

International Journal of Foundations of Computer Science
© World Scientific Publishing Company

ON-LINE SEAT RESERVATIONS VIA OFF-LINE SEATING ARRANGEMENTS*

JENS S. KOHRT and KIM S. LARSEN

*Department of Mathematics and Computer Science
University of Southern Denmark, Odense, Denmark
{svalle,kslarsen}@imada.sdu.dk
<http://www.imada.sdu.dk/~{svalle,kslarsen}>*

Received (Day Month Year)
Accepted (Day Month Year)
Communicated by (xxxxxxxxxx)

When reservations are made to for instance a train, it is an on-line problem to accept or reject, i.e., decide if a person can be fitted in given all earlier reservations. However, determining a seating arrangement, implying that it is safe to accept, is an off-line problem with the earlier reservations and the current one as input. We develop algorithms with optimal running time to handle problems of this nature.

Keywords: Channel-assignment problem; Seat reservation problem; On-line algorithms; Fairness.

1. Introduction

In Danish as well as other European long-distance train systems, it is very common to make reservations. Near weekends and holidays, almost all tickets are reserved in advance. In the current system, customers specify their starting and ending stations, and if there is a seat available for the entire distance between the two stations, a reservation is granted. Then the customer is given a car and seat number which uniquely specifies one seat in the train set. The problem of giving these seat numbers on-line has been studied extensively [9, 10, 8, 4, 3], and the conclusion is that no matter which algorithm is used, the result can be quite far from optimal. How far depends on the pricing policy. For unit price tickets, a factor of about two is lost, depending on more specific assumptions. If the price depends linearly on the distance, measured in number of stations, then the result can be much worse.

We give a very simple example of how mistakes are possible in this scenario. Much more advanced examples can be found in the literature cited above. In the

*A preliminary version of this paper appeared in the Eighth International Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, vol. 2748, pages 174–185, Springer-Verlag, 2003.

example, we assume the stations are numbered 1, 2, 3, and 4, and we assume that the train has only two seats, seat 1 and seat 2. The first reservation is (1, 2), and without loss of generality, we give the seat number 1. The next reservation is (3, 4). If we give seat 2 to this reservation, then the next reservation will be (1, 4), which we must reject, even though it could have been accommodated had we given seat 1 the second time as well. If, on the other hand, we give seat 1 to the reservation (3, 4), then we might get first (1, 3), which we can give seat 2, and then (2, 4), which we must reject. Thus, no matter which decision we make on the second reservation, we may accommodate fewer than possible, if we knew the entire future.

Because of these results, it is tempting to switch to a different system, where seat numbers are not given in response to a reservation, but instead announced later. Many people expect that soon we will almost all be equipped with PDAs (personal digital assistants) or just cell phones, so it will be practically feasible to send the seat number to the customer shortly before the train may be boarded. An electronic bulletin board inside the train could inform the remaining customers of their seat number. Notice that in both the example scenarios above, it would be possible to seat all customers, if seat numbers are not determined until after all reservations are made.

Computing a seating arrangement off-line is a well-known problem. The input to the problem is a sequence of requests for seat reservations of the form $(begin, end)$, where $begin$ and end indicates stations. The stations are assumed to be numbered consecutively from one up to the number of stations. We assume that the train travels through the stations in numerical order, so for all requests we assume that the number of the end station is strictly greater than the number of the begin station. A fixed number N of seats are available and they are numbered from one through N .

A seating arrangement is a function f from the sequence of requests into the set of seat numbers $\{1, \dots, N\}$. The arrangement is *feasible* if for all pairs of requests, (b, e) and (b', e') , where $f((b, e)) = f((b', e'))$, either $e \leq b'$ or $e' \leq b$, i.e., if two reservations are placed on the same seat, then one person gets off no later than at the station where the other person gets on. The objective of the off-line seating arrangement problem is to accommodate as many requests as possible.

The decision version of the off-line problem is equivalent to the channel-assignment problem [13] and both of these problems can be viewed as variants of coloring of interval graphs [14]. In [13], it is shown that the off-line problem can be solved in the optimal time $\Theta(n \log n)$ in the decision tree model, where n is the number of requests. The optimality follows by a reduction from the element uniqueness problem [11].

In the on-line seating arrangement problem, the requests are given one by one and the algorithm has to decide about each request before knowing the later requests. The papers [9, 10, 8, 4, 3] focus on the quality of on-line algorithms measured as the ratio of the number of requests for reservations which are accommodated by the on-line algorithm to the number of requests which are accommodated by an op-

timal off-line algorithm (optimal with regards to this quality measure; these papers are not concerned with computational efficiency).

The problem we consider is in some sense in between the on-line and off-line problems described above, since we wish to compute the final seating arrangement, but we must decide on-line for each reservation whether or not it can be accommodated. Here, a reservation can be accommodated if the inclusion of the reservation into the collection of already accepted reservations will still allow for a solution, given the number of seats available. Further, as in the on-line version of the problem, the off-line algorithm is given the reservation requests in a sequence and has to accept as many as possible under the two restrictions that the requests are treated in the order they are given, and that a request must be accepted whenever this is possible.

Thus, we want a data structure with an operation *insert*, which inserts a reservation into the data structure if the resulting collection allows for a solution using at most N seats, where N is a fixed constant. If not, then the request for a reservation should be rejected. We also want an operation *output*, which from the data structure extracts a seating arrangement. We assume that each reservation is accompanied by some form of identifier (reservation number, cell phone number, or similar) such that each customer can be notified regarding his or her allocated seat. The output must be sorted by increasing starting station. Finally, we want an operation *delete* that allows customers to cancel their reservation.

We provide a data structure where the running time of the operations are optimal in the pointer machine model [7]. Let n be the current number of reservations, and let p be the current number of different stations (which could be a lot smaller than n and also smaller than the number of possible stations). The time complexity of *insert* and *delete* is $O(\log p)$ and the time complexity of *output* is $O(n)$.

Given the motivating application for our data structure, it might be interesting to build pauses into the output operation such that it outputs data for the current station and then waits until the next station is reached, i.e., at a given station (possible shortly before the train arrives), all customers with reservations starting there are informed of their seat number. In such a scenario, our data structure allows us to perform insertions of reservation, provided that the starting station of the reservation is later than the current station, and the output would still be correct, i.e., it would be the same as if all reservations were made before the output operation was initiated. Similarly, deletions of reservations can be carried out when the starting station of the reservation has not yet been reached. The total time spent on outputting will still be $O(n)$, where n is the total number of intervals, which have been inserted and not deleted. The fact that this gradual outputting can be done efficiently may be even more interesting in non-train scenarios, if our algorithm is used to allow computers to reserve some resources for particular time intervals, e.g., in a variant of the channel-assignment problem.

Our structure is similar to segment trees (in [6], this data structure is reported

to have been described first in [5]) and dynamic segment trees [15]. Segment trees are data structures for storing intervals represented by their endpoints. This is similar to our problem since a reservation can be considered an interval where the begin and end station are the two endpoints. A segment tree is an extension of a standard balanced binary search tree. It is initialized by specifying a set of possible endpoints. Each possible endpoint is stored in a unique leaf ordered from left to right. Subsequently, intervals can be inserted into the segment tree under the restriction that endpoints for the intervals are chosen only from the set of possible endpoints from when the structure was initialized.

In contrast, to obtain optimal time complexities, we need to be able to insert new endpoints (stations) dynamically. Otherwise, in situations where there are sequences of stations where nobody gets on or off, we would arrive at a suboptimal complexity.

This can be handled by dynamic segment trees, but these are fairly complicated (which is not surprising because they solve a more involved problem). For the dynamic segments trees of [15], the time complexity of *insert* is $O(\log n)$ and the time complexity of *delete* is $O(a(i, n) \log n)$, where a is related to the inverse Ackermann function [1] and i is a constant. This function grows extremely slowly and can for all practical purposes be considered a constant. The time complexity is only amortized because the structure must be rebuilt occasionally. The space requirements are $O(n \log n)$. It may be possible to adjust dynamic segment trees to solve our problem. However, the problem scenarios are not comparable since dynamic segment trees must be able to answer stabbing queries, whereas we must be able to provide an efficient *output* operation and also efficiently disallow *insert* operations if and only if some stabbing query after the insertion would yield a set with a cardinality greater than N . In the main part of the paper, for simplicity, we refer to and compare with the better known segment trees.

2. The Algorithms

In this section, we follow the graph tradition and talk about intervals, endpoints, and colors instead of reservations, stations, and seat numbers, respectively. Thus, a reservation, which consists of a begin station and an end station, can be viewed as an interval with two endpoints, and instead of referring to a reservation being on a given seat, we say that the interval is given a color. The feasibility definition is then expressed in this setting as follows. A *coloring* of the intervals, which is a function from the sequence of intervals into the set of colors $\{1, \dots, N\}$, is feasible if for all pairs of intervals in the sequence, (b, e) and (b', e') , where (b, e) and (b', e') are given the same color, either $e \leq b'$ or $e' \leq b$.

We first discuss the simple datatypes used in our algorithms. *Intervals* have left and right endpoints, which we refer to as *begin* and *end*. When we simply refer to an endpoint, it can be either of the two. The intervals are closed to the left and open to the right. The only reason for this choice is ease of terminology. It is convenient that intervals are disjoint if and only if the corresponding reservations can be placed on

the same seat (which rules out making the intervals closed). At the same time it is convenient that the union of a number of consecutive intervals is again an interval (which rules out making the intervals open). The concrete choice is arbitrary, i.e., the intervals could just as well have been open to the left and closed to the right.

Intervals may also have a *color*. If necessary, we assume that intervals are also equipped with a unique identifier such that otherwise identical intervals can be distinguished.

The data structure we propose is a binary tree for storing intervals. The leaves represent the set of all the different endpoints which have been used. They appear in the leaves in sorted order from left to right. To be precise, if we consider all the intervals which have been inserted into the structure at a given time, the set of different endpoints from these intervals is stored in the leaves—one endpoint in each leaf, i.e., the number of leaves equals the cardinality of the set of endpoints from the intervals. Even if the same endpoint has been used more than once in the sequence of intervals, it only appears in one leaf.

The tree is built from nodes which contain the following information: a *left* and *right* reference to the left and right subtrees, respectively, and the attribute *cover*, which stores the interval covered by a node. For a leaf node, the cover interval is the interval from the endpoint of the leaf to the endpoint of the next leaf to the right, and for an internal node, this is the union of all the intervals of the leaves in its subtree. At any leaf node, the intervals which begin or end at the endpoint of the leaf are stored in the attributes *BeginList* and *EndList*, respectively.

We need to be able to determine efficiently whether or not we can fit in a given new interval, i.e., if we add this interval to the collection of intervals accepted so far, is it still possible to find a feasible coloring?

Let us consider the interval $[a, b]$ and let $x_1 < \dots < x_m$ be all the endpoints from previous accepted intervals which fall in between a and b . A necessary and sufficient condition in order to fit in $[a, b]$ is that among $[a, b]$ together with all previously accepted intervals, there are at most N intervals overlapping each of the cover intervals $[a, x_1), [x_1, x_2), \dots, [x_m, b]$ (see the correctness section). If a is not an endpoint in the structure at this time, $[x_0, x_1)$ is used instead of $[a, x_1)$, where x_0 is the largest endpoint smaller than x_1 . Similarly, $[x_n, x_{n+1})$ may be used instead of $[x_m, b]$. In any case, these intervals are cover intervals of subsequent leaf nodes. We cannot afford to check all these subintervals separately, but fortunately with our tree representation, we can do better. We refer to the number of accepted intervals overlapping one of the small subintervals, $[x_i, x_{i+1})$, as the *density* of this interval.

To verify this property efficiently, we introduce local values, k and Δk , in the nodes of the tree. Our manipulation of these values will ensure that the density information that we need can always be computed efficiently.

We can explain the intuition behind k and Δk from a static equivalent of our structure. In the dynamic case, these properties no longer hold, and k and Δk can only be understood as a means of maintaining a certain invariant which we will

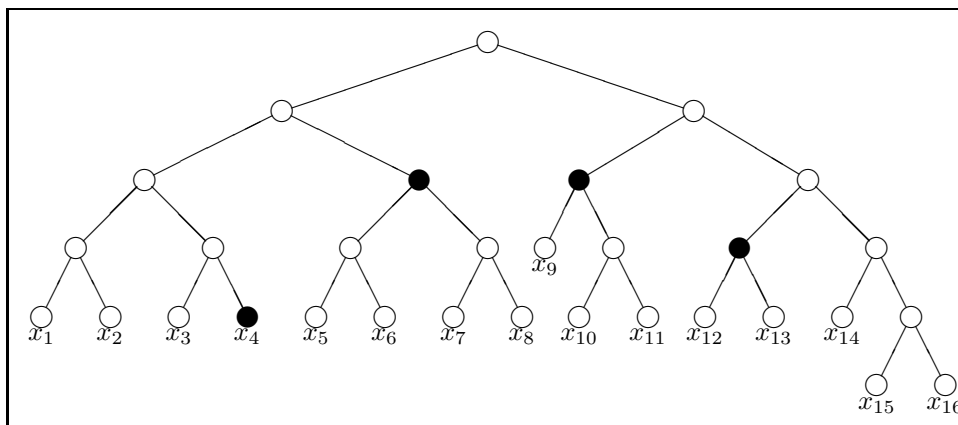


Fig. 1. An example of where Δk -values are incremented. This is done at all the filled nodes. The inserted interval is $[x_4, x_{14})$. Notice that for any leaf x_i where $4 \leq i < 14$, the Δk -value is incremented at exactly one node from the root of the tree to x_i . For all other x_i , all nodes from the root of the tree to x_i keep their Δk -values.

discuss later.

In the dynamic case, we do not store endpoints (stations) before they are used in some request (reservation). Now imagine instead that we build a tree where all possible endpoints are stored consecutively in the leaves initially, after which we move on to process intervals. Assume that we insert $[a, b)$. Then we would increment the Δk -value of exactly those nodes where the cover interval is contained in $[a, b)$ and no node closer to the root has that property. An example of this is given in Fig. 1.

The effect of these increments is that for any path from the root to some leaf, storing some endpoint x_i , $a \leq x_i < b$, exactly one node has its Δk -value incremented. We later argue that only a logarithmic number of nodes need to be visited. Furthermore, we maintain that the k -value of a node is the maximal sum of Δk -values from that node down to a leaf in its subtree. Thus, if we think of an interval as being *registered* at the nodes where we increment the Δk -values, the k -value of a node is the maximal density of positions in the node's subtree with regards to intervals registered in that subtree.

Now, to determine whether or not an interval $[a, b)$ can be accepted, we must check whether or not the inclusion of the interval will increase the density of any of the small subintervals, e.g., $[x_i, x_{i+1})$, to more than N . This is done by recursive checks down the tree as follows. Initially, we assume that we have N colors available (corresponding to N free seats). We adjust this number as we discover non-zero Δk -values on our way down the tree. There are two non-trivial cases. If the cover interval of the node we are currently at is contained in the interval we wish to insert, $[a, b)$, then the insertion of $[a, b)$ will increase the density by one for all the small subintervals under this node. The interval $[a, b)$ can therefore be included if

and only if the number of colors available are at least one more than the current maximal density (stored in the k -value of the node). If the cover interval is not contained in $[a, b)$, but overlaps it, then all the registered intervals at this node (counted in the Δk -value) overlap $[a, b)$; we reduce the number of colors available by Δk and proceed recursively to the subtrees.

When switching to the dynamic case, we are forced to move Δk -values around when we rebalance the tree. However, we maintain the two properties which we used to implement the density check described above. We define the Δ -length of a path from a node to a leaf as the sum of all Δk -values on that path. Now, the properties are the following:

- (1) The Δ -length of any path from the root to a leaf is exactly the density of the cover interval of that leaf.
- (2) The k -value of a node is the maximum Δ -length of any path from that node to any leaf in its subtree.

As a basis for our data structure, we use a worst-case logarithmically balanced search tree such as a red-black tree [12] or an AVL-tree [2]. This means that in addition to the attributes for tree nodes described above, attributes appropriate for rebalancing should also be present, but since the exact choice of tree is irrelevant, we just assume that the necessary attributes are present.

For clarity, we assume that the starting point is a leaf node covering the interval $-\infty$ to ∞ with $k = \Delta k = 0$ and empty *BeginList* and *EndList*.

To ensure that the two demands regarding k and Δk , as given above, are met, we initialize the Δk -values of new leaf nodes to zero. When inserting a new interval into the structure, we increment the Δk -value of exactly one node on any path from the root node to a leaf, the cover interval of which intersects the new interval. All other nodes maintain their Δk -values. Subsequently, we update the k -values bottom-up. The algorithm for insertion is given in Fig. 2. The procedure *insert* is called with the root of the tree as its first argument and the interval to be inserted as its second argument. We use “attribute” notation, so if x is an interval with attributes *begin* and *end* (the endpoints of the interval), then $x.begin$ and $x.end$ are used to denote these. Similarly, leaf nodes have two lists of intervals called *BeginList* and *EndList*. We assume that those are equipped with standard operations such as *append*, so we can write, e.g., $n.BeginList.append(x)$ to mean “append x to *BeginList* in node n ”.

As a first step in the *insert* procedure, we check if the new interval fits in, as described above. If it cannot, no further action is taken. In the motivating application, this is where the customer would be informed that the reservation is rejected. If, on the other hand, we get a positive response, we proceed to inserting the two endpoints of the interval. If an endpoint already exists, we merely *append* the interval to the appropriate list in the leaf of intervals beginning (or ending) at the endpoint stored in the leaf. If the endpoint is new, we must first introduce a new leaf, and possibly rebalance the tree. As the final step of an insertion, we update

```

proc insert(tree: Node, x: Interval)
  if okToInsert(tree, x, N) then
    insertEndpoint(tree, x.begin, true, x)
    insertEndpoint(tree, x.end, false, x)
    updateDensity(tree, x, 1)

func okToInsert(n: Node, x: Interval, c: Integer): Boolean
  if n.cover  $\cap$  x =  $\emptyset$  then
    return True
  else if n is a leaf or n.cover  $\subseteq$  x then
    return c  $\geq$  n.k + 1
  else
    c'  $\leftarrow$  c - n. $\Delta$ k # Calculates the number of colors left
    return okToInsert(n.left, x, c') and okToInsert(n.right, x, c')

proc insertEndpoint(tree: Node, b: Real, beginInterval: Boolean, x: Interval)
  n  $\leftarrow$  findLeaf(tree, b) # Finds leaf with maximal a such that a  $\leq$  b
  if n.cover.begin  $\neq$  b then
    split(n) # Splits n as described in the text
    n  $\leftarrow$  n.right
    rebalance(n) # Rebalances the tree bottom-up if necessary
  if beginInterval then n.BeginList.append(x) else n.EndList.append(x)

proc updateDensity(n: Node, x: Interval, d: Integer)
  if n.cover  $\subseteq$  x then
    n. $\Delta$ k  $\leftarrow$  n. $\Delta$ k + d
    n.k  $\leftarrow$  n.k + d
  else
    if n.left.cover  $\cap$  x  $\neq$   $\emptyset$  then
      updateDensity(n.left, x, d)
    if n.right.cover  $\cap$  x  $\neq$   $\emptyset$  then
      updateDensity(n.right, x, d)
    n.k  $\leftarrow$  max(n.left.k, n.right.k) + n. $\Delta$ k

```

Fig. 2. The insert operation.

the density information by incrementing Δk -values (notice that for now we only call `updateDensity` with parameter `d` equal to one). This is analogous to the check performed in `okToInsert`.

With slightly more complicated code, it is possible to combine searches down

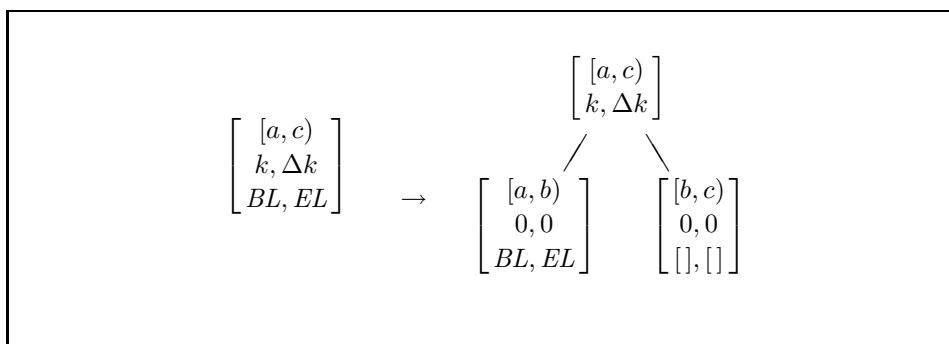


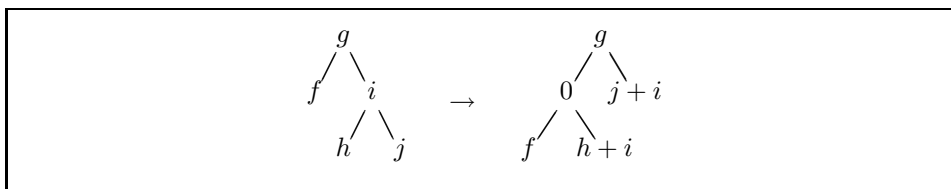
Fig. 3. A split operation performed on a leaf initially containing the interval $[a, c)$. In the nodes, the first line shows the *cover interval* and the second line shows the k -value and Δk -value of the node. The third line shows the *BeginList* and *EndList* of leaf nodes. The new endpoint b is inserted.

the tree. However, this will only improve the complexity by a constant factor. For readability, we have divided it up, so that we first check whether the insertion is at all possible, then we insert the endpoints (if they are not already present) and update the corresponding *BeginList* and *EndList*, and as the last step we update the counters.

With regards to the details of inserting a new leaf and rebalancing, a local operation is performed at the leaf where the search ends. The setting of the attributes in the new node is shown in Fig. 3, where it is demonstrated how one leaf is replaced by one internal node and two leaves. Note that the leaf before the operation represents the point a , so all intervals beginning or ending at that leaf are stored in *BL* and *EL*, respectively. After the operation, the left-most leaf represents the point a , so these *BL* and *EL* lists are stored there. The right-most leaf represents the new point b and therefore initially does not have any associated intervals. In the code segments, we use `split(n)` to denote this local operation at the leaf n where the search ends.

After this change, the tree may need rebalancing. This is done differently for different balanced tree schemes. However, we only assume that it is done bottom-up by at most a logarithmic number of local constant-sized transformation on the search path. Such transformations on a search tree can always be expressed as a constant number of rotations. In Fig. 4, we show how attributes should be set for Δk -values in connection with a left rotation. A right rotation is similar. In the code segments, `rebalance(n)` denotes the bottom-up rebalancing from node n .

Note that the new k -values can be calculated using the Δk -values, and the new *cover* values for the two internal nodes of the operation can be recomputed using their children. Notice also that the two properties regarding density information are preserved: The sum of Δk -values on any root to leaf path is unaltered by the rotation and as already remarked, the k -values of the nodes involved can be

Fig. 4. A left rotation with old and new Δk -values shown.

```

proc delete(tree: Node, x: Interval)
  updateDensity(tree, x, -1)
  deleteEndpoint(tree, x.begin, true, x)
  deleteEndpoint(tree, x.end, false, x)

proc deleteEndpoint(tree: Node, b: Real, beginInterval: Boolean, x: Interval)
  n ← findLeaf(tree, b) # Finds leaf containing the endpoint b
  if beginInterval then n.BeginList.remove(x) else n.EndList.remove(x)
  if n.BeginList.isEmpty() and n.EndList.isEmpty() then
    delete(n) # Deletes n as described in the text
    rebalance(n) # Rebalances the tree bottom-up if necessary

```

Fig. 5. The delete operation.

recomputed.

The considerations for *delete* are similar. We must update the density information by deleting the interval, we must remove the actual reservation from the two leaves, and we must delete the endpoints if no other intervals share them. These actions reverse actions taken during an *insert*. The *delete* operation is shown in Fig. 5. In Fig. 6, we show how a node is removed from the tree in the case where no other intervals share the endpoint. Notice how the updates to the Δk -values preserve the invariants. For the first case, where the node to be deleted is a left child of its parent, b must be changed to a c on the path from the point of deletion up towards the root, until the procedure reaches the root or a node which has the deleted node in its right subtree. From that node, the b 's must also be changed to c 's on the path down to the predecessor of the deleted node (the node containing $[a, b)$ before the update). In the code segments, we refer to this operation at node n as `delete(n)`.

As for insertion, rebalancing is a matter of carrying out a number of rotations, so the details given for insertions cover this case as well.

Finally, the *output* operation is shown in Fig. 7. We assume that we have a

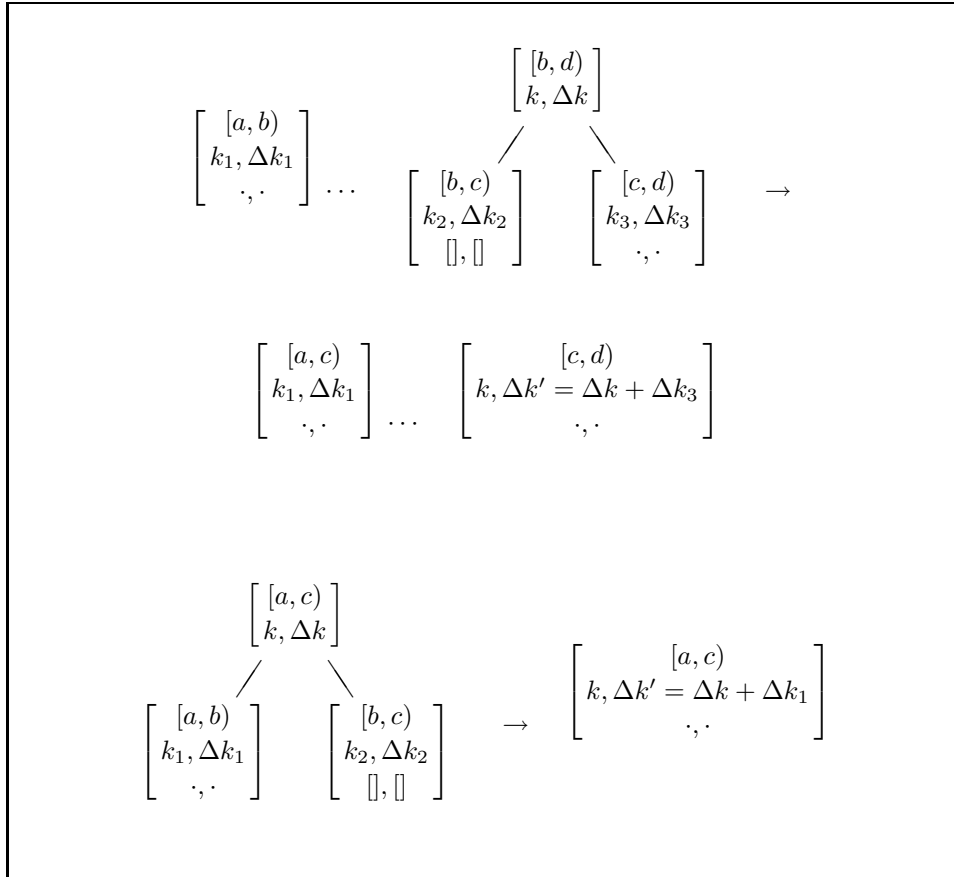


Fig. 6. A delete operation performed on a node with the cover interval $[b, c]$. There are two cases depending on whether the node to be deleted is the left or right child of its parent.

high-order operation, “using in-order”, to give us an in-order traversal of a tree. Alternatively, we can use a tree implementation where leaves are linked, so we can traverse them directly.

Finally, we note that in an actual implementation, some of the values we use can be computed rather than stored.

First, it is only necessary to store the k -values in the nodes, since the Δk -value for any node n can be calculated as $n.\Delta k = n.k - \max(n.left.k, n.right.k)$.

Second, it is sufficient to store the starting point of the cover intervals in the nodes. The other endpoint can be computed as we traverse the path. This would also eliminate the need for the traversal down towards the predecessor of a deleted node to change b 's to c 's.

```

proc output(tree: Node)
  s ← new Stack of N Colors
  # Optional wait until first station is reached can be inserted here
  for each Leaf v in tree using in-order do
    for each Interval x in v.EndList do
      s.push(x.color)
    for each Interval x in v.BeginList do
      x.color ← s.pop()
      print x
  # Optional wait until next station is reached can be inserted here

```

Fig. 7. The output operation.

3. Correctness and Complexity

The goal is to be able to seat all the customers and to accept insertions if and only if a seating arrangement can still be found if the insertion is accepted; and of course to carry out these operations efficiently. In terms of colors, we want to color each interval with one of the N colors (seats) such that no two overlapping intervals (reservations) receive the same color. For reference, the equivalence between such a coloring being obtainable and the density at all small subintervals being at most N is a consequence of interval graphs being perfect [14].

3.1. Correctness

Regarding correctness, there are three essential properties our structure should have. First, it should allow an insertion if and only if the resulting set of intervals can be colored under the constraints described above. Second, a deletion should correctly undo an insertion. Third, a legal coloring using at most N colors should be printed by the outputting procedure.

Regarding the first point, we claim that for any path from the root node to a leaf node, its Δ -length is exactly the same as the number of intervals inserted into the tree which intersect the cover interval of the leaf node, i.e., the density of the cover interval of the leaf. Furthermore, we claim that for any node, its k -value is the maximum Δ -length of a path to a leaf in its subtree. This is true because the insertion and the deletion of an interval ensures it and rotations preserve it.

An insertion of an interval ensures it by incrementing Δk in nodes such that their *cover* intervals are disjoint while together covering the inserted interval exactly and furthermore updating the k -values bottom up. Similarly for deletions. Rotations preserve it by ensuring that Δk -values remain associated with the correct intervals and recomputing the k -values based on the Δk -values.

When deciding whether or not the insertion of an interval is possible, *okToInsert* is used. By using the Δk -values, this function keeps track of how many colors are left in the recursion on the way to the bottom of the tree. An insertion is only accepted if it will not increase the maximum Δ -length from the root of the tree to more than the allowed number of colors.

Regarding the second point, deletion preserves the properties involving Δ -lengths and k -values, as described under insertion in the text above, and removes the interval from the lists in the leaves. If this deletion removes the last occurrence of an endpoint, the node representing that endpoint is removed, using the transformation in Fig. 6, which correctly updates the k -values and Δk -values.

Regarding the third point, we must argue that we output a legal coloring which means that we use at most N colors and no two overlapping intervals receive the same. The fact that no two overlapping intervals receive the same color is ensured by the stacking mechanism where the color is simply removed from the stack of available colors when it is used for an interval and it is not pushed onto the stack again until that interval has ended. The fact that we use at most N colors follows from the fact that the number of colors in use (the ones which are not on the stack) is exactly the density at the given point.

3.2. Complexity

If the underlying search tree guarantees time $O(\log p)$ searches and rebalancing, where p is the number of leaves (which is the same as the number of different endpoints), then *insertEndpoint* is also completed in $O(\log p)$ steps.

Regarding *updateDensity*, the argument for its complexity is similar to the corresponding argument for segment trees. At a first glance, it seems that the searching down the tree could split into many different paths. However, we argue that this is not the case.

In general, the search may stop (the first if-part) or continue (the else-part) either to the left or to the right, or possibly in both directions. For a number (possibly zero) of steps, we may from each node just continue down one of the two paths. Then at some node u , we may have to continue down both of them. We argue that there are no further real splits off the two search paths from that point on.

Let us consider the search down the left-most path. At the left child of u , we know (since there was also a search splitting off to the right) that the interval to be inserted covers the right-most point in our subtree. This is the essential property (we refer to it as the right-cover property), and it will be maintained on the rest of the search down the path.

At any node on this path, starting with the left child of u , if we continue down to our left child, then the recursive call to the right child will fall into the if-case and therefore terminate immediately because of the right-cover property. At the same time, the right-cover property will hold for the search to the left. If there is no

search to the left, but only to the right, the right-cover property also clearly holds in that case.

The analysis for *okToInsert* is similar to *updateDensity*, except that instead of checking directly before calling, we use an additional recursive call when deciding whether the cover interval of a node intersects the interval to be inserted.

For deletion, the argument is similar. However, we assume that the user reservation encodes a pointer to the reservation. The reservations stored in the *BeginLists* and *EndLists* are kept in a doubly-linked list such that they can be removed in constant time.

The work of *output* consists of a linear time traversal of the nodes of the tree which is done in time $O(p) \subseteq O(n)$, where p is the number of different endpoints used in the intervals, plus some constant work per interval which is then also $O(n)$.

Finally, the space requirements are $\Theta(n)$: the procedure *insertEndpoint* uses constant extra space per interval, and the procedure *updateDensity* only modifies integers already present in the structure.

3.3. *Optimality*

We consider the lower bound of the time complexity of the total process of inserting n intervals and outputting the result. If we only measure the running time in terms of the size of the input, n , then $O(n \log n)$ is an upper bound on the running time of the algorithm, and in the worst-case, time $\Omega(n \log n)$ is also necessary (see argument and reference below), so the time complexity of the problem is in $\Theta(n \log n)$.

We consider a more detailed analysis where we also include the number p of different endpoints. If p is $O(\log \log n)$, for instance, then the running time of our algorithm is $O(n \log \log n)$, and we would like to prove that this is optimal.

Clearly time $\Omega(n)$ is required to output the result. If, as we do, *output* is provided in time $O(n)$, *insert* requires time $\Omega(\log n)$ in the worst-case, in the cases where $p \in \Theta(n)$. Otherwise, we can solve the off-line problem in time $o(n \log n)$, and this has been proven impossible in the decision tree model in [13] by a simple reduction from the well-known element uniqueness problem [11], which is known to require time $\Theta(n \log n)$.

However, this only settles optimality for $p \in \Theta(n)$. We now assume that $p \in o(n)$ and argue that also in this case is the result optimal.

Let us first consider the following sorting problem: we are given a sequence of n distinct objects x_1, x_2, \dots, x_n , equipped with keys of which $p \in o(n)$ are distinct. We argue that in the decision tree model, the time to sort such sequences is $\Omega(n \log p)$. By sorting, we here mean outputting the objects in an order such that the keys of the objects are nondecreasing.

First, we obtain a lower bound on the number of possible outputs. We can think of the number of different ways we can place the x_i 's in p distinct boxes under the restriction that none of them may be empty. We first remove p objects with distinct keys from the sequence, placing them in each their box, thereby removing

the restriction. The remaining $n - p$ objects can be placed in the p different boxes in p^{n-p} different ways. The number of binary comparisons we would have to use in the worst-case to choose correctly between p^{n-p} different possible outputs is $\log(p^{n-p})$, assuming that we can balance our decision tree perfectly; otherwise it only gets worse. Now, $\log(p^{n-p}) = (n - p) \log p \in \Omega(n \log p)$, since $p \in o(n)$.

As a simple corollary, n intervals with at least p different endpoints cannot in general be sorted on starting point faster than $\Omega(n \log p)$.

However, this sorting problem can be solved using the data type discussed in this paper. Let $N = n$ so that all intervals will fit, use *insert* to insert each interval one at a time, and *output* to obtain the result. Hence, since the sorting problem requires time $\Omega(n \log p)$, the problem in this paper must also require time $\Omega(n \log p)$.

Note that even though $p \in o(n)$, the lower bound on the running time could still be $\Omega(n \log n)$. This happens for instance when $p \in \Theta(\sqrt{n})$, since $\Omega(n \log \sqrt{n}) = \Omega(n \log n)$. In those cases, the upper bound is of course also $O(n \log n)$.

4. Concluding Remarks

Without making the data structure more complicated, it is possible to make some minor extensions.

As presented here, we use a constant number N as the number of seats available. It would not be a problem to make this value dynamic, as long as it is never changed to a value smaller than the k -value of the root of the tree, i.e., the number of seats which currently are necessary in order to accommodate all reservations.

Furthermore, the intervals we consider are all closed to the left and open to the right. This can easily be extended to the general case as in [15], where either side may be open or closed, by using alternately open and closed intervals in the leaves of the structure: $(-\infty, a_1)$, $[a_1, a_1]$, (a_1, a_2) , $[a_2, a_2]$, \dots

In some special cases, it is also straight-forward to implement *split* and *join* operations on the tree. If we for *split* require that no intervals in the tree contain the splitting point inside the interval, and for *join* require that the intervals in the two trees do not intersect each other, then both operations can be implemented in $O(\log p)$ time.

In the seat reservation scenario, it is natural to ask if the techniques can be extended to include group reservations. Here, we interpret group reservations to mean that an additional requirement is imposed upon the seat allocation algorithm, namely that the members of the group must be assigned consecutive seat numbers (colors).

The answer to this question is negative as the following example shows. We give the reservations $[1, 5)$, $[4, 8)$, and then the two group reservations, both of size $n - 2$, where n is the number of seats, $[2, 3)$ and $[6, 7)$. Clearly the maximal density is at most $n - 1$. However, the reservation $[1, 8)$ cannot be placed, which is easily seen by considering all possible placements, up to symmetry, of the first four reservations.

As a more general remark, it is important to notice that we do not need to

assume that the stations which are used are numbered from 1 through p . In fact, we do not even need to assume that they are integers. One can think of the stations as floating point numbers. One could consider a less dynamic version of the problem and assume that stations are numbered from 1 through p , treating p as a constant. This would make it possible to obtain different theoretical results and better results in practice, in the cases where p really is small. However, the results would be less general and therefore not necessarily as easily applicable to other problems, such as the channel-assignment problem. The theoretical treatment would also be entirely different, since if elements are known to be from a small interval of integers, many problems become computationally much easier.

Acknowledgments

The authors would like to thank the anonymous referees for many constructive comments and very careful reading of our manuscript, clearly beyond what you can normally expect.

This work was supported in part by the Danish Natural Science Research Council (SNF) and in part by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

References

1. Wilhelm Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99:118–133, 1928.
2. G. M. Adel'son-Vel'skiĭ and E. M. Landis. An Algorithm for the Organisation of Information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962. In Russian. English translation in *Soviet Math. Doklady*, 3:1259–1263, 1962.
3. Eric Bach, Joan Boyar, Leah Epstein, Lene M. Favrholt, Tao Jiang, Kim S. Larsen, Guo-Hui Lin, and Rob van Stee. Tight Bounds on the Competitive Ratio on Accommodating Sequences for the Seat Reservation Problem. *Journal of Scheduling*, 6(2):131–147, 2003.
4. Eric Bach, Joan Boyar, Tao Jiang, Kim S. Larsen, and Guo-Hui Lin. Better Bounds on the Accommodating Ratio for the Seat Reservation Problem. In *Sixth Annual International Computing and Combinatorics Conference*, volume 1858 of *Lecture Notes in Computer Science*, pages 221–231. Springer-Verlag, 2000.
5. J. L. Bentley. Solutions to Klee's Rectangle Problems. Technical report, Carnegie-Mellon University, 1977.
6. Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.
7. Peter van Emde Boas. Machine Models and Simulations. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 1, pages 1–66. Elsevier Science Publishers, 1990.
8. Joan Boyar, Lene M. Favrholt, Kim S. Larsen, and Morten N. Nielsen. Extending the Accommodating Function. *Acta Informatica*, 40(1):3–35, 2003.
9. Joan Boyar and Kim S. Larsen. The Seat Reservation Problem. *Algorithmica*, 25(4):403–417, 1999.
10. Joan Boyar, Kim S. Larsen, and Morten N. Nielsen. The Accommodating Function: a

- generalization of the competitive ratio. *SIAM Journal on Computing*, 31(1):233–258, 2001.
11. David P. Dobkin and Richard J. Lipton. On the Complexity of Computations under Varying Sets of Primitives. *Journal of Computer and System Sciences*, 18(1):86–91, 1979.
 12. Leonidas J. Guibas and Robert Sedgwick. A Dichromatic Framework for Balanced Trees. In *19th Annual IEEE Symposium on the Foundations of Computer Science*, pages 8–21, 1978.
 13. Udaiprakash I. Gupta, D. T. Lee, and Joseph Y.-T. Leung. An Optimal Solution for the Channel-Assignment Problem. *IEEE Transactions on Computers*, 28(11):807–810, 1979.
 14. Tommy R. Jensen and Bjarne Toft. *Graph Coloring Problems*. John Wiley & Sons, 1995.
 15. Marc J. van Kreveld and Mark H. Overmars. Union-Copy Structures and Dynamic Segment Trees. *Journal of the ACM*, 40(3):635–652, 1993.