3 Data structures

We will now describe a couple of basic data structures that will be used extensively throughout the book.

3.1 Dynamic range minimum queries

A *binary tree* is a tree with each internal node having at most two children. We denote by $\overleftarrow{v}$ and $\overrightarrow{v}$ the left-most and right-most leaves in the subtree rooted at v. For a tree node v, we denote by $\texttt{key}(v)$ the *search key* and by $\texttt{value}(v)$ the *value* associated with the key. A *binary search tree* is a binary tree with the following properties:

(a) each leaf stores a pair $(\texttt{key}(v), \texttt{value}(v))$;
(b) leaves are ordered left to right with increasing $\texttt{key}$; and
(c) each internal node v stores $\texttt{key}(v)$ equalling $\texttt{key}(\overrightarrow{v})$.

The total order in (b) enforces the implicit assumption that all keys are unique. A *balanced binary search tree* is a binary search tree on n pairs $(\texttt{key}, \texttt{value})$ such that each leaf can be reached with a path of length $O(\log n)$ from the root. Such a tree has $n - 1$ internal nodes. Observe that our definition follows the computational geometry literature and deviates slightly from the more common definition with the data stored in all nodes and not only in leaves.

While there exist dynamic balanced binary search trees that allow elements to be inserted and deleted, a semi-dynamic version is sufficient for our purposes. That is, we can just construct a balanced search tree by sorting the set of keys, assigning them to the leaves of an empty tree in that order with values initialized to ∞, and building internal nodes level by level by pairing consecutive leaves (see Exercise 3.2). Such a fully balanced tree can be built in linear time after the keys have been sorted. Now we can consider how to update the values and how to query the content.

LEMMA 3.1 *The following two operations can be supported with a balanced binary search tree $\mathcal{T}$ in time $O(\log n)$, where n is the number of leaves in the tree.*

$\texttt{update}(k, \texttt{val})$: *For the leaf w with* $\texttt{key}(w) = k$, *update* $\texttt{value}(w) = \texttt{val}$.
$\mathsf{RMQ}(l, r)$: *Return* $\min_{w\,:\,l \le \texttt{key}(w) \le r} \texttt{val}(w)$ (Range Minimum Query*).*
Moreover, the balanced binary search tree can be built in $O(n)$ time, given the n pairs $(\texttt{key}, \texttt{value})$ *sorted by component* $\texttt{key}$.

https://doi.org/10.1017/CBO9781139940023.005 Published online by Cambridge University Press

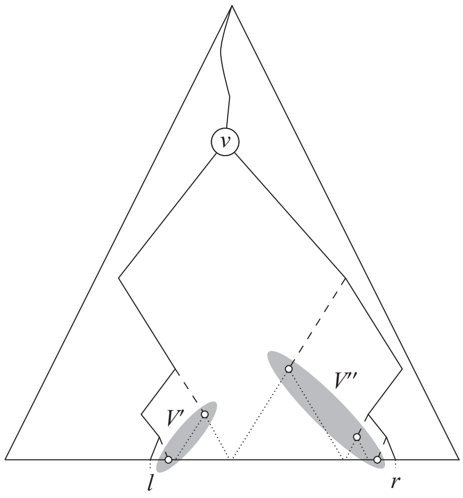


Figure 3.1 Answering a range minimum query with a binary search tree.

Proof Store for each internal node v the minimum among $\texttt{value}(i)$ associated with the leaves i under it. Analogously to the leaves, let us denote this minimum value by $\texttt{value}(v)$. These values can be easily updated after a call of an operation $\texttt{update}(k, \texttt{val})$; only the $O(\log n)$ values on the path from the updated leaf towards the root need to be modified.

It is hence sufficient to show that query $\mathsf{RMQ}(l, r)$ can be answered in $O(\log n)$ time. Find node v, where the search paths to keys l and r separate (it can be the root, or empty when there is at most one key in the query interval). Let $\texttt{path}(v, l)$ denote the set of nodes through which the path from v goes when searching for key l, excluding node v and leaf L where the search ends. Similarly, let $\texttt{path}(v, r)$ denote the set of nodes through which the path from v goes when searching for key r, excluding node v and leaf R where the search ends. Figure 3.1 illustrates these concepts.

Now, for all nodes in $\texttt{path}(v, l)$, where the path continues to the left, it holds that the keys k in the right subtree are at least l and at most r. Choose $vl = \min_{v' \in V'}(\texttt{value}(v'))$, where V' is the set of roots of their right subtrees. Similarly, for all nodes in $\texttt{path}(v, r)$, where the path continues to the right, it holds that the keys k in the left subtree are at most r and at least l. Choose $vr = \min_{v'' \in V''}(\texttt{value}(v''))$, where V'' is the set of roots of their left subtrees. If $L = l$ update $vl = \min(vl, \texttt{value}(L))$. If $R = r$ update $vr = \min(vr, \texttt{value}(R))$. The final result is $\min(vl, vr)$.

The correctness follows from the fact that the subtrees of nodes in $V' \cup V''$ contain all keys that belong to the interval $[l..r]$, and only them (excluding leaves L and R, which are taken into account separately). The running time is clearly $O(\log n)$. □

It is not difficult to modify the above lemma to support insertion and deletion operations as well: see Exercise 3.3.

3.2 Bitvector rank and select operations

A *bitvector* is an array $B[1..n]$ of Boolean values, that is, $B[i] \in \{0, 1\}$. An elementary operation on B is to compute

$$\mathtt{rank}_c(B, i) = |\{i' \mid 1 \leq i' \leq i, B[i'] = c\}|, \tag{3.1}$$

where $c \in \{0, 1\}$. Observe that $\mathtt{rank}_0(B, i) = i - \mathtt{rank}_1(B, i)$ for $B \in \{0, 1\}^*$. It is customary to use just $\mathtt{rank}(B, i)$ to denote $\mathtt{rank}_1(B, i)$.

Storing all values $\mathtt{rank}(B, i)$ as an array would take $O(n \log n)$ bits, which is too much for our later purposes. Before entering into the details on how to solve $\mathtt{rank}$ queries more space-efficiently, let us consider a motivating example on a typical use of such structure. Consider a *sparse* table $A[1..n] \in \{0, 1, 2, \ldots, u\}^n$ having k non-zero entries, where k is much smaller than n. Let $B[1..n]$ have $B[i] = 1$ iff $A[i] \neq 0$ and let $A'[1..k]$ have $A'[\mathtt{rank}(B, i)] = A[i]$ for $B[i] = 1$. We can now replace A that occupies $n \log u$ bits, with B and A' that occupy $n + k \log u$ bits plus the size of the data structure to support $\mathtt{rank}$ operations.

We shall soon see that $o(n)$ additional bits are enough to answer all $\mathtt{rank}$ queries in constant time. This solution is gradually developed in the following paragraphs. To simplify the formulas, we implicitly assume that all real numbers are rounded down to the nearest integer, unless explicitly indicated otherwise with, for example, the ceiling $\lceil\cdot\rceil$ operation.

Partial solution 1: Let us store each ℓth $\mathtt{rank}(B, i)$ as is and scan the rest of the bits (at most $\ell - 1$), during the query. We then have an array $\mathtt{first}[0..n/\ell]$, where $\mathtt{first}[0] = 0$ and $\mathtt{first}[i/\ell] = \mathtt{rank}(B, i)$ when $i \bmod \ell = 0$ ($/$ is here integer division). If we choose $\ell = (\lceil(\log n)/2\rceil)^2$, we need on the order of $n \log n/(\log^2 n) = n/(\log n)$ bits space for the array $\mathtt{first}$. We can answer $\mathtt{rank}(B, i)$ in $O(\log^2 n)$ time: $\mathtt{rank}(B, i) = \mathtt{first}[i/\ell] + \mathtt{count}(B, [\ell \cdot (i/\ell) + 1..i])$, where $\mathtt{count}(B, [i'..i])$ computes the amount of 1-bits in the range $B[i'..i]$.

Partial solution 2: Let us store more answers. We store inside each block of length ℓ induced by $\mathtt{first}$ answers for each kth position (how many 1-bits from the start of the block). We obtain an array $\mathtt{second}[0..\ell/k]$, where $\mathtt{second}[i/k] = \mathtt{rank}(B, [\ell(i/\ell) + 1..i])$, when $i \bmod k = 0$. This uses overall space $O((n/k) \log \ell)$ bits. Choosing $k = \lceil(\log n)/2\rceil$ gives $O(n \log\log n/(\log n))$ bits space usage. Now we can answer $\mathtt{rank}(B, i)$ in $O(\log n)$ time, as $\mathtt{rank}(B, i) = \mathtt{first}[i/\ell] + \mathtt{second}[i/k] + \mathtt{count}(B, [k \cdot (i/k) + 1..i])$.

Final solution: We use the so-called *four Russians technique* to improve the $O(\log n)$ query time to constant. This is based on the observation that there are fewer than $\sqrt{n}$ possible bitvectors of length $k - 1 = \lceil(\log n)/2\rceil - 1 < (\log n)/2$. We store a table $\mathtt{third}[0..2^{k-1} - 1][0..k - 2]$, which stores the answers for rank queries for all possible $k - 1 < (\log n)/2$ positions on all possible $2^{k-1} \leq \sqrt{n}$ block configurations, where the answer is an integer of size $\log(k) \leq \log\log n$ bits. This takes in total $O(\sqrt{n} \log n \log\log n)$ bits. Let c_i be the bitvector $B[k \cdot (i/k) + 1..k \cdot (i/k + 1) - 1]$.

We obtain the final formula to compute $\mathtt{rank}(B,i)$:

$$\mathtt{rank}(B,i) = \mathtt{first}[i/\ell] + \mathtt{second}[i/k] + \mathtt{third}[c_i][(i \bmod k) - 1]. \qquad (3.2)$$

Integer c_i can be read in constant time from the bitvector B: see Exercise 2.7.

Example 3.1 illustrates these computations.

Example 3.1 Simulate $\mathtt{rank}(B, 18)$ on $B = \mathtt{0110101100011101010110}\ldots$ assuming $\ell = 16$ and $k = 4$.

Solution

																		$i = 18$						
B		0	1	1	0	1	0	1	1	0	0	0	1	1	1	0	1	0	1	0	1	0	1	1	0...
first	0															9									
second	0				2				5				6				0				11				13

third	0	1	2
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	0	0
101	1	1	2
110	1	2	2
111	1	2	3

$$\begin{aligned}\mathtt{rank}(B,18) &= \mathtt{first}[18/16] + \mathtt{second}[18/4] + \mathtt{third}[\mathtt{010}][(18 \bmod 4) - 1]\\ &= \mathtt{first}[1] + \mathtt{second}[4] + \mathtt{third}[\mathtt{010}][1]\\ &= 9 + 0 + 1\\ &= 10\end{aligned}$$

THEOREM 3.2 *There is a bitvector representation using $o(n)$ bits in addition to the bitvector $B[1..n]$ itself, such that operation $\mathtt{rank}(B,i)$ can be supported in constant time, assuming a RAM model with computer word size $w = \Omega(\log n)$. This representation can be built in $O(n)$ time and $n + o(n)$ space.*

We will also need the inverse operation $\mathtt{select}_1(B,j) = i$ that gives the maximal i such that $\mathtt{rank}_1(B,i) = j$, that is, the position i of the jth bit set in B. This operation can also be supported in constant time, after building a similar but a bit more involved data structure than for $\mathtt{rank}$: Exercise 3.7 asks to prove the following claim.

THEOREM 3.3 *There is a bitvector representation using $o(n)$ bits in addition to the bitvector $B[1..n]$ itself, such that operations $\mathtt{select}_1(B,j)$ and $\mathtt{select}_0(B,j)$ can* *be supported in constant time, assuming a RAM model with computer word size $w = \Omega(\log n)$. This representation can be built in $O(n)$ time and $n + o(n)$ space.*

3.3 Wavelet tree

The *wavelet tree* generalizes `rank` and `select` queries to sequences from any alphabet size. Let us denote by $\mathtt{rank}_c(T, i)$ the count of characters c up to position i in T, and $\mathtt{select}_c(T, j)$ the position of the jth occurrence of c in T, for $T \in \Sigma^*$. Note that $\mathtt{rank}_c(T, \mathtt{select}_c(T, j)) = j$.

Let us first consider a naive representation of a string $T[1..n] \in \Sigma^*$ as σ binary strings $B_c[1..n]$ for all $c \in \Sigma$, such that $B_c[i] = 1$ if $T[i] = c$, otherwise $B_c[i] = 0$. Now, $\mathtt{rank}_c(T, i) = \mathtt{rank}(B_c, i)$. After preprocessing binary strings B_c for `rank` queries, we can answer $\mathtt{rank}_c(T, i)$ in constant time using $\sigma n(1 + o(1))$ bits of space.

We will next show a more space-efficient represenation that also reduces the problem of `rank` computation on general sequences to computation on binary sequences.

3.3.1 Balanced representation

Consider a perfectly balanced binary tree where each node corresponds to a subset of the alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. The children of each node partition the node subset into two. A bitvector B_v at node v indicates to which children each sequence position belongs. Each child then handles the subsequence of the parent's sequence corresponding to its alphabet subset. The root of the tree handles the sequence $T[1..n]$. The leaves of the tree handle single alphabet characters and require no space.

To answer query $\mathtt{rank}_c(T, i)$, we first determine to which branch of the root c belongs. If it belongs to the left ($c \leq \sigma/2$), then we recursively continue at the left subtree with $i = \mathtt{rank}_0(B_{\mathtt{root}}, i)$. Otherwise we recursively continue at the right subtree with $i = \mathtt{rank}_1(B_{\mathtt{root}}, i)$. The value reached by i when we arrive at the leaf that corresponds to c is $\mathtt{rank}_c(T, i)$.

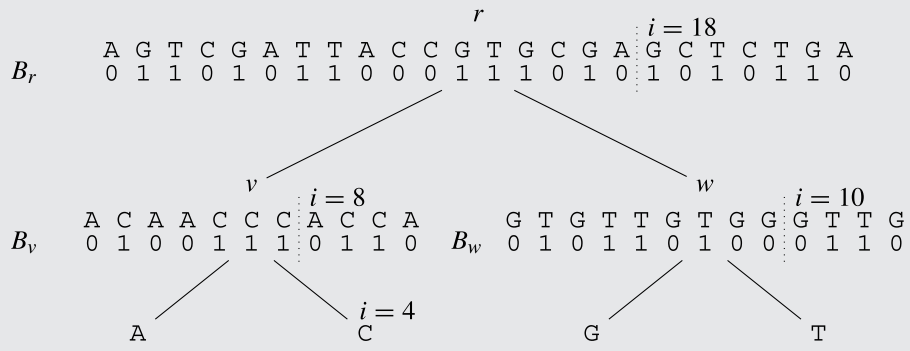
The character t_i at position i is obtained similarly, this time going left or right depending on whether $B_v[i] = 0$ or 1 at each level, and finding out which leaf we arrived at. Denote this operation $\mathtt{access}(T, i)$.

This hierarchical structure is called a *wavelet tree*. Example 3.2 illustrates `rank` and `access` operations. Note that string T can be replaced with its wavelet tree, and retain access to all its poistions.

Example 3.2 Visualize the wavelet tree of the string T = `AGTCGATTAC CGTGCGAGCTCTGA` and solve $\mathtt{rank}_{\mathtt{C}}(T, 18)$ and $\mathtt{access}(T, 18)$.

Solution

One can map the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ into $[1..4]$ by associating $\mathtt{A} = 1$, $\mathtt{C} = 2$, $\mathtt{G} = 3$, and $\mathtt{T} = 4$. Then $[1..2]$ ($\{\mathtt{A}, \mathtt{C}\}$) go to the left and $[3..4]$ ($\{\mathtt{G}, \mathtt{T}\}$) to the right, and so on. We obtain the following visualization, where the bitvectors and leaf labels form the wavelet tree.



For solving $\texttt{rank}_{\texttt{C}}(T, 18)$, we first compute $i = \texttt{rank}_0(B_r, 18) = 8$ as C goes left from the root node r. At node v, C goes right, so we compute $i = \texttt{rank}_1(B_v, 8) = 4$. We have reached the leaf, so this is our answer.

For solving $\texttt{access}(T, 18)$, we first compute $i = \texttt{rank}_{B_r[18]}(B_r, 18) = 10$ because $B_r[18] = 1$ indicates that the location is stored in the right subtree. Then we check that $B_w[10] = 0$, indicating that we should go left. To the left of w we have the leaf labeled G, so this is our answer.

One can construct the wavelet tree level by level, starting at the root. After marking $B_{\texttt{root}}[i] = 0$ if $T[i] \leq \sigma/2$ and $B_{\texttt{root}}[i] = 1$ otherwise, one can extract subsequence T' of T marked by 0-bits and send it to the left child handling alphabet interval $[1..\sigma/2]$, and subsequence T'' of T marked by 1-bits and send it to the right child handling alphabet interval $[\sigma/2 + 1..\sigma]$. After this T can be deleted, and construction can continue recursively to the next levels, creating leaf nodes on encountering subsequences consisting of a single character. At any moment, the space is bounded by $3n \log \sigma + O(\sigma \log n)$ for representing the set of distinct subsequences of T at two consecutive levels, for the bitvectors forming the wavelet tree, and for the pointers recording the shape of the tree. Creating rank structures separately for each node and dropping the subsequences, the wavelet tree occupies $n \log \sigma (1 + o(1)) + O(\sigma \log n)$ bits.

To improve the space requirement further, one can concatenate all bitvectors level by level and build just one rank structure. The tree shape can then be implicitly reconstructed from the way the alphabet is partitioned, and from the content of the bitvectors. We leave the details of navigating this concatenated bitvector on rank and access queries for the reader: see Exercise 3.9.

THEOREM 3.4 *The wavelet tree representation of string $T[1..n] \in \{1, 2, \ldots, \sigma\}^*$ occupies $n \log \sigma (1+o(1))$ bits and supports $\texttt{rank}_c(T, i)$, $\texttt{select}_c(T, i)$, and $\texttt{access}(T, i)$ operations in $O(\log \sigma)$ time, assuming a RAM model with computer word size $w = \Omega(\log n)$. The structure can be built in $O(n \log \sigma)$ time and bits of space.*

Proof See Exercise 3.10 for the solution for $\texttt{select}_c(T, i)$. The other operations were considered above. □

3.3.2 Range queries

The wavelet tree can be used for answering *two-dimensional range queries*. We are especially interested in the following counting query on string $T = t_1 \cdots t_n$:

$$\texttt{rangeCount}(T, i, j, l, r) = |\{k \mid i \leq k \leq j, l \leq t_k \leq r\}|. \tag{3.3}$$

Recall the binary search tree and the range minimum query studied in Section 3.1. One can associate the shape of the binary search tree with the shape of the wavelet tree, and consider the same $[l..r]$ range search. Recall the node sets V' and V'' and the leaf nodes labeled L and R in the proof of Lemma 3.1. The leaves under subtrees rooted at nodes of V' and V'' contain all characters c that appear in T such that $l < c < r$.

Remember that the $\texttt{rank}_l(T, i)$ operation maintains on each node v encountered when searching l an updated value of i: denote this by i_v. Similarly, we obtain j_v when simulating $\texttt{rank}_l(T, j)$ operation, as well as i_w and j_w when simulating $\texttt{rank}_l(T, i)$ and $\texttt{rank}_r(T, j)$ on nodes w encountered when searching r. For each $v' \in V'$ and $v'' \in V''$ we have thus i_v, j_v, i_w, and j_w computed, where v is the parent of v' and w is the parent of v''. Then $i_{v'} = \texttt{rank}_1(B_v, i_v - 1)$, $j_{v'} = \texttt{rank}_1(B_v, j_v)$, $i_{v''} = \texttt{rank}_0(B_w, i_w - 1)$, and $j_{v''} = \texttt{rank}_0(B_w, j_w)$. We obtain an $O(\log \sigma)$ time algorithm to answer the range counting query, as

$$\begin{aligned} \texttt{rangeCount}(T, i, j, l, r) = & \sum_{v' \in V'} j_{v'} - i_{v'} \\ & + \sum_{v'' \in V''} j_{v''} - i_{v''} \\ & + \texttt{lcount} + \texttt{rcount}, \end{aligned} \tag{3.4}$$

where $\texttt{lcount} = j_L - i_L$ if $L = l$, otherwise $\texttt{lcount} = 0$, and $\texttt{rcount} = j_R - i_R$ if $R = r$, otherwise $\texttt{rcount} = 0$.

One can also solve the $\texttt{rangeList}(T, i, j, l, r)$ operation that asks for a listing of all distinct characters inside $[l..r]$ appearing in $T[i..j]$ in lexicographic order. It is sufficient to continue from each $v' \in V'$ and $v'' \in V''$ down to the leaves, branching only to nodes w with $j_w - i_w > 0$. Labels of leaves reached occur at least once in $T[i..j]$. The path to each such reported leaf could be largely distinct from those for other reported leaves, so the running time is $O(d \log \sigma)$ for d being the size of the answer. An optimized search strategy yields $O(d \log(\sigma/d))$ time: see Exercise 3.11. For each such c, $l \leq c \leq r$, occurring at least once in $T[i..j]$, one can also compute the non-zero frequencies by

$$\texttt{freq}_c(T, i, j) = \texttt{rank}_c(T, j) - \texttt{rank}_c(T, i - 1). \tag{3.5}$$

We will also need an operation $\texttt{isRangeUnary}(T, i, j)$ that returns true if substring $T[i..j]$ consists of a run of only one character c. To support this operation, we can use $c = \texttt{access}(T, i)$ and return true if $\texttt{freq}_c(T, i, j) = j - i + 1$, otherwise return false.

We will later see that these four operations play vital roles in navigating certain implicit search trees.

COROLLARY 3.5 *The wavelet tree of string* $T[1..n] \in \{1,2,\ldots,\sigma\}^*$ *supports operations* `rangeCount`(T,i,j,l,r), `freq`$_c(T,i,j)$, *and* `isRangeUnary`(T,i,j) *in* $O(\log\sigma)$ *time, and operation* `rangeList`(T,i,j,l,r) *in* $O(d\log(\sigma/d))$ *time, where* d *is the size of the output.*

3.4 Literature

The range search with balanced binary search trees (Adelson-Velskii & Landis 1962; Bayer 1972) is typically presented as a listing problem in the computational geometry literature (de Berg *et al.* 2000, Chapter 5). We added here the aggregate operation of taking min over the elements in a range; one can verbatim change this to max or $\sum$, the latter leading to the *range counting* solution. For semi-infinite ranges one can obtain $O(\log\log n)$ time using *vEB trees* (van Emde Boas 1977) via a reduction (Gabow *et al.* 1984).

The $O(\log n)$ time solution we describe for dynamic range minimum queries is similar to the one presented by Arge *et al.* (2013). It is also possible to improve the time from $O(\log n)$ to $O(\log n/\log\log n)$ by using a balanced multiary tree in which every node has between $\log n/(4\log\log n)$ and $\log n/(2\log\log n)$ children instead of just 2. This will result in a tree of depth $O(\log n/\log\log n)$. In order to obtain the result, one needs to implement a local dynamic range minimum structure on $\log n/(2\log\log n)$ elements in every node that supports updates and queries in constant time. This is possible using the Q-heap (Fredman & Willard 1994), which requires the use of $o(n)$ bits of space in a precomputed table and allows predecessor queries in constant time and using the dynamic rank structure of Dietz (1989) to return the position of the predecessor in constant time too. This improved $O(\log n/\log\log n)$ time was only recently achieved (Brodal *et al.* 2011; Davoodi 2011) and is known to be the best possible by a lower bound proved in Alstrup *et al.* (1998).

Bitvector `rank` and `select` queries were first studied by Jacobson (1989) for the succinct representation of static trees supporting navigation. The constant time solutions in the RAM model described here are from Clark (1996) and Munro (1996), and the four Russians technique used inside the solutions is from Arlazarov *et al.* (1970). The construction time can be improved to $O(n/\log n)$ (Babenko *et al.* 2015). Wavelet trees were introduced by Grossi *et al.* (2003), although very similar concepts were implicitly known in computational geometry literature; the space-efficient improvement (Chazelle 1988) over *fractional cascading* (de Berg *et al.* 2000, Section 5.6) is almost identical to the two-dimensional range counting we covered (Mäkinen & Navarro 2007). Our description of `rank`, `select`, and wavelet trees follows closely the survey by Navarro & Mäkinen (2007). See Navarro (2012) for further extensions and applications of wavelet trees.

Exercises

3.1 Give an example of a perfectly balanced binary search tree storing eight (key,value) pairs in its leaves as described in Lemma 3.1. Give an example of a range minimum query for some non-empty interval.

3.2 Give a pseudocode for the algorithm to construct and initialize a balanced binary search tree given the sorted keys.

3.3 Recall how *red–black trees* work. Revisit the proof of Lemma 3.1 and consider how the tree can be maintained correctly updated for RMQ queries during the rebalancing operations needed if one adds the support for `Insert` and `Delete`.

3.4 Instead of taking the minimum among the values in Lemma 3.1 one could take a sum. If all leaves are initialized to value 1, what question does the operation analogous to RMQ answer?

3.5 Consider the sets V' and V'' in the proof of Lemma 3.1. The subtrees rooted at nodes in V' and V'' induce a *partitioning* of the set of characters appearing in the interval $[l..r]$ (see Figure 3.1). There are many other partitionings of $[l..r]$ induced by different subsets of nodes of the tree. Why is the one chosen in the proof the minimum size partitioning? Are there other partitionings that could give the same running time?

3.6 A *van Emde Boas tree* (vEB tree) supports in $O(\log\log n)$ time insertions, deletions, and *predecessor queries* for values in the interval $[1..n]$. A predecessor query returns the largest element i' stored in the vEB tree smaller than query element i. Show how the structure can be used instead of the balanced search tree of Lemma 3.1 to solve range minimum queries for semi-infinite intervals $(-\infty..i]$.

3.7 Prove Theorem 3.3. *Hint.* Start as in `rank` with $O(\log^2 n)$ size blocks, but this time on arguments of $\texttt{select}_1$. Call these *source blocks* and the areas they span in bitvector B *target blocks*. Define *long* target blocks so that there must be so few of those that you can afford to store all answers inside the corresponding source blocks. We are left with *short* target blocks. Apply the same idea recursively to these short blocks, adjusting the definition of a *long* target block in the second level of recursion. Then one should be left with short enough target blocks that the four Russians technique applies to compute answers in constant time. The solution to $\texttt{select}_0$ is symmetric.

3.8 Show how to reduce the preprocessing time in Theorem 3.2 and Theorem 3.3 from $O(n)$ to $O(n/\log n)$, by using the four Russians technique during the preprocessing.

3.9 Consider the wavelet tree in Example 3.2. Concatenate bitvectors B_r, B_v, and B_w. Give formulas to implement the example queries in Example 3.2 with just the concatenated bitvector. Derive the general formulas that work for any wavelet tree.

3.10 Consider the operation $\texttt{select}_c(A,j) = i$ that returns the position i of the jth occurrence of character c in string $A[1..n]$. Show that the wavelet tree with its bitvectors preprocessed for constant time $\texttt{select}_1(B,j)$ and $\texttt{select}_0(B,j)$ queries can answer $\texttt{select}_c(A,j)$ in $O(\log\sigma)$ time.

3.11 Show that the operation $\texttt{rangeList}(T,i,j,l,r)$ can be supported in $O(d\log(\sigma/d))$ time by optimizing the given search strategy. *Hint.* After finding the left-most element in the interval, go up until branching towards the second element occurs, and so on. Observe that the worst case is when the elements are equally distributed along the interval: see Section 8.1 for an analogous analysis.

3.12 Show how to efficiently implement the operation `rangeListExtended` (T, i, j, l, r) which returns not only the distinct characters from $[l..r]$ in $T[i..j]$, but also, for every such distinct character c, returns the pair $(\mathtt{rank}_c(T, i-1), \mathtt{rank}_c(T, j))$. *Hint.* Observe that the enumeration of the distinct characters uses rank queries on binary sequences that can also be used to compute the pairs of rank operations executed for the distinct characters.