10

B-Trees

Figure 10.1. Sequoia trees—*Sequoia sempervirens*—in Sequoia National Park, with average lifespans of 1800 years or more

Searching data that is too large to store in main memory introduces a number of complications. As with sorting, we focus on seeks during disk-based searching. For very large n, even a search that runs in $O(\lg n)$ seeks will be too slow. We're looking for something that runs closer to constant $O(1)$ time.

Given these needs, we specify two requirements for a disk-based search.

1. Searching must be faster than $O(\lg n)$ binary search.
2. Collections are dynamic, so insertion and deletion must be as fast as searching.

10.1 Improved Binary Search

One obvious approach is to try to improve binary search to meet our requirements. For example, data is often stored in a binary search tree (BST). As noted above, however, a BST's best-case search, insertion, and deletion performance is $O(\lg n)$. Moreover, as data is added it is not uncommon for the BST to become unbalanced, decreasing performance towards the worst-case $O(n)$ case.

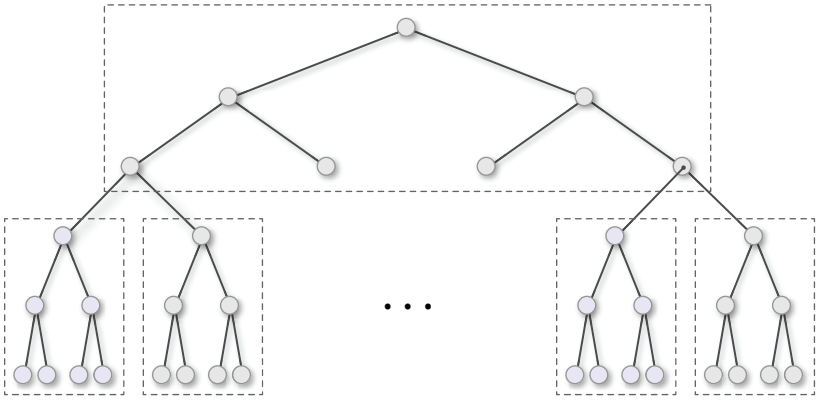

Figure 10.2. A paged BST with seven nodes per page and nine pages

10.1.1 Self-Correcting BSTs

Numerous improvements on BST exist. For example, AVL trees, red-black trees, and splay trees use rotations to guarantee a certain amount of balance after each insertion or deletion. Worst-case searching on an AVL tree is $1.44 \lg(n+2)$ versus $\lg n$ for a perfectly balanced tree. A single reorganization to re-balance the tree requires no more than five parent–child reassignments, and these reorganizations are applied for approximately every other insertion, and every fourth deletion.

Unfortunately, all of this effort simply guarantees $O(\lg n)$ performance. Although advantageous, for disk-based search an AVL tree's height of $O(\lg n)$ is still "too deep" for our needs.

10.1.2 Paged BSTs

One problem with BSTs is that they require too many seeks to find a record. For disk-based searching, each seek can efficiently retrieve a sector or a cylinder of data, but most of that data is being ignored. Suppose we could reorganize a BST to store a set of locally neighbouring nodes on a common disk page.

In Fig. 10.2 we are storing $k = 7$ nodes per page. A fully balanced tree holds $n = 63$ nodes, and any node can be found in no more than 2 seeks. Suppose we append another level to the paged BST, creating 64 new disk pages. We can now store $n = 511$ nodes in the tree and find any node in no more than 3 seeks. Appending another level allows us to search $n = 4095$ nodes in no more than 4 seeks. Compare this to a fully balanced BST, which would need up to $\lg(4095) \approx 12$ seeks to search.

Storing $k = 7$ nodes per page was chosen to make the example simple. Real disk pages are much larger, which will further improve performance. Suppose we used disk pages of 8KB, capable of holding 511 key–reference pairs that make up a file index containing $n = 100$ million records. To find any key–reference pair, we need

$$\log_{k+1}(n+1) = \log_{512}(100000001) = 2.95 \text{ seeks} \tag{10.1}$$

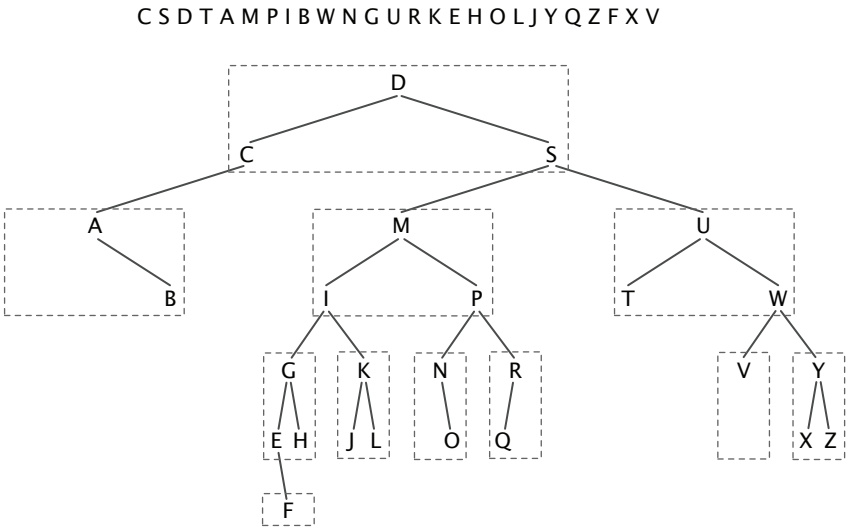

Figure 10.3. A paged BST with three nodes per page, constructed from letters of the alphabet in a random order

So, with a page holding 511 nodes, we can find any record in a 100 million record index file in at most 3 seeks. This is exactly the type of search performance we're looking for.

Space Efficiency. One problem with paged BSTs is that there is still significant wasted space within each page. For example, in our $k = 7$ node example, each page has 14 references, but 6 of them point to data inside the page. This suggests we could represent the same data with only 7 keys and 8 references.

One possibility is to hold the data in a page in an array rather than a BST. We would need to perform an in-memory linear search to find records within each page, but this is relatively fast. This change would allow us to store more elements per page, potentially reducing the total number of seeks needed to locate records and therefore offering a time savings much larger than the cost of the in-memory search.

Construction. A much more significant problem with paged BSTs is guaranteeing they are relatively balanced, both when they are initially constructed, and as insertions and deletions are performed. For example, supposed we insert the following data into a paged BST that holds $k = 3$ keys in each page.

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

Fig. 10.3 shows the paged BST tree that results. To try to maintain reasonable balance, we performed AVL rotations within a page during insertion to reorganize our keys. This is why the keys C, S, and D produced a page with D at the root and C and S as its left and right child nodes.

Even with page-based rotations the tree is not balanced. The problem is that the tree is built top-down. This means the first few keys **must** go at or near the root of the tree, and they can't be moved. The root elements partition the remainder of the tree, so if those elements aren't near the center of the set of keys we're storing, they produce an unbalanced tree.

In our example, D, C, and S are stored in the root page. These keys are located near the left and right ends of the key list, so they do a poor job of partitioning the list, and therefore produce unbalanced regions in the tree. In particular, D pushes three keys to the left, and 22 keys to the right. Rotating within a page can't correct this type of problem. That would require rotating keys *between* pages. Unfortunately, trying to keep individual pages balanced both internally and well placed relative to one another is a much more difficult problem.

In summary, grouping keys in individual disk pages is very good for improved search performance, but it introduces two important problems.

1. How can we ensure keys at the root of the tree are good partitioning keys for the current collection?
2. How can we avoid grouping keys like D, C, and S that shouldn't share a common page?

Any solution to our problem should also satisfy a third condition, to ensure each page's space is used efficiently.

3. How can we guarantee each page contains a minimum number of nodes?

10.2 B-Tree

B-trees were proposed by Rudolf Bayer and Ed McCreight of the Boeing Corporation in 1972[1]. Significant research followed in the 1970s to improve upon the initial B-tree algorithms. In the 1980s B-trees were applied to database management systems, with an emphasis on how to ensure consistent B-tree access patterns to support concurrency control and data recovery. More recently, B-trees have been applied to disk management, to support efficient I/O and file versioning. For example, ZFS is built on top of an I/O efficient B-tree implementation.

The original B-tree algorithm was designed to support efficient *access and maintenance* of an index that is too large to hold in main memory. This led to three basic goals for a B-tree.

1. Increase the tree's node size to minimize seeks, and maximize the amount of data transferred on each read.
2. Reduce search times to a very few seeks, even for large collections.
3. Support efficient local insertion, search, and deletion.

[1] Organization and maintenance of large ordered indexes. Bayer and McCreight. *Acta Informatica 1*, 3, 173–189, 1972.

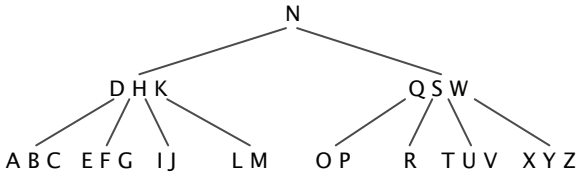

Figure 10.4. An order-4 B-tree with 3 keys per node, constructed from letters of the alphabet in a random order

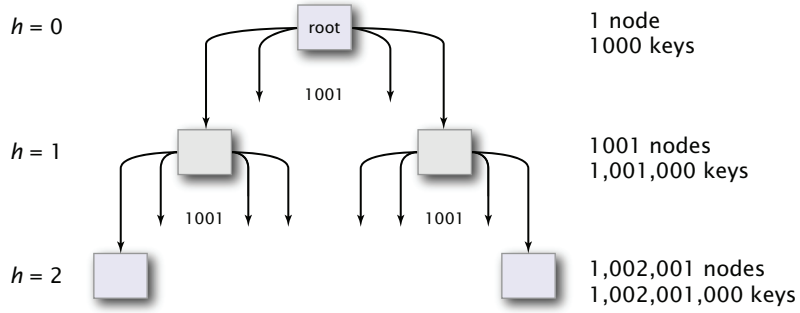

Figure 10.5. An order-1001 B-tree with 1000 keys per node, three levels yields enough space for about 1.1 billion keys

The key insight of B-trees is that the tree should be built bottom-up, and not top-down. We begin by inserting keys into a single leaf node. When this leaf node overflows, we split it into two half-full leaves and promote a single key upwards to form a new root node. Critically, since we defer the promotion until the leaf overflows, we can pick the key that does the best job of partitioning the leaf. This split–promote operation continues throughout the life of the B-tree.

A B-tree is a generalization of a BST. Rather than holding 1 key and pointers to two subtrees at each node, we hold up to $k - 1$ keys and k subtree references. This is called an order-k B-tree. Using this terminology, a BST is an order-2 B-tree. Fig. 10.4 shows an order-4 B-tree used to store the same collection of keys we inserted into the paged BST in Fig. 10.3.

Although our examples have low order, a B-tree node will normally hold hundreds or even thousands of keys per node, with each node sized to fill one or more disk pages. Fig. 10.5 shows the number of nodes and keys in the first three levels of an order-1001 B-tree. Even with a very small height of 3, the tree holds more than a billion keys. We need at most 3 seeks to find any key in the tree, producing exactly the type of size : seek ratio we are looking for.

The B-tree algorithm guarantees a number of important properties on order-k B-trees, to ensure efficient search, management, and space utilization.

1. Every tree node can hold up to $k - 1$ keys and k subtree references.

2. Every tree node except the root holds at least $\lceil {}^{k}/_{2} - 1 \rceil$ keys.
3. All leaf nodes occur at the same depth in the tree.
4. The keys in a node are stored in ascending order.

10.2.1 Search

Searching a B-tree is similar to searching a BST, however, rather than moving recursively left or right at each node, we need to perform a k-way search to see which subtree to probe.

```
search_btree(tg, node, k)
Input: tg, target key, node, tree node to search, k, tree's order
s = 0                                        // Subtree to recursively search
while s < k-1 do
    if tg == node.key[ s ] then
        return node.ref[ s ]                // Target found, return reference
    else if tg < node.key[ s ] then
        break                                                // Subtree found
    else
        s++
    end
end
if node.subtree[ s ] ≠ null then
    return search_btree( tg, node.subtree[ s ], k )
else
    return -1                                        // No target key in tree
end
```

The `btree_search` function walks through the keys in a node, looking for a target key. If it finds it, it returns a reference value attached to the key (*e.g.*, the position of the key's full record in the data file). As soon as we determine the target key is not in the node, we recursively search the appropriate subtree. This continues until the target key is found, or the subtree to search doesn't exist. This tells us the target key is not contained in the tree.

To initiate a search, we call `btree_search( tg, root, k )` to start searching at the root of the tree. Searching in this way takes $O(\log_k n)$ time.

10.2.2 Insertion

Inserting into a B-tree is more complicated than inserting into a BST. Both insertion algorithms search for the proper leaf node to hold the new key. B-tree insertion must also promote a median key if the leaf overflows.

1. Search to determine which leaf node will hold the new key.
2. If the leaf node has space available, insert the key in ascending order, then stop.
3. Otherwise, split the leaf node's keys into two parts, and promote the median key to the parent.
4. If the parent node is full, recursively split and promote to its parent.

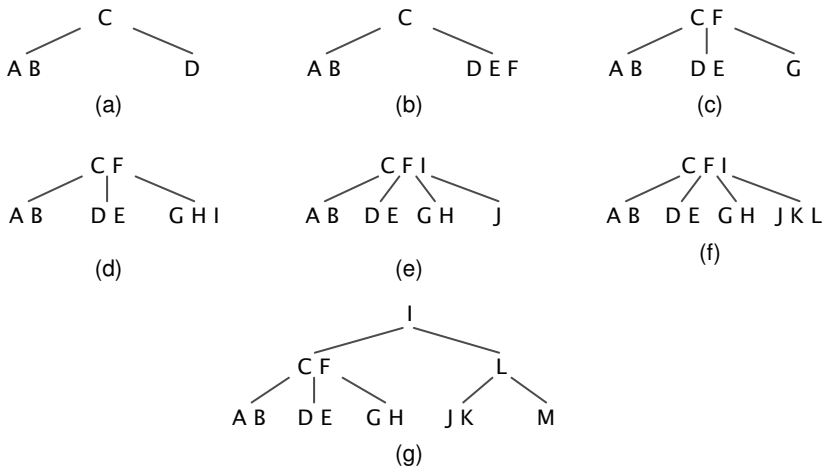

Figure 10.6. Inserting A B C D E F G H I J K L M into an order-4 B-tree: (a–f) insert, split and promote; (g) the final tree

5. If a promotion is made to a full root node, split and create a new root node holding only the promoted median key.

Consider inserting A B C D E F G H I J K L M into an order-4 B-tree.

- A, B, and C are inserted into the root node,
- D splits the root into two parts, promoting C to a new root node (Fig. 10.6a),
- E and F are inserted into a leaf node (Fig. 10.6b),
- G splits the leaf, promoting F to the root (Fig. 10.6c),
- H and I are inserted into a leaf node (Fig. 10.6d),
- J splits the leaf, promoting I to the root (Fig. 10.6e),
- K and L are inserted into a leaf node (Fig. 10.6f), and
- M splits the leaf, promoting L to the root. L splits the root, promoting I to a new root node (Fig. 10.6g).

Notice that at each step of the insertion, the B-tree satisfies the four required properties: nodes hold up to $k - 1 = 3$ keys, internal nodes hold at least $\lceil {}^{3}/_{2} - 1 \rceil = 1$ key, all leaf nodes are at the same depth in the tree, and keys in a node are stored in ascending order.

Each insertion walks down the tree to find a target leaf node, then may walk back up the tree to split and promote. This requires $O(\log_k n)$ time, which is identical to search performance.

10.2.3 Deletion

Deletion in a B-tree works similar to deletion in a BST. If we're removing from a leaf node, we can delete the key directly. If we're removing from an internal node,

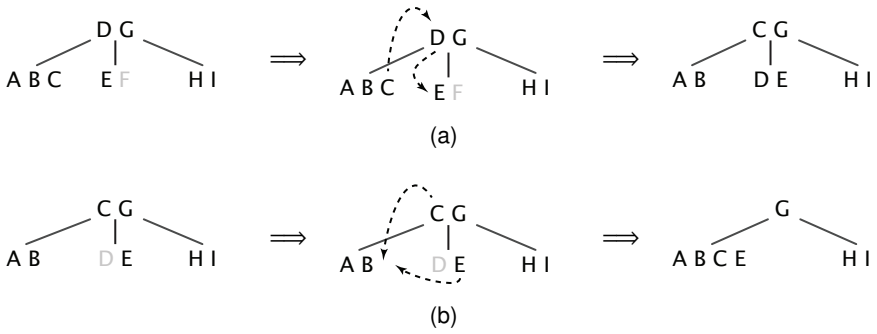

Figure 10.7. Deleting from an order-5 B-tree: (a) removing F produces an underfull leaf that borrows a key from its left sibling; (b) removing D produces an underfull leaf that coalesces with its left sibling and parent to form a full leaf

we delete the key and promote its predecessor or its successor—the largest key in the internal node's left subtree or the smallest key in its right subtree, respectively.

In all cases, either deleting directly from a leaf or promoting a predecessor or successor, a key is being removed from a leaf node, leaving l keys in the leaf. If $l \geq \lceil k/2 - 1 \rceil$ then we can stop. If $l < \lceil k/2 - 1 \rceil$, however, the leaf is underfull and we must rebalance the tree to correct this.

Borrow. We first check the leaf's left sibling, then its right sibling (if they exist) to see whether it has more than $\lceil k/2 - 1 \rceil$ keys. If it does, we can borrow one of the sibling's keys without making it underfull.

If we're borrowing from the left sibling, we do the following:

1. Take the key in the parent node that splits the left sibling and the leaf, the *split key*, and insert it into the leaf.
2. Replace the split key with the left sibling's largest key.

In Fig. 10.7a we delete F from an order-5 B-tree. Since each node must contain at least $\lceil 5/2 - 1 \rceil = 2$ keys, this causes the leaf to underflow. The leaf's left sibling contains 3 keys, so we can borrow one of its keys, placing it in the parent and pushing the parent's split key down to the leaf. Borrowing from the right sibling is identical, except that we replace the split key with the right sibling's *smallest* key.

Coalesce. If both siblings have only $\lceil k/2 - 1 \rceil$ keys, we coalesce the leaf with its left sibling and its parent's split key.

1. Combine the left sibling's keys, the split key, and the leaf's keys. This produces a node with $k - 1$ keys.
2. Remove the parent's split key and its now empty right subtree.

In Fig. 10.7b we delete D from an order-5 B-tree. Since each node must contain at least 2 keys, this causes the leaf to underflow. Neither sibling has more than 2 keys, so we coalesce the left sibling, the parent's split key, and the leaf to form a full leaf

node with 4 keys. The parent node is updated to remove its split key and the empty right subtree.

After coalescing, two checks must be made. First, if the parent node was the tree's root and if it is now empty, we make the coalesced node the new root node. Second, if the parent node is an internal node and if it now has fewer than $\lceil {}^{k}/_{2} - 1 \rceil$ keys, it must be recursively rebalanced using the same borrow–coalesce strategy.

10.3 B* Tree

A B* tree is a B-tree with a modified insertion algorithm[2]. B* trees improve the storage utilization of internal nodes from approximately ${}^{k}/_{2}$ to approximately ${}^{2k}/_{3}$. This can postpone increasing the height of the tree, resulting in improve search performance.

The basic idea of a B* tree is to postpone splitting a node by trying to redistribute records when it overflows to the left or right siblings. If a redistribution is not possible, two full nodes are split into three nodes that are about ${}^{2}/_{3}$ full. Compare this to a B-tree, which splits one full node into two nodes that are about ${}^{1}/_{2}$ full.

If a leaf node with $l = k$ keys overflows, and a sibling—say the left sibling—has $m < k - 1$ keys, we combine the sibling's keys, the split key, and the leaf node's keys. This produces a list of $l + m + 1$ keys that we redistribute as follows.

1. Leave the first $\lfloor {}^{l+m}/_{2} \rfloor$ keys in the left sibling.
2. Replace the split key with the key at position $\lfloor {}^{l+m}/_{2} \rfloor + 1$.
3. Store the last $\lceil {}^{l+m}/_{2} \rceil$ keys in the leaf.

Redistribution with the right sibling is identical, except that the first $\lfloor {}^{l+m}/_{2} \rfloor$ keys go in the leaf, and the last $\lceil {}^{l+m}/_{2} \rceil$ keys go in the right sibling.

If both siblings are themselves full, we choose one—say the left sibling—and combine the sibling's keys, the split key, and the leaf node's keys. This produces a list of $2k$ keys that we redistribute into three new nodes as follows.

1. Place $\lfloor {}^{2k-2}/_{3} \rfloor$ keys in the left node.
2. Use the next key as a split key.
3. Place the next $\lfloor {}^{2k-1}/_{3} \rfloor$ keys in the middle node.
4. Use the next key as a split key.
5. Place the remaining $\lfloor {}^{2k}/_{3} \rfloor$ keys in the right node.

If adding two split keys to the parent causes *it* to overflow, the same redistribute–split strategy is used to recursively rebalance the tree. Fig. 10.8 shows an example of inserting L A N M D F G H C E B I J K into an order-5 B* tree.

- Fig. 10.8a, an overfull root splits into a root and two leaf nodes,

[2] *The Art of Computer Programming, 2nd Edition. Volume 3, Sorting and Searching.* Knuth. Addison-Wesley, Reading, MA, 1998, 487–488.

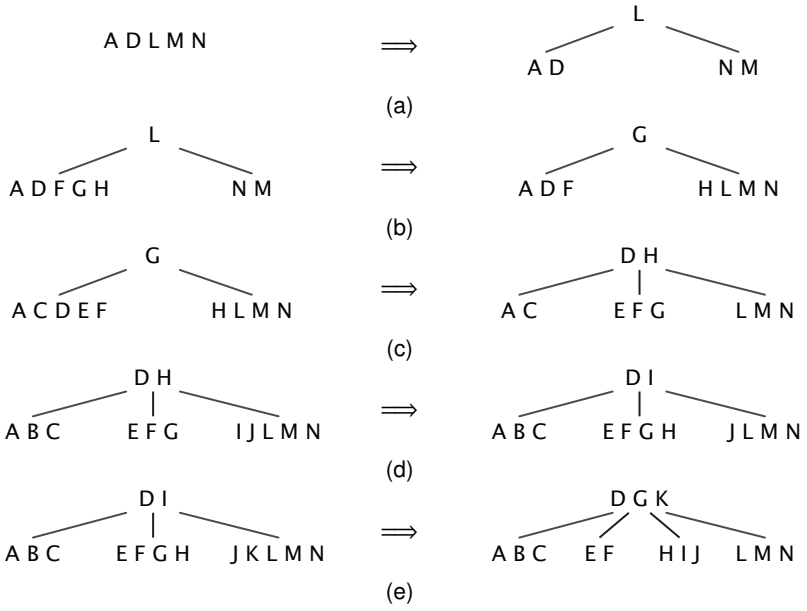

Figure 10.8. Inserting L A N M D F G H C E B I J K into an order-5 B* tree: (a) root node overflows and splits; (b) leaf redistributes to right sibling; (c) two full nodes split into three; (d) leaf redistributes to left sibling; (e) two full nodes split into three

- Fig. 10.8b, an overfull leaf with $l = 5$ keys redistributes to its right sibling with $m = 2$ keys, given $l + m + 1 = 8$ keys A D F G H L N M, we place $\lfloor {}^{l+m}/_{2} \rfloor = 3$ keys in the leaf, the key at position $\lfloor {}^{l+m}/_{2} \rfloor + 1 = 4$ in the parent, and $\lceil {}^{l+m}/_{2} \rceil = 4$ keys in the right sibling,

- Fig. 10.8c, an overfull leaf and its full right sibling split into three nodes, given $2k = 10$ keys A C D E F G H L M N, we place $\lfloor {}^{2k-2}/_{3} \rfloor = 2$ keys in the left sibling, the key at position 3 in the parent, $\lfloor {}^{2k-1}/_{3} \rfloor = 3$ keys in the leaf, the key at position 7 in the root, and $\lfloor {}^{2k}/_{3} \rfloor = 3$ keys in the right sibling,

- Fig. 10.8d, an overfull leaf redistributes to its left sibling, and

- Fig. 10.8e, an overfull leaf and its full left sibling split into three nodes.

Search and Deletion. Searching a B* tree is identical to searching a B-tree. Knuth provided no specific details on how to delete from a B* tree. The problem is how to perform deletions while maintaining the constraint that internal nodes stay at least ${}^{2}/_{3}$ full. In the absence of a deletion strategy for B* trees, we perform B-tree deletion. This means the ${}^{2}/_{3}$ full requirement may be violated after deletion, but should return following a sufficient number of additional insertions.

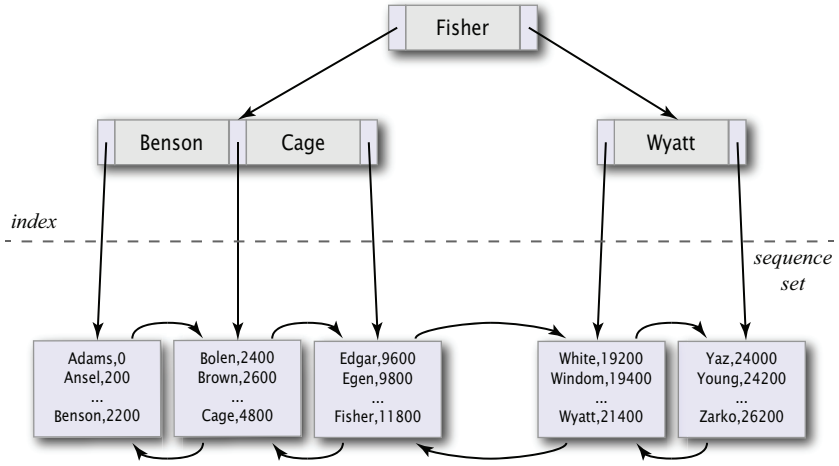

Figure 10.9. An order-3 B+ tree with last name acting as a key

10.4 B+ Tree

A B+ tree is a combination of a sequential, in-order set of key–value pairs, together with an index placed on top of the sequence set. The sequence set is a collection of blocks—usually disk pages—bound together as a doubly-linked list. These blocks form leaf nodes for a B-tree index of keys that allows us to rapidly identify the block that holds a target key (Fig. 10.9).

Placing all the data in the leaves of the index provides B+ trees a number of advantages over B-trees.

- the sequence set's blocks are linked, so scanning a range of key values requires a search for the first key and a single linear pass, rather than multiple queries on a B-tree index, and
- internal B+ tree nodes hold only keys and pointers—versus key–value pairs in a regular B-tree—so more keys can fit in each node, possibly leading to shorter trees.

The one disadvantage of B+ trees is that if the target key is found at an internal node, we must still traverse all the way to the bottom of the tree to find the leaf that holds the key's value. With a B-tree this value could be returned immediately. Since the tree's height is, by design, compact, this is normally a small penalty that we're willing to accept.

Search, insertion, and deletion into a B+ tree works identically to a B-tree, but with the understanding that all operations occur in the leaf nodes, where the key–value pairs are stored. For example, to insert a new key–value pair into a B+ tree, we would do the following.

1. Search the B+ tree to locate the block to hold the new key.
2. If space is available in the block, store the key–value pair and stop.

3. If the block is full, append a new block to the end of the file, keep the first $^{k}/_{2}$ keys in the existing block, and move the remaining $^{k}/_{2}$ keys to the new block.
4. Update the B+ tree's index (using the normal B-tree insertion algorithm) to link the new block into the index.

10.4.1 Prefix Keys

The B+ tree's internal nodes don't contain answers to search requests, they simply direct the search to a block that potentially holds a target key. This means we only need to store as much of each key as we need to properly separate the blocks.

In Fig. 10.9, two keys span the gap between the first and second blocks: Benson and Bolen. The parent node uses Benson to define that the first block contains all keys ≤ Benson. We don't need the entire key to separate the blocks, however. It would be sufficient use Bf, since that would correctly define the first block to contain keys ≤ Bf, and the second block to contain keys > Bf. Similarly, the key values Cage, Fisher, and Wyatt in the internal nodes could be changed to D, G, and Y, respectively.

In general, to differentiate block A with largest key k_A and block B with smallest key k_B, it is sufficient to select any separator key k_S such that

$$k_A \leq k_s < k_B \tag{10.2}$$

In our example with k_A = Benson and k_B = Bolen, k_S = Benson satisfies this requirement, but so too does the shorter k_S = Bf. If k_S has the following properties

1. k_S is a separator between k_A and k_B.
2. No other separator k'_S is shorter than k_S.

then k_S satisfies the *prefix property.* B+ trees constructed with prefix keys are known as simple prefix B+ trees[3].

By choosing the smallest k_S, we can increase the number of keys we can store in each internal node, potentially producing flatter trees with better search performance. This improvement does not come for free, however. Internal nodes must now manage variable length records, which means the number of entries in a fixed-sized node will vary. Modifications must be made to the search, insertion and deletion algorithms to support this.

Performance. Performance results vary depending on the type of data being stored. Experimental results suggest that, for trees containing between 400 and 800 pages, simple prefix B+ trees require 20-25% fewer disk accesses than B+ trees.

A more efficient method that removes redundant prefix information along any given path in the tree produces a small improvement over simple prefix B+ trees (≈2%), but demands 50-100% more time to construct the prefixes. This suggests that the overhead of generating more complex prefixes outweighs any savings from reduced tree height.

[3]Prefix B-trees. Bayer and Unterauer. *ACM Transactions on Database Systems 2*, 1, 11–26, 1977.