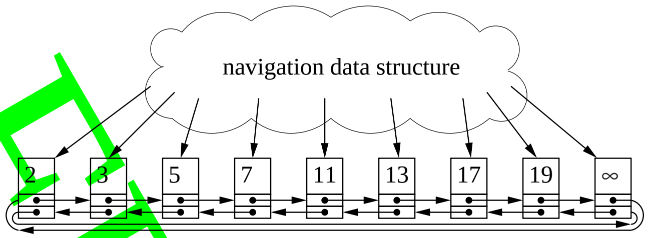


Fig. 7.1. A sorted sequence as a doubly linked list plus a navigation data structure

one for each element and one additional “dummy item”. We use the dummy item to store a special key value $+\infty$ which is larger than all conceivable keys. We can then define the result of $locate(k)$ as the handle to the smallest list item $e \geq k$. If k is larger than all keys in M, *locate* will return a handle to the dummy item. In Sect. 3.1.1, we learned that doubly linked lists support a large set of operations; most of them can also be implemented efficiently for sorted sequences. For example, we “inherit” constant-time implementations for *first*, *last*, *succ*, and *pred*. We shall see constant-amortized-time implementations for $remove(h : Handle)$, *insertBefore*, and *insertAfter*, and logarithmic-time algorithms for concatenating and splitting sorted sequences. The indexing operator $[\cdot]$ and finding the position of an element in the sequence also take logarithmic time. Before we delve into a description of the navigation data structure, let us look at some concrete applications of sorted sequences.

Best-first heuristics. Assume that we want to pack some items into a set of bins. The items arrive one at a time and have to be put into a bin immediately. Each item i has a weight $w(i)$, and each bin has a maximum capacity. The goal is to minimize the number of bins used. One successful heuristic solution to this problem is to put item i into the bin that fits best, i.e., the bin whose remaining capacity is the smallest among all bins that have a residual capacity at least as large as $w(i)$ [41]. To implement this algorithm, we can keep the bins in a sequence q sorted by their residual capacity. To place an item, we call $q.locate(w(i))$, remove the bin that we have found, reduce its residual capacity by $w(i)$, and reinsert it into q. See also Exercise 12.8.

Sweep-line algorithms. Assume that you have a set of horizontal and vertical line segments in the plane and want to find all points where two segments intersect. A sweep-line algorithm moves a vertical line over the plane from left to right and maintains the set of horizontal lines that intersect the sweep line in a sorted sequence q. When the left endpoint of a horizontal segment is reached, it is inserted into q, and when its right endpoint is reached, it is removed from q. When a vertical line segment is reached at a position x that spans the vertical range $[y, y']$, we call $s.locate(y)$ and scan q until we reach the key y'.[2] All horizontal line segments discovered during this scan define an intersection. The sweeping algorithm can be generalized to arbitrary line segments [21], curved objects, and many other geometric problems [46].

[2] This *range query* operation is also discussed in Sect. 7.3.

Database indexes. A key problem in databases is to make large collections of data efficiently searchable. A variant of the (a,b)-tree data structure described in Sect. 7.2 is one of the most important data structures used for databases.

The most popular navigation data structure is that of *search trees*. We shall frequently use the name of the navigation data structure to refer to the entire sorted sequence data structure.[3] We shall introduce search tree algorithms in three steps. As a warm-up, Sect. 7.1 introduces (unbalanced) *binary search trees* that support *locate* in $\mathrm{O}(\log n)$ time under certain favorable circumstances. Since binary search trees are somewhat difficult to maintain under insertions and removals, we then switch to a generalization, (a,b)-trees that allows search tree nodes of larger degree. Section 7.2 explains how (a,b)-trees can be used to implement all three basic operations in logarithmic worst-case time. In Sects. 7.3 and 7.5, we shall augment search trees with additional mechanisms that support further operations. Section 7.4 takes a closer look at the (amortized) cost of update operations.

7.1 Binary Search Trees

Navigating a search tree is a bit like asking your way around in a foreign city. You ask a question, follow the advice given, ask again, follow the advice again, ..., until you reach your destination.

A *binary search tree* is a tree whose leaves store the elements of a sorted sequence in sorted order from left to right. In order to locate a key k, we start at the root of the tree and follow the unique path to the appropriate leaf. How do we identify the correct path? To this end, the interior nodes of a search tree store keys that guide the search; we call these keys *splitter* keys. Every nonleaf node in a binary search tree with $n \geq 2$ leaves has exactly two children, a *left* child and a *right* child. The splitter key s associated with a node has the property that all keys k stored in the left subtree satisfy $k \leq s$ and all keys k stored in the right subtree satisfy $k > s$.

With these definitions in place, it is clear how to identify the correct path when locating k. Let s be the splitter key of the current node. If $k \leq s$, go left. Otherwise, go right. Figure 7.2 gives an example. Recall that the height of a tree is the length of its longest root–leaf path. The height therefore tells us the maximum number of search steps needed to *locate* a leaf.

Exercise 7.1. Prove that a binary search tree with $n \geq 2$ leaves can be arranged such that it has height $\lceil \log n \rceil$.

A search tree with height $\lceil \log n \rceil$ is called *perfectly balanced*. The resulting logarithmic search time is a dramatic improvement compared with the $\Omega(n)$ time needed for scanning a list. The bad news is that it is expensive to keep perfect balance when elements are inserted and removed. To understand this better, let us consider the "naive" insertion routine depicted in Fig. 7.3. We locate the key k of the new element e before its successor e', insert e into the list, and then introduce a new node v with

[3] There is also a variant of search trees where the elements are stored in all nodes of the tree.

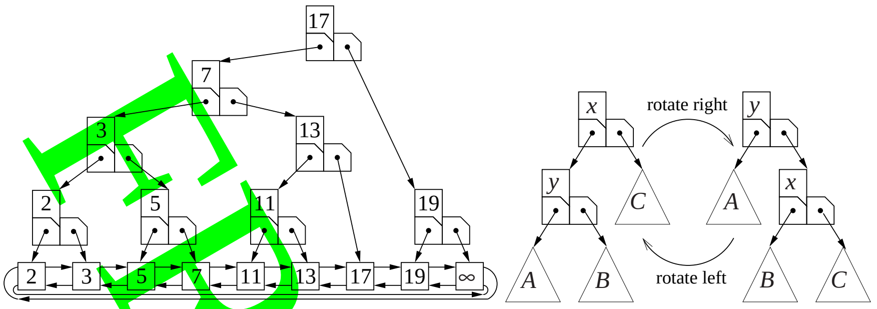


Fig. 7.2. *Left*: the sequence $\langle 2,3,5,7,11,13,17,19\rangle$ represented by a binary search tree. In each node, we show the splitter key at the top and the pointers to the children at the bottom. *Right*: rotation of a binary search tree. The triangles indicate subtrees. Observe that the ancestor relationship between nodes x and y is interchanged

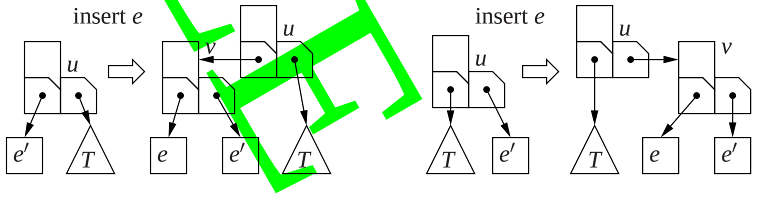


Fig. 7.3. Naive insertion into a binary search tree. A triangle indicates an entire subtree

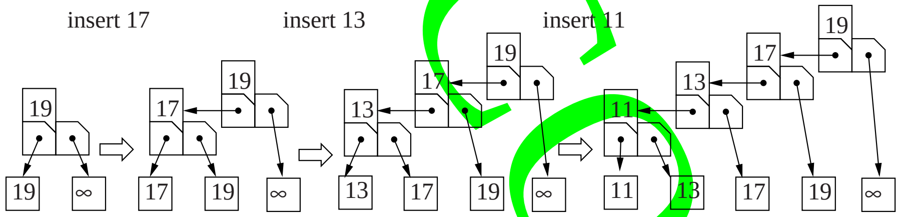


Fig. 7.4. Naively inserting sorted elements leads to a degenerate tree

left child e and right child e'. The old parent u of e' now points to v. In the worst case, every insertion operation will locate a leaf at the maximum depth so that the height of the tree increases every time. Figure 7.4 gives an example: the tree may degenerate to a list; we are back to scanning.

An easy solution to this problem is a healthy portion of optimism; perhaps it will not come to the worst. Indeed, if we insert n elements in *random* order, the expected height of the search tree is $\approx 2.99 \log n$ [51]. We shall not prove this here, but outline a connection to quicksort to make the result plausible. For example, consider how the tree in Fig. 7.2 can be built using naive insertion. We first insert 17; this splits the set into subsets $\{2,3,5,7,11,13\}$ and $\{19\}$. From the elements in the left subset, we first insert 7; this splits the left subset into $\{2,3,5\}$ and $\{11,13\}$. In quicksort terminology, we would say that 17 is chosen as the splitter in the top-level call and that 7 is chosen as the splitter in the left recursive call. So building a binary search tree and quicksort are completely analogous processes; the same comparisons are made, but at different times. Every element of the set is compared with 17. In quicksort, these comparisons take place when the set is split in the top-level call. In building a binary search tree, these comparisons take place when the elements of the set are inserted. So the comparison between 17 and 11 takes place either in the top-level call of quicksort or when 11 is inserted into the tree. We have seen (Theorem 5.6) that the expected number of comparisons in a randomized quicksort of n elements is $\mathrm{O}(n\log n)$. By the above correspondence, the expected number of comparisons in building a binary tree by random insertions is also $\mathrm{O}(n\log n)$. Thus any insertion requires $\mathrm{O}(\log n)$ comparisons on average. Even more is true; with high probability each single insertion requires $\mathrm{O}(\log n)$ comparisons, and the expected height is $\approx 2.99\log n$.

Can we guarantee that the height stays logarithmic in the worst case? Yes and there are many different ways to achieve logarithmic height. We shall survey these techniques in Sect. 7.7 and discuss two solutions in detail in Sect. 7.2. We shall first discuss a solution which allows nodes of varying degree, and then show how to balance binary trees using rotations.

Exercise 7.2. Figure 7.2 indicates how the shape of a binary tree can be changed by a transformation called *rotation*. Apply rotations to the tree in Fig. 7.2 so that the node labelled 11 becomes the root of the tree.

Exercise 7.3. Explain how to implement an *implicit* binary search tree, i.e., the tree is stored in an array using the same mapping of the tree structure to array positions as in the binary heaps discussed in Sect. 6.1. What are the advantages and disadvantages compared with a pointer-based implementation? Compare searching in an implicit binary tree with binary searching in a sorted array.

7.2 (a,b)-Trees and Red–Black Trees

An (a,b)-tree is a search tree where all interior nodes, except for the root, have an outdegree between a and b. Here, a and b are constants. The root has degree one for a trivial tree with a single leaf. Otherwise, the root has a degree between 2 and b. For $a \geq 2$ and $b \geq 2a-1$, the flexibility in node degrees allows us to efficiently maintain the invariant that *all leaves have the same depth*, as we shall see in a short while. Consider a node with outdegree d. With such a node, we associate an array $c[1..d]$ of pointers to children and a sorted array $s[1..d-1]$ of $d-1$ splitter keys. The splitters guide the search. To simplify the notation, we additionally define $s[0] = -\infty$ and $s[d] = \infty$. The keys of the elements e contained in the i-th child $c[i]$, $1 \leq i \leq d$, lie between the $i-1$-th splitter (exclusive) and the i-th splitter (inclusive), i.e., $s[i-1] < key(e) \leq s[i]$. Figure 7.5 shows a $(2,4)$-tree storing the sequence $\langle 2,3,5,7,11,13,17,19\rangle$.

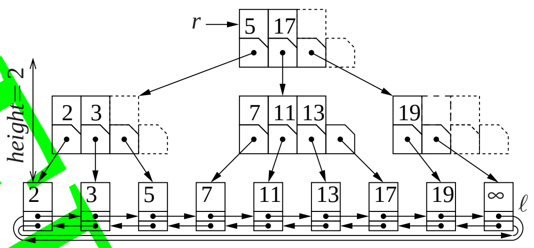
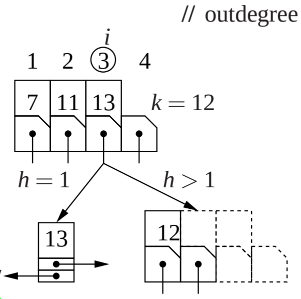


Fig. 7.5. Representation of $\langle 2,3,5,7,11,13,17,19\rangle$ by a $(2,4)$-tree. The tree has height 2

```
Class ABHandle : Pointer to ABItem or Item
// an ABItem (Item) is an item in the navigation data structure (doubly linked list)

Class ABItem(splitters : Sequence of Key, children : Sequence of ABHandle)
    $d = |children| : 1..b$                                   // outdegree
    $s$ = splitters  : Array $[1..b-1]$ of Key
    $c$ = children  : Array $[1..b]$ of Handle

    Function locateLocally($k$ : Key) : $\mathbb{N}$
        return $\min\{i \in 1..d : k \le s[i]\}$

    Function locateRec($k$ : Key, $h$ : $\mathbb{N}$) : Handle
        $i$:=locateLocally($k$)
        if $h = 1$ then return $c[i]$
        else return $c[i] \rightarrow$locateRec($k$, $h-1$)

Class ABTree($a \ge 2 : \mathbb{N}$, $b \ge 2a-1 : \mathbb{N}$) of Element
    $\ell = \langle\rangle$  : List of Element
    $r$ : ABItem($\langle\rangle$, $\langle \ell$.head$\rangle$)
    height = 1  : $\mathbb{N}$

    // Locate the smallest Item with key $k' \ge k$
    Function locate($k$ : Key) : Handle return $r$.locateRec($k$, height)
```



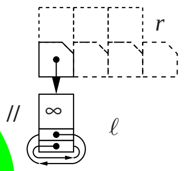


Fig. 7.6. (a,b)-trees. An *ABItem* is constructed from a sequence of keys and a sequence of handles to the children. The outdegree is the number of children. We allocate space for the maximum possible outdegree b. There are two functions local to *ABItem*: *locateLocally*(k) locates k among the splitters and *locateRec*(k,h) assumes that the *ABItem* has height h and descends h levels down the tree. The constructor for *ABTree* creates a tree for the empty sequence. The tree has a single leaf, the dummy element, and the root has degree one. Locating a key k in an (a,b)-tree is solved by calling r.*locateRec*(k,h), where r is the root and h is the height of the tree

Lemma 7.1. *An (a,b)-tree for n elements has a height at most*

$$1+\left\lfloor \log_a \frac{n+1}{2} \right\rfloor .$$

Proof. The tree has $n+1$ leaves, where the “+1” accounts for the dummy leaf $+\infty$. If $n=0$, the root has degree one and there is a single leaf. So, assume $n \geq 1$. Let h be the height of the tree. Since the root has degree at least two and every other node has degree at least a, the number of leaves is at least $2a^{h-1}$. So $n+1 \geq 2a^{h-1}$, or $h \leq 1+\log_a(n+1)/2$. Since the height is an integer, the bound follows. □

Exercise 7.4. Prove that the height of an (a,b)-tree for n elements is at least $\lceil \log_b(n+1) \rceil$. Prove that this bound and the bound given in Lemma 7.1 are tight.

Searching in an (a,b)-tree is only slightly more complicated than searching in a binary tree. Instead of performing a single comparison at a nonleaf node, we have to find the correct child among up to b choices. Using binary search, we need at most $\lceil \log b \rceil$ comparisons for each node on the search path. Figure 7.6 gives pseudocode for (a,b)-trees and the *locate* operation. Recall that we use the search tree as a way to locate items of a doubly linked list and that the dummy list item is considered to have key value ∞. This dummy item is the rightmost leaf in the search tree. Hence, there is no need to treat the special case of root degree 0, and the handle of the dummy item can serve as a return value when one is locating a key larger than all values in the sequence.

Exercise 7.5. Prove that the total number of comparisons in a search is bounded by $\lceil \log b \rceil (1+\log_a(n+1)/2)$. Assume $b \leq 2a$. Show that this number is $\mathrm{O}(\log b)+\mathrm{O}(\log n)$. What is the constant in front of the $\log n$ term?

To *insert* an element e, we first descend the tree recursively to find the smallest sequence element $e' \geq e$. If e and e' have equal keys, e' is replaced by e.

Otherwise, e is inserted into the sorted list ℓ before e'. If e' was the i-th child $c[i]$ of its parent node v, then e will become the new $c[i]$ and $key(e)$ becomes the corresponding splitter element $s[i]$. The old children $c[i..d]$ and their corresponding splitters $s[i..d-1]$ are shifted one position to the right. If d was less than b, d can be incremented and we are finished.

The difficult part is when a node v already has a degree $d=b$ and now would get a degree $b+1$. Let s' denote the splitters of this illegal node, c' its children, and

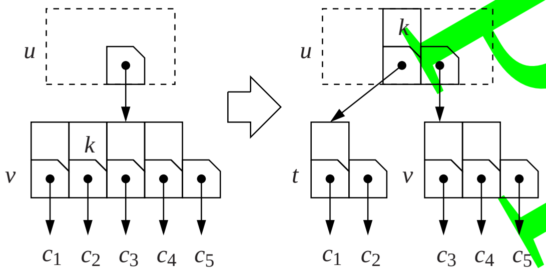
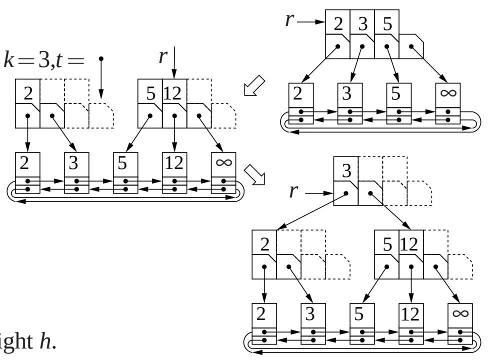
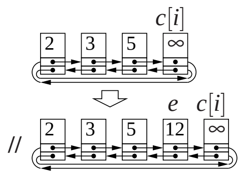
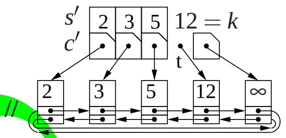


Fig. 7.7. Node splitting: the node v of degree $b+1$ (here 5) is split into a node of degree $\lfloor (b+1)/2 \rfloor$ and a node of degree $\lceil (b+1)/2 \rceil$. The degree of the parent increases by one. The splitter key separating the two “parts” of v is moved to the parent



```
// Example:
// $\langle 2,3,5\rangle.insert(12)$
**Procedure** $ABTree::insert(e : Element)$
    $(k,t) := r.insertRec(e, height, \ell)$
    **if** $t \neq$ **null then**    // root was split
        $r :=$ **allocate** $ABItem(\langle k\rangle, \langle r,t\rangle)$
        $height$++

// Insert a new element into a subtree of height $h$.
// If this splits the root of the subtree,
// return the new splitter and subtree handle
**Function** $ABItem::insertRec(e : Element, h : \mathbb{N}, \ell : List$ **of** $Element) : Key \times ABHandle$
    $i := locateLocally(e)$
    **if** $h = 1$ **then**    // base case
        **if** $key(c[i] \to e) = key(e)$ **then**
            $c[i] \to e := e$
            **return** $(\perp,$ **null** $)$
        **else**
            $(k,t) := (key(e), \ell.insertBefore(e, c[i]))$
    **else**
        $(k,t) := c[i] \to insertRec(e, h-1, \ell)$
        **if** $t =$ **null then return** $(\perp,$ **null** $)$
    **endif**

    $s' := \langle s[1], \ldots, s[i-1], k, s[i], \ldots, s[d-1]\rangle$
    $c' := \langle c[1], \ldots, c[i-1], t, c[i], \ldots, c[d]\rangle$

    **if** $d < b$ **then** // there is still room here
        $(s,c,d) := (s', c', d+1)$
        **return** $(\perp,$ **null** $)$
    **else** // **split** this node
        $d := \lfloor (b+1)/2 \rfloor$
        $s := s'[b+2-d..b]$
        $c := c'[b+2-d..b+1]$
        **return** $(s'[b+1-d],$ **allocate** $ABItem(s'[1..b-d], c'[1..b+1-d]))$
```





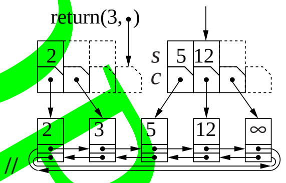


Fig. 7.8. Insertion into an (a,b)-tree

u the parent of v (if it exists). The solution is to *split* v in the middle (see Fig. 7.7). More precisely, we create a new node t to the left of v and reduce the degree of v to $d = \lceil (b+1)/2 \rceil$ by moving the $b+1-d$ leftmost child pointers $c'[1..b+1-d]$ and the corresponding keys $s'[1..b-d]$. The old node v keeps the d rightmost child pointers $c'[b+2-d..b+1]$ and the corresponding splitters $s'[b+2-d..b]$.

The “leftover” middle key $k = s'[b+1-d]$ is an upper bound for the keys reachable from t. It and the pointer to t are needed in the predecessor u of v. The situation for u is analogous to the situation for v before the insertion: if v was the i-th child of u, t displaces it to the right. Now t becomes the i-th child, and k is inserted as the i-th splitter. The addition of t as an additional child of u increases the degree of u. If the degree of u becomes $b+1$, we split u. The process continues until either some ancestor of v has room to accommodate the new child or the root is split.

In the latter case, we allocate a new root node pointing to the two fragments of the old root. This is the only situation where the height of the tree can increase. In this case, the depth of all leaves increases by one, i.e., we maintain the invariant that all leaves have the same depth. Since the height of the tree is $O(\log n)$ (see Lemma 7.1), we obtain a worst-case execution time of $O(\log n)$ for *insert*. Pseudocode is shown in Fig. 7.8.[4]

We still need to argue that *insert* leaves us with a correct (a,b)-tree. When we split a node of degree $b+1$, we create nodes of degree $d = \lceil (b+1)/2 \rceil$ and $b+1-d$. Both degrees are clearly at most b. Also, $b+1-\lceil (b+1)/2 \rceil \geq a$ if $b \geq 2a-1$. Convince yourself that $b = 2a-2$ will not work.

Exercise 7.6. It is tempting to streamline *insert* by calling *locate* to replace the initial descent of the tree. Why does this not work? Would it work if every node had a pointer to its parent?

We now turn to the operation *remove*. The approach is similar to what we already know from our study of *insert*. We locate the element to be removed, remove it from the sorted list, and repair possible violations of invariants on the way back up. Figure 7.9 shows pseudocode. When a parent u notices that the degree of its child $c[i]$ has dropped to $a-1$, it combines this child with one of its neighbors $c[i-1]$ or $c[i+1]$ to repair the invariant. There are two cases illustrated in Fig. 7.10. If the neighbor has degree larger than a, we can *balance* the degrees by transferring some nodes from the neighbor. If the neighbor has degree a, balancing cannot help since both nodes together have only $2a-1$ children, so that we cannot give a children to both of them. However, in this case we can *fuse* them into a single node, since the requirement $b \geq 2a-1$ ensures that the fused node has degree b at most.

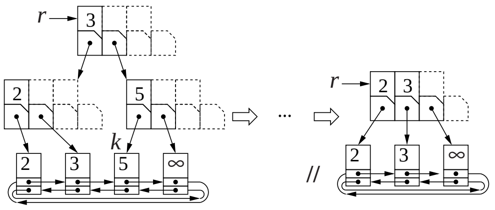
To fuse a node $c[i]$ with its right neighbor $c[i+1]$, we concatenate their child arrays. To obtain the corresponding splitters, we need to place the splitter $s[i]$ of the parent between the splitter arrays. The fused node replaces $c[i+1]$, $c[i]$ is deallocated, and $c[i]$, together with the splitter $s[i]$, is removed from the parent node.

Exercise 7.7. Suppose a node v has been produced by fusing two nodes as described above. Prove that the ordering invariant is maintained: an element e reachable through child $v.c[i]$ has key $v.s[i-1] < key(e) \leq v.s[i]$ for $1 \leq i \leq v.d$.

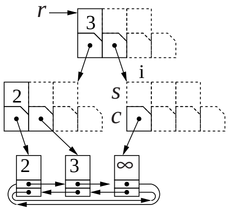
Balancing two neighbors is equivalent to first fusing them and then splitting the result, as in the operation *insert*. Since fusing two nodes decreases the degree of their

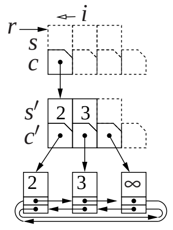
[4] We borrow the notation $C::m$ from C++ to define a method m for class C.

```
// Example: $\langle 2,3,5\rangle$.remove(5)
Procedure ABTree::remove($k$ : Key)    //
    $r$.removeRec($k$, height, $\ell$)
    if $r.d = 1 \wedge height > 1$ then
        $r' := r$; $r := r'.c[1]$; dispose $r'$          //
```



```
Procedure ABItem::removeRec($k$ : Key, $h$ : $\mathbb{N}$, $\ell$ : List of Element)
    $i := locateLocally(k)$
    if $h = 1$ then    // base case
        if $key(c[i] \to e) = k$ then    // there is sth to remove
            $\ell.remove(c[i])$
            $removeLocally(i)$                                    //
    else
        $c[i] \to removeRec(e, h-1, \ell)$
        if $c[i] \to d < a$ then                                  // invariant needs repair
            if $i = d$ then $i$--                  // make sure $i$ and $i+1$ are valid neighbors
            $s' := concatenate(c[i] \to s, \langle s[i]\rangle, c[i+1] \to s))$
            $c' := concatenate(c[i] \to c, c[i+1] \to c)$
            $d' := |c'|$
            if $d' \leq b$ then  // fuse
                $(c[i+1] \to s, c[i+1] \to c, c[i+1] \to d) := (s', c', d')$
                dispose $c[i]$;    $removeLocally(i)$                //
            else              // balance
                $m := \lceil d'/2 \rceil$
                $(c[i] \to s, c[i] \to c, c[i] \to d) := (s'[1..m-1], c'[1..m], m)$
                $(c[i+1] \to s,$              $c[i+1] \to c,$    $c[i+1] \to d) :=$
                    $(s'[m+1..d'-1],$         $c'[m+1..d'],$     $d'-m)$
                $s[i] := s'[m]$
```





```
// Remove the $i$-th child from an ABItem
Procedure ABItem::removeLocally($i$ : $\mathbb{N}$)
    $c[i..d-1] := c[i+1..d]$
    $s[i..d-2] := s[i+1..d-1]$                               //
    $d$--
```

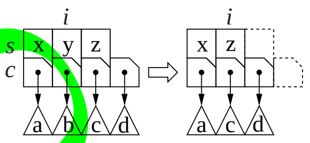


Fig. 7.9. Removal from an (a,b)-tree

parent, the need to fuse or balance might propagate up the tree. If the degree of the root drops to one, we do one of two things. If the tree has height one and hence contains only a single element, there is nothing to do and we are finished. Otherwise, we deallocate the root and replace it by its sole child. The height of the tree decreases by one.

The execution time of *remove* is also proportional to the height of the tree and hence logarithmic in the size of the sorted sequence. We summarize the performance of (a,b)-trees in the following theorem.

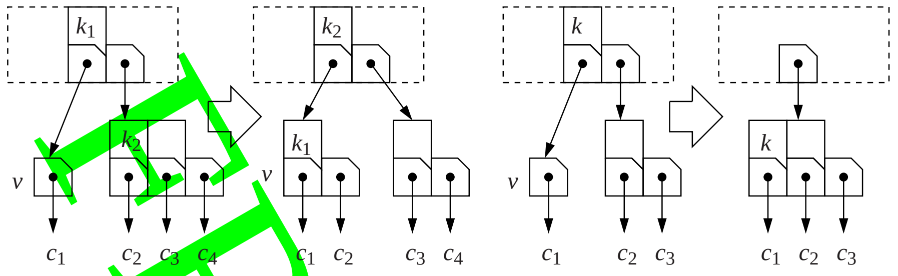


Fig. 7.10. Node balancing and fusing in (2,4)-trees: node v has degree $a-1$ (here 1). In the situation on the *left*, it has a sibling of degree $a+1$ or more (here 3), and we *balance* the degrees. In the situation on the *right*, the sibling has degree a and we *fuse* v and its sibling. Observe how keys are moved. When two nodes are fused, the degree of the parent decreases

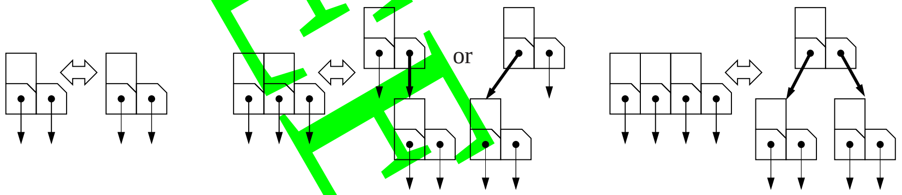


Fig. 7.11. The correspondence between (2,4)-trees and red–black trees. Nodes of degree 2, 3, and 4 as shown on the *left* correspond to the configurations on the *right*. Red edges are shown in **bold**

Theorem 7.2. *For any integers a and b with $a \geq 2$ and $b \geq 2a-1$, (a,b)-trees support the operations insert, remove, and locate on sorted sequences of size n in time* $O(\log n)$.

Exercise 7.8. Give a more detailed implementation of *locateLocally* based on binary search that needs at most $\lceil \log b \rceil$ comparisons. Your code should avoid both explicit use of infinite key values and special case treatments for extreme cases.

Exercise 7.9. Suppose $a = 2^k$ and $b = 2a$. Show that $(1+\frac{1}{k})\log n + 1$ element comparisons suffice to execute a *locate* operation in an (a,b)-tree. Hint: it is *not* quite sufficient to combine Exercise 7.4 with Exercise 7.8 since this would give you an additional term $+k$.

Exercise 7.10. Extend (a,b)-trees so that they can handle multiple occurrences of the same key. Elements with identical keys should be treated last-in first-out, i.e., *remove*(k) should remove the least recently inserted element with key k.

***Exercise 7.11 (red–black trees).** A *red–black tree* is a binary search tree where the edges are colored either red or black. The *black depth* of a node v is the number of black edges on the path from the root to v. The following invariants have to hold:

(a) All leaves have the same black depth.
(b) Edges into leaves are black.
(c) No path from the root to a leaf contains two consecutive red edges.

Show that red–black trees and $(2,4)$-trees are isomorphic in the following sense: $(2,4)$-trees can be mapped to red–black trees by replacing nodes of degree three or four by two or three nodes, respectively, connected by red edges as shown in Fig. 7.11. Red–black trees can be mapped to $(2,4)$-trees using the inverse transformation, i.e., components induced by red edges are replaced by a single node. Now explain how to implement $(2,4)$-trees using a representation as a red–black tree.[5] Explain how the operations of expanding, shrinking, splitting, merging, and balancing nodes of the $(2,4)$-tree can be translated into recoloring and rotation operations in the red–black tree. Colors are stored at the target nodes of the corresponding edges.

7.3 More Operations

Search trees support many operations in addition to *insert*, *remove*, and *locate*. We shall study them in two batches. In this section, we shall discuss operations directly supported by (a,b)-trees, and in Sect. 7.5 we shall discuss operations that require augmentation of the data structure.

- *min/max*. The constant-time operations *first* and *last* on a sorted list give us the smallest and the largest element in the sequence in constant time. In particular, search trees implement *double-ended priority queues*, i.e., sets that allow locating and removing both the smallest and the largest element in logarithmic time. For example, in Fig. 7.5, the dummy element of list ℓ gives us access to the smallest element, 2, and to the largest element, 19, via its *next* and *prev* pointers, respectively.
- *Range queries*. To retrieve all elements with keys in the range $[x,y]$, we first locate x and then traverse the sorted list until we see an element with a key larger than y. This takes time $\mathrm{O}(\log n + \text{output size})$. For example, the range query $[4,14]$ applied to the search tree in Fig. 7.5 will find the 5, it subsequently outputs 7, 11, 13, and it stops when it sees the 17.
- *Build/rebuild*. Exercise 7.12 asks you to give an algorithm that converts a sorted list or array into an (a,b)-tree in linear time. Even if we first have to sort the elements, this operation is much faster than inserting the elements one by one. We also obtain a more compact data structure this way.

Exercise 7.12. Explain how to construct an (a,b)-tree from a sorted list in linear time. Which $(2,4)$-tree does your routine construct for the sequence $\langle 1..17\rangle$? Next, remove the elements 4, 9, and 16.

[5] This may be more space-efficient than a direct representation, if the keys are large.

7.3.1 *Concatenation

Two sorted sequences can be concatenated if the largest element of the first sequence is smaller than the smallest element of the second sequence. If sequences are represented as (a,b)-trees, two sequences q_1 and q_2 can be concatenated in time $\mathrm{O}(\log\max(|q_1|,|q_2|))$. First, we remove the dummy item from q_1 and concatenate the underlying lists. Next, we fuse the root of one tree with an appropriate node of the other tree in such a way that the resulting tree remains sorted and balanced. More precisely, if $q_1.\mathit{height} \geq q_2.\mathit{height}$, we descend $q_1.\mathit{height} - q_2.\mathit{height}$ levels from the root of q_1 by following pointers to the rightmost children. The node v, that we reach is then fused with the root of q_2. The new splitter key required is the largest key in q_1. If the degree of v now exceeds b, v is split. From that point, the concatenation proceeds like an *insert* operation, propagating splits up the tree until the invariant is fulfilled or a new root node is created. The case $q_1.\mathit{height} < q_2.\mathit{height}$ is a mirror image. We descend $q_2.\mathit{height} - q_1.\mathit{height}$ levels from the root of q_2 by following pointers to the leftmost children, and fuse .... If we explicitly store the heights of the trees, the operation runs in time $\mathrm{O}(1 + |q_1.\mathit{height} - q_2.\mathit{height}|) = \mathrm{O}(\log(|q_1| + |q_2|))$. Figure 7.12 gives an example.

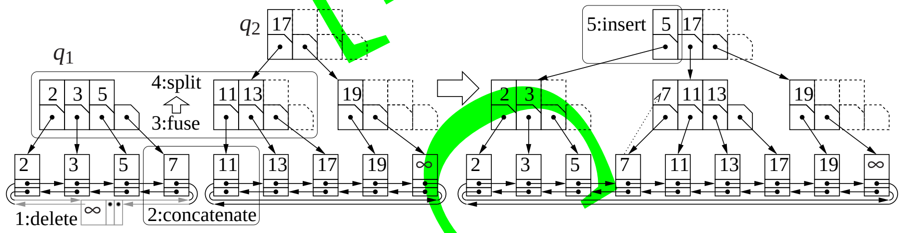


Fig. 7.12. Concatenating $(2,4)$-trees for $\langle 2,3,5,7\rangle$ and $\langle 11,13,17,19\rangle$

7.3.2 *Splitting

We now show how to split a sorted sequence at a given element in logarithmic time. Consider a sequence $q = \langle w,\ldots,x,y,\ldots,z\rangle$. Splitting q at y results in the sequences $q_1 = \langle w,\ldots,x\rangle$ and $q_2 = \langle y,\ldots,z\rangle$. We implement splitting as follows. Consider the path from the root to leaf y. We split each node v on this path into two nodes, v_ℓ and v_r. Node v_ℓ gets the children of v that are to the left of the path and v_r gets the children, that are to the right of the path. Some of these nodes may get no children. Each of the nodes with children can be viewed as the root of an (a,b)-tree. Concatenating the left trees and a new dummy sequence element yields the elements up to x. Concatenating $\langle y\rangle$ and the right trees produces the sequence of elements starting from y. We can do these $\mathrm{O}(\log n)$ concatenations in total time $\mathrm{O}(\log n)$ by exploiting the fact that the left trees have a strictly decreasing height and the right trees have a strictly increasing height. Let us look at the trees on the left in more detail. Let

r_1, r_2 to r_k be the roots of the trees on the left and let h_1, h_2 to h_h be their heights. Then $h_1 \geq h_2 \geq \ldots \geq h_k$. We first concatenate r_{k-1} and r_k in time $\mathrm{O}(1+h_{k-1}-h_k)$, then concatenate r_{k-2} with the result in time $\mathrm{O}(1+h_{k-2}-h_{k-1})$, then concatenate r_{k-3} with the result in time $\mathrm{O}(1+h_{k-2}-h_{k-1})$, and so on. The total time needed for all concatenations is $\mathrm{O}\left(\sum_{1\leq i<k}(1+h_i-h_{i+1})\right) = \mathrm{O}(k+h_1-h_k) = \mathrm{O}(\log n)$. Figure 7.13 gives an example.

Exercise 7.13. We glossed over one issue in the argument above. What is the height of the tree resulting from concatenating the trees with roots r_k to r_i? Show that the height is $h_i + \mathrm{O}(1)$.

Exercise 7.14. Explain how to remove a subsequence $\langle e \in q : \alpha \leq e \leq \beta \rangle$ from an (a,b)-tree q in time $\mathrm{O}(\log n)$.

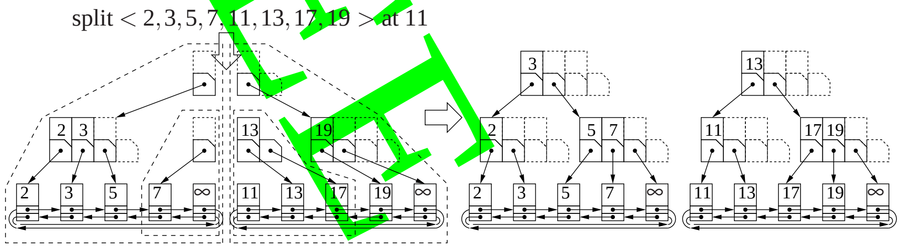


Fig. 7.13. Splitting the $(2,4)$-tree for $\langle 2,3,5,7,11,13,17,19\rangle$ shown in Fig. 7.5 produces the subtrees shown on the *left*. Subsequently concatenating the trees surrounded by the dashed lines leads to the $(2,4)$-trees shown on the *right*

7.4 Amortized Analysis of Update Operations

The best-case time for an insertion or removal is considerably smaller than the worst-case time. In the best case, we basically pay for locating the affected element, for updating the sequence, and for updating the bottommost internal node. The worst case is much slower. *Split* or *fuse* operations may propagate all the way up the tree.

Exercise 7.15. Give a sequence of n operations on $(2,3)$-trees that requires $\Omega(n \log n)$ *split* and *fuse* operations.

We now show that the *amortized* complexity is essentially equal to that of the best case if b is not at its minimum possible value but is at least $2a$. In Sect. 7.5.1, we shall see variants of *insert* and *remove* that turn out to have constant amortized complexity in the light of the analysis below.

Theorem 7.3. *Consider an (a,b)-tree with $b \geq 2a$ that is initially empty. For any sequence of n insert or remove operations, the total number of split or fuse operations is* $\mathrm{O}(n)$.

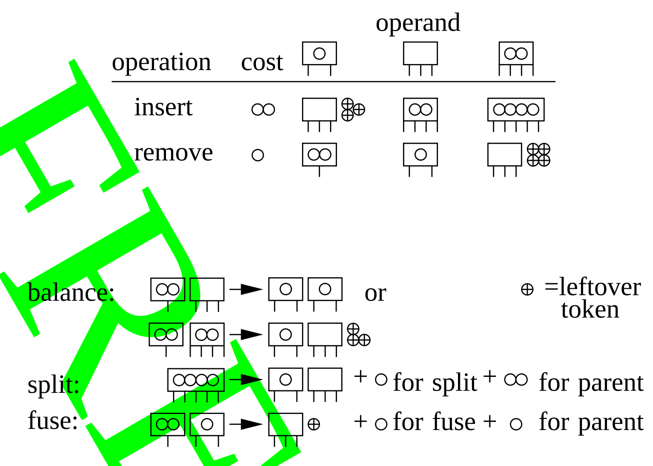


Fig. 7.14. The effect of (a,b)-tree operations on the token invariant. The *upper part* of the figure illustrates the addition or removal of a leaf. The two tokens charged for an insert are used as follows. When the leaf is added to a node of degree three or four, the two tokens are put on the node. When the leaf is added to a node of degree two, the two tokens are not needed, and the token from the node is also freed. The *lower part* illustrates the use of the tokens in *balance*, *split*, and *fuse* operations

Proof. We give the proof for $(2,4)$-trees and leave the generalization to Exercise 7.16. We use the bank account method introduced in Sect. 3.3. *Split* and *fuse* operations are paid for by tokens. These operations cost one token each. We charge two tokens for each *insert* and one token for each *remove*. and claim that this suffices to pay for all *split* and *fuse* operations. Note that there is at most one *balance* operation for each *remove*, so that we can account for the cost of *balance* directly without an accounting detour. In order to do the accounting, we associate the tokens with the nodes of the tree and show that the nodes can hold tokens according to the following table (*the token invariant*):

degree	1	2	3	4	5
tokens	○○	○		○○	○○○○

Note that we have included the cases of degree 1 and 5 that occur during rebalancing. The purpose of splitting and fusing is to remove these exceptional degrees.

Creating an empty sequence makes a list with one dummy item and a root of degree one. We charge two tokens for the *create* and put them on the root. Let us look next at insertions and removals. These operations add or remove a leaf and hence increase or decrease the degree of a node immediately above the leaf level. Increasing the degree of a node requires up to two additional tokens on the node (if the degree increases from 3 to 4 or from 4 to 5), and this is exactly what we charge for an insertion. If the degree grows from 2 to 3, we do not need additional tokens and we are overcharging for the insertion; there is no harm in this. Similarly, reducing the degree by one may require one additional token on the node (if the degree decreases from 3 to 2 or from 2 to 1). So, immediately after adding or removing a leaf, the token invariant is satisfied.

We need next to consider what happens during rebalancing. Figure 7.14 summarizes the following discussion graphically.

A *split* operation is performed on nodes of (temporary) degree five and results in a node of degree three and a node of degree two. It also increases the degree of the parent. The four tokens stored on the degree-five node are spent as follows: one token pays for the *split*, one token is put on the new node of degree two, and two tokens are used for the parent node. Again, we may not need the additional tokens for the parent node; in this case, we discard them.

A *balance* operation takes a node of degree one and a node of degree three or four and moves one child from the high-degree node to the node of degree one. If the high-degree node has degree three, we have two tokens available to us and need two tokens; if the high-degree node has degree four, we have four tokens available to us and need one token. In either case, the tokens available are sufficient to maintain the token invariant.

A *fuse* operation fuses a degree-one node with a degree-two node into a degree-three node and decreases the degree of the parent. We have three tokens available. We use one to pay for the operation and one to pay for the decrease of the degree of the parent. The third token is no longer needed, and we discard it.

Let us summarize. We charge two tokens for sequence creation, two tokens for each *insert*, and one token for each *remove*. These tokens suffice to pay one token each for every *split* or *fuse* operation. There is at most a constant amount of work for everything else done during an *insert* or *remove* operation. Hence, the total cost for n update operations is $\mathrm{O}(n)$, and there are at most $2(n+1)$ *split* or *fuse* operations. □

***Exercise 7.16.** Generalize the above proof to arbitrary a and b with $b \geq 2a$. Show that n *insert* or *remove* operations cause only $\mathrm{O}(n/(b-2a+1))$ *fuse* or *split* operations.

***Exercise 7.17 (weight-balanced trees [150]).** Consider the following variant of (a,b)-trees: the node-by-node invariant $d \geq a$ is relaxed to the global invariant that the tree has at least $2a^{height-1}$ leaves. A *remove* does not perform any *fuse* or *balance* operations. Instead, the whole tree is rebuilt using the routine described in Exercise 7.12 when the invariant is violated. Show that *remove* operations execute in $\mathrm{O}(\log n)$ amortized time.

7.5 Augmented Search Trees

We show here that (a,b)-trees can support additional operations on sequences if we augment the data structure with additional information. However, augmentations come at a cost. They consume space and require time for keeping them up to date. Augmentations may also stand in each other's way.

Exercise 7.18 (reduction). Some operations on search trees can be carried out with the use of the navigation data structure alone and without the doubly linked list. Go through the operations discussed so far and discuss whether they require the *next* and *prev* pointers of linear lists. Range queries are a particular challenge.

7.5.1 Parent Pointers

Suppose we want to remove an element specified by the handle of a list item. In the basic implementation described in Sect. 7.2, the only thing we can do is to read the key k of the element and call $\mathit{remove}(k)$. This would take logarithmic time for the search, although we know from Sect. 7.4 that the amortized number of *fuse* operations required to rebalance the tree is constant. This detour is not necessary if each node v of the tree stores a handle indicating its *parent* in the tree (and perhaps an index i such that $v.\mathit{parent}.c[i] = v$).

Exercise 7.19. Show that in (a,b)-trees with parent pointers, $\mathit{remove}(h : \mathit{Item})$ and $\mathit{insertAfter}(h : \mathit{Item})$ can be implemented to run in constant amortized time.

***Exercise 7.20 (avoiding augmentation).** Outline a class *ABTreeIterator* that allows one to represent a position in an (a,b)-tree that has no parent pointers. Creating an iterator I is an extension of *search* and takes logarithmic time. The class should support the operations *remove* and *insertAfter* in constant amortized time. Hint: store the path to the current position.

***Exercise 7.21 (finger search).** Augment search trees such that searching can profit from a "hint" given in the form of the handle of a *finger element* e'. If the sought element has rank r and the finger element e' has rank r', the search time should be $O(\log|r - r'|)$. Hint: one solution links all nodes at each level of the search tree into a doubly linked list.

***Exercise 7.22 (optimal merging).** Explain how to use finger search to implement merging of two sorted sequences in time $O(n\log(m/n))$, where n is the size of the shorter sequence and m is the size of the longer sequence.

7.5.2 Subtree Sizes

Suppose that every nonleaf node t of a search tree stores its *size*, i.e., $t.\mathit{size}$ is the number of leaves in the subtree rooted at t. The k-th smallest element of the sorted sequence can then be selected in a time proportional to the height of the tree. For simplicity, we shall describe this for binary search trees. Let t denote the current search tree node, which is initialized to the root. The idea is to descend the tree while maintaining the invariant that the k-th element is contained in the subtree rooted at t. We also maintain the number i of elements that are to the *left* of t. Initially, $i = 0$. Let i' denote the size of the left subtree of t. If $i + i' \geq k$, then we set t to its left successor. Otherwise, t is set to its right successor and i is increased by i'. When a leaf is reached, the invariant ensures that the k-th element is reached. Figure 7.15 gives an example.

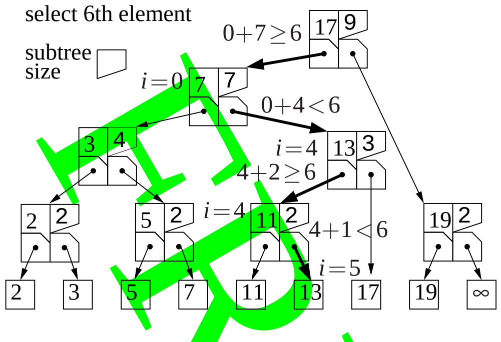


Fig. 7.15. Selecting the 6th smallest element from $\langle 2,3,5,7,11,13,17,19\rangle$ represented by a binary search tree. The **thick** arrows indicate the search path

Exercise 7.23. Generalize the above selection algorithm to (a,b)-trees. Develop two variants: one that needs time $O(b \log_a n)$ and stores only the subtree size and another variant that needs only time $O(\log n)$ and stores $d-1$ sums of subtree sizes in a node of degree d.

Exercise 7.24. Explain how to determine the rank of a sequence element with key k in logarithmic time.

Exercise 7.25. A colleague suggests supporting both logarithmic selection time and constant amortized update time by combining the augmentations described in Sects. 7.5.1 and 7.5.2. What will go wrong?

7.6 Implementation Notes

Our pseudocode for (a,b)-trees is close to an actual implementation in a language such as C++ except for a few oversimplifications. The temporary arrays s' and c' in the procedures *insertRec* and *removeRec* can be avoided by appropriate case distinctions. In particular, a *balance* operation will not require calling the memory manager. A *split* operation of a node v might be slightly faster if v keeps the left half rather than the right half. We did not formulate the operation this way because then the cases of inserting a new sequence element and splitting a node would no longer be the same from the point of view of their parent.

For large b, *locateLocally* should use binary search. For small b, a linear search might be better. Furthermore, we might want to have a specialized implementation for small, fixed values of a and b that *unrolls*[6] all the inner loops. Choosing b to be a power of two might simplify this task.

Of course, the values of a and b are important. Let us start with the cost of *locate*. There are two kinds of operation that dominate the execution time of *locate*: besides their inherent cost, element comparisons may cause branch mispredictions (see also Sect. 5.9); pointer dereferences may cause cache faults. Exercise 7.9 indicates that

[6] *Unrolling* a loop "**for** $i := 1$ **to** K **do** $body_i$" means replacing it by the *straight-line program* "$body_1; \ldots; body_K$". This saves the overhead required for loop control and may give other opportunities for simplifications.

element comparisons can be minimized by choosing a as a large power of two and $b = 2a$. Since the number of pointer dereferences is proportional to the height of the tree (see Exercise 7.4), large values of a are also good for this measure. Taking this reasoning to the extreme, we would obtain the best performance for $a \geq n$, i.e., a single sorted array. This is not astonishing. We have concentrated on searches, and static data structures are best if updates are neglected.

Insertions and deletions have an amortized cost of one *locate* plus a constant number of node reorganizations (*split*, *balance*, or *fuse*) with cost $\mathrm{O}(b)$ each. We obtain a logarithmic amortized cost for update operations if $b = \mathrm{O}(\log n)$. A more detailed analysis (see Exercise 7.16) would reveal that increasing b beyond $2a$ makes *split* and *fuse* operations less frequent and thus saves expensive calls to the memory manager associated with them. However, this measure has a slightly negative effect on the performance of *locate* and it clearly increases *space consumption*. Hence, b should remain close to $2a$.

Finally, let us take a closer look at the role of cache faults. A cache of size M can hold $\Theta(M/b)$ nodes. These are most likely to be the frequently accessed nodes close to the root. To a first approximation, the top $\log_a(M/b)$ levels of the tree are stored in the cache. Below this level, every pointer dereference is associated with a cache fault, i.e., we will have about $\log_a(bn/\Theta(M))$ cache faults in each *locate* operation. Since the cache blocks of processor caches start at addresses that are a multiple of the block size, it makes sense to *align* the starting addresses of search tree nodes with a cache block, i.e., to make sure that they also start at an address that is a multiple of the block size. Note that (a,b)-trees might well be more efficient than binary search for large data sets because we may save a factor of $\log a$ in cache faults.

Very large search trees are stored on disks. Under the name *B-trees* [16], (a,b)-trees are the workhorse of the indexing data structures in databases. In that case, internal nodes have a size of several kilobytes. Furthermore, the items of the linked list are also replaced by entire data blocks that store between a' and b' elements, for appropriate values of a' and b' (see also Exercise 3.20). These leaf blocks will then also be subject to splitting, balancing, and fusing operations. For example, assume that we have $a = 2^{10}$, the internal memory is large enough (a few megabytes) to cache the root and its children, and the data blocks store between 16 and 32 Kbyte of data. Then two disk accesses are sufficient to *locate* any element in a sorted sequence that takes 16 Gbyte of storage. Since putting elements into leaf blocks dramatically decreases the total space needed for the internal nodes and makes it possible to perform very fast range queries, this measure can also be useful for a cache-efficient internal-memory implementation. However, note that update operations may now move an element in memory and thus will invalidate element handles stored outside the data structure. There are many more tricks for implementing (external-memory) (a,b)-trees. We refer the reader to [79] and [141, Chaps. 2 and 14] for overviews. A good free implementation of B-trees is available in STXXL [48].

From the augmentations discussed in Sect. 7.5 and the implementation trade-offs discussed here, it becomes evident that *the* optimal implementation of sorted sequences does not exist but depends on the hardware and the operation mix relevant to the actual application. We believe that (a,b)-trees with $b = 2^k = 2a = \mathrm{O}(\log n)$, augmented with parent pointers and a doubly linked list of leaves, are a sorted-sequence data structure that supports a wide range of operations efficiently.

Exercise 7.26. What choice of a and b for an (a,b)-tree guarantees that the number of I/O operations required for *insert*, *remove*, or *locate* is $\mathrm{O}(\log_B(n/M))$? How many I/O operations are needed to *build* an n-element (a,b)-tree using the external sorting algorithm described in Sect. 5.7 as a subroutine? Compare this with the number of I/Os needed for building the tree naively using insertions. For example, try $M = 2^{29}$ bytes, $B = 2^{18}$ bytes[7], $n = 2^{32}$, and elements that have 8-byte keys and 8 bytes of associated information.

7.6.1 C++

The STL has four container classes *set*, *map*, *multiset*, and *multimap* for sorted sequences. The prefix *multi* means that there may be several elements with the same key. *Map*s offer the interface of an associative array (see also Chap. 4). For example, $someMap[k] := x$ inserts or updates the element with key k and sets the associated information to x.

The most widespread implementation of sorted sequences in STL uses a variant of red–black trees with parent pointers, where elements are stored in all nodes rather than only in the leaves. None of the STL data types supports efficient splitting or concatenation of sorted sequences.

LEDA [118] offers a powerful interface *sortseq* that supports all important operations on sorted sequences, including finger search, concatenation, and splitting. Using an implementation parameter, there is a choice between (a,b)-trees, red–black trees, randomized search trees, weight-balanced trees, and skip lists.

7.6.2 Java

The Java library *java.util* offers the interface classes *SortedMap* and *SortedSet*, which correspond to the STL classes *set* and *map*, respectively. The corresponding implementation classes *TreeMap* and *TreeSet* are based on red–black trees.

7.7 Historical Notes and Further Findings

There is an entire zoo of sorted sequence data structures. Just about any of them will do if you just want to support *insert*, *remove*, and *locate* in logarithmic time. Performance differences for the basic operations are often more dependent on implementation details than on the fundamental properties of the underlying data structures. The differences show up in the additional operations.

[7] We are making a slight oversimplification here, since in practice one will use much smaller block sizes for organizing the tree than for sorting.

The first sorted-sequence data structure to support *insert*, *remove*, and *locate* in logarithmic time was AVL trees [4]. AVL trees are binary search trees which maintain the invariant that the heights of the subtrees of a node differ by one at the most. Since this is a strong balancing condition, *locate* is probably a little faster than in most competitors. On the other hand, AVL trees do *not* have constant amortized update costs. Another small disadvantage is that storing the heights of subtrees costs additional space. In comparison, red–black trees have slightly higher costs for *locate*, but they have faster updates and the single color bit can often be squeezed in somewhere. For example, pointers to items will always store even addresses, so that their least significant bit could be diverted to storing color information.

$(2,3)$-trees were introduced in [6]. The generalization to (a,b)-trees and the amortized analysis of Sect. 3.3 come from [95]. There, it was also shown that the total number of splitting and fusing operations at the nodes of any given height decreases exponentially with the height.

Splay trees [183] and some variants of randomized search trees [176] work even without any additional information besides one key and two successor pointers. A more interesting advantage of these data structures is their *adaptability* to nonuniform access frequencies. If an element e is accessed with probability p, these search trees will be reshaped over time to allow an access to e in a time $\mathrm{O}(\log(1/p))$. This can be shown to be asymptotically optimal for any comparison-based data structure. However, this property leads to improved running time only for quite skewed access patterns because of the large constants.

Weight-balanced trees [150] balance the size of the subtrees instead of the height. They have the advantage that a node of weight w (= number of leaves of its subtree) is only rebalanced after $\Omega(w)$ insertions or deletions have passed through it [26].

There are so many *search tree* data structures for *sorted sequences* that these two terms are sometimes used as synonyms. However, there are also some equally interesting data structures for sorted sequences that are *not* based on search trees. Sorted arrays are a simple *static* data structure. Sparse tables [97] are an elegant way to make sorted arrays dynamic. The idea is to accept some empty cells to make insertion easier. Reference [19] extended sparse tables to a data structure which is asymptotically optimal in an amortized sense. Moreover, this data structure is a crucial ingredient for a sorted-sequence data structure [19] that is *cache-oblivious* [69], i.e., it is cache-efficient on any two levels of a memory hierarchy without even knowing the size of caches and cache blocks. The other ingredient is oblivious *static* search trees [69]; these are perfectly balanced binary search trees stored in an array such that any search path will exhibit good locality in any cache. We describe here the *van Emde Boas layout* used for this purpose, for the case where there are $n = 2^{2^k}$ leaves for some integer k. We store the top 2^{k-1} levels of the tree at the beginning of the array. After that, we store the 2^{k-1} subtrees of depth 2^{k-1}, allocating consecutive blocks of memory for them. We recursively allocate the resulting $1+2^{k-1}$ subtrees of depth 2^{k-1}. Static cache-oblivious search trees are practical in the sense that they can outperform binary search in a sorted array.

Skip lists [159] are based on another very simple idea. The starting point is a sorted linked list ℓ. The tedious task of scanning ℓ during *locate* can be accelerated by producing a shorter list ℓ' that contains only some of the elements in ℓ. If corresponding elements of ℓ and ℓ' are linked, it suffices to scan ℓ' and only descend to ℓ when approaching the searched element. This idea can be iterated by building shorter and shorter lists until only a single element remains in the highest-level list. This data structure supports all important operations efficiently in an expected sense. Randomness comes in because the decision about which elements to lift to a higher-level list is made randomly. Skip lists are particularly well suited for supporting finger search.

Yet another family of sorted-sequence data structures comes into play when we no longer consider keys as atomic objects. If keys are numbers given in binary representation, we can obtain faster data structures using ideas similar to the fast integer-sorting algorithms described in Sect. 5.6. For example, we can obtain sorted sequences with w-bit integer keys that support all operations in time $O(\log w)$ [198, 129]. At least for 32-bit keys, these ideas bring a considerable speedup in practice [47]. Not astonishingly, string keys are also important. For example, suppose we want to adapt (a,b)-trees to use variable-length strings as keys. If we want to keep a fixed size for node objects, we have to relax the condition on the minimal degree of a node. Two ideas can be used to avoid storing long string keys in many nodes. *Common prefixes* of keys need to be stored only once, often in the parent nodes. Furthermore, it suffices to store the *distinguishing prefixes* of keys in inner nodes, i.e., just enough characters to be able to distinguish different keys in the current node [83]. Taking these ideas to the extreme results in *tries* [64], a search tree data structure specifically designed for string keys: tries are trees whose edges are labeled by characters or strings. The characters along a root–leaf path represent a key. Using appropriate data structures for the inner nodes, a trie can be searched in time $O(s)$ for a string of size s.

We shall close with three interesting generalizations of sorted sequences. The first generalization is *multidimensional objects*, such as intervals or points in d-dimensional space. We refer to textbooks on geometry for this wide subject [46]. The second generalization is *persistence*. A data structure is persistent if it supports nondestructive updates. For example, after the insertion of an element, there may be two versions of the data structure, the one before the insertion and the one after the insertion – both can be searched [59]. The third generalization is *searching many sequences* [36, 37, 130]. In this setting, there are many sequences, and searches need to locate a key in all of them or a subset of them.