Module 8 - Selection Algorithms

Order Statistics

We begin our discussion by considering a particular problem in identifying a specific element in an unordered list based on its position in a corresponding ordered list.

Definition: The ith order statistic of a set of n elements is the ith smallest element in the set.

Definition: The minimum of a set is the first order statistic.

Definition: The maximum of a set is the nth order statistic.

Definition: The median of a set is the midpoint in the ordered set such that there are as many elements less than the median as there are greater than the median.

Given our definition of order statistics, the problem we will solve here is called the "selection" problem.

Definition: The selection problem is defined as follows:

- Input: A set $\mathbf{A}$ of n distinct elements and a number i where $1 \leq i \leq n$.
- Output: The element that is greater than exactly $i - 1$ other elements in $\mathbf{A}$.

As a naïve approach, the selection problem can be solved in $O(n \lg n)$ time by first sorting the elements in $\mathbf{A}$ and then selecting the ith element in the sorted set. This appears to be pretty good, but we will consider algorithms that improve on this bound.

Above, we defined minimum and maximum in terms of order statistics. These can be found in $O(n)$ time simply by scanning the data set as follows:

Algorithm 1 Minimum

```
MINIMUM(A)
  min ← A[1];
  for i ← 2 to length(A) do
    if min > A[i] then
      min ← A[i];
    end if;
  end for;
  return min;
```

The above provides the code for minimum, and maximum can be constructed similarly. An interesting question: What if we want to find the minimum and maximum element at the same time? Can we improve the performance beyond $O(n)$? As we will see, we cannot improve beyond $O(n)$, but we can improve the constant. Specifically, the naïve approach where we do two scans would require $2n - 2$ comparisons, but we can improve this to find both the minimum and maximum in comparisons. How?

To do this, all we have to do is maintain the minimum and maximum seen so far as you scan the list in order. Initially, these will be the first values in the list. Next, scan the list in pairs (i.e., i and $i + 1$) and compare these to determine the smaller. Then compare the smaller to the current min and swap if necessary. Finally, compare the larger to the current max and swap if necessary.

RandomizedSelect

Now let's consider the more general problem of finding the ith order statistic. For this algorithm we will, once again, apply a divide-and-conquer strategy. As we will see the approach will be very similar to Quicksort in that we will partition the data around a pivot. The difference is that, instead of sorting the data, we will only process one partition.

To find the pivot, we will apply a randomized method:

Algorithm 2 Randomized Partition

```
RandomizedPartition(A, p, r)
  i ← Rand(p, r);
  exchange A[p] and A[i];
  return Partition(A, p, r);
```

As we see, the difference is that the pivot is selected at random from the current partition. That element is then swapped with the first element in the partition, and Partition is used from there. Now we are ready to define the RandomizedSelect algorithm:

Algorithm 3 Randomized Select

```
RandomizedSelect(A, p, r, i)
  if p = r then
    return A[p];
  end if;
  q ← RandomizedPartition(A, p, r);
  k ← q − p + 1;
  if i ≤ k then
    return RandomizedSelect(A, p, q, i);
  else
    return RandomizedSelect(A, q + 1, r, i − k);
  end if;
```

As we see, the algorithm is recursive (in fact, tail-recursive). For the base case, it tests to see if there is only one element in the partition. If so, that must be the order statistic. Otherwise, the current subarray is partitioned. The algorithm is called with i to indicate the desired order statistic. Now that the partitions have been found, we need to search in the appropriate partition for the order statistic. Specifically, if i is less than the index splitting the array, then we continue searching in the "left" partition. However, if i is greater than the index splitting the array, we need to search in the "right" partition, but we need to adjust the order statistic itself. Specifically, we now search the right-hand partition with order statistic $i - k$ to account for removing the k elements in the left partition.

Unfortunately, the worst case performance for this algorithm, just like Quicksort, is $O(n^2)$. This is because, even with the randomized strategy, it is possible to be unlucky every time when we partition the data such that we only reduce the needed partition by 1. Even so, the algorithm performs well on average. Let's determine the average case complexity.

First, we observe that the probability of the left partition having a particular size is $1/n$. As a result, the probability of either the left or the right partition having exactly one element is $2/n$. From here, assume we randomly split according to this distribution, but we always end up with the order statistic being in the larger of the two partitions. (If the two partitions are the same size, then it does not matter which one we have to search.) Then we can derive the following recurrence:

$$T(n) \leq \frac{1}{n}\left(T(\max\{1, n-1\}) + \sum_{k=1}^{n-1} T(\max\{k, n-k\})\right) + O(n).$$

Note that $\max\{1, n-1\} = n-1$. In addition, $\max\{k, n-k\} = k$ if $k \geq \lceil n/2 \rceil$ and $n-k$ otherwise. Because of this, we see that the term $T(\lceil n/2 \rceil + i)$appears twice for $i = 2, \ldots, n-1$. In addition, if n is odd, $T(\lceil n/2 \rceil)$ will also appear twice. Therefore, we can refine the above expression to be:

$$T(n) \leq \frac{1}{n}\left(T(n-1) + 2\sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} T(k)\right) + O(n).$$

Recall that $T(n) = T(n-1) + O(n) = O(n^2)$. Since we are dividing by n, the $T(n-1)$ part of the above expression is absorbed into the $O(n)$ term at the end of the expression. This leaves the following recurrence to solve:

$$T(n) = \frac{2}{n} \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} T(k) + O(n).$$

We can solve this by substitution. First assume $T(n) \leq cn$. If we are correct, this means the average case performance for RandomizedSelect is linear in the input! So let's use the above as our inductive hypothesis as follows:

$$\begin{aligned}
T(n) &\leq \frac{2}{n} \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} ck + O(n) \\
&\leq \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k \right) + O(n) \\
&= \frac{2c}{n} \left(\frac{1}{2}(n-1)n - \frac{1}{2} \left(\left\lceil \frac{n}{2} \right\rceil - 1 \right) \left\lceil \frac{n}{2} \right\rceil \right) + O(n) \\
&\leq c(n-1) - \frac{c}{n} \left(\frac{n}{2} - 1 \right) \left(\frac{n}{2} \right) + O(n) \\
&= c \left(\frac{3}{4}n - \frac{1}{2} \right) + O(n) \\
&\leq cn.
\end{aligned}$$

In this derivation, the second line follows from the fact that we are considering the full sum and subtracting out the portion not included in the original sum. This allows us to apply the closed form solution to the resulting arithmetic sums to get the third line. From there, the rest is algebra.

Augmenting Data Structures

On the surface the topic of augmenting data structures appears to have nothing to do with selection; however, we will use a selection problem as our primary example for how to augment a data structure.

Often, the basic "textbook" data structures are deemed to be insufficient to solve real-world problems. This is because the data structures, as defined, require a lot of additional work to maintain the data structure and pass supporting information around. Sometimes, these data structures can be "augmented" with information not normally maintained with the data structure to improve what would otherwise be unwieldy performance.

The process of augmenting a data structure follows four steps:

1. Choose an appropriate underlying data structure.
2. Determine what additional information is required to support the problem to be solved.
3. Verify that the additional information can be maintained with the data structure's basic modification operators without increasing the computational complexity of those operators.
4. Modify the operators to maintain the required information.

Augmented Red-Black Trees

To illustrate how to augment a data structure for a specific purpose, we will focus here on the red-black tree. Based on the four-step process in the following section, we will begin by proving a theorem that, while specific to the red-black tree, applies in principle to any augmentation.

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

Proof: Recall that, for red-black trees, insertion occurs in two phases. During the first phase, x is inserted as a child of an existing node, which we denote $parent[x]$. The value of $f(x)$ can be computed in $O(1)$ time since, by supposition, it only depends on information in the current node and the node's children (of which there are none). Once $f(x)$ is determined, the changes to any other f values propagate up the tree. Thus phase 1 requires $O(\lg n)$ time. For phase 2, structural changes only occur via rotations. Since only two nodes are affected by a rotation, the total time for updating f is $O(1)$ during the rotation. Since there are at most two nodes per rotation and at most $O(\lg n)$ rotations, total time to update f in phase 2 is $O(\lg n)$.

Like insertion, deletion occurs in two phases. In phase 1, a structural change occurs if the deleted node is replaced by its successor or whenever either the deleted node or the successor node is spliced out. Such changes propagate up the tree, causing $O(\lg n)$ updates to f. The repair in phase 2 is $O(1)$, and there are $O(\lg n)$ repairs, so the total time to update f in phase 2 is $O(\lg n)$.

Since all phases for insertion and deletion require $O(\lg n)$ updates to f, the asymptotic running time for insertion and deletion in the augmented red-black tree is still $O(\lg n)$. QED

Order Statistic Trees

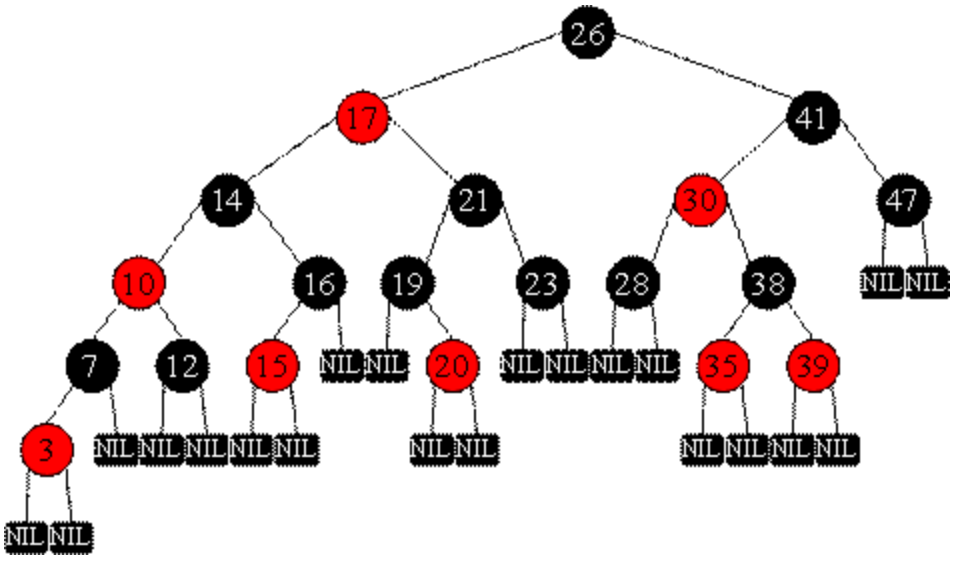
Consider now the application of using a red-black tree to determine order statistics. Recall that, in an unordered array, we can find an arbitrary order statistic in $O(n)$ time. We will now show that, by augmenting a red-black tree, we can find an arbitrary order statistic in $O(\lg n)$ time. We will also consider the following variant on the problem. Let the rank of an element in a sorted list be the position in the linear order of that list. We will also be able to find the rank of a particular data element in $O(\lg n)$ time using the augmented red-black tree.

Definition: An order statistic tree ($\mathbf{T}$) is a red-black tree that has been augmented such that each node of the tree holds the size (i.e., number of nodes) of the subtree rooted at that node.

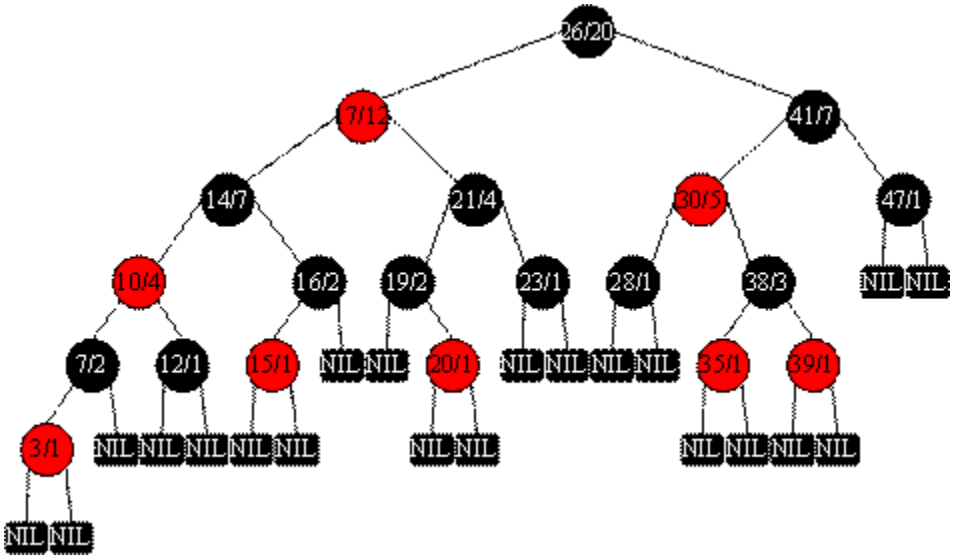
Recall that a typical node in a red-black tree includes fields key, color, parent, left, and right. As suggested by the definition above, we will augment the red-black tree to include a new field, size. Notice that the size of a subtree can be determined simply as

$$size(x) = size(left[x]) + size(right[x]) + 1.$$

Recall the red-black tree above, which we repeat here.



Now consider the following "augmented" version of this same tree.



In this tree, each node shows two fields (in addition to the pointers)—the key value and the size of the subtree rooted at that node. For this size field, the NIL leaf nodes are not included.

Operations on the Order-Statistic Tree We need to make sure that we have not violated the augmentation theorem in the specific way we have augmented the tree. Recall that red-black trees are dynamic sets, and that the structure of the tree changes as nodes are added or deleted (as described in the augmentation theorem). When the structure changes, the size field will need to be changed as well.

For insertion, maintaining the size in phase 1 simply involves incrementing size as we walk down the tree to find the insertion point. Phase 2 only requires a size change whenever a rotation occurs. Note that rotation is a local operation and only invalidates the size on the two nodes around which rotation occurs. Without loss of generality, consider left rotation (right rotation is symmetric). If we assume that, initially, x is the parent and y is the child, then we can update size in $O(1)$ time as follows:

$$\begin{aligned} size(y) &\leftarrow size(x) \\ size(x) &\leftarrow size(left[x]) + size(right[x]) + 1. \end{aligned}$$

For deletion, since phase 1 results in a node being spliced out, we need to repair the size. This can be done by traversing from y (the spliced out node) to the root, decrementing size along the way. As with insertion, phase 2 in deletion just involves rotations, and the size updates for deletion rotations are the same.

Since both insertion and deletion involve $O(\lg n)$ updates to size, and these updates require $O(1)$ time, the insertion and deletion operations are still $O(\lg n)$ in accordance with the augmentation theorem.

OSSelect Now that we know we can construct and maintain our order-statistic tree, let's see what we can do with it relative to the order-statistic problem. We define an algorithm, OSSelect, that is designed to return the element from the order-statistic tree corresponding to the ith order statistic.

Observe that the value of $size(left[x])$ for any given node is the number of nodes that come before x during an inorder walk of the subtree rooted at x. This means $size(left[x])+1$ is the rank of x in the subtree rooted at x. We can use this property to find the ith order statistic as follows:

Algorithm 4 OS-Select

```
OSSelect(x, i)
  r ← size(left[x]) + 1;
  if i = r then
    return x;
  else if i < r then
    return OSSelect(left[x], i);
  else
    return OSSelect(right[x], i − r);
  end if;
```

The way the algorithm works is to start by determining the rank of the current node within the subtree rooted at that node. This rank is compared to the desired order statistic. If they match, the pointer to that node is returned. If the order statistic is less than the current rank, then we traverse to the node's left child and tree again. If the order statistic is greater than the rank, then we subtract the rank from the target order statistic and traverse to the right child.

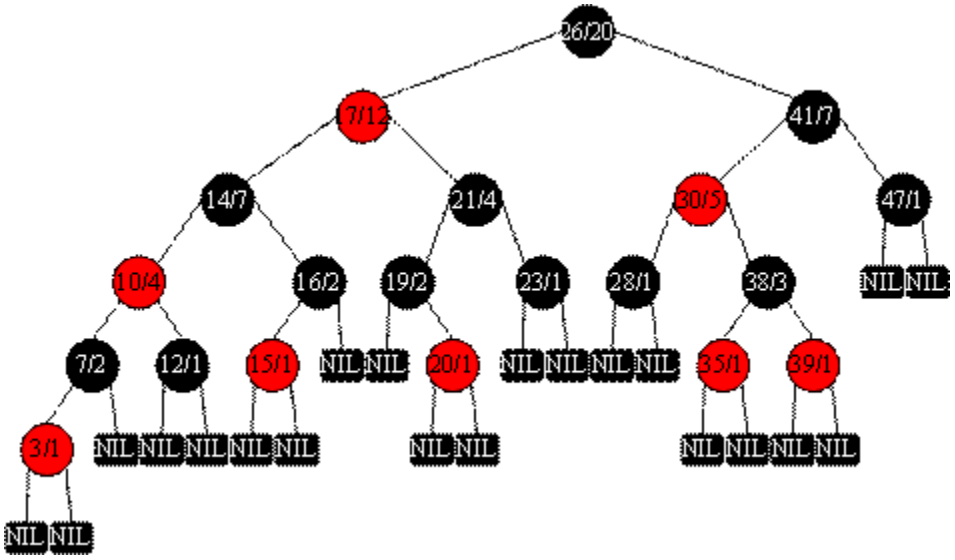
Notice that this is a recursive (in fact tail-recursive) algorithm. Each recursive call goes down one level in the order-statistic tree and never goes back up. Thus the complexity must be bound by the height of the tree. Since this is a red-black tree, it is balanced (i.e., $h = O(\lg n)$), so the running time for OSSelect must be $O(\lg n)$.

OSRank Next consider the problem where we know a particular node in the order-statistic tree but want to know its rank in the linear order of all elements in the tree. The rank of node x is simply the number of nodes that precede x in a inorder walk of the tree (plus one). We do not want to do an inorder walk of the tree since that will require $O(n)$ time.

To build up rank as we search the tree, we need to consider the size of x's left subtree along the way. In addition, we must consider the sizes of the left subtrees (plus one) of all of x's ancestors that do not contain x. For example, in the order-statistic tree above (which we repeat here),

Algorithm 5 OS-Rank

```
OSRANK(T, x)
  r ← size(left[x]) + 1;
  y ← x;
  while r ≠ root[T] do
    if y = right[parent[y]] then
      r ← r + size(left[parent[y]]) + 1;
    end if;
    y ← parent[y];
  end while;
  return r;
```



suppose we want to know the rank of node 38. We see that its rank within its subtree is $size(left[38])+1 = 2$. We also need to consider the size of the left subtrees (plus one) of node 26 and node 30. We do not need to worry about node 41 because node 38 is in its left subtree already. So we get for node 26, $size(left[26])+1 = 13$, and for node 30, $size(left[30]) + 1 = 2$. Adding all three terms together, we find that the rank of node 38 is $2 + 13 + 2 = 17$. The algorithm for finding this is as follows:

Since we know x to start, we do not have to search the tree for that node. Instead, we start at the node and work our way back to the root. The initial estimate for the rank r is the size of x's left subtree plus one. Then, as we traverse up the tree to the root, if we cross a right branch, we add the size of that ancestor's left subtree plus one to the rank.

An iteration of the loop requires constant time since it only involves a test and (possibly) an addition. We traverse up the tree and never walk back down. This means the algorithm is bound by the height of the tree, which is $O(\lg n)$. Thus, the complexity of the algorithm is $O(\lg n)$.

Interval Trees

Let's consider another application where we will augment the red-black tree to help solve a problem. In this case, suppose our red-black tree implements a dynamic set that is storing intervals.

Definition: A closed interval is an ordered pair of real numbers $[t_1, t_2]$ such that $t_1 \leq t_2$ corresponding to $\{t \in \Re | t_1 \leq t \leq t_2\}$.

Definition: An open interval is an ordered pair of real numbers (t_1, t_2) such that $t_1 \leq t_2$ corresponding to $\{t \in \Re | t_1 < t < t_2\}$. Thus, the endpoints t_1 and t_2 are not included in the interval.

Definition: A half interval is an ordered pair of real numbers $[t_1, t_2)$ such that $t_1 \leq t_2$ corresponding to $\{t \in \Re | t_1 \leq t < t_2\}$ or $(t_1, t_2]$ such that $t_1 \leq t_2$ corresponding to $\{t \in \Re | t_1 < t \leq t_2\}$. Thus only one of the endpoints, either t_1 or t_2 is not included in the interval.

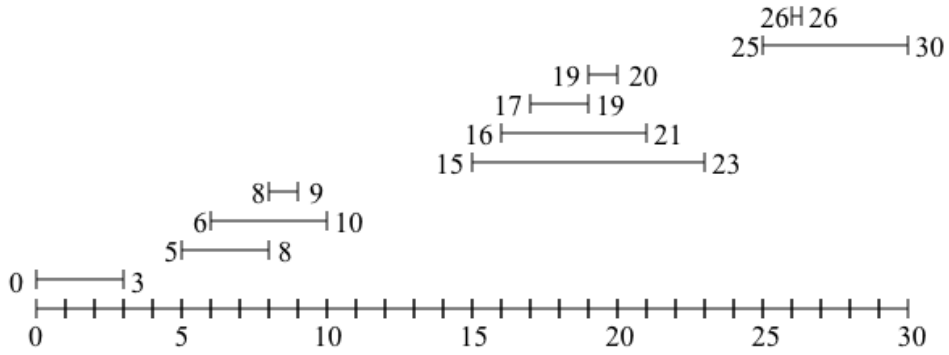
For our interval trees, we will assume we are storing closed intervals.

We can represent a closed interval $[t_1, t_2]$ in the red-black tree as an object with fields $low[i] = t_1$ and $high[i] = t_2$. Then we can say that two intervals, i_1 and i_2, overlap if $i_1 \cap i_2 \neq \oslash$. This can be expressed equivalently with the inequalities $low[i_1] \leq high[i_2]$, and $low[i_2] \leq high[i_1]$.

A key property held by any pair of intervals is referred to as the interval trichotomy. Specifically, any pair of intervals i_1 and i_2 must satisfy exactly one of the following:

1. $i_1$and i_2 overlap
2. $high[i_1] < low[i_2]$
3. $high[i_2] < low[i_1]$

For example, suppose we wish to work with the following intervals:



Now we are prepared to define the interval tree data structure. An interval tree is a red-black tree that maintains a dynamic set of intervals where each element x stored in the tree contains an interval $int[x]$. Given this data structure, an interval tree supports the following operations:

- IntervalInsert($\mathbf{T}$, x)—adds element x to $\mathbf{T}$ (where x contains the interval to be added).
- IntervalDelete($\mathbf{T}$, x)—deletes element x from $\mathbf{T}$.
- IntervalSearch($\mathbf{T}$, i)—returns a pointer to element x in $\mathbf{T}$ such that $int[x]$ overlaps i. It returns NIL otherwise.

Next we need to augment the interval tree to ensure that these operations can be performed without increasing the complexity over the basic red-black tree operations. To do this, we complete the following steps.

1. Choose an underlying data structure

   We have already chosen the red-black tree as the data structure. For this data structure, each node will include $int[x]$ in addition to the other, standard fields. The key of a node, however, will be the low endpoint of $int[x]$. Thus performing an inorder walk of the tree will return the intervals sorted on their low endpoints.

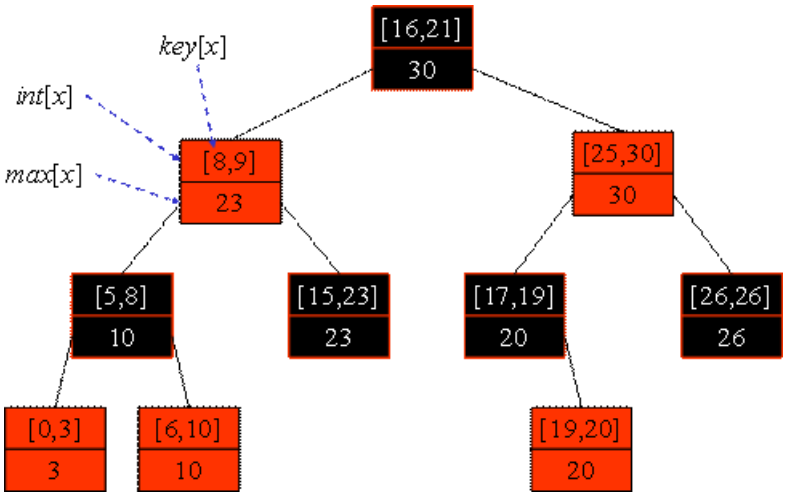
2. Determine additional information

   Now we need to include the information on the intervals themselves. In addition, we will augment the nodes to include $max[x]$, which is the maximum value of any interval endpoint in the subtree rooted at x.

Algorithm 6 Interval Search

```
INTERVALSEARCH(T, i)
  x ← root[T];
  while x ≠ NIL and i does not overlapint[x] do
    if left[x] ≠ NIL and max[left[x]] ≥ low[i] then
      x ← left[x];
    else
      x ← right[x];
    end if;
  end while;
  return x;
```

An example interval tree with this type of augmentation is as follows:



3. Verify the augmented information can be maintained without affecting the asymptotic behavior of the operators.

   The intervals are stored directly and not derived from other elements in the tree. The $max[x]$ field is determined by examining the current interval at x relative to the maximum endpoint of its two children. Thus

   $$max[x] = \max\{high[int[x]], max[left[x]], max[right[x]]\}.$$

   Care should be taken in reading this equation due to the overloading of notation. Specifically, the italicized "max" refers to the maximum endpoint field in the node x, but the un-italicized "max" refers to finding the maximum value of the set. Note also that the augmentation theorem requires us to perform insertion, deletion, and search in $O(\lg n)$ time.

4. Modify the operators to maintain the performance bounds.

   We already have INTERVALINSERT and INTERVALDELETE as modified with the above maximization step. We can prove this calculation does not modify the performance of insertion or deletion using an argument parallel to the order-statistic tree. For INTERVALSEARCH, we assume searching will return the first interval in thee tree that overlaps the target interval i. If no interval is found, traversal will hit a leaf and return NIL. Pseudocode for the algorithm is as follows:

   This algorithm scans from the root downward until it finds the first overlapping interval (we will

discuss correctness in a moment). Since the traversal is limited by the height of the tree, searching must require $O(\lg n)$ time.

Now let's consider the correctness of searching for an overlapping interval in the interval tree. We claim that, at any node x, if $int[x]$ does not overlap interval i, then the search algorithm will always proceed in a safe direction (i.e., one that does not require backtracking).

Theorem: Consider any pass through the while loop in IntervalSearch. We consider two cases.

- If we traverse to $left[x]$, the x's left subtree either contains an interval that overlaps i or no overlap exists in x's right subtree.
- If we traverse to $right[x]$, then no overlap exists in x's left subtree.

Proof: As we will see, the proof for this theorem depends on the interval trichotomy.

We start the proof with case 2. If we branch to $right[x]$, then either $left[x] = \text{NIL}$ or $max[left[x]] < low[i]$. This is evident from the conditional statement in the algorithm. Note that if $left[x] = \text{NIL}$, then there is no subtree to the left containing an overlapping interval because there is no left subtree. Suppose $left[x] \neq \text{NIL}$ and $max[left[x]] < low[i]$, and let i' be some interval in x's left subtree. Since $max[left[x]]$ is the largest endpoint in x's left subtree, we have $high[i'] \leq max[left[x]] < low[i]$. Thus, by the interval trichotomy, i and i' cannot overlap.

Now consider case 1. We start by assuming that no interval in x's left subtree overlaps i (since if any do, we would find one). This means we only need to prove that no intervals in x's right subtree overlap i. If we branch left, then $max[left[x]] \geq low[i]$ must hold. By definition of $max[x]$, some interval must exist in x's subtree such that $high[i'] \max[left[x]] \geq low[i]$, but by our assumption i and i' do not overlap. Since i and i' don't overlap, and since $high[i'] \geq low[i]$, it follows from the interval trichotomy that $high[i] < low[i']$. Now recall that interval trees are keyed on the low endpoints of the intervals. This means that, for any interval i'' in x's right subtree,

$$\begin{aligned} high[i] &< low[i'] \\ &\leq low[i'']. \end{aligned}$$

So by the interval trichotomy, i and i'' cannot overlap. QED