7

Sorted Sequences

All of us spend a significant part of our time on searching, and so do computers: They look up telephone numbers, balances of bank accounts, flight reservations, bills and payments, ... In many applications, we want to search dynamic collections of data. New bookings are entered into reservation systems, reservations are changed or canceled, and bookings turn into actual flights. We have already seen one solution to the problem, namely hashing. However, it is often desirable to keep a dynamic collection sorted. The "manual data structure" used for this purpose is a filing-card box. We can insert new cards at any position, we can remove cards, we can go through the cards in sorted order, and we can use some kind of binary search to find a particular card. Large libraries used to have filing-card boxes with hundreds of thousands of cards.[1]

Formally, we wish to maintain a set S of elements e, which are equipped with keys $key(e)$ from a linearly ordered set *Key*. As before we write $e \leq e'$ if $key(e) \leq key(e')$. This induces a linear preorder on S. Not only should searches be possible, but insertions and deletions as well. This leads to the following basic operations of a *sorted sequence*:

- $S.locate(k : Key)$: **return** $\min\{e \in S : key(e) \geq k\}$
  (i.e., return an element e with $key(e) \geq k$ as small as possible).
- $S.remove(k : Key)$: $S := S \setminus \{e \in S : key(e) = k\}$
- $S.insert(e : Element)$: $S := (S \setminus \{e' \in S : key(e') = key(e)\}) \cup \{e\}$

The operation $locate(k)$ *locates* key k in S, i.e., it finds an element with key k if it exists, and finds an element of S with a minimum key larger than S otherwise. If k is larger than all keys that appear in S, no element is returned. The operation $insert(e)$ inserts an element e with a new key or replaces an element e' in S that has the same key as e. Note that this way of specifying *insert* implies that all elements in S have pairwise different keys. We shall reconsider this issue in Exercise 7.10. The operation $delete(k)$ removes the element with key k, if S has such an element. We shall show that these operations can be implemented to run in time $O(\log n)$, where

[1] The above photograph is from the library catalog of the University of Graz (Dr. M. Gossler).

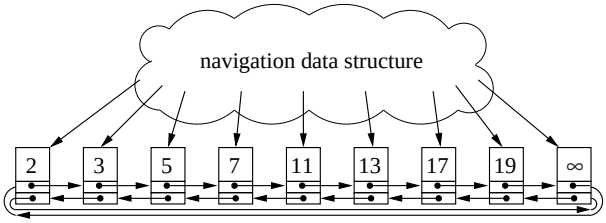


Fig. 7.1. A sorted sequence as a doubly linked list plus a navigation data structure.

n denotes the size of the sequence. How do sorted sequences compare with the data structures discussed in earlier chapters? They are more flexible than sorted arrays, because they efficiently support *insert* and *remove*. They are slower but also more powerful than hash tables, since $locate(k)$ also works when there is no element with key k in S. Priority queues are a special case of sorted sequences; they can only locate and remove the smallest element.

Our basic realization of a sorted sequence consists of a sorted doubly linked list with an additional navigation data structure supporting *locate*. Figure 7.1 illustrates this approach. Recall that a doubly linked list for n elements consists of $n+1$ items, one for each element and one additional "dummy item". We use the dummy item to store a special key value ∞, which is larger than all keys that might possibly appear in an element of S. 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 S, $locate(k)$ will return a handle to the dummy item. In Sect. 3.2.1, we saw 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 $remove(h : Handle)$ (note that here the object to be removed is given as a handle to a list item), *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 all bins have the same 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 $w(i)$ [73]. To implement this algorithm, we can keep the bins in a sequence S sorted by their residual capacity. To place an item i, we call $S.locate(w(i))$, remove the bin that we have found, reduce its residual capacity by $w(i)$, and reinsert it into S. See also Exercise 12.9. Note that for this application we must allow distinct elements (bins) with identical keys, see Exercise 7.10.

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, in a sorted sequence S, the set of horizontal line segments that intersect the sweep line. When the left endpoint of a horizontal segment is reached, it is inserted into S, with its y-coordinate as key, and when its right endpoint is reached, it is removed from S. (In the case where there are overlapping horizontal segments with identical y-coordinates, the sorted sequence must admit different elements with identical keys.) When a vertical line segment s is reached at a position x, we determine its vertical range $[y,y']$, call $S.locate(y)$, and, starting from the position thus found, scan S to the right until we reach an element with key y' or larger.[2] Exactly those horizontal line segments that are discovered during the scan have an intersection with s. The sweep-line algorithm can be generalized to arbitrary line segments [42], curved objects, and many other geometric problems [85].

Database indexes. A key problem in databases is to make large collections of data efficiently searchable. So-called *B-trees*, which are a variant of the (a,b)-tree data structure to be described in detail in Sect. 7.2, belong to the most important data structures used for databases.

The most popular navigation data structures belong to the class of *search trees*. Generally speaking, a search tree is a rooted ordered tree whose leaves correspond to the items in the doubly linked list. Internal nodes do not hold elements, but contain only keys to support navigation, and possibly other auxiliary information.[3] We shall frequently use the name of the navigation data structure to refer to the entire sorted-sequence data structure. Search tree algorithms will be introduced in three steps. As a warm-up, in Sect. 7.1 we consider (unbalanced) *binary search trees*, in which inner nodes have two children. They support *locate* in $O(\log n)$ time under certain favorable circumstances. Since binary search trees that guarantee this time bound are somewhat difficult to maintain under insertions and removals, we then switch to a generalization, namely (a,b)-trees, which allow search tree nodes of larger degree than 2. 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 augment search trees with additional mechanisms that support further operations. Section 7.4 takes a closer look at the (amortized) cost of update operations. Parallelizing search trees is briefly discussed in Sect. 7.6. As with priority queues, we shall concentrate on bulk operations that access many elements at once.

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, and so on, until you reach your destination.

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

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

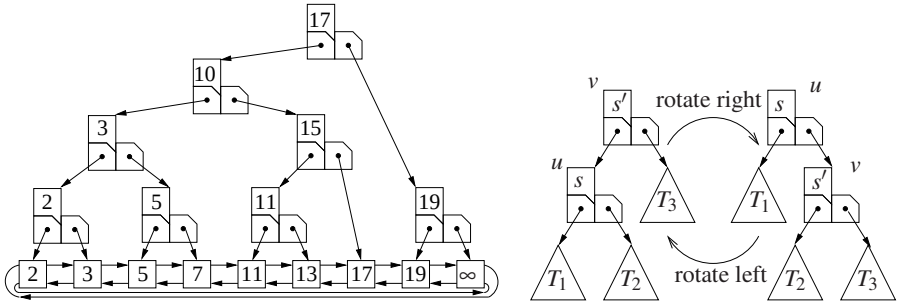


Fig. 7.2. *Left*: the sequence $\langle 2,3,5,7,11,13,17,19\rangle$ represented by a binary search tree. In each node, we show the splitter key at the top and the pointers to the children at the bottom. *Right*: rotation in a binary search tree. The triangles indicate subtrees. Observe that the ancestor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

A *binary search tree* is an ordered 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 would like to 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 (not elements!) that guide the search; we call these keys *splitter keys* or *splitters*. 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 nonleaf node v has the property that all elements stored in the left subtree of v have a key less than or equal to s, while all elements stored in the right subtree of v have a key larger than s.

Assume now that we are given a binary search tree T and a search key k. It is not hard to find the path to the correct leaf, i.e., the leaf with the entry e' that has the smallest key $k' \geq k$. Start at the root, and repeat the following step until a leaf is reached: If $k \leq s$ for the splitter key in the current node, go to the left child, otherwise go to the right child. If the data structure contains an element e with key k, it is quite easy to see that the process reaches the leaf that contains e. However, we also have to cover the case where no element with key k exists, and $k' = key(e') > k$. An easy case analysis shows that in this case the search procedure has the following invariant: Either the subtree rooted at the current node contains e' or this subtree contains the immediate predecessor of e', in its rightmost leaf. The search ends in a leaf with an element e'' and a key $k'' = key(e'')$, which is then compared with k. Because of the invariant, there are only two possibilities: If $k \leq k''$, then e'' is the desired element e'; if $k > k''$, then e' is the immediate successor of e'' in the doubly linked list. Figure 7.2 (left) shows an example of a binary search tree. You may want to test the behavior of the search procedure with keys such as 1, 9, 13, and 25. 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 search key k.

Exercise 7.1. Prove that, for every sorted sequence with $n \geq 1$ elements, there is a binary search tree with $n+1$ leaves and height $\lceil \log(n+1) \rceil$.

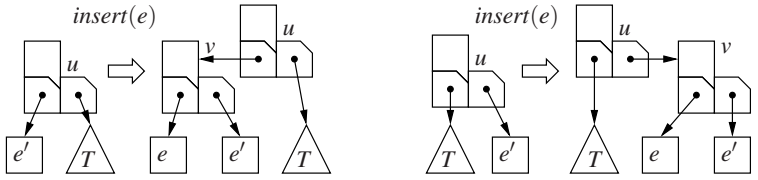


Fig. 7.3. Naive insertion into a binary search tree. A triangle indicates an entire subtree.

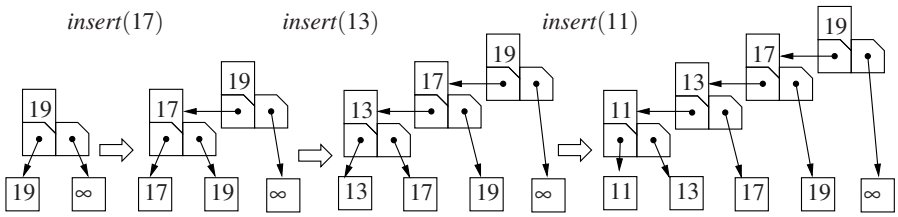


Fig. 7.4. Naively inserting elements in sorted order leads to a degenerate tree.

A binary search tree with $n+1$ leaves and height $\lceil \log(n+1) \rceil$ 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 left child e, right child e', and splitter key k. 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 with each insertion. Figure 7.4 gives an example: The tree may degenerate into a list, and 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 $\approx 2.99 \log n$ [91]. We shall not prove this here, but will outline a connection to quicksort to make the result plausible. As an example, consider the tree in Fig. 7.2 (left), with the splitters 10 and 15 replaced by 7 and 13, respectively. This tree can be built by 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 next recursive call for the left subarray. 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 in

the top-level call of quicksort and when 11 is inserted into the tree, respectively. We have seen (Theorem 5.6) that the expected number of comparisons in a randomized quicksort of n elements is $O(n \log n)$. By the above correspondence, the expected number of comparisons in building a binary tree by inserting n elements in random order is also $O(n \log n)$. Thus an insertion requires $O(\log n)$ comparisons on average. Even more is true. With high probability, each single insertion requires $O(\log n)$ comparisons, and the expected height is $\approx 2.99 \log n$.

Can we guarantee that the height stays logarithmic in the worst case? The answer is yes, and there are many different ways to achieve logarithmic height. We shall survey these techniques in Sect. 7.8, 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 (right) indicates how the shape of a binary tree can be changed by transformations called (left and right) *rotations*. Apply rotations to subtrees of the tree in Fig. 7.2 so that the node labeled 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 search 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 1 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 $s[0] = -\infty$

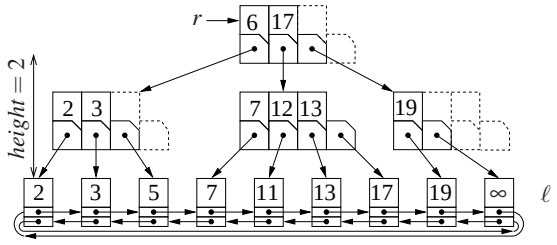


Fig. 7.5. Representation of $\langle 2,3,5,7,11,13,17,19 \rangle$ by a $(2,4)$-tree. The tree has height 2.

and $s[d] = \infty$. The keys of the elements e contained in the subtree rooted at the ith child $c[i]$, $1 \le i \le d$, lie between the $(i-1)$th splitter (exclusive) and the ith splitter (inclusive), i.e., we have $s[i-1] < key(e) \le s[i]$. Figure 7.5 shows a $(2,4)$-tree storing the sequence $\langle 2,3,5,7,11,13,17,19\rangle$.

Lemma 7.1. *An (a,b)-tree for n elements has height at most $1 + \lfloor \log_a(n+1)/2 \rfloor$, provided $n \ge 1$.*

Proof. The tree has $n+1$ leaves, where the "+1" accounts for the dummy leaf with key ∞. If $n = 0$, the root has degree 1 and there is a single leaf. So, assume $n \ge 1$. Let h be the height of the tree. Since the root has degree at least 2 and all other nodes have degree at least a, the number of leaves is at least $2a^{h-1}$. So $n+1 \ge 2a^{h-1}$, or $h \le 1 + \log_a(n+1)/2$. 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. Let k be the search key and let k' be the smallest key in the doubly linked list that is at least as large as k. Instead of performing a single comparison at a nonleaf node, we have to find the correct child from among up to b choices, i.e., the child $c[i]$ with the property that $s[i-1] < k \le s[i]$. Using binary search on the sorted sequence of the splitters, we need at most $\lceil \log b \rceil$ comparisons for each node on the search path. Correctness of the search procedure is proven using the same invariant as in the case of binary search: If key k is present in the tree, the search ends at the leaf with key k; otherwise, the subtree rooted at the current node either contains the element with key k' or contains the immediate predecessor of this element, in its rightmost leaf. Figure 7.6 gives pseudocode for initializing (a,b)-trees and for 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 $\lceil \log b \rceil (1 + \log_a((n+1)/2))$. Assuming in addition that $b \le 2a$, show that this number is $O(\log b) + O(\log n)$. What is the constant in front of the $\log n$ term?

To *insert* an element e with key $k = key(e)$, we first descend the tree recursively to locate k. This leads to a sequence element e' with key k'. Either k' is the smallest key $\ge k$ in the sequence or k' is the largest key strictly smaller than k. The latter situation can arise only if k is not present in the sequence. If $k = key(e')$, we simply replace e' by e in the list item. Note that this choice ensures that each key occurs at most once in the sorted sequence. If $k < k'$, we have to insert a new list item for e immediately before the item for e'. This case applies in particular if $k' = \infty$. We can avoid treating the case $k > k'$ separately, as follows: If this happens, we create

```
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                                              // Degree
    s = splitters : Array [1..b−1] of Key
    c = children : Array [1..b] of ABHandle

    Function locateLocally(k : Key) : ℕ
        return min{i ∈ 1..d : k ≤ s[i]}

    Function locateRec(k : Key, h : ℕ) : ABHandle
        i := locateLocally(k)
        if h = 1 then
            if c[i]→e ≥ k then return c[i]
            else return c[i]→next
        else return c[i]→locateRec(k, h−1)                             //

Class ABTree(a ≥ 2 : ℕ, b ≥ 2a−1 : ℕ) of Element
    ℓ = ⟨⟩ : List of Element
    r : ABItem(⟨⟩, ⟨ℓ.head⟩)
    height = 1 : ℕ

    // Locate the item with the smallest key k′ ≥ k
    Function locate(k : Key) : ABHandle return r.locateRec(k, height)
```

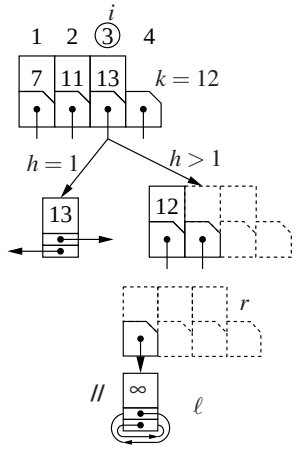


Fig. 7.6. (a,b)-trees. An *ABItem* is constructed from a sequence of keys and a sequence of handles to the children. The outdegree d is the number of children. We allocate space for the maximum possible outdegree b. There are two functions local to *ABItem*: $locateLocally(k)$ locates key k among the 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 1. 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.

a new item for e, swap it with the e'-item in the linked list, and redirect the pointer to the e'-item in the navigation data structure to the e-item. Then the e'-item can be inserted immediately before the e-item, just as in the first case. Since we found e' when localizing k, and the splitter keys in the tree have not changed, we know that localizing k' in the changed structure would lead to the e-item. Now we focus on the case where the new e-item has to be inserted before the e'-item. If e' was the ith child $c[i]$ of its parent node v, then e becomes the new child $c[i]$ and $k = key(e)$ becomes the corresponding splitter $s[i]$. The old children $c[i..d]$ and their corresponding splitters $s[i..d-1]$ are shifted one position to the right. If the degree d of v was smaller than b before, d can be incremented by 1, 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 $s'[1..b]$ denote the array of splitters of this illegal node, $c'[1..b+1]$ its child array, and 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 $s'[1..b-d]$ to this new node. The old node v keeps the d rightmost child pointers $c'[b+2-d..b+1]$ and the corresponding splitters $s'[b+2-d..b]$, but moved to the left by $b+1-d$ positions.

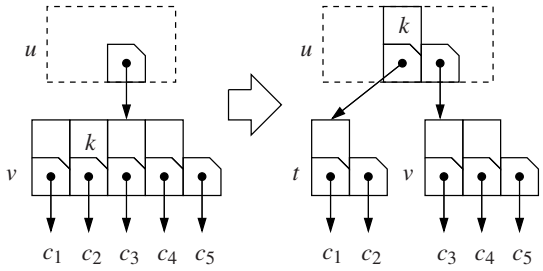


Fig. 7.7. Node splitting: The node v of degree $b+1$ (here 5) is split into a node of degree $\lfloor (b+1)/2 \rfloor$ and a node of degree $\lceil (b+1)/2 \rceil$. The degree of the parent increases by 1. The splitter key separating the two "parts" of v is moved to the parent.

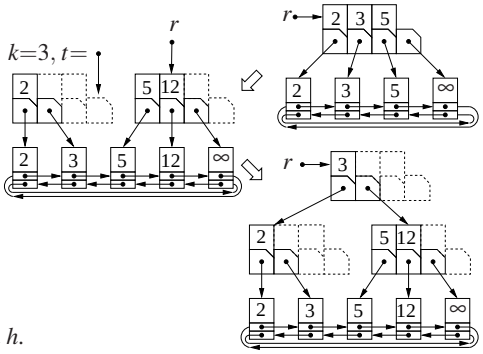
The "leftover" middle key $k = s'[b+1-d]$ is an upper bound for the keys reachable from t. This key and the pointer to t are needed in the predecessor u of v. The situation for u is analogous to the situation for v before the insertion: If v was the ith child of u, t displaces it to the right; children $i+1,\ldots,d$ are pushed one position to the right as well. Now t becomes the ith child, and k is inserted as the ith 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 1, i.e., the invariant that all leaves have the same depth is maintained. 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?

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

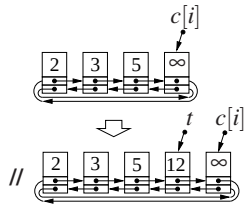


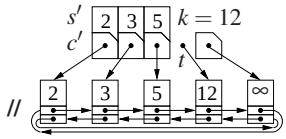

```
// Example:
// ⟨2,3,5⟩.insert(12)
Procedure ABTree::insert(e : Element)
    (k,t) := r.insertRec(e, height, ℓ)
    if t ≠ null then    // root was split
        r := allocate ABItem(⟨k⟩, ⟨t,r⟩)
        height++

// Insert an 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 : ℕ, ℓ : List of Element) : Key×ABHandle
    i := locateLocally(e)
    if h = 1 then    // base case
        if key(c[i]→e) = key(e) then    // same key already present
            c[i]→e := e    // overwrite with new element
            return (⊥, null)
        else
            if key(c[i]→e) > key(e) then
                (k,t) := (key(e), ℓ.insertBefore(e, c[i]))
            else    // key(c[i]→e) < key(e)
                (k,t) := (key(c[i]→e), ℓ.insertAfter(e, c[i]))
                swap(c[i], t)
        endif
    else
        (k,t) := c[i]→insertRec(e, h−1, ℓ)
        if t = null then return (⊥, null)
    endif

    s' := ⟨s[1],...,s[i−1], k, s[i],...,s[d−1]⟩
    c' := ⟨c[1],...,c[i−1], t, c[i],...,c[d]⟩

    if d < b then    // there is still room here
        (s,c,d) := (s',c',d+1)
        return (⊥, null)
    else    // split this node
        d := ⌊(b+1)/2⌋
        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]))
```





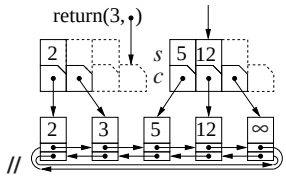
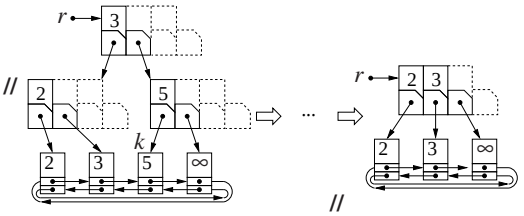
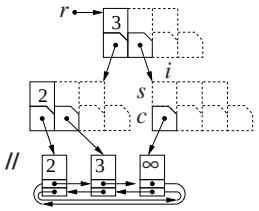


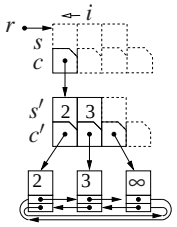
Fig. 7.8. Insertion into an (a,b)-tree

```
// Example: ⟨2,3,5⟩.remove(5)
Procedure ABTree::remove(k : Key)
    r.removeRec(k, height, ℓ)
    if r.d = 1 ∧ height > 1 then
        r' := r; r := r'.c[1];
        dispose r'; height−−
```




```
Procedure ABItem::removeRec(k : Key, h : ℕ, ℓ : List of Element)
    i := locateLocally(k)
    if h = 1 then    // base case, bottom of tree
        if key(c[i]→e) = k then    // there is sth. to remove
            ℓ.remove(c[i])
            removeLocally(i)
    else
        c[i]→removeRec(e, h−1, ℓ)
        if c[i]→d < a then                    // invariant needs repair
            if i = d then i−−                 // make sure i and i+1 are valid children
            s' := concatenate(c[i]→s, ⟨s[i]⟩, c[i+1]→s)
            c' := concatenate(c[i]→c, c[i+1]→c)
            d' := |c'|
            if d' ≤ b then    // fuse
                (c[i+1]→s, c[i+1]→c, c[i+1]→d) := (s', c', d')
                dispose c[i];    removeLocally(i)
            else              // balance
                m := ⌈d'/2⌉
                (c[i]→s, c[i]→c, c[i]→d) := (s'[1..m−1], c'[1..m], m)
                (c[i+1]→s,          c[i+1]→c,    c[i+1]→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 : ℕ)
    c[i..d−1] := c[i+1..d]
    s[i..d−2] := s[i+1..d−1]
    d−−
```

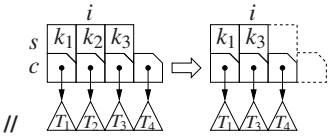


Fig. 7.9. Removal from an (a,b)-tree

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 one child or several children from that neighbor to $c[i]$. 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 new splitter array, we place the splitter $s[i]$ of the parent between the splitter arrays of $c[i]$ and $c[i+1]$. The new arrays are stored in $c[i+1]$, node $c[i]$ is deallocated, and the pointer to $c[i]$, 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 parent, the need to fuse or balance might propagate up the tree. If the degree of the root drops to 1, we do one of two things. If the tree has height 1 and hence the linked list contains only a single item (the dummy node), 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 1.

The execution time of *remove* is 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.

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.

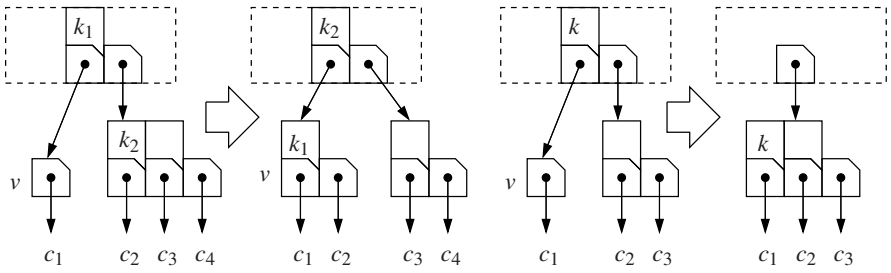


Fig. 7.10. Node balancing and fusing in (2,4)-trees: Node v has degree $a-1$ (here 1). In the situation on the *left*, it has a sibling of degree $a+1$ or more (here 3), and we *balance* the degrees. In the situation on the *right*, the sibling has degree a, and we *fuse* v and its sibling. Observe how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

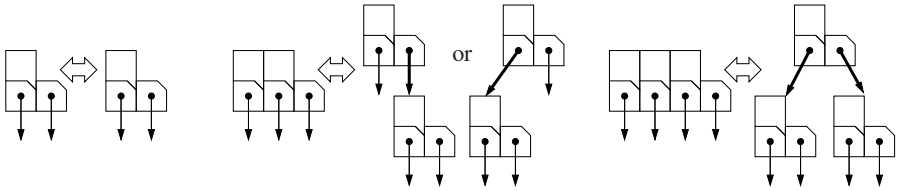


Fig. 7.11. The correspondence between (2,4)-trees and red–black trees. Nodes of degree 2, 3, and 4 as shown on the *left* correspond to the configurations on the *right*. Red edges are shown in **bold**.

Exercise 7.9. Suppose $a = 2^k$ and $b = 2a$. Show that $(1 + 1/k)\log n + 1$ 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 adding a child to a node, removing a child from a node, splitting, fusing, and balancing nodes in the $(2,4)$-tree can be translated into recoloring and rotation operations (see Fig. 7.2) 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:

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

- *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. 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. In particular, search trees implement *double-ended priority queues*, i.e., sets that allow finding and removing both the smallest and the largest element; finding takes constant time and removing takes logarithmic time.
- *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 $O(\log n + \text{output size})$. 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 $\langle 1,\ldots,17\rangle$? Next, remove the elements 4, 9, and 16.

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 S_1 and S_2 can be concatenated in time $O(\log(\max\{|S_1|,|S_2|\}))$. First, using the pointers to their dummy items, we find the beginning and end of both lists and make a note of the largest key in S_1. We remove the dummy item from S_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 $S_1.\mathit{height} \geq S_2.\mathit{height}$, we descend $S_1.\mathit{height} - S_2.\mathit{height}$ levels from the root of S_1 by following pointers to the rightmost children. The node v thus reached is then fused with the root of S_2. The new splitter key required is the largest key in S_1. If the degree of v now exceeds b, node v is split. From that point on, 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 $S_1.\mathit{height} < S_2.\mathit{height}$ is a mirror image. We descend $S_2.\mathit{height} - S_1.\mathit{height}$ levels from the root of S_2 by following pointers to the leftmost children, fuse nodes, and propagate splits upwards, if necessary. Figure 7.12 gives an example of this case. The operation runs in time $O(\log(\max\{|S_1|,|S_2|\}))$, even if the heights of the trees have to be calculated. If the heights of the trees are known, time $O(1+|S_1.\mathit{height} - S_2.\mathit{height}|)$ is sufficient.

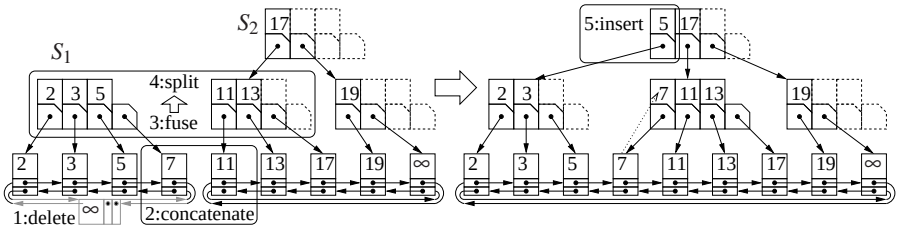


Fig. 7.12. Concatenating $(2,4)$-trees for $\langle 2,3,5,7\rangle$ and $\langle 11,13,17,19\rangle$.

7.3.2 *Splitting

We now show how to split a sorted sequence at a given element in logarithmic time. Consider a sequence $S = \langle w,\ldots,x,y,\ldots,z\rangle$. Splitting S at y results in the sequences $S_1 = \langle w,\ldots,x\rangle$ and $S_2 = \langle y,\ldots,z\rangle$. We implement splitting as follows. Consider the path p from the root to leaf y. We split each node v on p into two nodes, v_ℓ and v_r. Node v_ℓ gets the children of v that are to the left of p, and v_r gets the children that are to the right of p. Some of these nodes may get only one child or no children. Such a node with at least one child can be viewed as the root of an (a,b)-tree (slightly generalized by waiving the degree bound of 2 for the root). Note that we can record the heights of all these trees at practically no extra cost. The doubly linked list is split between the x-node and the y-node; the first part gets a new dummy item. Successively concatenating the left trees and the new dummy element yields an (a,b)-tree for the sorted list $S_1 = \langle w,\ldots,x\rangle$. Concatenating $\langle y\rangle$ and the right trees produces the (a,b)-tree for $S_2 = \langle y,\ldots,z\rangle$. The concatenation is simpler than in the general case, as we do not have to worry about the linked lists. These $O(\log n)$ concatenations can even be carried out in total time $O(\log n)$, by exploiting the fact that the heights of the left trees are strictly decreasing and the heights of the right trees are strictly increasing. Let us look at the trees to the left of p in more detail. Let $r_1, r_2, \ldots, r_k$ be the roots of these trees and let $h_1, h_2, \ldots, h_k$ be their heights. Then $h_1 > h_2 > \cdots > h_k$. Recall that these heights are known. We first concatenate the tree with root r_k with the new dummy element, which in time $O(1+h_k)$ yields a new tree with height at most $h_k + O(1)$. This tree is concatenated with the tree with root r_{k-1}, which in time $O(1+h_{k-1}-h_k)$ leads to a new tree with height at most $h_{k-1}+O(1)$. This is iterated: In the round for $j = k-2,\ldots,1$ the tree with root r_j is concatenated with the tree resulting from the previous rounds; the time needed is $O\big(1+h_j-h_{j+1}\big)$; the new tree has height at most $h_j + O(1)$ (see Exercise 7.13). After k rounds, only one tree is left. The total time needed is $O\big(1+h_k+\sum_{1\le i<k}(1+h_i-h_{i+1})\big) = O(k+h_1) = O(\log n)$. The trees on the right side are treated analogously, starting with the smallest tree and the single element y. Figure 7.13 gives an example.

Exercise 7.13. We glossed over one issue in the argument above. How can we bound the height of the tree resulting from concatenating the trees with roots r_k to r_i? Show (by induction) that the height is not larger than $h_i + 1$.

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

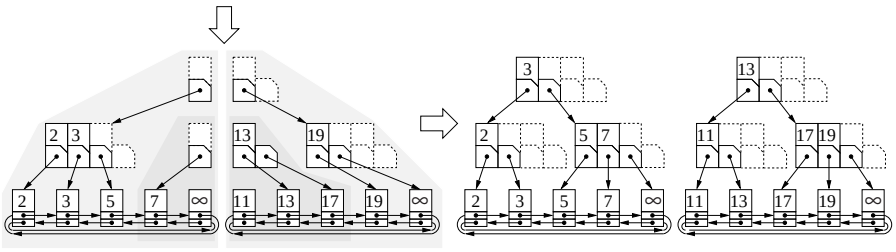


Fig. 7.13. Splitting the $(2,4)$-tree for $\langle 2,3,5,7,11,13,17,19 \rangle$ shown in Fig. 7.5 at element 11 produces the four (shaded) subtrees shown on the *left* and a single list node (with key 11). The list is split, which creates a new dummy item. Successively concatenating the trees indicated by the shaded areas leads to the $(2,4)$-trees shown on the *right*.

7.4 Amortized Analysis of Update Operations

Inserting or deleting an element in an (a,b)-tree costs $\Theta(\log n)$ time. Let us take a closer look at the best and the worst case. In the best case, we have to locate the affected element and to update the linked list and the bottommost internal node on the search path. In the worst case, *split* or *fuse* operations may propagate all the way up the tree. Since allocating (or deallocating) nodes is usually much slower than just reading them, these transformation steps make a significant difference in the running time.

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

In this section we study the cost of update operations in terms of the number of *split* and *fuse* operations. We show that, with repect to this cost measure, 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, in the light of the analysis below, turn out to have constant amortized complexity even in the standard sense.

Theorem 7.3. *Consider a sorted sequence S organized as an (a,b)-tree with $b \geq 2a$. Assume that S is initially empty. Then, for any sequence of n insert or remove operations, the total number of split or fuse operations is $O(n)$.*

Proof. We give the proof for $(2,4)$-trees and leave the generalization to the reader (Exercise 7.16). We use the bank account method introduced in Sect. 3.5. The internal operations *split* and *fuse* are paid for by tokens; they 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 we can account for the cost of *balance* directly without amortized analysis. 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 which, (apart from the special case of a root of an empty sequence) occur temporarily 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 1. 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 1 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.

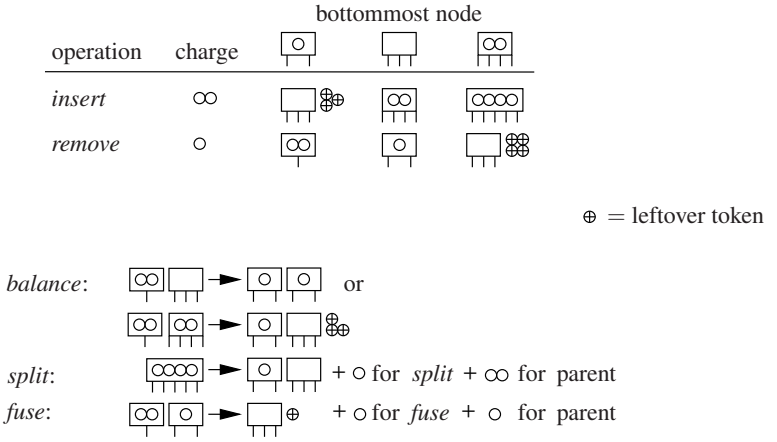


Fig. 7.14. The effect of (a,b)-tree operations on the token invariant. The *upper part* of the figure illustrates the addition or removal of a leaf. The two tokens charged for an insert are used as follows. When the leaf is added to a node of degree 3 or 4, the two tokens are put on the node. When the leaf is added to a node of degree 2, the two tokens are not needed, and the token on the node is also freed. The *lower part* illustrates the use of the tokens in *balance*, *split*, and *fuse* operations.

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 5 and results in a node of degree 3 and a node of degree 2. It also increases the degree of the parent by 1. The four tokens stored on the degree-5 node are spent as follows: One token pays for the *split*, one token is put on the new node of degree 2, 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 1 and a node of degree 3 or 4 and moves one child from the high-degree node to the node of degree 1. If the high-degree node has degree 3, two tokens are available and two tokens are needed; if the high-degree node has degree 4, four tokens are available and one token is needed. In either case, the available tokens are sufficient to maintain the token invariant.

A *fuse* operation fuses a degree-1 node with a degree-2 node into a degree-3 node and decreases the degree of the parent. Three tokens are available. We use one to pay for the operation and one to pay for the decrease in 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 of n update operations is $O(n)$, 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 $O(n/(b-2a+1))$ *fuse* or *split* operations.

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 7.17 (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 *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 $v.parent.c[i] = v$).

Exercise 7.18. Assume $b \geq 2a$. Show that in (a,b)-trees with parent pointers, *remove*$(h : Handle)$ and *insertAfter*$(h : Handle)$ can be implemented to run in constant amortized time.

***Exercise 7.19 (avoiding augmentation).** Assume $b \geq 2a$. Design an iterator class that allows one to represent a position in an (a,b)-tree that has no parent pointers. Creating an iterator is an extension of *locate* 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 7.20 (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 7.21 (optimal merging).** Section 5.3 discusses how to merge two sorted lists or arrays. Explain how to use finger search to implement merging of two sorted sequences in (optimal) 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., $t.size$ is the number of leaves in the subtree rooted at t. The kth 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 kth 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 kth element is reached. Figure 7.15 gives an example.

Exercise 7.22. 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 7.23. Explain how to determine the rank of a sequence element with key k in logarithmic time.

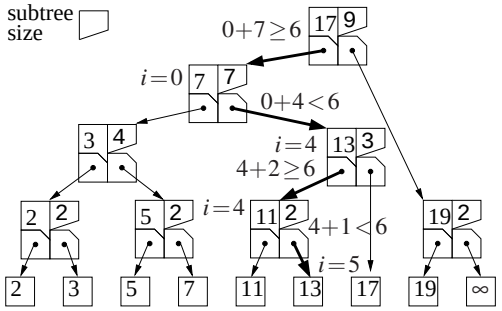


Fig. 7.15. Selecting the 6th smallest element from $\langle 2,3,5,7,11,13,17,19\rangle$ represented by a binary search tree. The **thick** arrows indicate the search path.

Exercise 7.24. 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 Parallel Sorted Sequences

Sorted sequences are difficult to parallelize for distributed-memory machines. Even on shared-memory machines, this is not easy. There has been a lot of work on sorted sequences that allow concurrent access [56, 134, 196]. However, it seems to be very difficult to achieve significant speedup compared with a tuned sequential implementation of a cache-friendly sorted sequence such as (a,b)-trees [291]. This is particularly difficult if there are a many *insert*, *remove*, and *update* operations and contention for the same keys.

If one needs only concurrent access, the (a,b)-trees introduced in Sect. 7.2 are a promising starting point [62, 134, 196], since their constant amortized update cost (see Sect. 7.4) requires concurrency control on only a constant number of tree nodes when averaged over all operations.

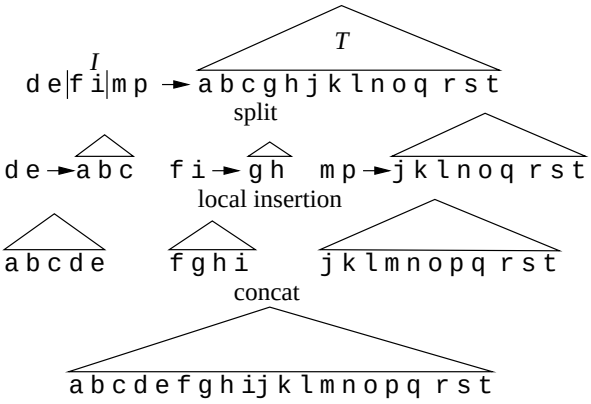


Fig. 7.16. Inserting $\langle d,e,f,i,m,p\rangle$ into $\langle a,b,c,g,h,j,k,l,n,o,q,r,s,t\rangle$ using three PEs.

We next discuss bulk operations on sorted sequences and show that simple algorithms can achieve sizable speedups in practice. The solution works for any sorted-sequence data structure with efficient support for concatenation (Sect. 7.3.1) and splitting (Sect. 7.3.2). We describe bulk insertion in detail. Bulk remove, bulk update, or arbitrary operation mixes are easy generalizations. One can also adapt the techniques described here to obtain algorithms for operations on two trees such as union, intersection, or difference [12, 46].

We first sort the insertion operations by their keys. Then, we split the sequence I of insertions and the sorted sequence T into corresponding pieces. We can then, independently and in parallel, perform the insertions in each piece of I into its corresponding piece of T. Concatenating the resulting sequences into a single sequence yields the desired result. Figure 7.16 gives an example. Figure 7.17 shows pseudocode for a parallel divide-and-conquer algorithm implementing the split–insert–concat approach. Not counting the cost of sorting I, we obtain a span of $\log k \log|T| + \frac{I}{k}\log|T|$; $\log k$ levels of recursion which take time $O(\log|T|)$ for splitting and concatenation in each level and time $O(\frac{|I|}{k}\log|T|)$ for the insertions. Again ignoring the time for sorting I, the overall work is $O(k\log|T|)$ for splitting and concatenation and $O(|I|\log|T|)$ for the insertions. Setting $k = p$, this yields a parallel execution time

$$O\left(\frac{|I|}{p}\log|T| + \log p \log|T|\right).$$

This result can be improved to $O((|I|/p)\log|T| + \log|T|)$ by splitting and concatenating with a more scalable parallel algorithm [12].

```
Procedure bulkInsertPar(T : SortedSequence, I : Array of Element, k : ℕ)
    assert I is sorted
    if k = 1 then                                                    // base case
        foreach e ∈ I do T.insert(e)
    else
        m := ⌊I/2⌋
        (T1, T2) := T.split(I[m])
        bulkInsertRec(T1, I[1..m], ⌈k/2⌉) ||                         // parallel recursion
        bulkInsertRec(T2, I[m+1..|I|], ⌊k/2⌋)
        T := concat(T1, T2)
```

Fig. 7.17. Task-parallel pseudocode for bulk insertion. The (presorted) insertion sequence I and the sorted sequence T are recursively split into k independent insertion tasks with an approximately equal number of insertions per task.

****Exercise 7.25.** Show, using a more careful analysis, that with an appropriate choice of k, the algorithm in Fig. 7.17 actually only needs parallel execution time $O((|I|/p)\log(|T|/|I|) + \log^2|T|)$. Hint: The *average* size of a subtree used for insertion is $|T|/k$. Argue that the overall work for insertions is maximized when all subtrees have average size.

7.7 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 may want to have a specialized implementation for small, fixed values of a and b that *unrolls*[6] all the inner loops.

Of course, the values of a and b are important. Let us start with the cost of *locate*. There are two kinds of operations that dominate the execution time of *locate*: Besides their inherent cost, element comparisons may cause branch mispredictions (see also Sect. 5.17). Furthermore, dereferencing pointers may cause cache faults. Exercise 7.9 indicates that element comparisons can be minimized by choosing a as a large power of two and $b = 2a$. Since the number of pointers that have to be dereferenced 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 $O(b)$ each. We obtain a logarithmic amortized cost for update operations as long as $b = O(\log n)$. A more detailed analysis (see Exercise 7.16) reveals 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 time a pointer is dereferenced, a cache fault will occur, 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.

[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.

Very large search trees are stored on disks. Under the name *B-trees* [34], (a,b)-trees are the workhorse of the indexing data structures in databases. Internal nodes of B-trees 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 3.31). 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 KB of data. Then two disk accesses are sufficient to *locate* any element in a sorted sequence that takes 16 GB 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 [135] and [228, Chaps. 2 and 14] for overviews. A good free implementation of external B-trees is available in STXXL [88]. For an internal memory version, consider `github.com/bingmann/stx-btree`.

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 $b = 2^k = 2a = \mathrm{O}(\log n)$, with a doubly linked list for the leaves, and augmented with parent pointers, 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. 5.12 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.

7.7.1 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

[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.

than only in the leaves. None of the STL data types supports efficient splitting or concatenation of sorted sequences.

LEDA [194] 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 (for the last three, see Sect. 7.8).

7.7.2 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.

7.8 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.

The first sorted-sequence data structure to support *insert*, *remove*, and *locate* in logarithmic time was AVL trees [5]. AVL trees are binary search trees which maintain the invariant that the heights of the subtrees of a node differ by at most one. Since this is a strong balancing condition, *locate* is probably a little faster than most competitors. On the other hand, AVL trees do *not* have constant amortized update costs. In each node, one has to store a "balance factor" from $\{-1,0,1\}$. 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 [7]. The generalization to (a,b)-trees and the amortized analysis in Sect. 3.5 come from [158]. There, it was also shown that the total number of splitting and fusing operations at the nodes of any given height decreases exponentially with height.

Splay trees [300] and some variants of randomized search trees [289] 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 time $O(\log(1/p))$. This can be shown to be asymptotically optimal for any comparison-based data structure. However, because of the large constants, this property leads to improved running times only for quite skewed access patterns.

Weight-balanced trees [243] 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 [49].

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 [162] are an elegant way to make sorted arrays dynamic. The idea is to accept some empty cells to make insertion easier. Bender et al. [39] 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 [39] that is *cache-oblivious* [116], i.e., it is cache-efficient on any two levels of a memory hierarchy without even knowing the size of the caches and cache blocks. The other ingredient is oblivious *static* search trees [116]; 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 $2^{2^{k-1}}$ subtrees of depth 2^{k-1}, allocating consecutive blocks of memory for them. The resulting $1 + 2^{2^{k-1}}$ subtrees of depth 2^{k-1} are stored recursively in the same manner. Static cache-oblivious search trees are practical in the sense that they can outperform binary search in a sorted array.

Skip lists [257] 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. 5.10. For example, we can obtain sorted sequences with w-bit integer keys that support all operations in time $O(\log w)$ [215, 319]. At least for 32-bit keys, these ideas lead to a considerable speedup in practice [87]. 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 minimum 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 [139]. Taking these ideas to the extreme results in *tries* [111], 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 search in a trie for a string of size s can be carried out in time $O(s)$.

We 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 [85]. 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 [99]. The third generalization is *searching many sequences* [66, 67, 216]. In this setting, there are many sequences, and searches need to locate a key in all of them or a subset of them.