On-line seat reservations via off-line seating arrangements

Jens S. Frederiksen and Kim S. Larsen

University of Southern Denmark
The Seat Reservation Problem

[Boyar, Larsen: Algoritmica 99]

Accommodate as many requests as possible.
As an on-line problem:
e.g.: [Boyar, Larsen: Algoritmica 99]
Seat numbers given at reservation time.
On-line results can be far from optimal.

As an off-line problem (coloring interval graphs):
[Gupta, Lee, Leung 79]
Optimal $\Theta(n \log n)$ solution.
Previous Work

From another angle:

[Bentley 1977]
Semi-dynamic segment trees.
Endpoints known in advance.
Insert $O(\log p)$. Stabbing query $O(\log p + k)$.
Space $O(n \log p)$.

[Kreveld, Overmars 1993]
Fully dynamic segment trees.
Insert $O_A(\log n)$. Delete $O_A(a(i, n) \log n)$.
Stabbing query $O(\log p + k)$. Space $O(n \log n)$
A New Problem

On-line
- intervals given one-by-one.
- endpoints not known in advance.

Off-line
- seat numbers given at end of $\sigma$
- compute optimal solution
On-line Off-line

A new data structure

- $p_i$ – endpoints from a fully ordered set.
- $I$ – intervals: $[p_i; p_j)$.
- $A = \text{set of accepted intervals.}$
- $N = \text{number of colors available.}$
On-line Off-line

A new data structure with three operations

- **insert**(interval).
  If $\text{color}(A \cup \{\text{interval}\}) \leq N$, then accept the interval: $A := A \cup \{\text{interval}\}$. Otherwise: reject it.

- **delete**(interval).
  Delete a previously inserted interval from the structure. $A := A \setminus \{\text{interval}\}$.

- **output**( ).
  Print all accepted intervals with a legal coloring.
Our Solution

Use almost any balanced binary tree with $O(\log n)$ insert and rebalancing, e.g., a red-black or AVL-tree.

The leaves correspond to the endpoints $p_i$ of accepted intervals in sorted order.
Attributes

Leaves contain the following attributes:

- An endpoint $p_i$.
- $\text{BeginList}/\text{EndList}$ – the set of all accepted intervals so far which begin/end in $p_i$.
- $\text{Cover}$ – the interval $[p_i; p_{i+1})$.

Internal nodes contain the following attribute:

- $\text{Cover}$ where $n.\text{cover} = n.\text{left}.\text{cover} \cup n.\text{right}.\text{cover}$
Invariant #1 – $k$ & $\Delta k$

For the root node $n$ and any leaf node $l$, 

$$|\{I \in A|I \cap l.\text{cover} \neq \emptyset\}|$$

$$= \sum_{n_i \in \{n \rightarrow l\}} n_i.\Delta k.$$
Invariant #2 – $k$ & $\Delta k$

For any node $n$,

$$n.k := \max_l \sum_{n_i \in \{n \rightarrow l\}} n_i.\Delta k.$$
Rebalancing

A left rotation with old and new $\Delta k$ values shown.

Afterward, k and Cover are updated bottom-up.
A Split Operation

As this is a dynamic tree, it can be necessary to split a leaf node.

\[
\begin{bmatrix}
  [a, c) \\
  k, \Delta k \\
  BL, EL
\end{bmatrix}
\rightarrow
\begin{bmatrix}
  [a, b) \\
  0, 0 \\
  BL, EL
\end{bmatrix}
\begin{bmatrix}
  [b, c) \\
  0, 0 \\
  \emptyset, \emptyset
\end{bmatrix}
\]
proc insert(x: Interval)
    if okToInsert(tree, x, N) then
        insertEndpoint(tree, x.begin, true, x)
        insertEndpoint(tree, x.end, false, x)
        insertInterval(tree, x)
**insertInterval**

```plaintext
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) + n.Δk
```

```
insertInterval(n, [2; 11])
```

```
On-line seat reservations via off-line seating arrangements – WADS 2003 – p.15/21
```
The search is said to *stop* at a node, if recursion does not continue farther.
For any node $n$ on the left path after the split, either

1. $n.left.cover \cap I = \emptyset$.  
2. $n.right.cover \subseteq I$.

Note that the cover intervals of the leaves form a continuous interval.
Only one split occurs:

\[ \text{insertInterval}(n, [2; 11]) \]

Thus, this runs in \( O(\text{height of tree}) \), i.e., \( O(\log p) \).
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
Conclusion

- We have looked at a data structure with
  - insert, delete – $O(\log p)$
  - output – $O(n)$
  - total running time – $O(n \log p)$
  - this is optimal

$p$ is the number of different endpoints.

$n \geq p$ is the number of intervals.

- Furthermore,
  - space usage – $O(n)$
  - rebalancing – $O_A(1)$
Possible Extensions & Future Work

- $N$ can be dynamic.
- Intervals do not have to be $[a; b)$.
- \textit{split} and \textit{join} can be implemented in $O(\log p)$.
- Special case, when the endpoints are integers from \{1, \ldots, p\}. 
proc insertEndpoint(tree: Node, b: Real, d: Boolean, x: Interval)

# Find maximal \( a \) such that \( a \leq b \)

n ← findLeaf(tree, b)

if n.cover.begin \( \neq b \) then

# split \( n \)

n ← n.right

# rebalance tree bottom-up if necessary

if d then n.BeginList.append(x)
else n.EndList.append(x)
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
    # Calculate the number of colors left
    c' \leftarrow c - n.\Delta k
    return otI(n.left, x, c') and otI(n.right, x, c')