense that every sequence of B *insert* and *delete_min* operations requires only $O(\log_m n)$ I/Os. Practical implementations of priority queues based upon these ideas are examined in [90, 302]. Brodal and Fagerberg [91] examine I/O tradeoffs

between update and search for comparison-based EM dictionaries. Matching upper bounds for several cases can be achieved with a truncated version of the buffer tree.

In Section 12.2, we report on some timing experiments involving buffer trees for use in bulk loading of R-trees. Further experiments on buffer trees appear in [200].