Chapter 14: Data Structure Augmentation - Introduction

- This topic deals with augmenting an existing data structure for a new purpose
- The data structure used in all examples is a red-black tree
- Generally, the adapted data structure is enhanced with additional structure and additional operations to support the new application
In this example, RBTs will be adapted for providing order statistics for a data set.

Nodes in the RBT will be enhanced with a new attribute: `size`

- This attribute stores the number of nodes that are rooted at the given node, including the node itself.
- \( x.size = x.left.size + x.right.size + 1 \)

In this representation, the tree may have duplicate key values.

Finding rank

- Since duplicate values are allowed, a node’s rank is defined as its numeric position in an in-order traversal of the tree.
- To retrieve the element with rank \( i \):

```plaintext
OS-Select (x, i)
{
  1 L = x.left.size + 1
  2 if (i == L)
  3     return x
  4   else if (i < L)
  5     return OS-Select(x.left, i)
  6   else
  7     return OS-Select(x.right, i - L)
}
```
Chapter 14: Data Structure Augmentation - Order Statistics (2)

- Analysis:
  * Each call of the algorithm descends one level in the tree
  * In the worst case, reach lowest level
  * For RBT, tree’s height ∈ \( \mathcal{O}(\lg n) \)
  * So, run time \( T_{\text{worst}} \) ∈ \( \mathcal{O}(\lg n) \)

  - Since insertion and deletion operations affect the structure of the tree, they need to be modified to properly maintain size

- The real issue is whether the size attribute can be maintained in \( \mathcal{O}(\lg n) \) time

- Insertion
  
  - This operation is performed in two steps:
    1. Descend from the root toward the leaves to find the insertion location
    2. Ascend from the insertion location toward the root repairing RBT violations

  - Adjustments to algorithm:
    1. Step 1
      * No structural changes occur
      * Simply traverse the tree from the point of insertion to the root, adding 1 to the size attribute of each node encountered
    2. Step 2
      * Involves rotations, at most two
      * During a rotation, only two nodes change height
      * Note that the number of nodes rooted at the pivot location before and after the rotation does not change
      * The node that moves a level down can have it’s size recomputed based on the sizes of its new children
Chapter 14: Data Structure Augmentation - Order Statistics (3)

The adjustment to the insertion algorithm:

\[ y.\text{size} = x.\text{size} \]
\[ x.\text{size} = x.\text{left}.\text{size} + x.\text{right}.\text{size} + 1 \]

- Analysis
  * Step 1 traverses at most the tree’s height \( \in O(lg n) \)
  * Step 2 runs in constant time
  * RB-Insert \( \in O(lg n) \)

- Deletion
  - This operation also performed in two steps:
    1. Node \( y \) is either deleted or migrated toward the root
    2. Tree is reorganized to repair RBT violations using at most three rotations
  - Adjustments to algorithm:
    1. Step 1
      * No structural changes occur
      * Simply traverse the tree from \( y \)'s location to the root, subtracting 1 from the \textit{size} attribute of each node encountered
    2. Step 2
      * Treat the same as for insertion
  - Analysis
    * Same as for insertion
Chapter 14: Data Structure Augmentation - Guidelines

• Augmenting an existing data structure to support additional functionality entails four steps:

  1. Choose the underlying data structure
     − Given the functionality that is desired, select a DS that already supports some of the desired functionality
     − Choose a DS that will support the functionality efficiently
  2. Determine any additional values required
     − These are usually chosen to facilitate the additional processing required in an efficient manner
  3. Insure that these values can be maintained in an efficient manner by the operations intrinsic to the DS
  4. Develop new operations required to implement the new functionality

• In augmenting RBTs for use in order statistics,

  1. RBTs were chosen because they guarantee efficient run times for dynamic set operations like Minimum, Search, etc.
  2. The size attribute was chosen to facilitate efficient determination of rank
  3. The operations that can affect the size attribute - insertion and deletion - were modified to insure efficient maintenance of size
  4. New operations Select and Rank were designed to extend the functionality of the existing DS to the new application

• Theorem 14.1: This theorem states that any extension to RBTs that involve new features added to a node can be maintained in $O(lg n)$ time
Chapter 14: Data Structure Augmentation - Interval Trees

• Given any two intervals \( i_1 \) and \( i_2 \), they obey the interval trichotomy:
  1. Either \( i_1 \) precedes \( i_2 \),
  2. \( i_2 \) precedes \( i_1 \), or
  3. \( i_1 \) and \( i_2 \) overlap.

• Furthermore, there are four possible ways in which the two intervals can overlap:
  1. Either \( i_1 \) contains \( i_2 \),
  2. \( i_2 \) contains \( i_1 \),
  3. \( i_1 \)'s end overlaps \( i_2 \)'s start, or
  4. \( i_2 \)'s end overlaps \( i_1 \)'s start.

• An interval can be represented by an object with the following attributes:
  – high: The high limit of the interval
  – low: The low limit of the interval

• An interval tree is an RBT that stores intervals
  – Want to be able to support the following operations:
    1. \( \text{Interval-Insert}(T, x) \)
    2. \( \text{Interval-Delete}(T, x) \)
    3. \( \text{Interval-Search}(T, i) \)
      * Returns a pointer to the element that overlaps interval \( i \) (or to \( T.\text{nil} \))
The adaptation process applied to this problem:

1. Step 1: Choose the DS
   – RBTs chosen for the same reasons as for order statistics

2. Step 2: Identify additional information to be represented
   (a) Each node will hold an interval (or pointer to one)
      – Key will be the low limit of the interval
   (b) Max attribute
      – The value of the largest high limit of the intervals rooted at the node

3. Step 3: Maintaining the new attributes (max)
   – Refer to Theorem 14.1

4. Step 4: Developing new functionality
   – This entails \texttt{Interval-Search}(T, i)
     \begin{verbatim}
     Interval-Search (T, i)
     {
     1     x = T.root
     2     while (x != T.nil AND !overlap(i, x.interval))
     3       if (x.left != T.nil AND x.left.max >= i.low)
     4           x = x.left
     5         else
     6           x = x.right
     7     return x
     \end{verbatim}

• Theorem 14.2: Any execution of \texttt{Interval-Search}(T, i)) either returns a node whose interval overlaps i, or returns T.nil and the tree contains no node with an interval that does overlap i.