14. Augmenting Data Structures

Yu-Shuen Wang, CS, NCTU

Augmenting Data Structures

- It will suffice to augment a textbook data structure by storing additional information in it. You can then program new operations for the data structure to support the desired application. Augmenting a data structure is not always straightforward, however, since the added information must be updated and maintained by the ordinary operations on the data structure.

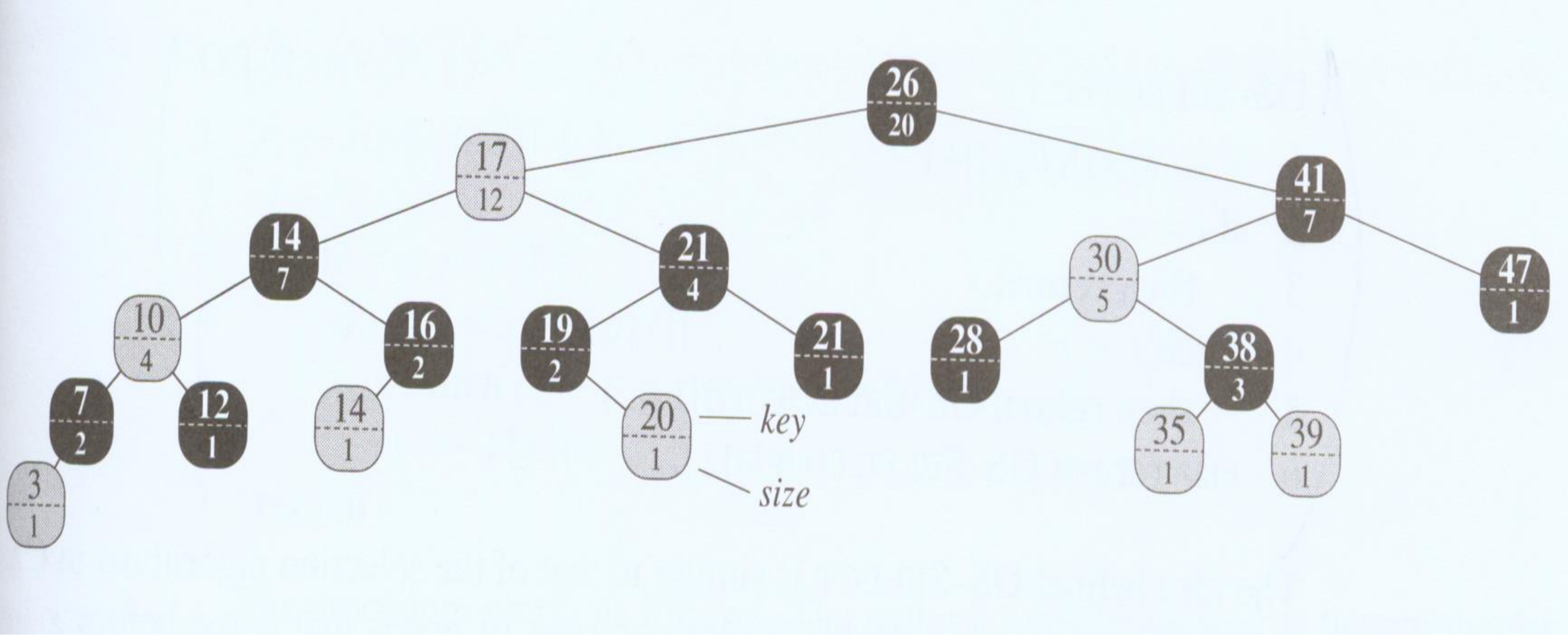
14.1 Dynamic order statistics

- We shall also see the **rank** of an element—its position in the linear order of the set—can likewise be determined in $O(\lg n)$ time.

- Beside the usual red-black tree fields $key[x]$, $color[x]$, $p[x]$, $left[x]$, and $right[x]$ in a node x, we have another field $size[x]$. This field contains the number of (internal) nodes in the subtree rooted at x (including x itself), that is the size of the subtree. If we define the sentinel's size to be 0, that is, we set $size[nil[T]]$ to be 0, then we have the identity

$$size[x] = size[left[x]] + size[right[x]] + 1$$

An order-statistic tree



Retrieving an element with a given rank

OS-SELECT(*x*, *i*)

```
r ← size[left[x]]+1
if i = r
   then return x
else if i < r
   then return OS-SELECT(left[x], i)
else return OS-SELECT(right[x], i – r)
```

Time complexity :O(lg n)

Determining the rank of an element

OS-RANK(T, x)

```
r ← size[left[x]] + 1
y ← x
while y ≠ root[T]
    do if y = right[p[y]]
            then r ← r + size[left[p[y]]] + 1
        y ← p[y]
return r
```

The running time of OS-RANK is at worst proportional to the height of the tree: O(lg n)

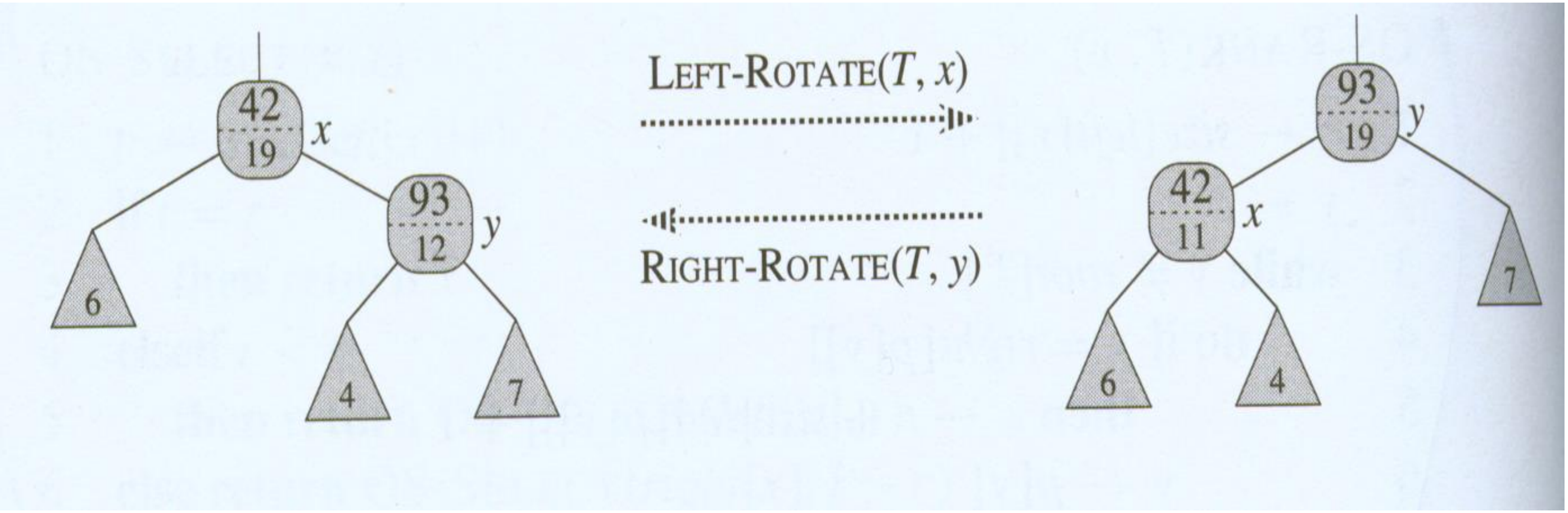
Maintaining subtree sizes

- Referring to the code for LEFT-ROTATE(T, x) in *section 13.2* we add the following lines:

  12 $size[y] \leftarrow size[x]$

  13 $size[x] \leftarrow size[left[x]] + size[right[x]] + 1$

Updating subtree sizes during rotations



14.2 How to augment a data structure

1. Choosing an underlying data structure,
2. Determining additional information to be maintained in the underlying data structure,
3. Verifying that the additional information can be maintained for the basic modifying operations on the underlying data structure, and
4. Developing new operations.

Augmenting red-black trees

- ***Theorem 14.1*** (Augmenting a red-black tree)

  Let f be a field that augments a red-black tree T of n nodes, and suppose that the contents of f for a node x can be computed using only the information in nodes x, $left[x]$, and $right[x]$, including $f[left[x]]$ and $f[right[x]]$. Then, we can maintain the values of f in all nodes of T during insertion and deletion without asymptotically affecting the $O(\lg n)$ performance of these operations.

Proof.

- The main idea of the proof is that a change to an f field in a node x propagates only to ancestors of x in the tree.

14.3 Interval trees

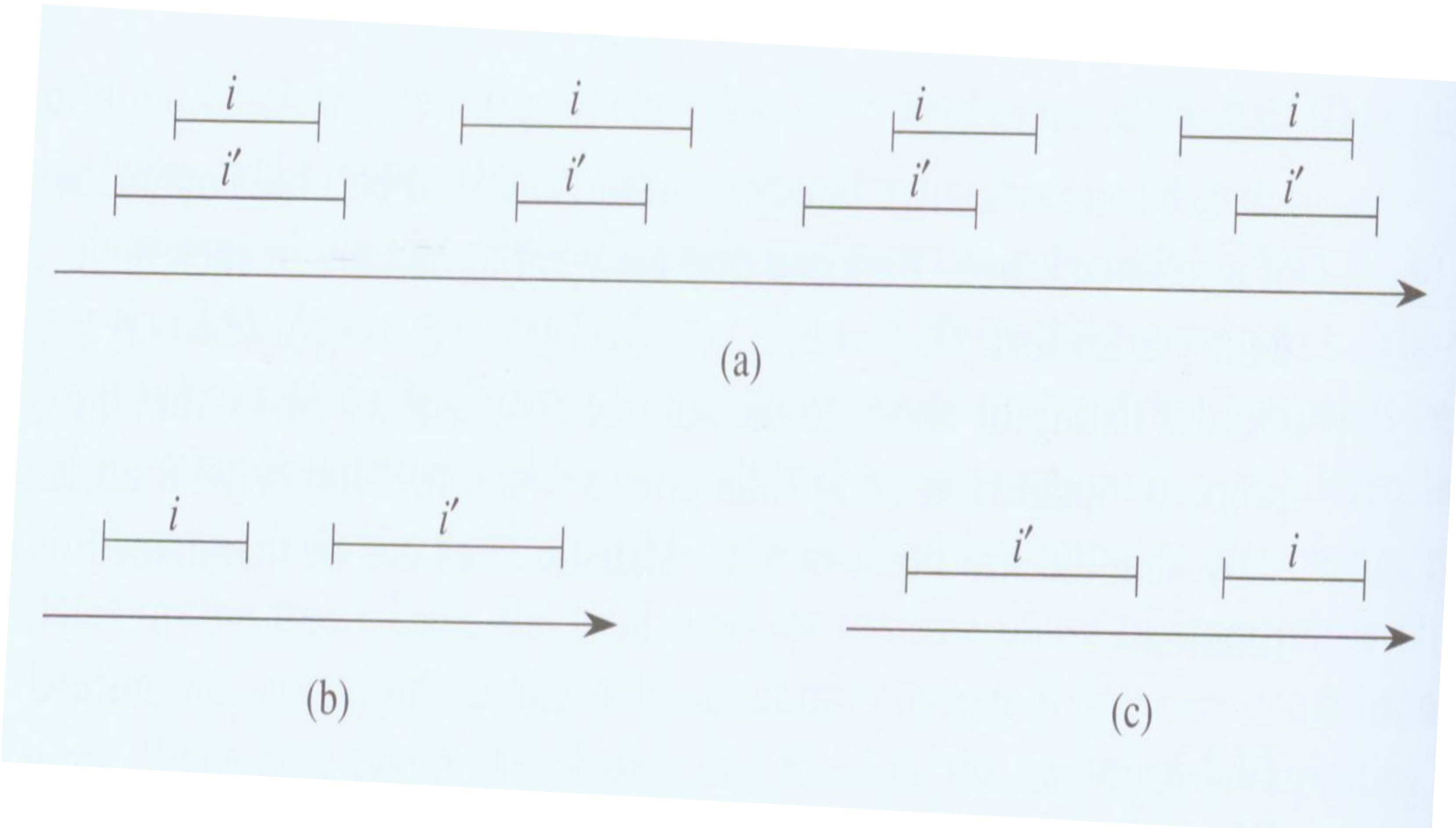
- We can represent an interval$[t_1,t_2]$ as an object i, with fields $low[i] = t_1$ (the **low endpoint**) and $high[i] = t_2$ (the **high endpoint**). We say that intervals i and i' **overlap** if $i \cap i' \neq \emptyset$, that is, if $\mathrm{low}[i] \leq \mathrm{high}[i']$ and $low[i'] \leq high[i]$. Any two intervals i and i' satisfy the **interval trichotomy**; that exactly one of the following three properties holds:

  a. i and i' overlap,

  b. i is to the left of i' (i.e., $high[i] < low[i']$),

  c. i is to the right of i' (i.e., $high[i'] < low[i]$)

The interval trichotomy for two colsed intervals i and i'

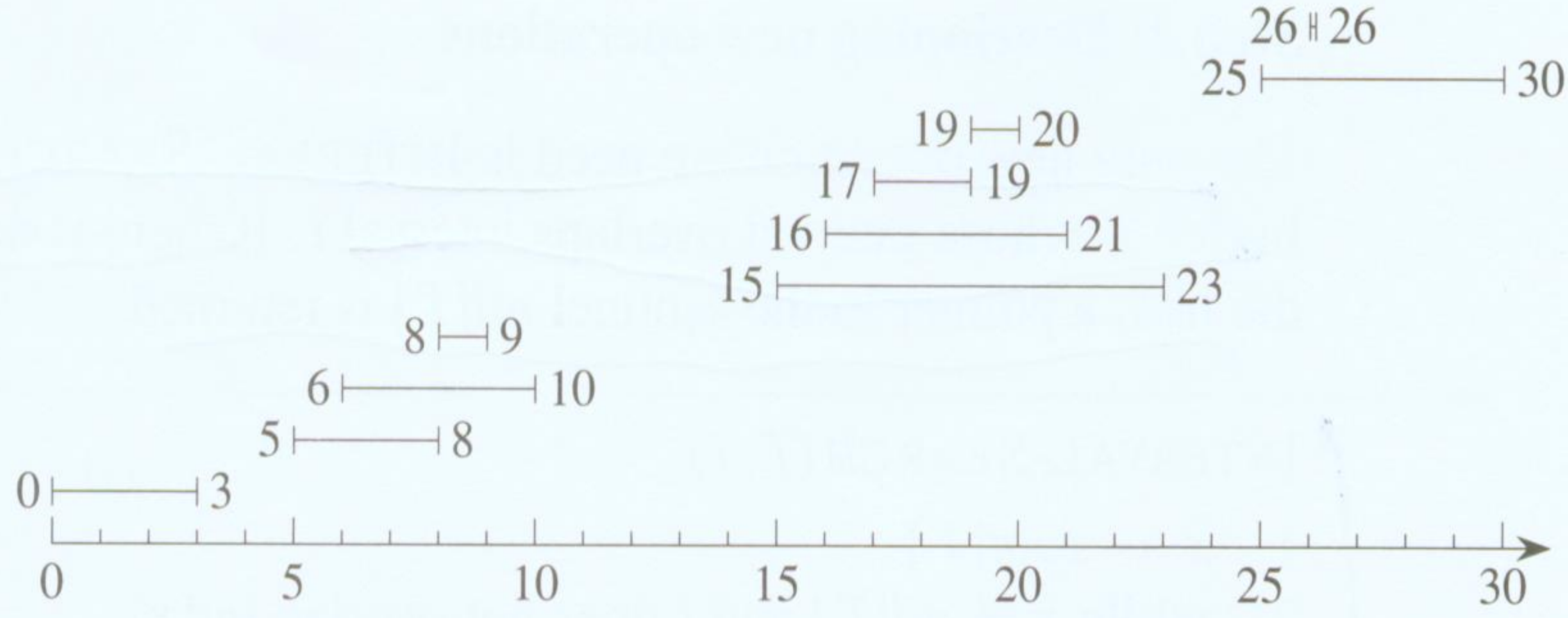


- An interval tree is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval $\mathrm{int}[x]$, and the key of *x* is the low endpoint, $low[int[x]]$, of the interval.

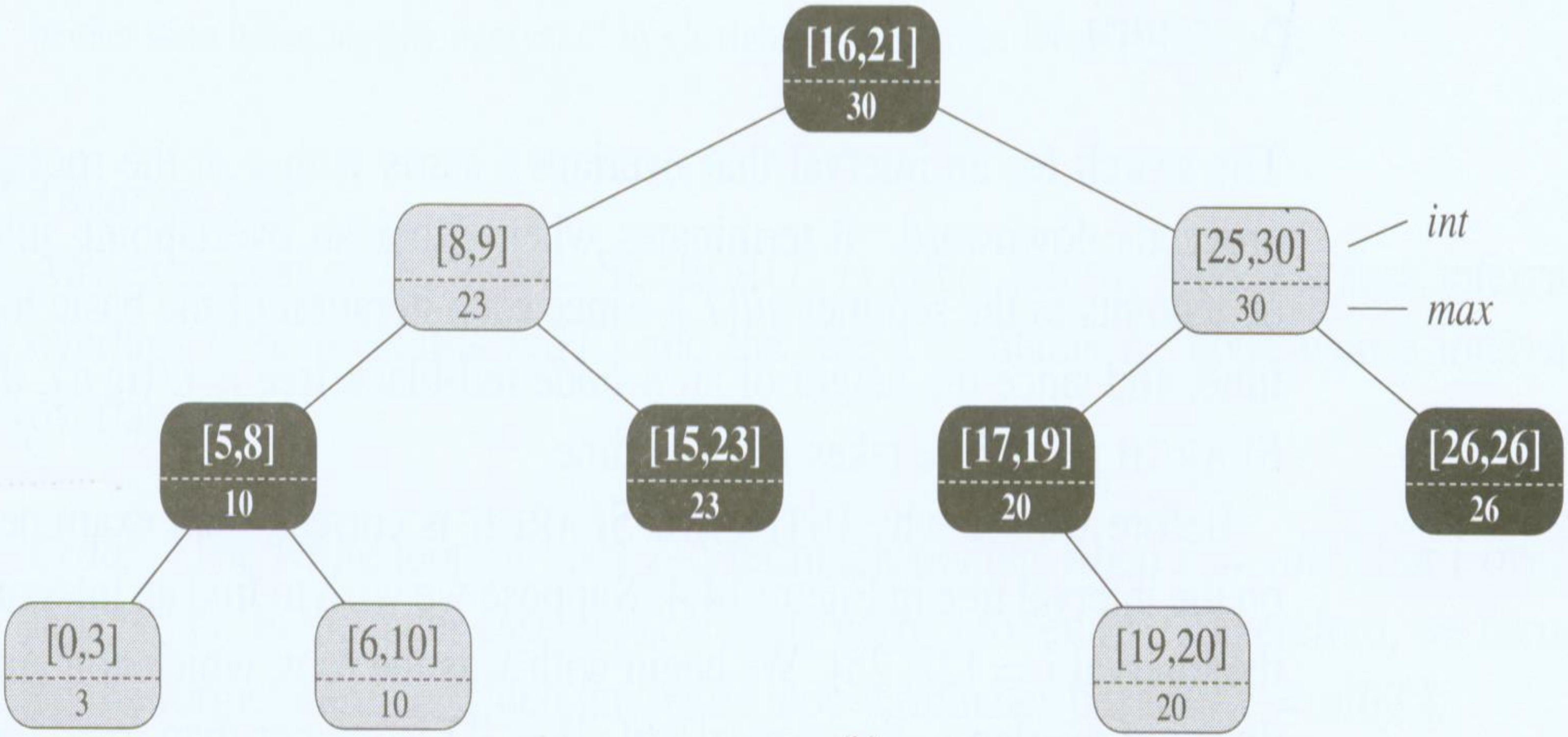
Operations

- Interval trees support the following operations.
  - INTERVAL-INSERT(T,x)
  - INTERVAL-DELETE(T,x)
  - INTERVAL-SEARCH(T,i)

An interval tree



(a)


[16,21]
30
[8,9]
23
[25,30]
30
int
max
[5,8]
10
[15,23]
23
[17,19]
20
[26,26]
26
[0,3]
3
[6,10]
10
[19,20]
20
(b)

Design of an interval tree

- Step 1: underlying data structure
  - Red-black tree
- Step 2: Additional information
  - Each node x contains a value $max[x]$, which is the maximum value of any interval endpoint stored in the subtree rooted at x.
- Step 3: Maintaining the information
  - We determine $max[x]$ given interval $int[x]$ and the max values of node x's children:

    $$max[x] = max(high[int[x]], max[left[x]], max[right[x]]).$$
  - Thus, by Theorem 14.1, insertion and deletion run in $O(\lg n)$ time.

- Step 4: Develop new operations
  - The only new operation we need is INTERVAL-SEARCH(T,i), which finds a node in tree T whose interval overlaps interval i. If there is no interval that overlaps i in the tree, a pointer to the sentinel $nil[T]$ is returned.

- INTERVAL-SEARCH(*T*, *i*)

```
x ← root [T]
while x ≠ nil[T] and i does not overlap int[x]
    do if left[x] ≠ nil[T] and max[left[x]] ≥ low[i]
            then x ← left[x]
            else  x ← right[x]
return x
```

Theorem 14.2

Any execution of INTERVAL-SEARCH(T,i) either returns a node whose interval overlaps i, or it returns $nil[T]$ and the tree T contain node whose interval overlaps i.