Module 1 Part C - Trees and Forests

Athens State University

January 5, 2016

Outline

Contents

1 The Tree ADT

Binary tree

Definition 1 (Definition)**.** A *binary tree* is an ADT which defines a set of "nodes", with each node containing a data element, a left pointer, and a right pointer.

- The left and right pointers recursively point to smaller "subtrees" on each side.
- The empty tree is a null pointer that represents a binary tree with no element
- A *rooted binary tree* includes a pointer to the topmost node in the tree.
- The bottom nodes of the tree have no pointers to subtrees and are denoted as a *leaf node*.

Binary tree

Interface

- *Insert(n: Node)*: Add the node n to a tree.
- *Delete(n: Node)*: Delete the node n from a tree.
- *Left(n: Node)*: Return the left sub-tree at node n.
- *Right(n: Node)*: Return the right sub-tree at node n.

Definitions

- *Ancestor*: a parent of a node
- *Subtree*: Any node in a tree can be viewed as the root of a new, smaller tree that contains the descendants of that node. You can have *left subtrees* and *right subtrees*.

Types of binary trees

- *Full binary tree*: every node other than the leaves has two children
- *Perfect binary tree*: A full tree in which all leaves have the same depth or same level
- *Complete binary tree*: Every level, except possibly the last, is completely filled, and all nodes are as far left as possible
- *Balanced binary tree*: A balanced binary tree has the minimum possible maximum height.

Properties

- A binary tree of n elements has $n - 1$ edges
- The height h of a binary tree with n elements is at most n and at least the ceiling of $\log_2(n + 1)$
- The number of nodes n in a full binary tree is at least $n = 2^h - 1$.
- The number of leaf nodes l in a full binary tree is $l = (n + 1)/2$

Properties

- The number of internal nodes in a complete binary tree of n nodes is equal to the floor of $n/2$
- In a complete binary tree, the number of nodes at depth d is 2^d
- If a tree has λ levels, then the number of leaves is at most $2^{\lambda-1}$

Binary search tree

Definition 2 (Definition)**.** A *binary search tree* is a binary tree where each node in the tree satisifes the condition that the data element in the node is larger than all data elements in left subtree and smaller than all of the data elements in its right subtree.

Binary Tree Interface: Inorder

Algorithm 1: Tree: Inorder Tree Walk

Input: A search tree node x
Output: The data values of the tree pointed to by x

```
1 if x is not nil then
2     Inorder-Walk (Left (x));
3     DoSomething (x);
4     Inorder-Walk (Right (x));
```

Binary Tree Interface: Other tree walks

- Preorder tree walk
- Postorder tree walk

Algorithm 2: Tree: Preorder Tree Walk

Input: A search tree node x
Output: The data values of the tree pointed to by x

```
1 if x is not nil then
2     DoSomething (x);
3     PreOrder-Walk (Left (x));
4     PreOrder-Walk (Right (x));
```

Algorithm 3: Tree: PostOrder Tree Walk

Input: A search tree node x
Output: The data values of the tree pointed to by x

```
1 if x is not nil then
2     PostOrder-Walk (Left (x));
3     PostOrder-Walk (Right (x));
4     DoSomething (x);
```

Binary Tree Interface: Binary Search

Algorithm 4: Binary Search

Input: A search tree node x and a search key k
Output: The data values of the tree pointed to by x

```
1 if x is nil or k is equal to data value in x then
2     return x;
3 if k < x.data then
4     return BinSearch (Left (x),k);
5 else
6     return BinSearch (Right (x),k);
```

Binary Tree Interface: Insertion

Algorithm 5: BST Insertion

Input: A tree root x and insertion value v
Output: A revised tree rooted at x with v inserted

```
1  if v < x.data then
2      if Left (x) = nil then
3          Create new node and set Left (x) equal to it;
4      else
5          Insert (Left (x),v));
6  else
7      if Right (X) = nil then
8          Create new node and set Right (x) equal to it;
9      else
10         Insert (Right (x),v);
```

2 Implementation of binary trees

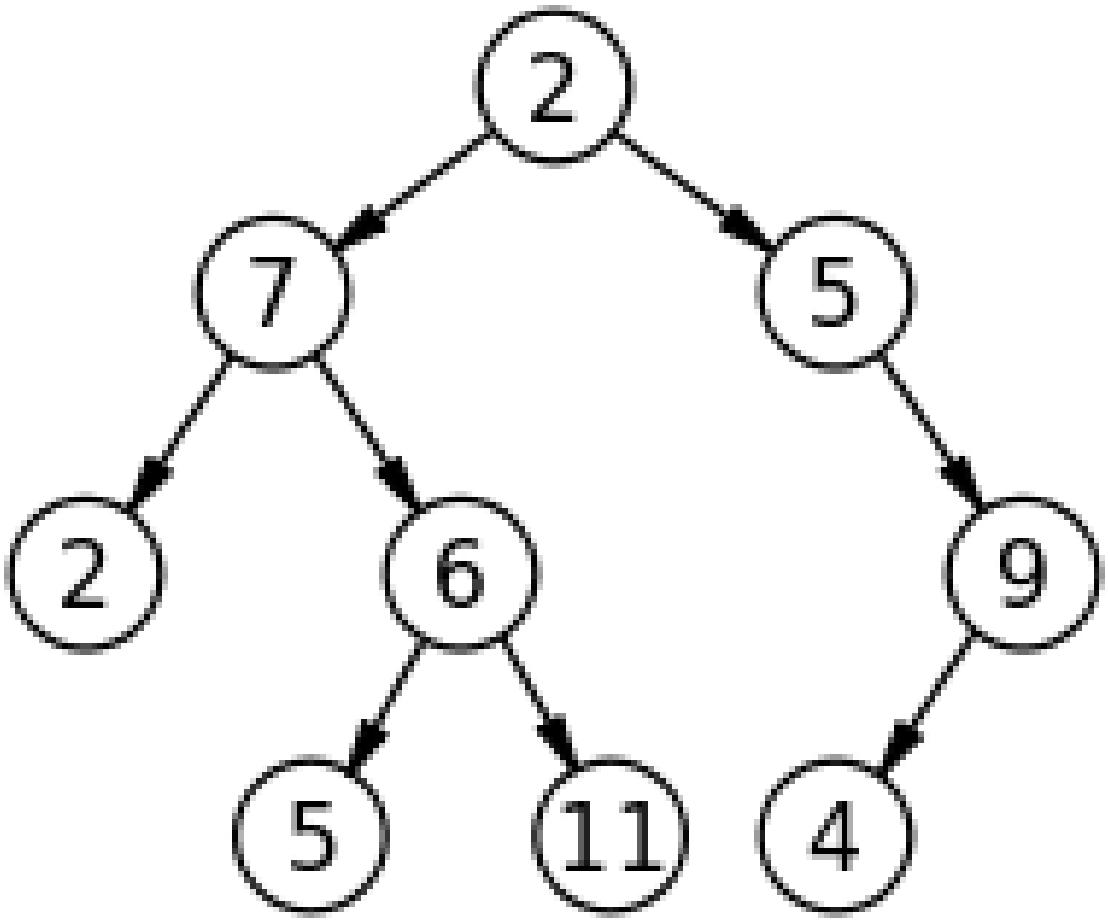
Implementation binary trees: Pointers

```
struct bintree {
   Node n;
   struct bintree *left;
   struct bintree *right;
}
```

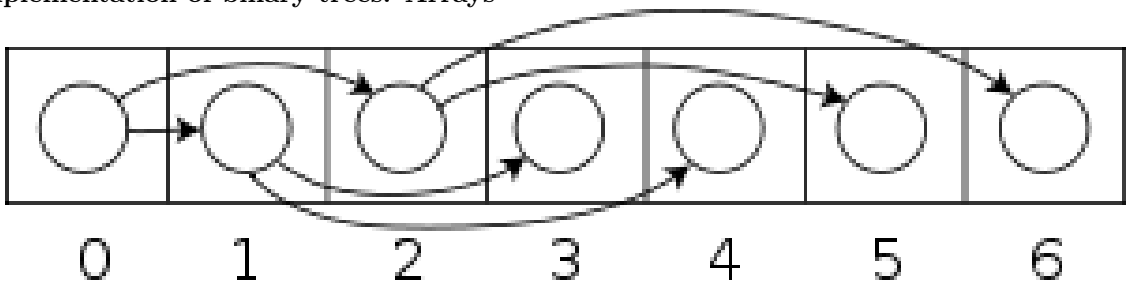
- Recursive definition, lots of pointer madness
- This becomes an implied definition for languages such as Lisp or Ruby

```
s = {#a {#b {}{}} {#c {#d {}}{} #e {}{}}}
```

Implementation of binary trees: Arrays



Implementation of binary trees: Arrays



3 Beyond binary: n-way trees

n Way Trees

- Suppose rather than just 2 children, we allowed an arbitrary number of children per node
- BUT with same number at each node...
- Left-child, right sibling representation
  - Make each node only have two children
  - *Left-child(x)*: points to the leftmost child of x
  - *Right-sibling(x)*: points to the sibling of x immediately to the right
  - If node x has no children then *Left-child(x)* will return nil
  - If node x is the rightmost child of its parent, then *Right-sibling(x)* will return nil.

4 Beyond binary: red-black trees

Red-black trees: definition

- Extend the binary search tree by keeping one additional piece of information: the node *color*
- Color can be either *Red* or *Black*
- A red-black tree must have the following properties
  - Every node is either red or black
  - Every leaf is black
  - If a node is red, then both its children are black
  - Every simple path from a node to a descendant leaf contains the same number of black nodes

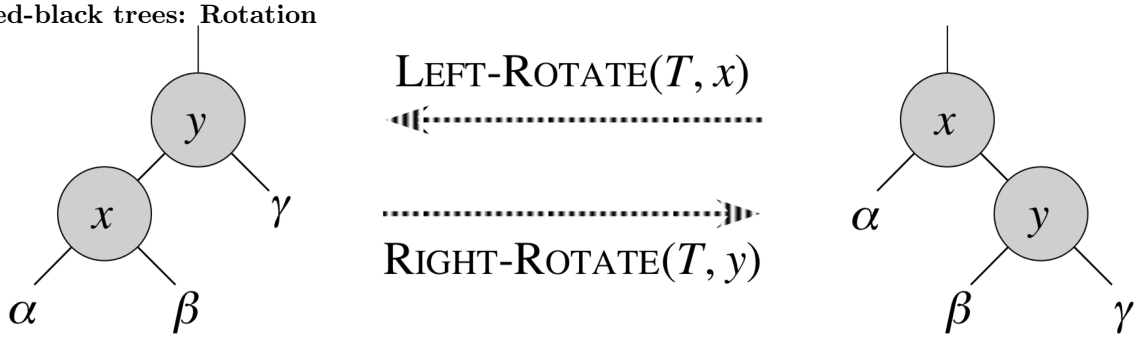
Red-black trees: height

- Height of a node is the number of edges in a longest path to a leaf
- The *black-height* of a node is the number of black nodes on the path from the node to leaf, not counting the node
- Important claims
  - Any node with height h has black-height greater than or equal $h/2$.
  - The subtree rooted at any node x contains $2^{bh(x)} - 1$ internal nodes
  - A red-black tree with n internal nodes has height less than or equal $2 * lg(n + 1)$

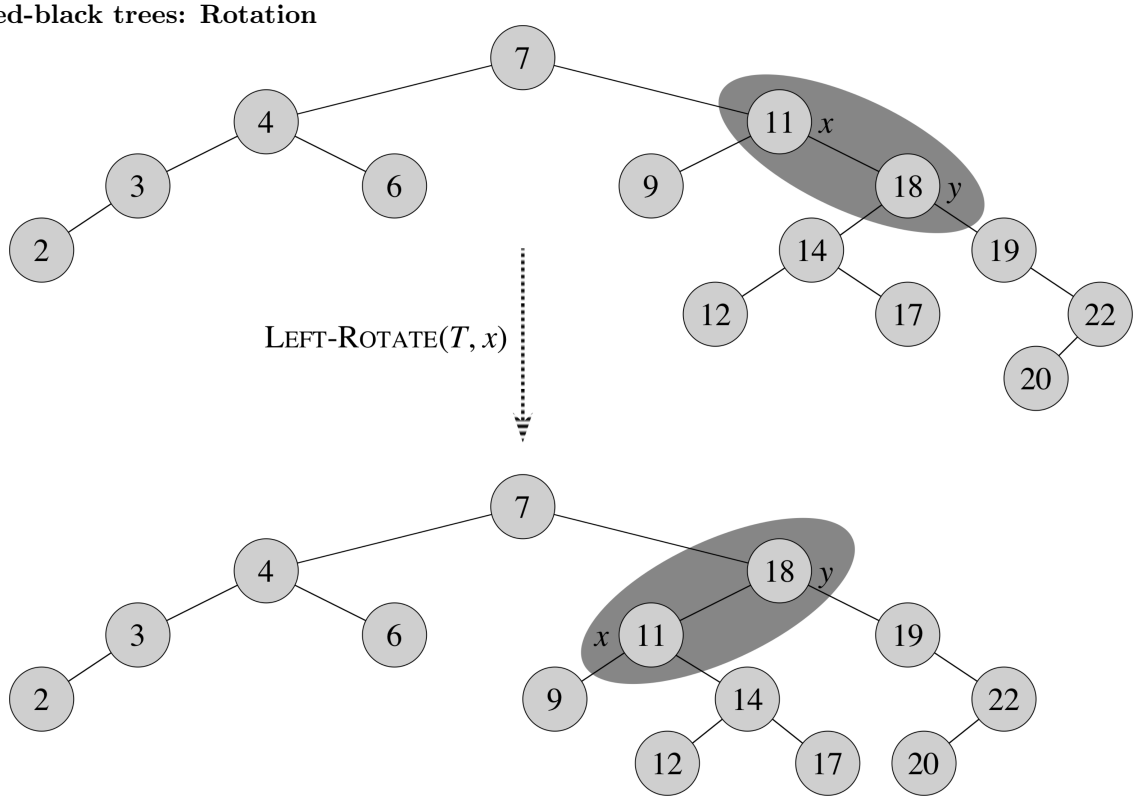
Red-black trees: insertion and deletion

- Non-modifying operations work like comparable BST operations
- Insertion and deletion are most definitely a horse of a different color

Red-black trees: Rotation



Red-black trees: Rotation

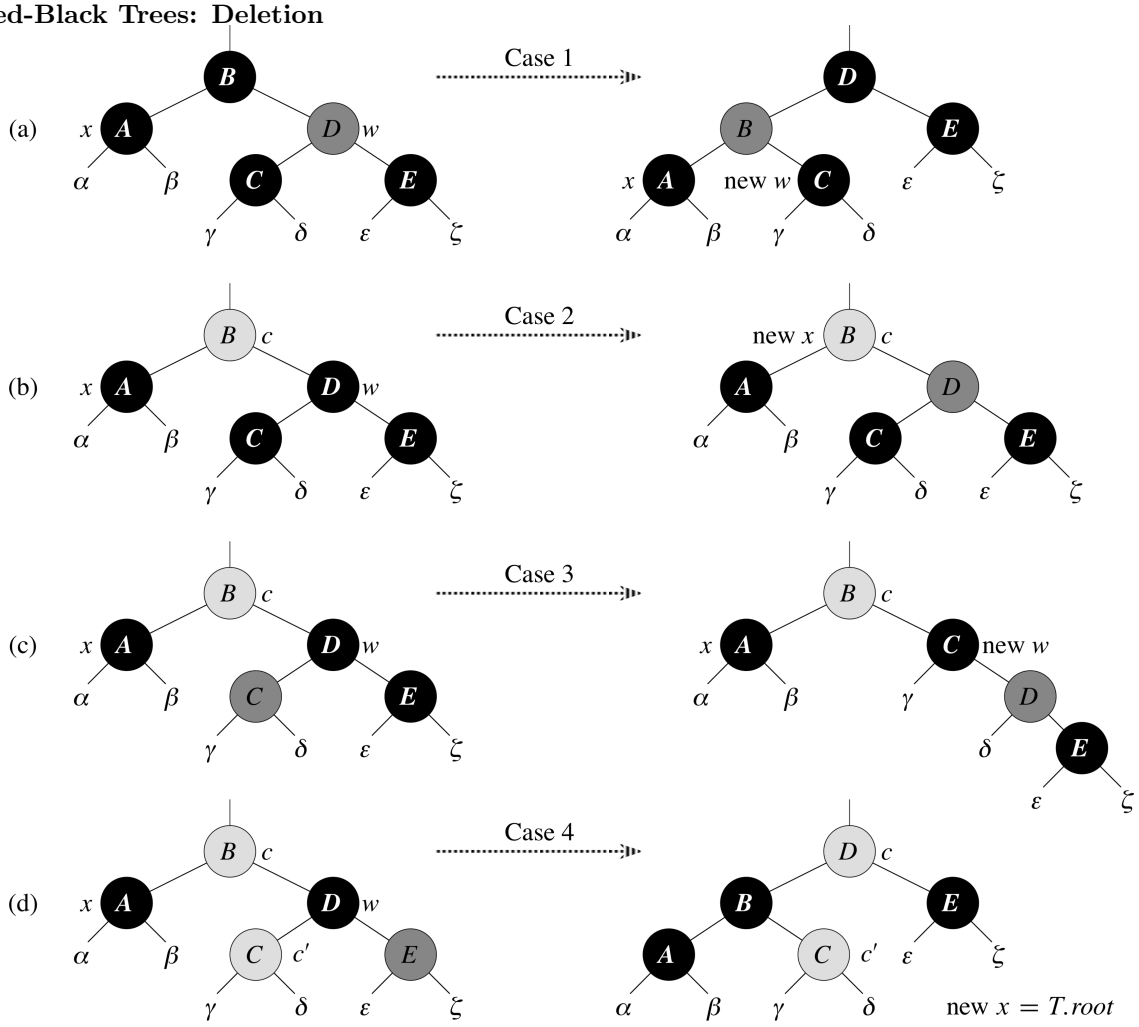


Red-Black trees: Insertion

Algorithm 6: Red-Black Insert

```
   Input: A tree T and node x to be inserted
   Output: A rearranged R-B tree T
 1 Insert(T,x) color[x] ← RED;
 2 while x is not root and Parent (x).color is RED do
 3     if Parent (x) = Left (Parent (Parent (x))) then
 4         y ← Right (Parent (Parent (x)));
 5         if y.color is READ then
 6             Parent (x).color ← BLACK;
 7             y.color ← BLACK;
 8             Parent (Parent (x)).color ← RED;
 9             x ← Parent (Parent (x));
10         else
11             if x = Right (Parent (x)) then
12                 x ← Parent (x);
13                 LeftRotate (T,x)
14             Parent (x).color ← BLACK;
15             Parent (Parent (x)) ← RED;
16             RightRotate (T,Parent (Parent (x)));
17     else
18         Same as then clause with "right" and "left" exchanged;
19 Root (T).color ← BLACK;
```

Red-Black Trees: Deletion



Red-Black Trees: Why?

Self-Balancing Tree a tree that keeps it height small in the face of many inserts and deletes

- Examples: R-B trees, AVL trees, B-trees of order 4
- Often used in computational geometry algorithms and operating system scheduling algorithms (example: the Linux CFS scheduler)

Red-Black Trees: Why?

op	Average	Worst
Space	$O(n)$	$O(n)$
Search	$O(lg(n))$	$O(lg(n))$
Insert	$O(lg(n))$	$O(lg(n))$
Delete	$O(lg(n))$	$O(lg(n))$

5 Key Points

Key Points

- Understand the care and feeding of the tree data structure
- Learn the use of some of the variants
  - N-way trees
  - R-B trees
  - Many others (AVL and B-trees, for example)