# Priority queues, binary search trees and variations on them

Presentation by Antonio-Gabriel Sturzu

# Agenda

- Priority queues
- Binary search trees
- Augmenting binary search trees
- Practical applications

# Max Priority queues

- Operations supported by a priority queue:
  - Insertion
  - Retrieval of the element with maximum priority
  - Extraction of the element with maximum priority
  - Increase the priority of an element to a larger specified value

# Max Priority queues

- All these operations can be implemented using a max-heap
- The retrieval of the maximum element can be done in O(1) by accessing the element at A[1] where is our max-heap represented as a vector

# Max Priority queues

- The extraction of the maximum element can be done by replacing the maximum element at position A[1] with the last element at position A[heap-size] and calling the max-heapify procedure in order to maintain the heap property in the root of the heap

# Max Priority queues

- A sample pseudocode for max extraction:

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

# Max Priority queues

- Increase priority operation
  - The basic idea is to go up until we find a parent that will have a greater priority than the new priority of the changed element

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

# Max Priority queues

- The insertion operation can be implemented using the increase key operation
- We just have to add a new element to the end of the heap and call increase key on it with the priority that we want to set

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

# Max Priority queues

- Exercises:
  - The swap operation in increase key requires three assignements. How to reduce this to just one assignement ?
  - How to implement FIFO and LIFO using a priority queue
  - How to implement the decrease priority operation ?
  - How to implement the deletion of an element from a max-heap ?

# Binary search trees

- Why do we need such a data structure ?
  - It permits fast deletion, insertion and search of an element
  - Using the heap we could only do fast delete and insert operations but the search operation would require linear time
  - They have wide applications in database search querys and in computational geometry ( although some more advanced variations are used the basic idea is the same)

# Binary search trees

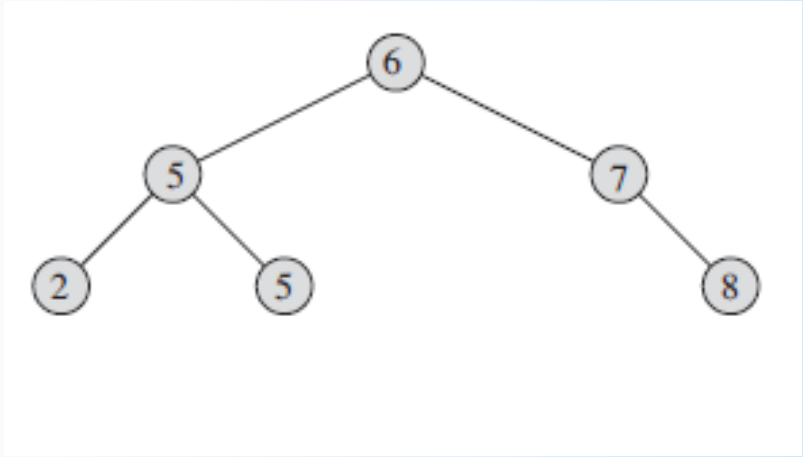
- Each node generally contains at least the following information:
  - Key
  - Left – the left child
  - Right – the right child
  - Parent – the parent node

# Binary search trees

- The property maintained by a binary search tree is the following:
  - For any node x if y is a node is the left subtree of x then the key of y must be less than or equal to the key of x
  - If y is a node in the right subtree of x then the key of y must be greater than or equal to the key of x
- From this we can conclude two important observations:
  - The element with the minimum key will be the leftmost node of the tree
  - The element with the maximum key will be the rightmost node of the tree

# Binary search trees

- Example of a binary search tree:



# Binary search trees

- Operations on binary search trees:
  - Searching a node given a key
  - Finding the maximum and the minimum
  - Finding the successor of a node
  - Inserting a new node with a given key
  - Deleting a node

# Binary search trees

- Searching a node for a given key can be implemented efficiently because of the property that the binary search tree has
- We begin at the root of the tree and each time compare the given key to the key of the current node
- If it is less than or equal we go left, else we go right
- We repeat this procedure until we find the node with the given key

# Binary search trees

- Pseudocode (Cormen, Third Edition)

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2       return x
3  if k < x.key
4       return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

# Binary search trees

- Finding the minimum and maximum in a binary search tree
- The idea here is simple
  - For the minimum, just go every time left until we meet a leaf
  - Similar for the maximum, just that we go right every time
- Pseudocode for the minimum

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2        x = x.left
3  return x
```

# Binary search trees

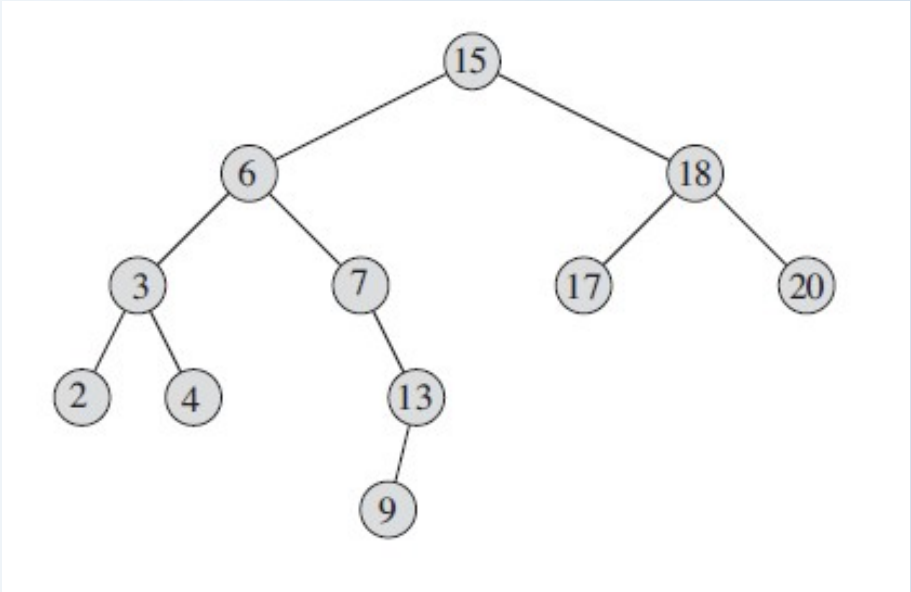
- Finding the successor of a given node
  - The successor is the node which appears after the given node in the array produced by the inorder traversal of the binary search tree
  - If all the keys are distinct it is the node with the smallest key greater than the key of the given node
  - How to find this node ?

# Binary search trees

- We have two cases:
  - If the node X has a right child then the successor node will be the node with the minimum key in the right subtree of the node
  - Else, we need to go up until we find a node whose left child is an ancestor of node X. This node will be the predecessor
- Pseudocode

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5       x = y
6       y = y.p
7  return y
```

# Binary search trees



# Binary search trees

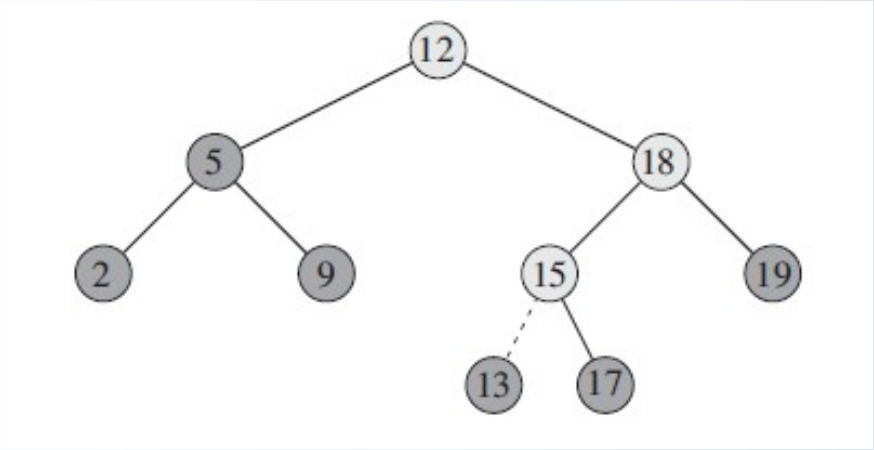
- Inserting a node in a binary search tree
  - Insertion is straightforward. Given the key of the node that has to be inserted we start from the root and just go left or right depending on the values of the keys found on the descending path. We stop when we reach a leaf and insert the node in the left or right of the leaf depending on the value of the given key. The leaf will become the parent of our newly inserted node.

# Binary search trees

- Pseudocode

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == NIL
10        T.root = z          // tree T was empty
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
```

# Binary search trees



# Binary search trees

- Deleting a node from a binary search tree
- Here we have three cases:
  - In the first case the node is a leaf
  - In the second case the node has only one child
  - In the third case the node has two childs
- The first case is straightforward because we just eliminate the leaf from the tree
- In the second case we replace the node with its child and the binary search tree property will be preserved

# Binary search trees

- The third case is the most difficult one because we can't repeat the trick from the previous case
- The key idea is to reduce this case to the previous one by using the successor
- If the node has two children then it means that the successor will be located in the right subtree and it will be the leftmost node in this subtree
- This means that the successor will not have a left child !

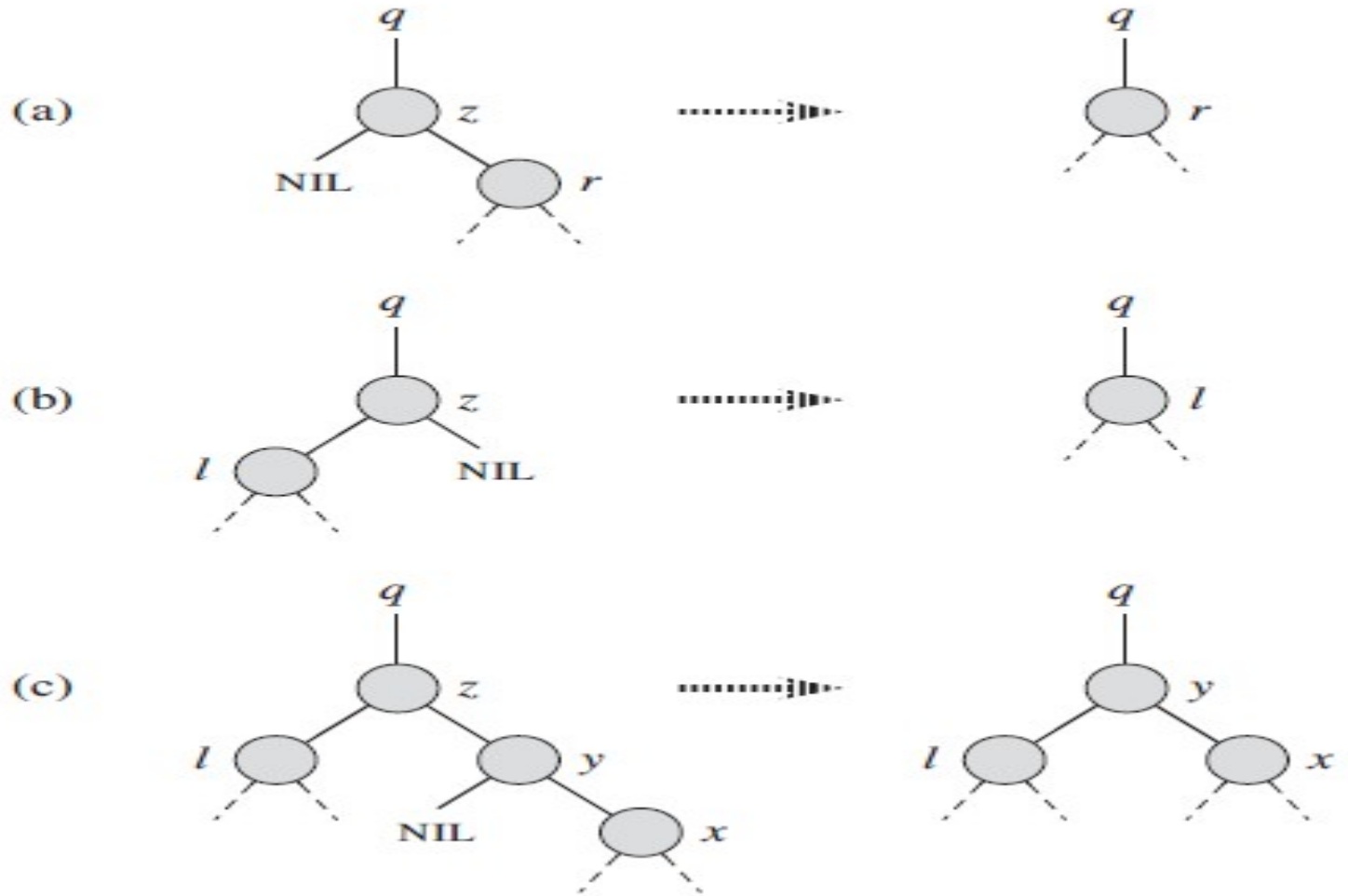
# Binary search trees

- What we have to do is to replace our node with its successor
- This can be done in two steps:
  - First, we must remove the successor from its current place
  - We can do this because it has only a child and we can use the idea from the previous case
  - Second, we replace the node with the successor
- The important observation is that when we replace the node with the successor the binary search tree property will be preserved
- Why ?

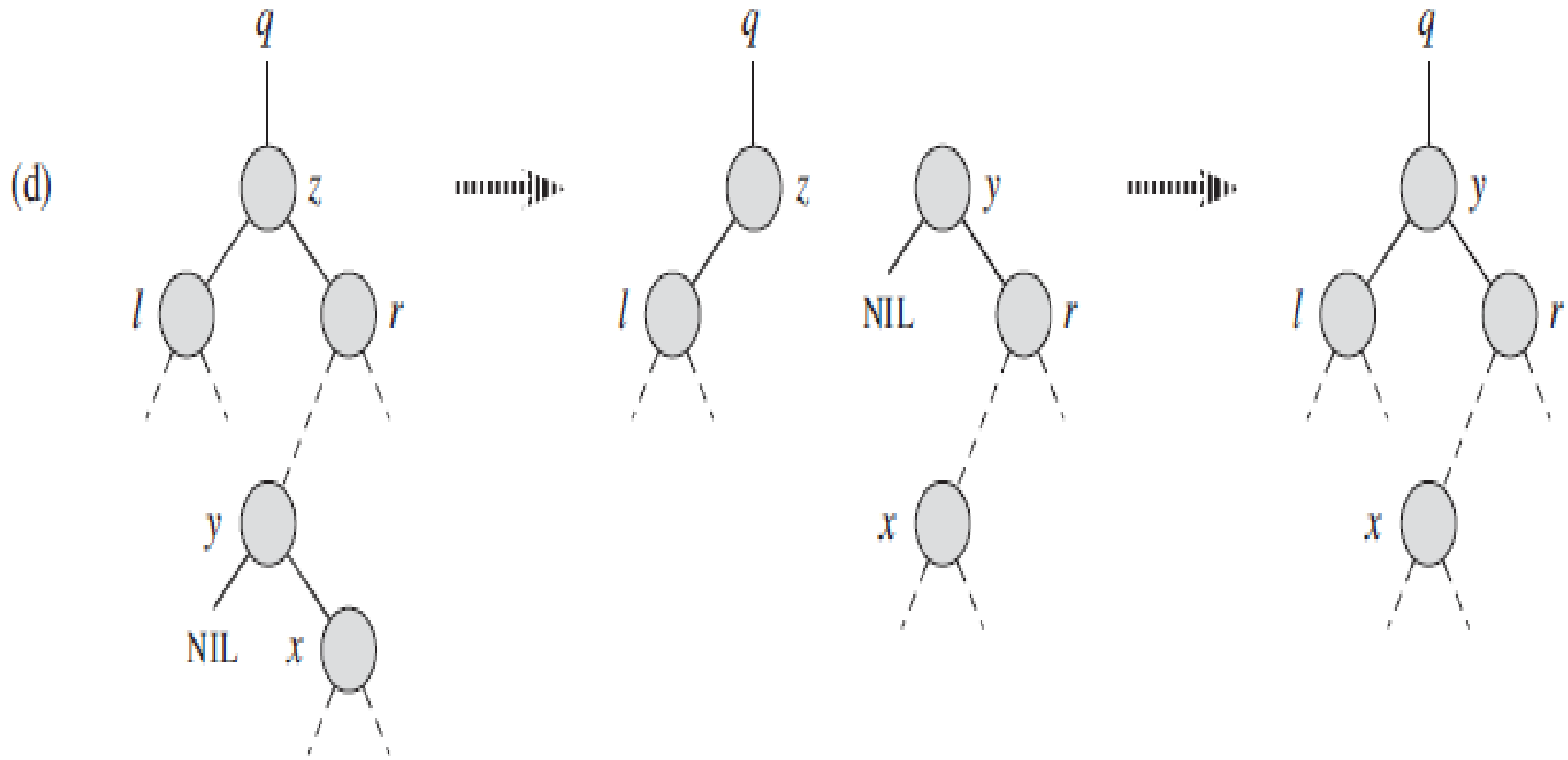
# Binary search trees

- Because the successor has its key greater than or equal to the node to delete, the binary search tree property in the left will be preserved
- In the right the property will be preserved because the successor is the node with the minimum key in the right subtree of our current node
- In the parent of the node the property is preserved because the node and the successor are located in the same subtree rooted at the parent

# Binary search trees



# Binary search trees



# Binary search trees

- Pseudocode

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

# Binary search trees

- Pseudocode

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

# Binary search trees

- The time complexity of all the operations on the binary search tree is O(h) where h is the height of the binary search tree
- How fast will this be ?
- The answer is that it heavily depends on the values of the keys of the nodes that we insert and on the type of operations that are performed on the tree
- Currently, the average height of the tree when both insertions and deletions are performed is an open subject
- But, if only insertions are peformed it can be proved that the height of a randomly built binary search tree is on average O(logn)

# Binary search trees

- This means that given a sequence of keys, any permutation of it is equally possible when performing the chain of insertions in the empty binary search tree
- A major problem with binary search trees is how to maintain the tree as balanced as possible
- Problems:
- Given a binary tree how to verify that it is a binary search tree
- How can we build a binary search tree when we have a lot of keys with equal values in order to balance the tree as much as possible

# Binary search trees

- How to use the idea from the binary search tree in order to build an efficient data structure that can tell if a string has been found in a text and how many times it has been found (In the literature this is known as a trie tree)
- If we are given a vector of strings how to use the above data structure in order to sort it lexicographically in $\theta(n)$ where n is the sum of the lenghts of all the strings in the array
- Using the binary searh tree data structure how to retrieve all the numbers located in the closed interval [x,y]

# Augmenting data structures

- Augmenting the binary search data structure in order to retrieve rank information efficiently
- This is known in literature as an order statistic tree
- You can view the order statistic tree as a data structure built on top of a binary search tree like data structure
- Example:
  - Given a stream of numbers, how to answer each time a new number is added to the stream what is the k'th smallest number

# Augmenting data structures

- The main idea here is to add aditional information in the binary search tree in order to be able to answer such queries
- Also, this information must be preserved when performing insertions and all the operations on the binary search tree must have the same time complexity
- The solution is to store in each node of the binary search tree the number of nodes in its subtree including the node itself
- Using this information we will be able to answer the rank queries in O(h) time, where h is the height of the binary search tree

# Augmenting data structures

- To retrieve the node with the k'th smallest key in the subtree rooted at node x we can use the following pseudocode
- GetNode(Node x, int rank )
  - nr=1+x.left.size
  - if(nr==rank)
    - return x
  - else if(nr>rank)
    - GetNode(x.left, rank)
  - else
    - GetNode(x.right, rank-nr)

# Augmenting data structures

- The last step needed in order to solve the initial problem is how to maintain the additional information when performing insertions
- This is straightforward
  - We just follow the descending path from the root to the place where we will insert our new node
  - For each node encountered in the path we just increment the variable that tells us the number of nodes in the subtree rooted at the current node
  - The newly inserted node will have this variable set to 1

# Augmenting data structures

- Now, we are ready to solve the initial problem
- When a new number arrives we just insert a new node with its key set to the number
- When a rank query is made, we just call the GetNode procedure on the root of the tree
- All the operations will still be done in O(h) time
- Another additional requirement would be to also support deletion
- How would you do this ?

# Augmenting data structures

- Interval trees
- Problem:
  - Given a set of closed intervals and a query interval find an interval from the set that overlaps the query interval in an efficient way
- If A and B are two intervals they will ovelap only if:
  - A.high>=B.low and B.high>=A.low
- Otherwise they will not overlap

# Augmenting data structures

- We build the interval tree on top of the binary search tree in the following way:
  - The keys of the binary search tree will be left endpoint of the interval
  - We will store in each node the following additional information:
    - The left and right endpoints of an interval
    - The maximum value of all the right endpoints in the subtree rooted at each node including the node itself
- The only thing that we have to maintain is this maximum value as we do insertions and deletions

# Augmenting data structures

- When performing insertions we update all the maximum values in all the nodes on the descending path
- For deletions we will go upward from the node to delete if it doesn't have two children or upward from the successor node if the node to delete has two children and update the maximum value in each node along the path to the root
- So, the insertion and deletion operations will preserve their O(h) time complexity

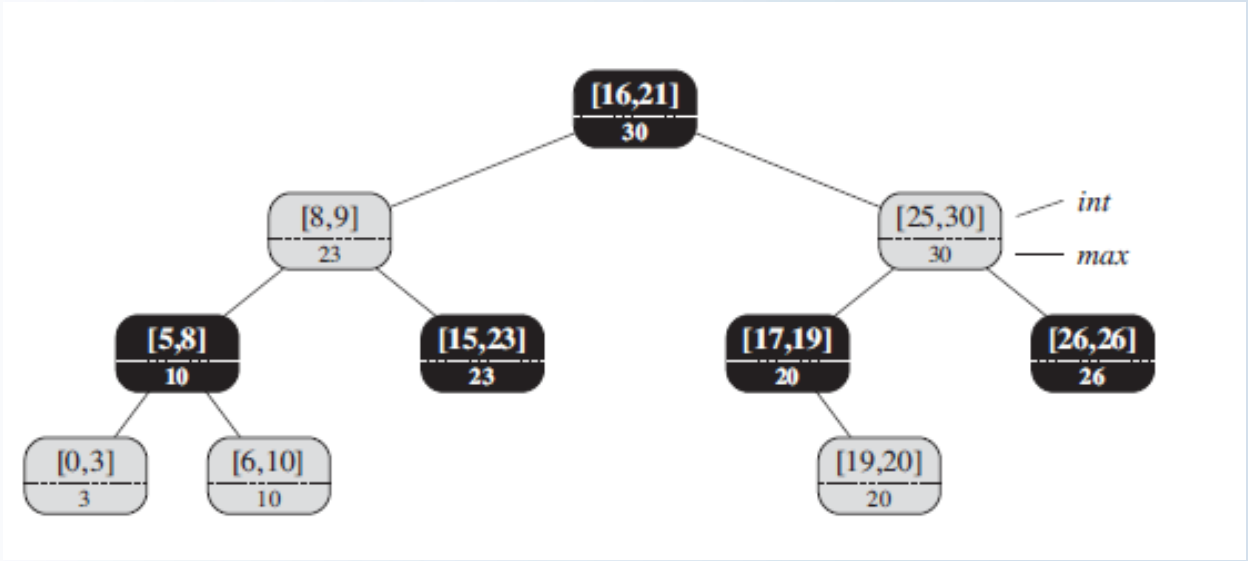
# Augmenting data structures

- The only thing that remains to do is to add the new query operation that finds for a query interval i an interval from the interval tree that overlaps with it
- Here is the pseudocode

```
INTERVAL-SEARCH(T, i)
1  x = T.root
2  while x ≠ T.nil and i does not overlap x.int
3      if x.left ≠ T.nil and x.left.max ≥ i.low
4          x = x.left
5      else x = x.right
6  return x
```

# Augmenting data structures

- An example interval tree



# Augmenting data structures

- The time complexity for this new operation will be O(h) because we always go left or right
- How could we find all the intervals that overlap the query interval i ?