4) develop new operations

Interval-Search(T,i)
  \[x = \text{root}[T]\]
  \[\text{while } x \neq \text{NIL} \text{ and } i \text{ does not overlap int}[x] \]
  \[\text{if } \text{left}[x] \neq \text{NIL} \text{ and } \max[\text{left}[x]] \geq \text{low}[i] \text{ then } x = \text{left}[x]\]
  \[\text{else } x = \text{right}[x]\]

Theorem: 1) if search goes right,
  then x’s left subtree contains no interval that overlaps i
  2) if search goes left, either x’s left subtree contains an interval that overlaps i, or no interval in x’s right subtree overlaps i

Proof: 1) any interval i’ in left subtree must end before low[i]

2) there must be interval i’ in x’s left subtree with high[i’]=\max[\text{left}[x]]
any interval i” in x’s right subtree has \(\text{low}[i’’] \geq \text{low}[i’]\)
3) verify that additional information can be maintained efficiently

\[ \text{max}[x] = \text{maximum of } \begin{cases} \text{high}[\text{int}[x]], \\ \text{max}[\text{left}[x]], \\ \text{max}[\text{right}[x]] \end{cases} \]
Interval Trees

data are closed intervals on a line

interval given as $i=[l,h]$

$l=$ low endpoint of $i$  
$h=$ high endpoint of $i$

interval $i$ overlaps interval $i'$ if $i$ and $i'$ share more than 1 point of line

we need to maintain operations

Interval-Insert($T,x$)
Interval-Delete($T,x$)
Interval-Search($T,i$) -- returns some interval that overlaps interval $i$

or NIL if no such interval exists

1) choose an underlying data structure: red-black tree
   key of a node is the low endpoint

2) determine additional information to be maintained:
   $\max[x]=$ maximum value of any interval endpoint stored in subtree rooted at $x$
during RB-Delete
  phase 1: splice out the node being deleted
    new: decrement size information of all the nodes on the path from the root to the deleted node
  phase 2: go up the tree fixing color information
    new: during rotations recompute size information

Running time: $O(lg n)$ time

General technique for augmenting data structures
  1) choose an underlying data structure
  2) determine additional information to be maintained
  3) verify that additional information can be maintained efficiently
  4) develop new operations
Maintaining variable size during inserts and deletes during RB-Insert

phase 1: go down the tree from root and insert new node as child of existing node
new: add 1 to each node along the way

phase 2: go up the tree fixing color information
new: during rotations recompute size information

example: when doing Right-Rotate,
size[x]=size[y]
size[y]=size[β]+size[γ]+1

Running time: O(lg n) time
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)

Running time: \(O(\lg n)\), where \(n\) is the number of keys in the tree

OS-Rank(T,x)
    r=size[left[x]]+1
    y=x
    while y ≠ root [T]
        if y=right[p[y]] then r=r+size[left[p[y]]]+1
        y=p[y]
    return r

Running time: \(O(\lg n)\)
Dynamic Order Statistics: have a set of keys and want to support the following operations:
1) determine the i-th smallest key
2) determine the rank of key x
3) add a key
4) delete a key

Order-statistics tree -- red-black binary search tree, with addition: at each node x keep a variable
size[x]=number of internal nodes in subtree rooted at x (including x itself)

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