Introduction to Algorithms

Advanced Data Structures: I

Xiang-Yang Li and Haisheng Tan

School of Computer Science and Technology
University of Science and Technology of China (USTC)

Fall Semester 2024

Outline of Topics

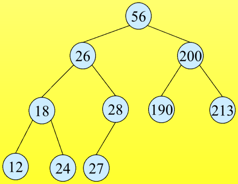
Binary Search Trees

Red-Black Trees

Augmenting Data Structures

Binary Trees

- Recursive definition
  1. An empty tree is a binary tree
  2. A node with at most two child subtrees is a binary tree
  3. Only what you get from 1 by a finite number of applications of 2 is a binary tree
- Is this a binary tree?

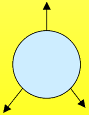


Binary Search Trees

- View today as data structures that can support dynamic set operations
  Search,Minimum,Maximum,Predecessor,Successor,Insert,and Delete
- Can be used to build
  Dictionaries
  Priority Queues
- Basic operations take time proportional to the height of the tree $-O(h)$

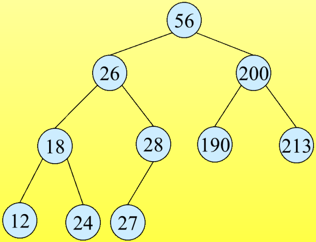
BST–Representation

- Represented by a linked data structure of nodes
- *root*(T) points to the root of tree T
- Each node contains field:
  key
  left- pointer to left child: root of left subtree
  right- pointer to right child : root of right subtree
  p- pointer to parent. $p[root[T]] = \text{NIL}$ (optional)



Binary Search Tree Property

- Stored keys must satisfy the *binary search tree* property
  1. $\forall y$ in left subtree of x, then $key[y] \leq key[x]$
  2. $\forall y$ in right subtree of x, then $key[y] \geq key[x]$

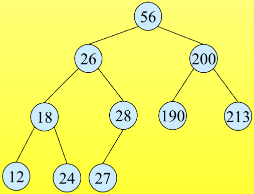


Inorder Traversal

- The *binary search tree* property allows the keys of a binary search tree to be printed, in (monotonically increasing) order, recursively

INORDERTREEWALK(x)

```
if x ≠ NIL then
    INORDERTREEWALK(left[x])
    print key[x]
    INORDERTREEWALK(right[x])
```



- How long does the walk take?
- Can you prove its correctness?

Correctness of Inorder-Walk

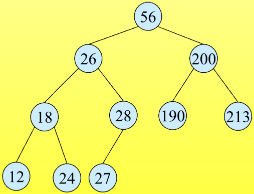
- Must prove that it prints all elements, in order, and that it terminates
- 1. By induction on size of tree, Size $= 0$: Easy
  2. Size ≥ 1:
     a. Prints left subtree in order by induction
     b. Prints root, which comes after all elements in left subtree (still in order)
     c. Prints right subtree in order (all elements come after root, so still in order)

Preorder Traversal

- The *binary search tree* property allows the keys of a binary search tree to be printed recursively

PREORDERTREEWALK(x)

if $x \neq$ NIL **then**
  print $key[x]$
  PREORDERTREEWALK($left[x]$)
  PREORDERTREEWALK($right[x]$)



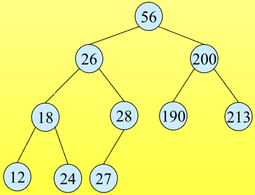
- How long does the walk take?
- Can you prove its correctness?

Postorder Traversal

- The *binary search tree* property allows the keys of a binary search tree to be printed recursively

POSTORDERTREEWALK(x)

```
if x ≠ NIL then
    POSTORDERTREEWALK(left[x])
    POSTORDERTREEWALK(right[x])
    print key[x]
```



- How long does the walk take?
- Can you prove its correctness?

Querying a Binary Search Tree

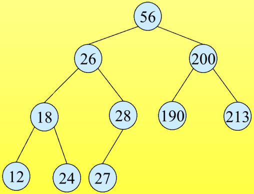
- All dynamic-set search operations can be supported in $O(h)$ time
- $h = \Theta(\lg\ n)$ for a balanced binary tree (and for an average tree built by adding nodes in random order.)
  1. Self-balanced binary search trees will have $h = \Theta(\lg\ n)$
  2. Examples of such trees, red-black tree, AVL tree, 2-3 tree
- $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of n nodes in the worst case

Tree Search

$\textsc{TreeSearch}(x, k)$,

x is a node, k is a value

if $x =$ NIL or $k = \text{key}[x]$ **then**
    return x
if $k < \text{key}[x]$ **then**
    return $\textsc{TreeSearch}(\mathit{left}[x], k)$
else
    return $\textsc{TreeSearch}(\mathit{right}[x], k)$

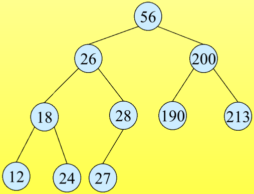


- Running time: $O(h)$
- Aside: tail-recursion

Iterative Tree Search

ITERATIVETREESEARCH(x, k)

```
while x ≠ NIL and k ≠ key[x] do
    if k < key[x] then
        x ← left[x]
    else
        x ← right[x]
return x
```



- The iterative tree search is more efficient on most computers
- The recursive tree search is more straightforward

Finding Min & Max

- The binary-search-tree property guarantees that:
  1. The minimum is located at the left-most node
  2. The maximum is located at the right-most node

TREEMINIMUM(x)

```
while left ≠ NIL do
    x ← left[x]
return x
```

TREEMAXIMUM(x)

```
while right ≠ NIL do
    x ← right[x]
return x
```

- Q: How long do they take?

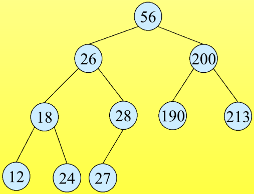
Predecessor and Successor

- Successor of node x is the node y such that $key[y]$ is the smallest key greater than $key[x]$
- The successor of the largest key is NIL
- Search consists of two cases:
  1. If node x has a non-empty right subtree, then x's successor is the minimum in the right subtree of x
  2. If node x has an empty right subtree, then:
     a. As long as we move to the left up the tree (move up through right children), we are visiting smaller keys
     b. x's successor $y$is the node that x is the predecessor of (x is the maximum in y's left subtree)
     c. In other words, x's successor y, is the lowest ancestor of x whose left child is also an ancestor of x

Pseudo-code for Successor

TREESUCCESSOR(x)

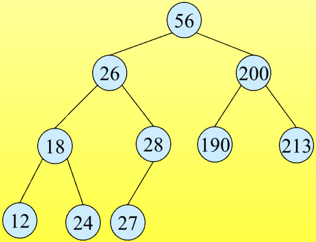
if $right[x] \neq$ NIL **then**
    return TREEMINIMUM($right[x]$)
$y \leftarrow p[x]$
while $y \neq$ NIL and $x = right[y]$ **do**
    $x \leftarrow y$
    $y \leftarrow p[y]$
return y



- Code for predecessor is symmetric
- Running time: $O(h)$

Insertion and Deletion

- Change the dynamic set represented by a BST
- Ensure the *binary search tree* property holds after change
- Insertion is easier than deletion



BST Insertion –Pseudocode

TREEINSERT(T, z)

```
y ← NIL
x ← root[T]
while x ≠ NIL do
    y ← x
    if key[z] < key[x] then
        x ← left[x]
    else
        x ← right[x]
p[z] ← y
if y = NIL then
    root[T] ← z
else if key[z] < key[y] then
    left[y] ← z
else
    right[y] ← z
```

Analysis of Insertion

- Initialization: $O(1)$
- While loop in lines 3-10 searches for place to insert z, maintaining parent y. This takes $O(h)$ time
- Lines 11-18 insert the value: $O(1)$
- Total: $O(h)$ time to insert a node

Exercise: Sorting Using BSTs

SORT(A)

1: **for** $i \leftarrow 1$ to n **do**
2: TREEINSERT($A[i]$)
3: INORDERTREEWALK(*root*)

- What are the worst case and best case running times?
- In practice, how would this compare to other sorting algorithms?

BST Deletion

- Case 0: if x has no children:
  then remove x
- Case 1: if x has one child:
  then make $p[x]$ point to child
- Case 2: if x has two children (subtrees):
  then swap x with its successor
  perform case 0 or case 1 to delete it
- Total: $O(h)$ time to delete a node

BST Deletion – Pseudocode

```
TreeDelete(T, z)
if left[z] = NIL
or right[z] = NIL then
    y ← z
else
    y ← TreeSuccessor[z]
if left[y] ≠ NIL then
    x ← left[y]
else
    x ← right[y]
if x ≠ NIL then
    p[x] ← p[y]
if p[y] = NIL then
    root[T] ← x
else if y ← left[p[i]] then
    left[p[y]] ← x
else
    right[p[y]] ← x
if y ≠ z then
    key[z] ← key[y]
return y
```

Correctness of TreeDelete

- How do we know case 2 should go to case 0 or case 1 instead of back to case 2?

Correctness of TreeDelete

- How do we know case 2 should go to case 0 or case 1 instead of back to case 2?
  Because when x has 2 children, its successor is the minimum in its right subtree, and that successor has no left child (hence 0 or 1 child)

Correctness of TreeDelete

- How do we know case 2 should go to case 0 or case 1 instead of back to case 2?
  Because when x has 2 children, its successor is the minimum in its right subtree, and that successor has no left child (hence 0 or 1 child)
- Any other choice?

Correctness of TreeDelete

- How do we know case 2 should go to case 0 or case 1 instead of back to case 2?
  Because when x has 2 children, its successor is the minimum in its right subtree, and that successor has no left child (hence 0 or 1 child)
- Any other choice?
  Equivalently, we could swap with predecessor instead of successor. It might be good to alternate to avoid creating lopsided tree

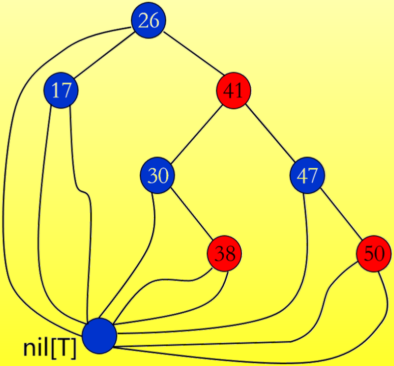
Red-Black Trees: Overview

- Red-black trees are a variation of binary search trees to ensure that the tree is *balanced*
- Height is $O(\lg n)$, where n is the number of nodes
- Operations take $O(\lg\ n)$ time in the worst case

Red-Black Tree

- Binary search tree + 1 bit per node: the attribute *color*, which is either **red** or **black**
- All other attributes of BSTs are inherited: *key*, *left*, *right*, and *p*
- All empty trees (leaves) are colored black
  1. We use a single sentinel, *nil*, for all the leaves of red-black tree T, with $color[nil]$ = black
  2. The root's parent is also $nil[T]$

Red-Black Tree – Example

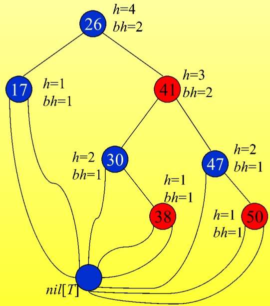


Red-Black Properties

- 1. Every node is either red or black
- 2. The root is black
- 3. Every leaf (*nil*) is black
  · Note: this means every "real" node has 2 children
- 4. If a node is red, then both its children are black
  · Note: can't have 2 consecutive reds on a path
- 5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Height of a Red-Black Tree

- Height of a node:
  Number of edges in a longest path to a leaf
- Black-height of a node x, $bh(x)$:
  Number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x
- Black-height of a red-black tree is the black-height of its root:
  By Property 5, black height is well defined



Height of Red-Black Trees

- What is the minimum black-height of a node with height h?
  A height-h node has black-height $\geq h/2$
- Theorem: A red-black tree with n internal nodes has height $h \leq 2\,lg(n+1)$
  How do you suppose we'll prove this?

RB Trees: Proving Height Bound

- Prove: n-node RB tree has height $h \leq 2\,lg(n+1)$
- Claim: A subtree rooted at a node x contains at least $2^{bh(x)} - 1$ internal nodes
- Proof by induction on height h

RB Trees: Proving Height Bound

- Base step: x has height 0 (i.e., NULL leaf node)
  1. So $bh(x) = 0$
  2. So subtree contains $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes (TRUE)
- Inductive step: x has positive height and 2 children
  1. Each child has black-height of $bh(x)$ or $bh(x) - 1$
  2. So the subtrees rooted at each child contain at least $2^{bh(x)-1} - 1$ internal nodes
  3. Thus subtree at x contains $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nodes (TRUE)

RB Trees: Proving Height Bound

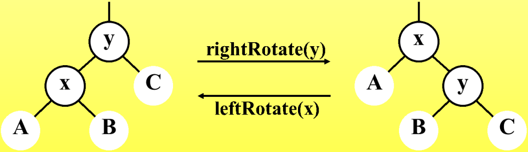
- Thus at the root of the red-black tree:
  $n \geq 2^{bh(root)} - 1 \Rightarrow n \geq 2^{h/2} - 1 \Rightarrow h \leq 2\, lg(n+1)$
- Thus $h = O(lg\ n)$

RB Trees: Worst-Case Time

- So we've proved that a red-black tree has $O(lg\ n)$ height
- Corollary: These operations take $O(lg\ n)$ time:
  Minimum(), Maximum()
  Successor(), Predecessor()
  Search()
- Insert() and Delete():
  1. Will also take $O(lg\ n)$ time
  2. But will need special care since they modify tree

RB Trees: Rotation

- Our basic operation for changing tree structure is called rotation:



- Preserves BST key ordering
- $O(1)$ time...just changes some pointers

RB Trees: Insertion

- Insertion: the basic idea
  1. Insert x into tree, color x red
  2. r-b property 2 could be violated (if x is root and red)
     If so, no other property is violated, we make x black.
  3. Otherwise,r-b property 4 might be violated (if $p[x]$ red)
     If so, move violation up tree until a place is found where it can be fixed
  4. Total time will be $O(lg\ n)$

RB Insertion – Pseudocode I

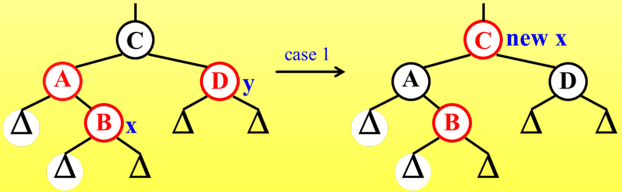
```
RBINSERT(T, x)
TREEINSERT(T, x)
color[x] ← RED
while x ≠ root[T] and color[p[x]] = RED do
    if p[x] = left[p[p[x]]] then
        y ← right[p[p[x]]]
        if color[y] = RED then
            color[p[x]] ← BLACK
            color[y] ← BLACK
            color[p[p[x]]] ← RED
            x ← p[p[x]]
```

RB Insertion – Pseudocode II

```
        else
            if x = right[p[x]] then
                x ← p[x]
                LeftRotate(x)
            color[p[x]] ← BLACK
            color[p[p[x]]] ← RED
            RightRotate p[p[x]]
    else
        same as above, but exchanging 'right' and 'left'
color[root[T]] ← BLACK
```

RB Insert: Case 1

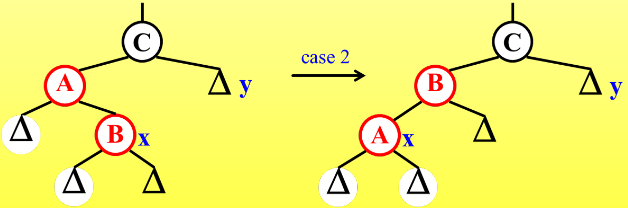
- Case 1: "uncle" is red:
  In figures below, all △'s are equal-black-height subtrees



- Change colors of some nodes, preserving r-b property 5: all downward paths have equal b.h.The while loop now continues with x's grandparent as the new x

RB Insert: Case 2

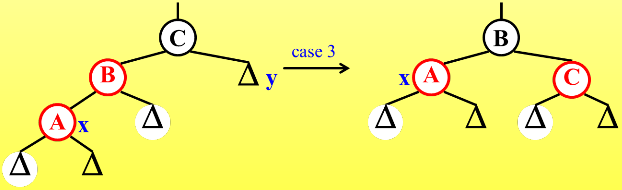
- Case 2: “uncle” is black
  Node *x* is a right child



- Set x=p[x]. Transform to case 3 via a left-rotation
- This preserves property 5: all downward paths contain same number of black nodes

RB Insert: Case 3

- Case 3: "uncle" is black
  Node x is a left child



- Perform some color changes and do a right rotation
- Again, preserves property 5: all downward paths contain same number of black nodes

Red-Black Trees

- Red-black trees do what they do very well
- What do you think is the worst thing about red-black trees?
  A: coding them up

Recall "Ordinary" BST Delete

- Case 1: If vertex to be deleted is a leaf, just delete it
- Case 2: If vertex to be deleted has just one child, replace it with that child
- Case 3: If vertex to be deleted has two children, then swap it with its successor

Bottom-Up Deletion

- Do ordinary BST deletion. Eventually a “case 1” or “case 2” will be conducted. If deleted node, U, is a leaf, think of deletion as replacing with the NULL pointer, V. If U had one child, V, think of deletion as replacing U with V
- What can go wrong? If U is red? If U is black?

Fixing the problem

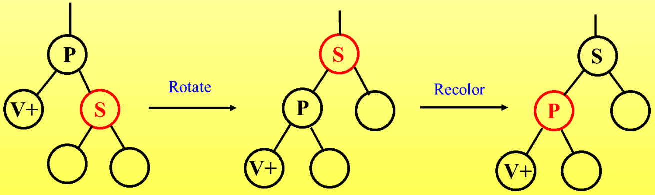
- Think of V as having an "extra" unit of blackness. This extra blackness must be absorbed into the tree (by a red node), or propagated up to the root and out of the tree
- There are four cases –our examples and "rules" assume that V is a left child. There are symmetric cases for V as a right child

Terminology

- The node just deleted was U
- The node that replaces it is V, which has an extra unit of blackness
- The parent of V is P
- The sibling of V is S

RB Delete: Case 1

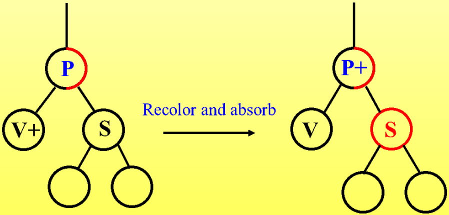
- Case 1: V's sibling, S, is red



- NOT a terminal case –One of the other cases will now apply
- All other cases apply when S is black

RB Delete: Case 2

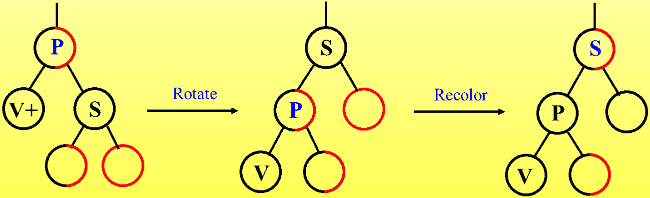
- Case 2: V's sibling, S, is black and has two black children



- Recolor S to be red
- P absorbs V's extra blackness:
  1. If P is red, we're done
  2. If P is black, it now has extra blackness and problem has been propagated up the tree

RB Delete: Case 3

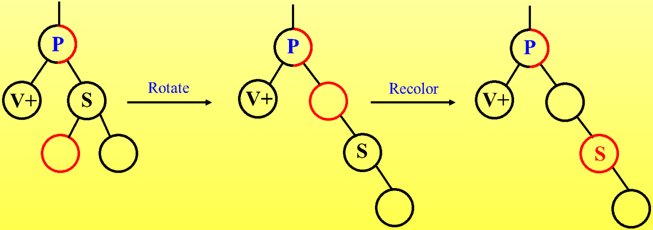
- Case 3: S is black, S's right child is red



- 1. Rotate S around P
  2. Swap colors of S and P, and color S's right child black
- This is the terminal case – we're done

RB Delete: Case 4

- Case 4: S is black, S's right child is black and S's left child is red



- 1. Rotate S's left child around S
  2. Swap color of S and S's left child before rotation
  3. Now in case 3. e.g., V's sibling is black, which has a red right child.

Augmenting Data Structures: Overview

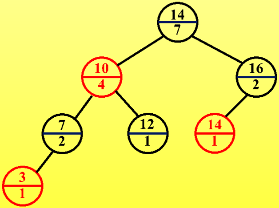
- A "textbook" data structure is enough in some situations
- Many others require a dash of creativity and it will suffice to augment a textbook data structure by storing additional information in it
- The added information must be updated and maintained by the ordinary operations on the data structure
- Then we can program new operations for the data structure to support the desired application

Dynamic Order Statistics

- We want to augment red-black trees so that they can support fast order-statistic operations
- So introducing an augmenting data structure: order-statistic tree:
  1. Besides the usual red-black tree fields $key[x]$, $color[x]$, $p[x]$, $left[x]$, and $right[x]$ in a node x, it has additional information: field $size[x]$
  2. $Size[x]$ is the number of (internal) nodes in the subtree rooted at x(including x itself)
  3. $Size[x] = size[left[x]] + size[right[x]] + 1(size[nil[T]] = 0)$

Order-Statistic Tree - Example

- Keys need not to be distinct in an order-statistic tree
- In the presence of equal keys, it is well defined that the rank of an element is the position at which it would be printed in an inorder walk of the tree



Retrieving an Element with a Given Rank i

$\text{OSSELECT}(x,\ i)$ // return the i-th smallest element

$r \leftarrow size[left[x]] + 1$
if $i = r$ **then**
  return x
else if $i < r$ **then**
  return $\text{OSSELECT}(left[x],\ i)$
else
  return $\text{OSSELECT}(right[x],\ i - r)$

- The running time of OSSELECT is $O(log\ n)$ for a dynamic set of n elements

Determining the Rank of an Element x in tree T

$\textsc{OSRank}(T, x)$

```
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 $\textsc{OSRank}$ is at worst proportional to the height of the tree: $O(\log n)$ on an n-node order-statistic tree

Maintaining Subtree Sizes

- The *size* field in each node should be efficiently maintained by the basic modifying operations on red-black trees
- Insertion:
  1. TREEINSERT : we simply increment size[x] for each node x on the path traversed from the root down toward the leaves. $O(log\ n)$
  2. ROTATE : each rotation only have the sizefields of two nodes invalidated. $O(1)$
- How about deletion?

How to Augment a Data Structure

- Augmenting a data structure can be broken into four steps:
  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
  4. Developing new operations

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 $O(log\ n)$ performance of these operations
- Proof?

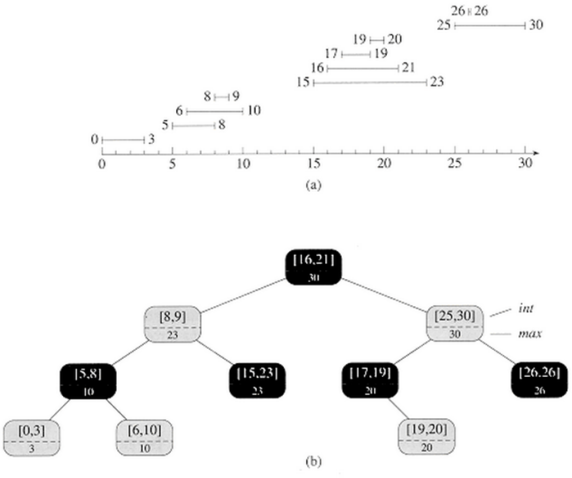
Interval Trees: Overview

- 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)
- Any two intervals i and i' satisfy the interval trichotomy, that is, exactly one of the following three properties holds:
  1. i and i' overlap
  2. i is to the left of i'
  3. i is to the right of i'
- An interval tree is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval $int[x]$
- Interval trees support the following operations $\textsc{IntervalInsert}(T, x)$, $\textsc{IntervalDelete}(T, x)$, $\textsc{IntervalSearch}(T, i)$

Design of an Interval Tree

- Underlying Data Structure:
  Choose a red-black tree in which each node x contains an interval $int[x]$ and the key of x is $low[int[x]]$
- 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
- Maintaining the Information
  $max[x] = max(high[int[x]],\ max[left[x]],\ max[right[x]])$
  $O(log\ n)$
- Developing New Operations
  $\textsc{IntervalSearch}(T,\ i)$

Interval Tree - Example



Interval Search

INTERVALSEARCH(T, i)

//Given interval i, return a node whose interval overlaps i, or it returns $nil[T]$ and the tree T contains no node whose interval overlaps i

$x \leftarrow root[T]$
while $x \neq nil[T]$ and i does not overlap $int[x]$ **do**
  if $left[x] \neq nil[T]$ and $max[left[x]] \geq low[i]$ **then**
    $x \leftarrow left[x]$
  else
    $x \leftarrow right[x]$
return x

Analysis of Interval Search

- Time complexity
  1. The search for an interval that overlaps i starts with x at the root of the tree and proceeds downward
  2. Each iteration of the basic loop takes $O(1)$ time. The height of an n-node red-black tree is $O(log\ n)$
  3. Thus, the INTERVALSEARCH procedure takes $O(log\ n)$ time