Alcom-FT Technical Report Series
ALCOMFT-TR-03-31

On-Line Seat Reservations via Off-Line Seating Arrangements*

Jens S. Frederiksen and Kim S. Larsen

Department of Mathematics and Computer Science
University of Southern Denmark, Odense, Denmark
{svalle,kslarsen}@imada.sdu.dk

Abstract. 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 optimal algorithms to handle problems of this nature.

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, and 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 [7, 8, 6, 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

* 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). To appear in the proceedings of the Eighth International Workshop on Algorithms and Data Structures, Ottawa, Canada, July 30 – August 1, 2003.

literature cited above. In the example, we assume the stations are named A, B, C, and D, and we assume that the train has only two seats, seat 1 and seat 2. The first reservation is (A, B), and without loss of generality, we give the seat number 1. The next reservation is (C, D). If we give seat 2 to this reservation, then the next reservation will be (A, D), 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 (C, D), then we might get first (A, C), which we can give seat 2, and then (B, D), 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 of 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. Partly because the problem is equivalent to the channel-assignment problem [12] and partly because the underlying abstract problem is coloring of interval graphs [13]. In [12], it is shown that the problem can be solved in the optimal time $\theta(n \log n)$ in the decision tree model, where the optimality follows by a reduction from the element uniqueness problem [10].

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 optimal off-line solution, but we must decide for each reservation whether or not the inclusion of this current reservation into the collection of already accepted reservations will still allow for a solution, given the number of seats available. 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 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* such that customers may cancel their reservation.

We show that in the pointer machine model [14], we can provide a data structure with the optimal complexities of $O(\log p)$ for *insert* and $O(n)$ for *output*, where n is the current number of reservations, and p is the current number of different stations, which could be a lot smaller than n and also smaller than the number of possible stations. The complexity of *delete* will also be $O(\log p)$. Furthermore, the updating operations make only $O(1)$ structural changes if a red-black tree is used as the underlying search tree, and the space requirements are $O(n)$.

In fact, our data structure will allow us to perform more insertions of reservations during the process, provided that the outputting process has not yet gotten to the starting station of the reservation to be inserted. 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 [9], this data structure is reported to have been described first in [5]) and dynamic segment trees [15]. However, segment trees have a fixed number of leaves, whereas we want to add new endpoints dynamically as they are required. 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], *insert* is $O(\log n)$ and *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 rebuild 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 as a starting point 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.

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.

We first discuss which attributes we expect different units of data to be equipped with in our algorithms.

Intervals have left and right endpoints, which we refer to as *begin* and *end*. The intervals are closed to the left and open 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 where 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. The tree is build from nodes which contain the following information: Since the tree is binary, there is a *left* and *right* reference. The attribute *cover* stores the interval covered by a node. For a leaf node, this is the interval from its endpoint to the endpoint of the next leaf, 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.

To help us decide how many colors are necessary to color the intervals, we use two extra variables in each node, k and Δk. For any path from a node in the tree to a leaf in its subtree, we define its Δ-length as the sum of all the Δk values of the nodes of the path. By updating the Δk and k values appropriately, we first of all make sure that the Δ-length of a path from the root to any leaf is exactly the density of the cover interval of the leaf, i.e., the number of intervals added to the structure which overlap the cover interval. Furthermore, we ensure that the k value of any node is the maximal Δ-length from this node to any leaf in its subtree. For the root, this is the maximal density of the tree.

As a basis for our data structure, we use a worst-case logarithmically balanced search tree such as a red-black tree [11] or an AVL-tree [2], for instance. 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. Our use of it is similar to segment trees. However, segment trees have a fixed number of leaves, whereas we want to add new endpoints dynamically as they are required.

A segment tree is designed for storing intervals and the leaves represent all possible endpoints in sorted order from left to right. The tree is used for searching for intervals which contain a certain point. Each interval (an identifier or a reference to it) is stored in at least one, but maybe in several nodes of the tree. This can be in internal nodes as well as leaves. An interval is stored in a given node u if and only if all the possible endpoints in the leaves of the subtree rooted by u are contained in the interval and no node between u and the root of the entire tree has that property. The effect obtained by this is that on a search for a point in the data structure, each interval containing that point will be found exactly once on the search path.

Our approach is similar in the way that we initially update at exactly the same locations. However, most places we only increase a counter. The actual interval is only stored at the leaves corresponding to its endpoints. Another difference is that the counter values cannot necessarily remain in the same location throughout the computations (as intervals would in a segment tree) because the tree structure is altered dynamically.

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 are met, we initialize the Δk values 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 algorithms for insertion can be seen in Fig. 1.

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

It is *insertEndpoint* which uses the tree dynamically. If an endpoint is not present, it is inserted by performing a local operation at the leaf where the search ends. The setting of the attributes in the new node is shown in Fig. 2, where it is demonstrated how one leaf is replaced by one internal node and two leaves.

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. 3, we show how attributes should be set in connection with a left rotation. A right rotation is similar.

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.

The considerations for *delete* are similar. We must update the density information by deleting the interval, we must remove the actual reservation from a leaf, and we must delete the endpoints if no other intervals share them. The actions reverse actions taken during an *insert*. The *delete* operation is shown in Fig. 4. In Fig. 5, we show how a node is removed from the tree in the case where no

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

func okToInsert(n: Node, x: Interval, c: Integer): Boolean
    if n.cover ∩ x = ∅ then
        return True
    else if n is a leaf or n.cover ⊆ x then
        return c ≥ n.k + 1
    else
        c′ ← c − n.Δk # Calculate the number of colors left
        return okToInsert(n.left, x, c′) and okToInsert(n.right, x, c′)

proc insertEndpoint(tree: Node, b: Real, d: Boolean, x: Interval)
    n ← findLeaf(tree, b) # Finds maximal a such that a ≤ b
    if n.cover.begin ≠ b then
        # Split n as described
        n ← n.right
        # Rebalance tree bottom-up if necessary
    if d then n.BeginList.append(x) else n.EndList.append(x)

proc insertInterval(n: Node, x: Interval)
    if n.cover ⊆ x then
        n.Δk ← n.Δk + 1
        n.k ← n.k + 1
    else
        if n.left.cover ∩ x ≠ ∅ then
            insertInterval(n.left, x)
        if n.right.cover ∩ x ≠ ∅ then
            insertInterval(n.right, x)
        n.k ← max(n.left.k, n.right.k) + Δk
```

Fig. 1. The insert operation.

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 changes to c's

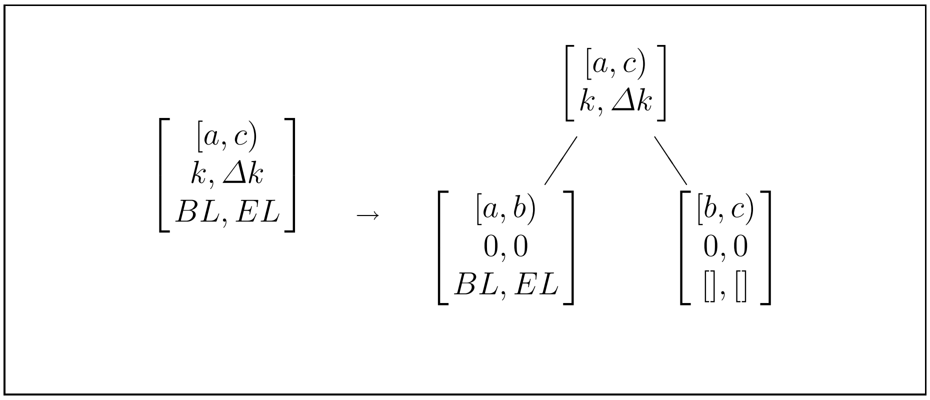

Fig. 2. A split operation performed on a leaf initially containing the interval $[a, c)$. In the nodes, the first line shows the *cover interval* and the second line shows the k and Δk value of the node. The third line shows the *BeginList* and *EndList* of leaf nodes. The new endpoint b is inserted.

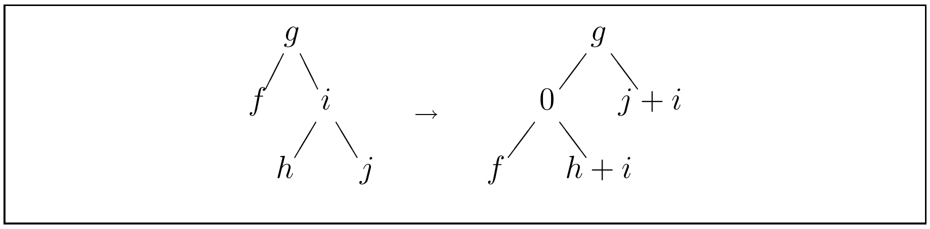

Fig. 3. A left rotation with old and new Δk values shown.

on the path down to the predecessor of the deleted node (the node containing $[a, b)$ before the update).

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. 6.

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 $n.\Delta k = n.k - \max(n.\mathit{left}.k, n.\mathit{right}.k)$.

Second, it is sufficient to store the starting 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 delete(tree: Node, x: Interval)
    deleteInterval(tree, x)
    deleteEndpoint(tree, x.begin, true, x)
    deleteEndpoint(tree, x.end, false, x)

proc deleteEndpoint(tree: Node, b: Real, d: Boolean, x: Interval)
    n ← findLeaf(tree, b) # Finds leaf containing the endpoint b
    if d then n.BeginList.remove(x) else n.EndList.remove(x)
    if n.BeginList.isEmpty() and n.EndList.isEmpty() then
        # Delete n as described
        # Rebalance tree bottom-up if necessary

proc deleteInterval(n: Node, x: Interval)
    if n.cover ⊆ x then
        n.Δk ← n.Δk − 1
        n.k ← n.k − 1
    else
        if n.left.cover ∩ x ≠ ∅ then
            delete(n.left, x)
        if n.right.cover ∩ x ≠ ∅ then
            delete(n.right, x)
        n.k ← max(n.left.k, n.right.k) + Δk
```

Fig. 4. The delete operation.

Correctness and Complexity

We argue that the algorithms presented are correct, and discuss their complexity.

Correctness

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

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

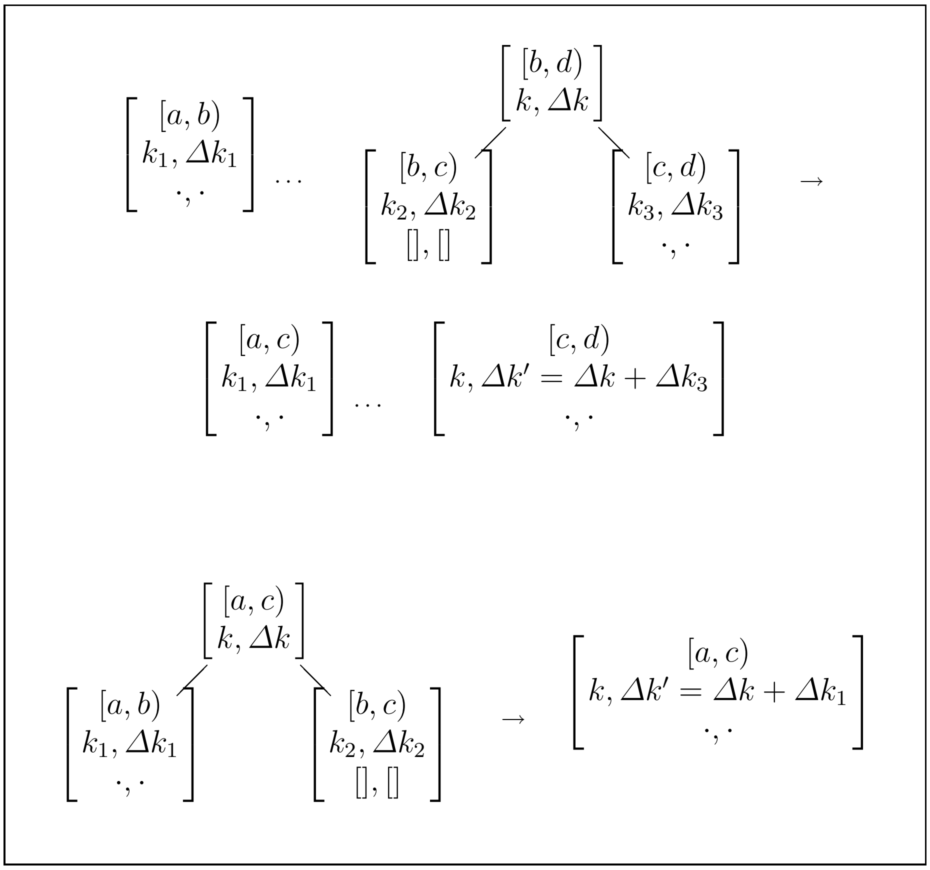

Fig. 5. A delete operation performed on a node containing the interval $[b, c)$. There are two cases depending on whether the node to be deleted is the left or right child of its parent.

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 increasing 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.

```
proc output(tree: Node)
   s ← new Stack of N Colors
   # Optional wait until first station is reached
   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
```

Fig. 6. The output operation.

Now, if k intervals overlap at a given point, this defines a clique of size k in the corresponding interval graph. Interval graphs are *perfect* [13] which means that the size of the largest clique equals the minimum number of colors required to color the graph.

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, 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.

Complexity

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

Regarding *insertInterval*, 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 *insertInterval*, 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 $O(p) \subseteq O(n)$, 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 *insertInterval* only increments integers already present in the structure.

Optimality

Regarding optimality, clearly $\Omega(n)$ is required to output the result. If, as we do, *output* is provided in $O(n)$, *insert* must be $\Omega(\log n)$, in the cases where $p \in \Theta(n)$. Otherwise, we can solve the off-line problem in $o(n \log n)$, and this has been proven impossible in the decision tree model in [12] by a simple reduction from the more well-known element uniqueness problem [10], which is known to be $\theta(n \log n)$.

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 $x_1, x_2, \ldots, x_n$, 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 most 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. Thus, if the problem in this paper is not in $\Omega(n \log p)$, the sorting problem above would not be either, and that would be a contradiction.

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.

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), $[a_2, a_2]$, ...

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.

As a more general remark, it is important to notice that we do not assume that the stations which are used are numbered from 1 through p. In fact, we do not even 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.

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 Akadamii 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. Favrholdt, 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. Tech. report, Carnegie-Mellon University, 1977.
6. Joan Boyar, Lene M. Favrholdt, Kim S. Larsen, and Morten N. Nielsen. Extending the Accommodating Function. In *Eighth Annual International Computing and Combinatorics Conference*, volume 2387 of *Lecture Notes in Computer Science*, pages 87–96. Springer-Verlag, 2002.
7. Joan Boyar and Kim S. Larsen. The Seat Reservation Problem. *Algorithmica*, 25(4):403–417, 1999.
8. 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.
9. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.
10. 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.
11. Leo J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *19th Annual IEEE Symposium on the Foundations of Computer Science*, pages 8–21, 1978.
12. U. 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.
13. Tommy R. Jensen and Bjarne Toft. *Graph Coloring Problems*. John Wiley & Sons, 1995.
14. 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.
15. Marc J. van Kreveld and Mark H. Overmars. Union-Copy Structures and Dynamic Segment Trees. *Journal of the Association for Computing Machinery*, 40(3):635–652, 1993.