ICT-2101
Data Structure

Lecture 12

B-Tree

Md. Mahmudur Rahman
Assistant Professor, IIT

B-Tree

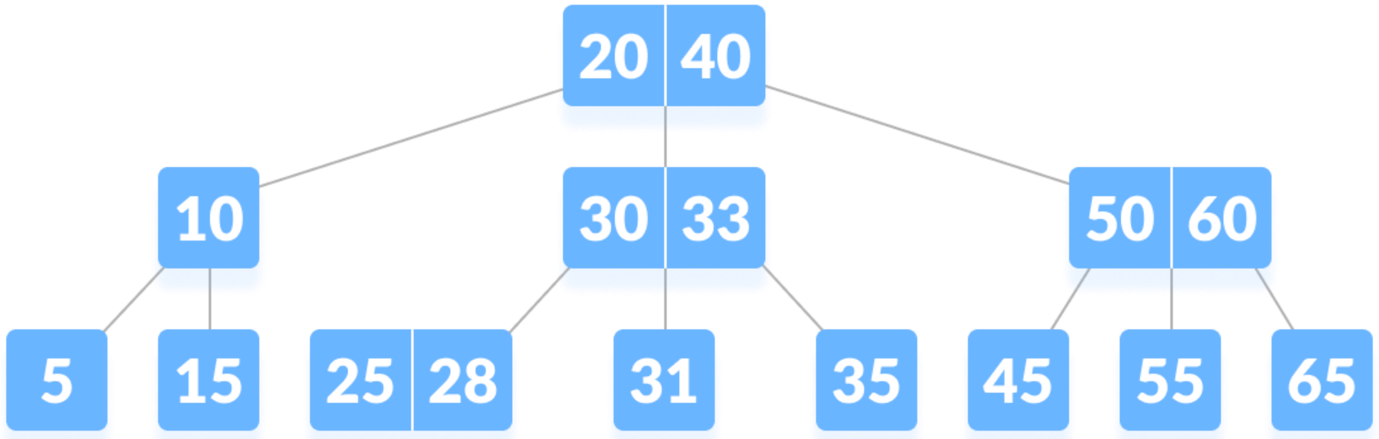
- **B-tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.

- B-tree is a special type of self-balancing search tree in which each node can contain **more than one key** and can have more than **two children**. It is a generalized form of the binary search tree. It is also known as a height-balanced m-way tree.

- Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write large blocks of data. B-trees are a good example of a data structure for external memory. It is commonly used in databases and file systems.

The B-Tree Rules

- Important properties of a B-tree:
  - B-tree nodes have many more than two children.
  - A B-tree node may contain more than just a single element.



Need of B-tree

- The need for B-tree arose with the rise in the need for lesser time in accessing physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity. There was a need for such types of data structures that minimize the disk access.

- Other data structures such as a binary search tree, avl tree, red-black tree, etc can store only one key in one node. If you have to store a large number of keys, then the height of such trees becomes very large, and the access time increases.

- However, B-tree can store many keys in a single node and can have multiple child nodes. This decreases the height significantly allowing faster disk accesses.

Applications of B-Trees

1. It is used in large databases to access data stored on the disk
2. Searching for data in a data set can be achieved in significantly less time using the B-Tree
3. With the indexing feature, multilevel indexing can be achieved.
4. Most of the servers also use the B-tree approach.

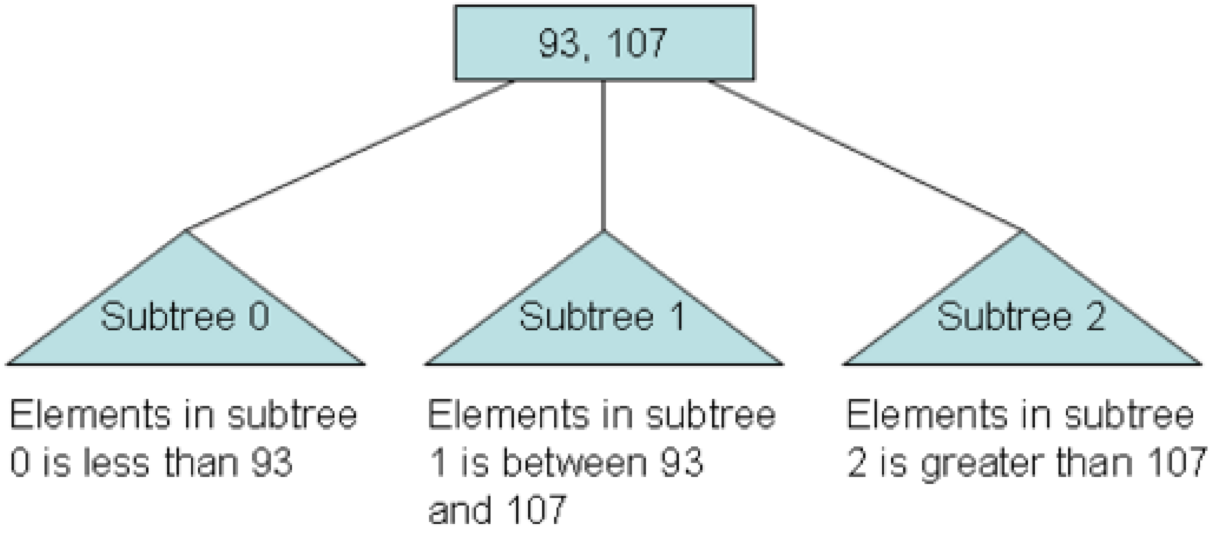
The B-Tree Rules

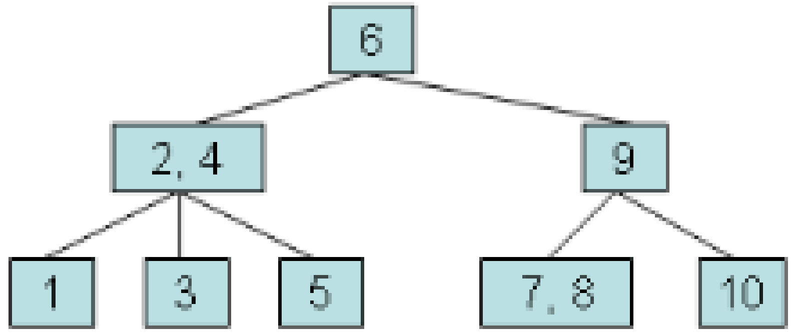
- The set formulation of the B-tree rules: Every B-tree depends on a positive constant integer called MINIMUM, which is used to determine how many elements are held in a single node.
- Rule 1: The root can have as few as one element (or even no elements if it also has no children); every other node has at least MINIMUM elements.
- **Rule 2:** The **maximum** number of elements in a node is **twice** the value of MINIMUM.
- Rule 3: The elements of each B-tree node are stored in a partially filled array, sorted from the smallest element (at index 0) to the largest element (at the final used position of the array).

The B-Tree Rules

- **Rule 4:** The number of subtrees below a non-leaf node is always one more than the number of elements in the node.
  - Subtree 0, subtree 1, …
- **Rule 5:** For any non-leaf node:
  - An element at index i is greater than all the elements in subtree number i of the node, and
  - An element at index i is less than all the elements in subtree number $i + 1$ of the node.
- **Rule 6:** Every leaf in a B-tree has the same depth. Thus it ensures that a B-tree avoids the problem of a unbalanced tree.

The B-Tree Rules


93, 107
Subtree 0
Subtree 1
Subtree 2
Elements in subtree 0 is less than 93
Elements in subtree 1 is between 93 and 107
Elements in subtree 2 is greater than 107



MINIMUM = 1

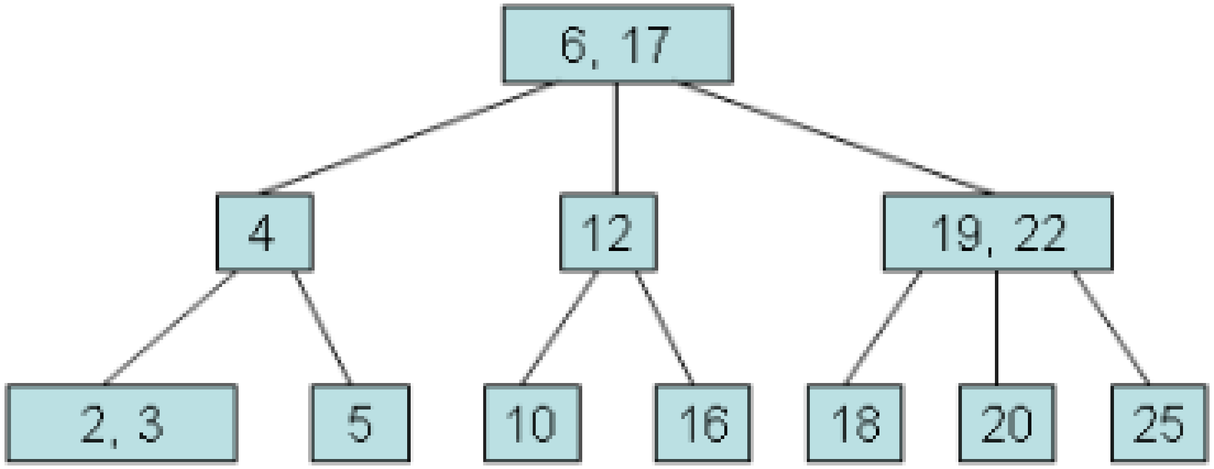
Searching for a Target in a Set

Algorithm:

- Start at the root node and check if the key is in the current node.
- If the key is found, return the node and the index of the key.
- If the key is not found and the node is a leaf, return None (the key does not exist in the tree).
- If the key is not found and the node is not a leaf, recurse into the appropriate child node:
- Compare the key with the keys in the current node to determine the correct child to search.

Searching for a Target in a Set

- See the following example, try to search for 10.

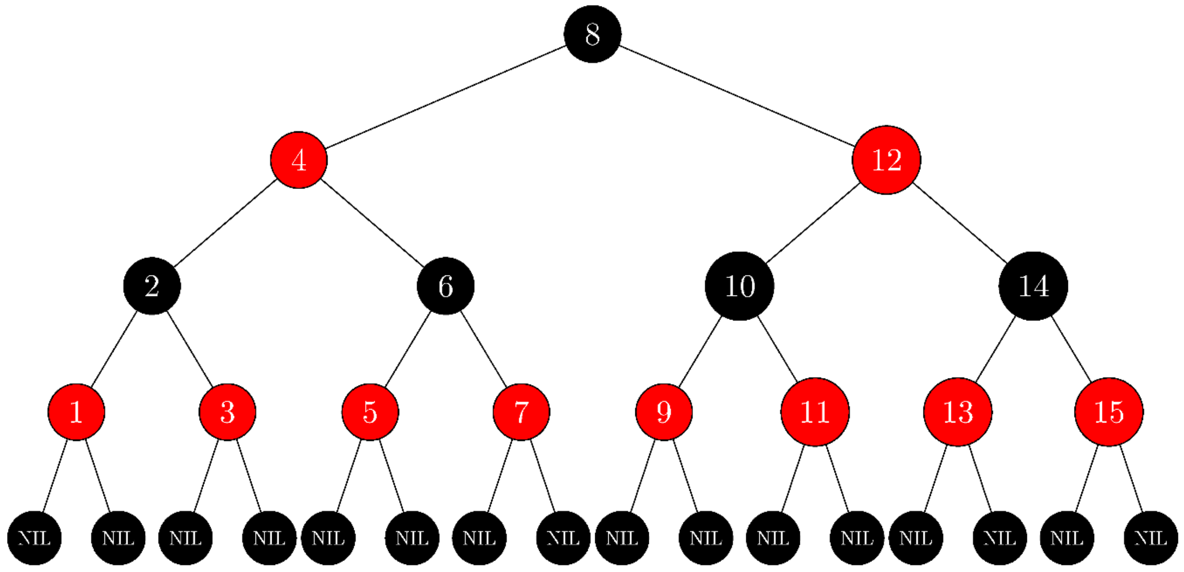


Searching for a Target in B-Tree

```
struct Node {
    int n;
    int key[MAX_KEYS];
    Node* child[MAX_CHILDREN];
    bool leaf;
};

Node* BtreeSearch(Node* x, int k) {
    int i = 0;
    while (i < x->n && k > x->key[i]) {
        i++;
    }
    if (i < x->n && k == x->key[i]) {
        return x;
    }
    if (x->leaf) {
        return nullptr;
    }
    return BtreeSearch(x->child[i], k);
}
```

RED BLACK TREE



Red-Black Trees

- A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK.

- By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

Red-Black Trees

- Each node of the tree now contains the attributes *color, key, left, right, and p.*

- *If* a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL.

- We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

Red-Black Trees

- A red-black tree is a binary tree that satisfies the following ***red-black properties***:

  1. Every node is either red or black.

  2. The root is **black**.

  3. Every leaf (NIL) is black.

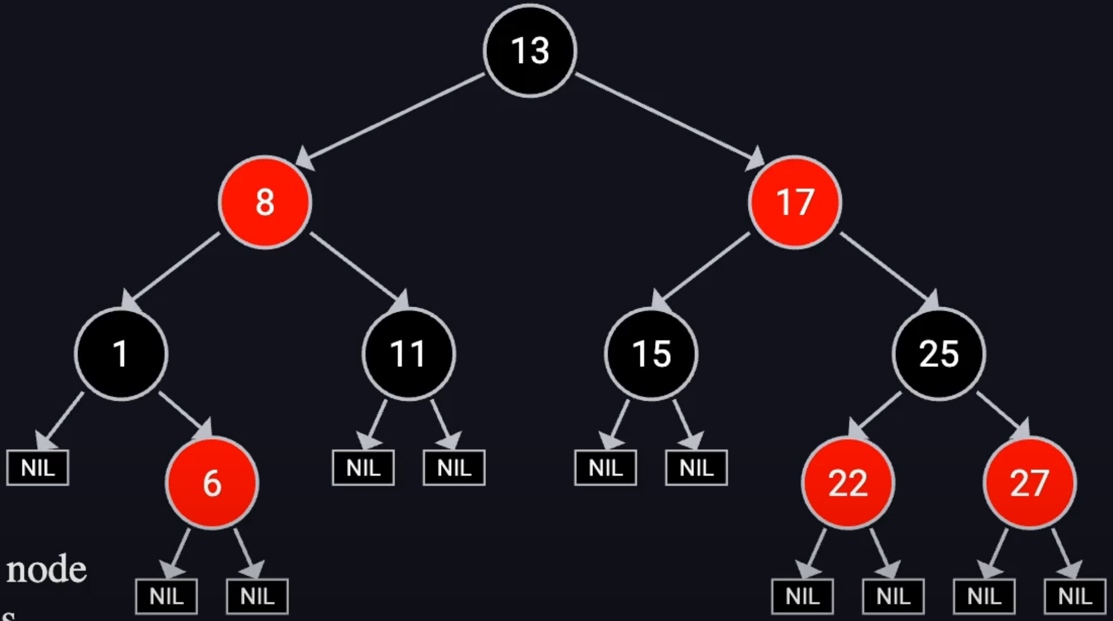
  4. If a node is **red**, then both its children are **black**.

  5. For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.

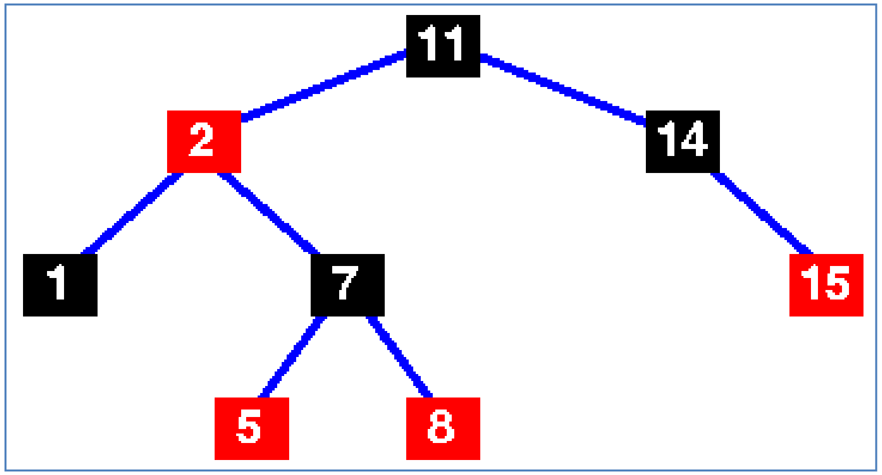
Derived property: The longest path from the root to a leaf is at most twice as long as the shortest path.

5 Rules of Red-Black Tree

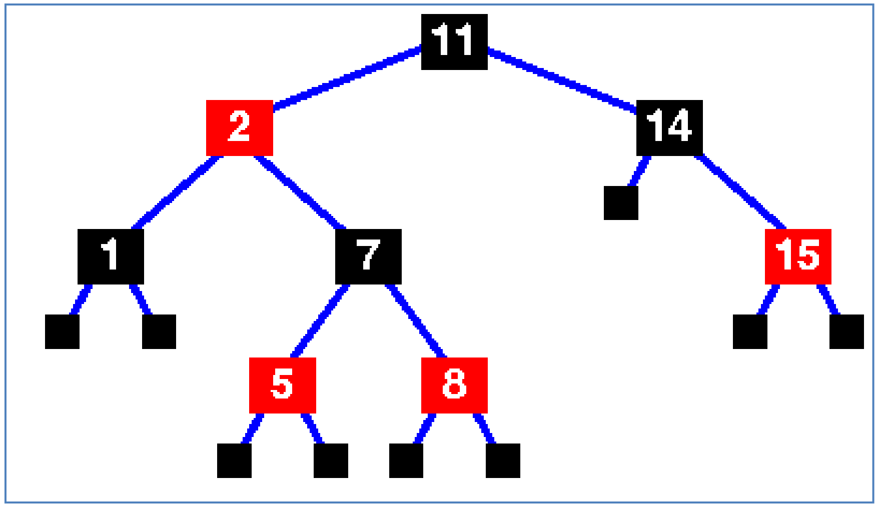
1. Every node is either **red** or black
2. Root node is black
3. All Leaf(NIL) nodes are black
4. If a node is **red**, then both its children are black
5. For each node, every path to a NIL node has the same number of black nodes.



Red-Black Trees



A basic red-black tree

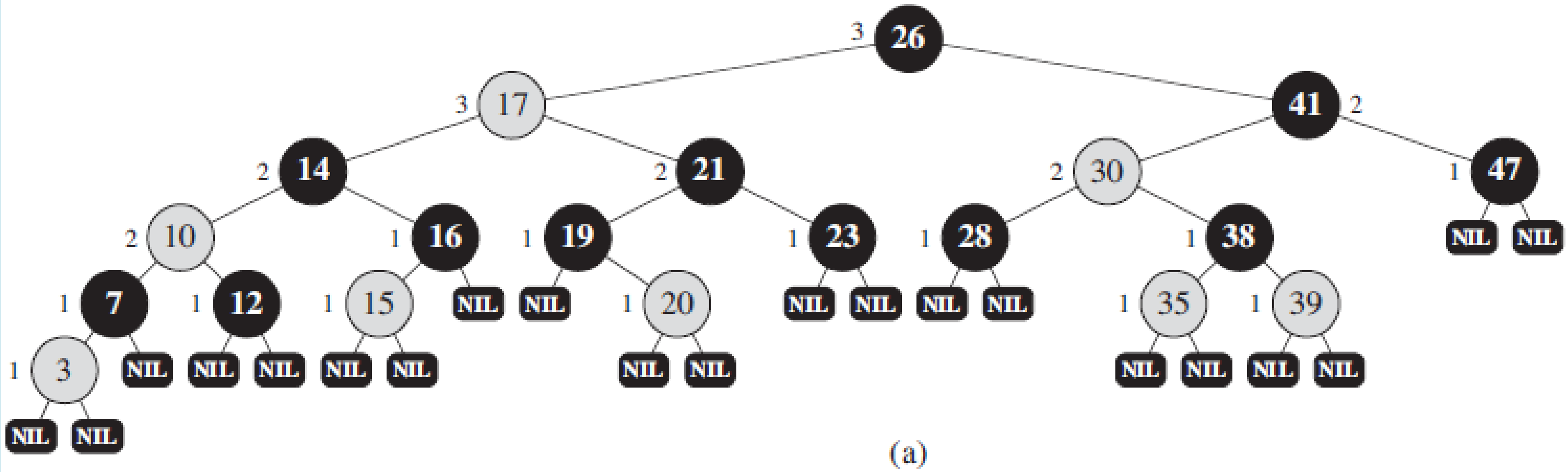


Basic red-black tree with the **sentinel** nodes added. Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node. They are the NULL black nodes of property 3.

The number of black nodes on any path from, but not including, a node ***x*** to a leaf is called the *black-height* of a node, denoted $\mathbf{b_h(x)}$.

Red-Black Trees

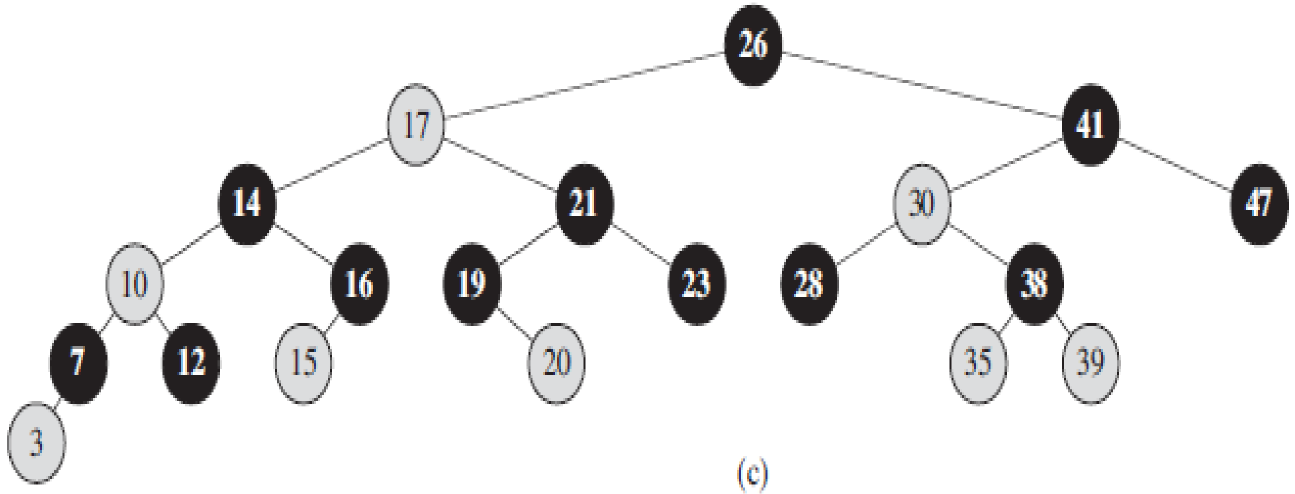
- An example of a red-black tree. Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0.



(a)

Red-Black Trees

- The same red-black tree but with leaves and the root's parent omitted entirely.

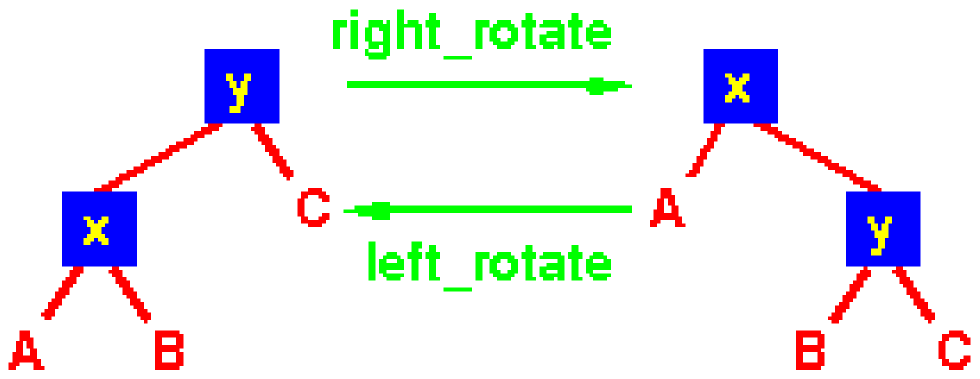


(c)

Rotations on red-black trees

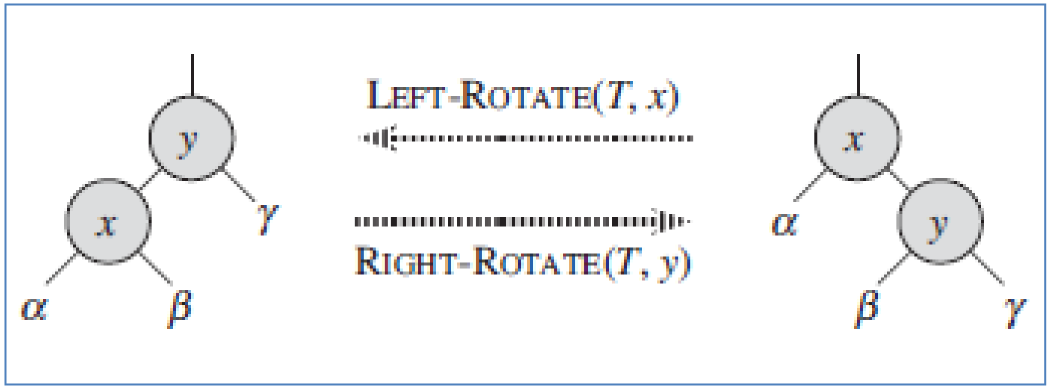
- As with heaps, additions and deletions from red-black trees destroy the red-black property, so we need to restore it. To do this we need to look at some operations on red-black trees.
- **A rotation** is a local operation in a search tree that preserves *in-order* traversal key ordering. Note that in both trees, an in-order traversal yields:

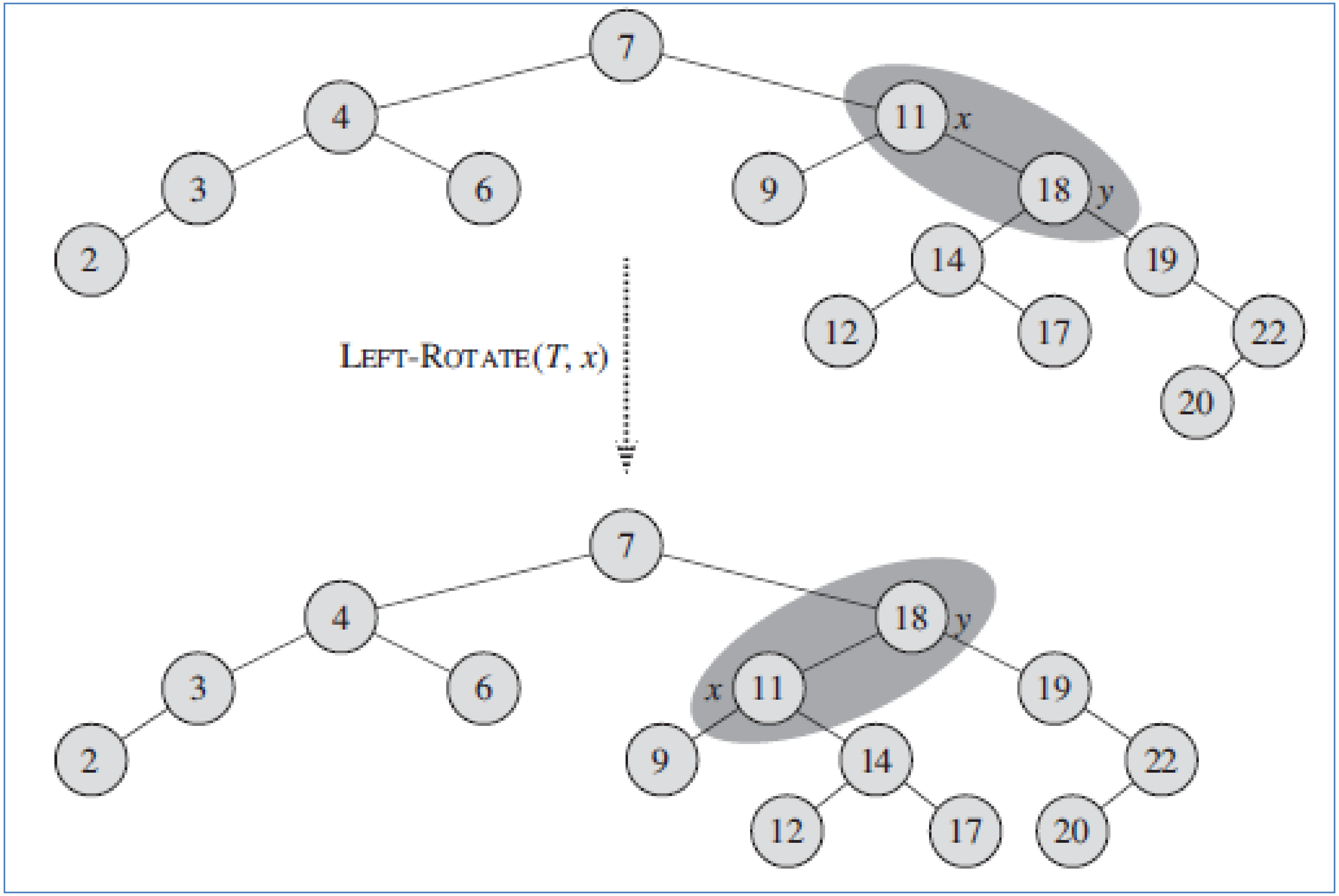
A x B y C



Rotations on red-black trees

```
LEFT-ROTATE(T, x)
    $y = x.right$
    $x.right = y.left$
    if $y.left \neq T.nil$
        $y.left.p = x$
    $y.p = x.p$
    if $x.p == T.nil$
        $T.root = y$
    elseif $x == x.p.left$
        $x.p.left = y$
    else $x.p.right = y$
    $y.left = x$
    $x.p = y$
```





An example of how the procedure LEFT-ROTATE (T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

Operations on red-black trees

- We can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in O(log n) time on red-black trees, since each can run in O(**h**) time on a binary search tree of height **h** and any red-black tree on n nodes is a binary search tree with height O(log n).

Thank you