File Edit Format View

COMP8547 - Advanced Computing Concepts

B *I* U

Lab - 3
Search Trees

LAB EXERCISES

SEARCH TREES

- Use classes BinarySearchTree, AVLTree, RedBlackBST, and SplayTree to create four search trees: BST, AVL, red-black, and splay trees.
- For each tree:
  - Insert 100,000 integer keys, from 1 to 100,000 (in that order). Find the average time of each insertion
  - Do 100,000 searches of random integer keys between 1 and 100,000. Find the average time of each search. (2b)
  - Delete all the keys in the trees, starting from 100,000 down to 1 (in that order). Find the average time of each deletion. (2c)

LAB EXERCISES

SEARCH TREES

- For each tree:
  - Insert 100,000 random keys between 1 and 100,000. Find the average time of each search
  - Repeat #2.b.
  - Repeat #2.c but with random keys between 1 and 100,000. Note that not all the keys may be found in the tree.
- Draw a table that contains all the average times found in #2 and #3
- Comment on the results obtained and compare them with the worst-case and average-case running times of each operation for each tree.
- Which search tree will you use in your application? Why?

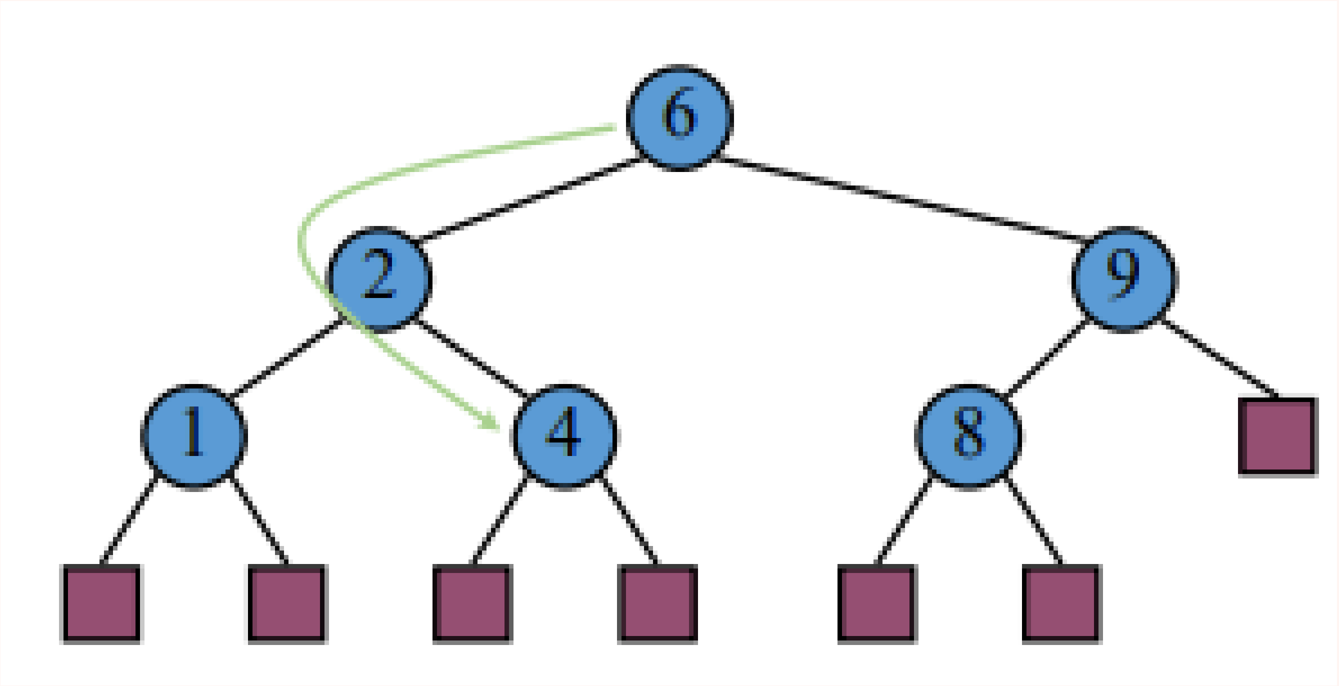
A quick recap

BST

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

BST Search

```
Algorithm TreeSearch(k, v)
    if T.isExternal (v) {base case}
        return v
    if k < key(v) {general case}
        return TreeSearch(k, T.left(v))
    else if k > key(v)
        return TreeSearch(k, T.right(v))
    else {k = key(v) -- base case}
        return v
```



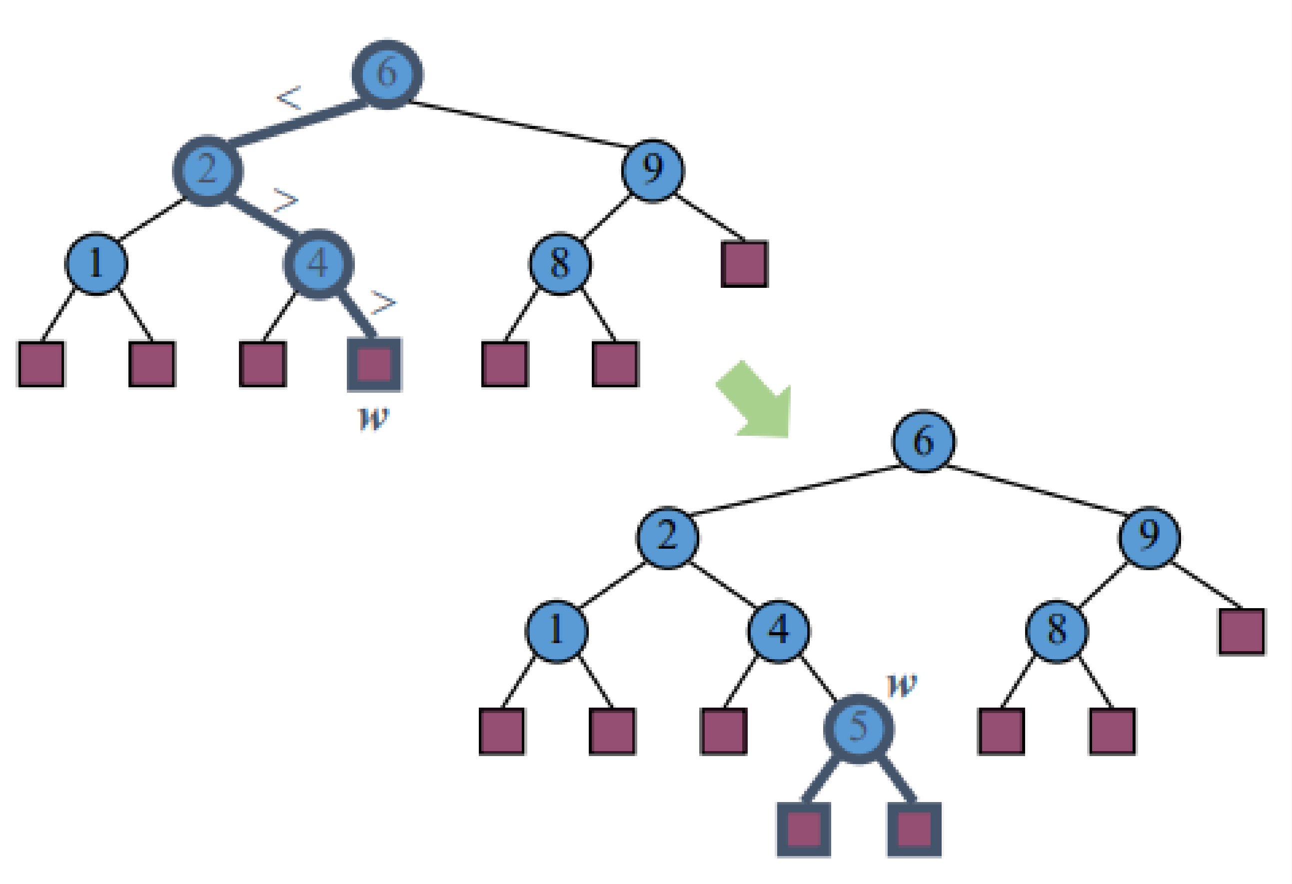
BST

- Insertion Algorithm

BST Search

```
Algorithm TreeInsert(k,x,v)
Input: A key k, a value x, and a node v of T
Output: A new node w of T

w <-- TreeSearch(k,v)
if k = key(w) then
    return TreeInsert(k,x,T.left(w))
T.insertAtExternal(k,x,w)
return w
```



BST

- **Node to be deleted is leaf**: Simply remove it from the tree.
- **Node to be deleted has only one child**: Copy the child to the node and delete the child
- **Node to be deleted has two children**: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor.

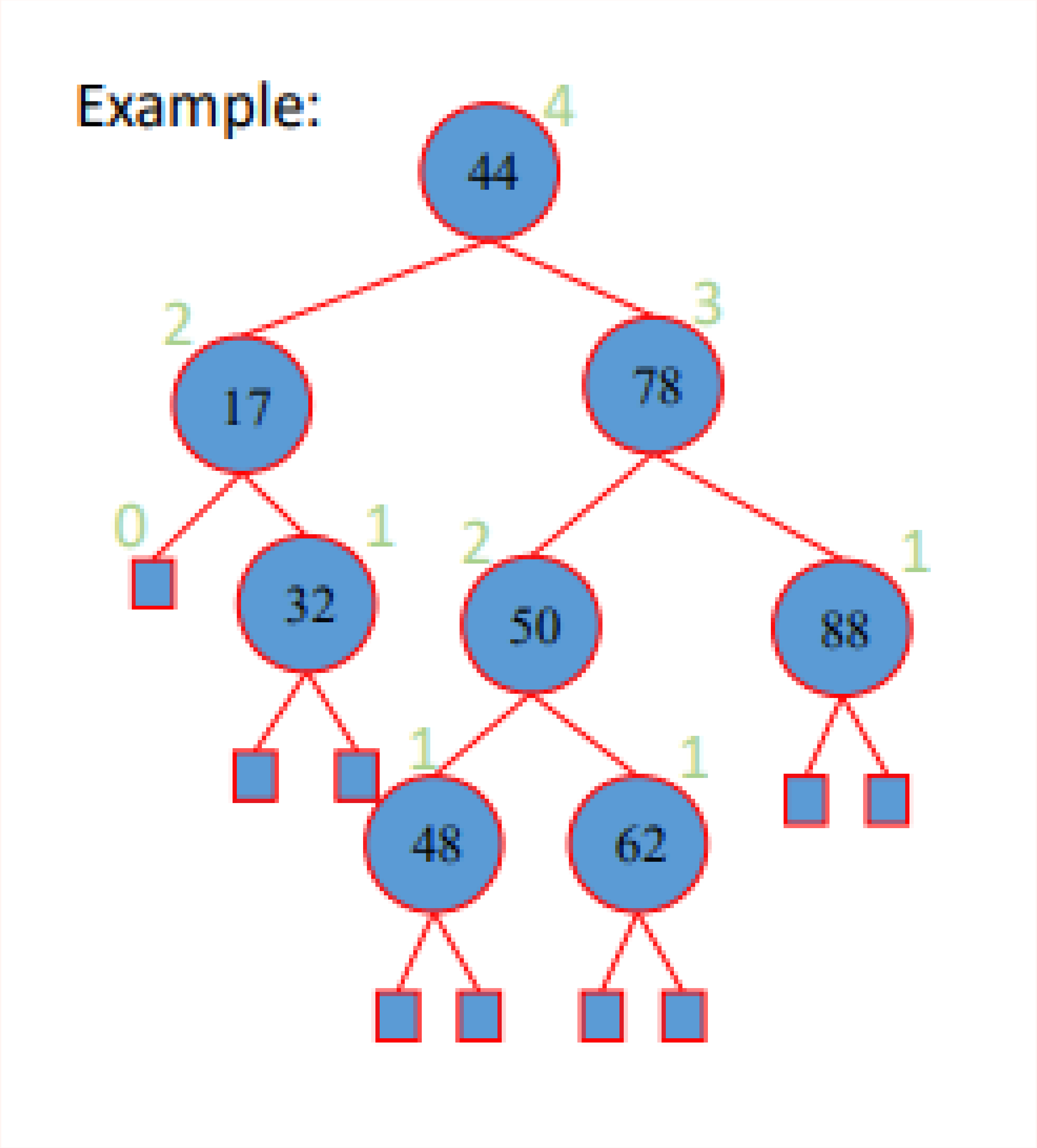
```
      50                              50
    /     \         delete(30)       /   \
   30      70       --------->     40      70
     \    /  \                            /  \
     40  60   80                         60   80

      50                              60
    /     \         delete(50)       /   \
   40      70       --------->     40      70
          /  \                               \
         60   80                              80

      50                              50
    /     \         delete(20)       /   \
   30      70       --------->     30      70
  /  \    /  \                       \    /  \
20   40  60   80                     40  60   80
```

AVL Trees

- AVL trees are balanced
- Adel’son-Vel’ski and Landis
- The height of an AVL tree storing n keys is O(log n)
- Search in an AVL tree works as it does in a BST



AVL Trees

- AVL trees are balanced

AVL

```java
private static final int ALLOWED_IMBALANCE = 1;

// Assume t is either balanced or within one of being balanced
private AvlNode<AnyType> balance( AvlNode<AnyType> t )
{
    if( t == null )
        return t;

    if( height( t.left ) - height( t.right ) > ALLOWED_IMBALANCE )
        if( height( t.left.left ) >= height( t.left.right ) )
            t = rotateWithLeftChild( t );
        else
            t = doubleWithLeftChild( t );
    else
    if( height( t.right ) - height( t.left ) > ALLOWED_IMBALANCE )
        if( height( t.right.right ) >= height( t.right.left ) )
            t =rotateWithRightChild( t );
        else
            t = doubleWithRightChild( t );

    t.height = Math.max( height( t.left ), height( t.right ) ) + 1;
    return t;
}
```

AVL Trees

- AVL trees removal

AVL

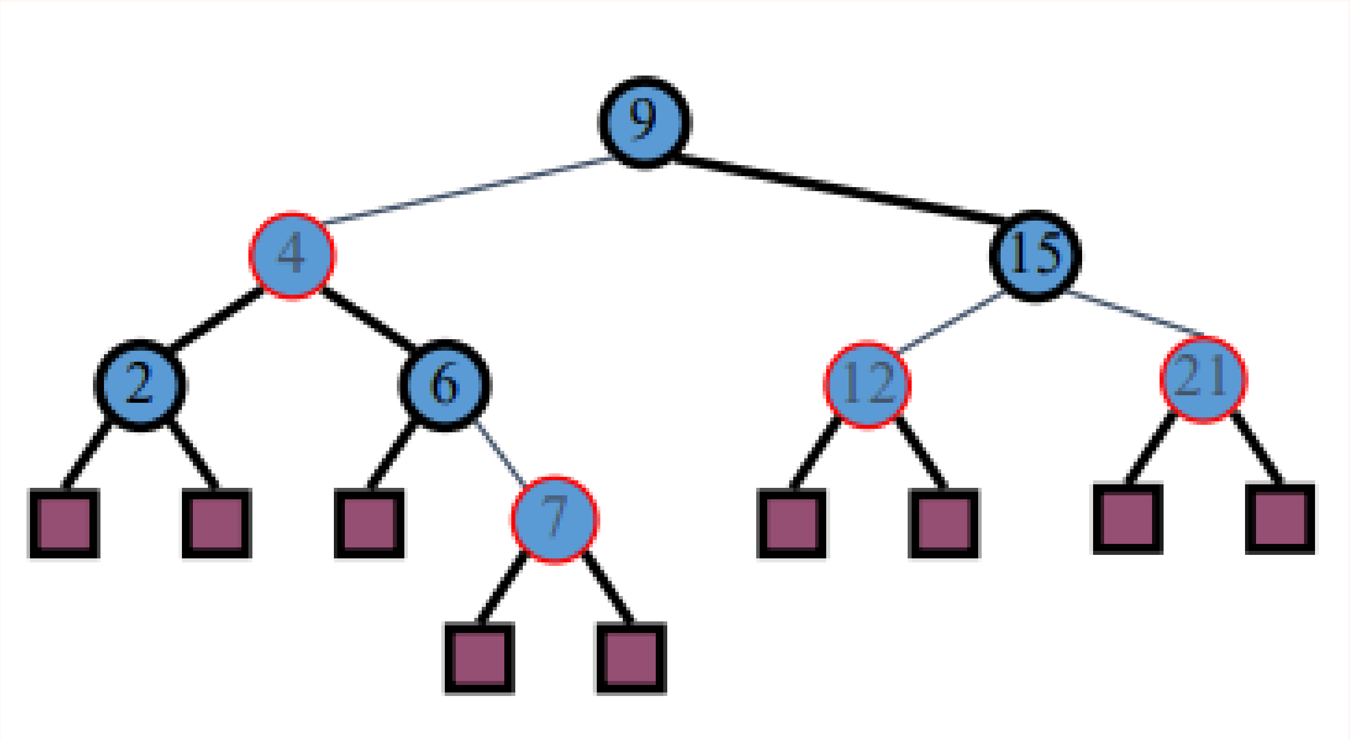
```
private AvlNode<AnyType> remove( AnyType x, AvlNode<AnyType> t )
{
    if( t == null )
        return t;   // Item not found; do nothing

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        t.left = remove( x, t.left );
    else if( compareResult > 0 )
        t.right = remove( x, t.right );
    else if( t.left != null && t.right != null ) // Two children
    {
        t.element = findMin( t.right ).element;
        t.right = remove( t.element, t.right );
    }
    else
        t = ( t.left != null ) ? t.left : t.right;
    return balance( t );
}
```

Red-Black Trees

- A red-black tree is a BST that satisfies the following properties:
  - **Root Property**: the root is black
  - **External Property**: every leaf is black
  - **Internal Property**: the children of a red node are black.
  - **Depth Property**: all the leaves have the same black depth



- RBT Insertion

RBT - Insertion

```java
private Node put(Node h, Key key, Value val) {
        if (h == null) return new Node(key, val, RED, 1);

        int cmp = key.compareTo(h.key);
        if      (cmp < 0) h.left  = put(h.left,  key, val);
        else if (cmp > 0) h.right = put(h.right, key, val);
        else              h.val   = val;

        // fix-up any right-leaning links
        if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
        if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
        if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);
        h.N = size(h.left) + size(h.right) + 1;

        return h;
    }
```

RBT Deletion

- **Case 1: node is black and has a red child,**
  - We perform a restructuring, and we are done
- **Case 2: node is black and its children are both black**
  - We perform a recoloring, which may propagate up the double black violation
- **Case 3: node is red (and hence it has a black child)**
  - We perform an adjustment, after which either Case 1 or Case 2 applies

RBT - Deletion

```java
private Node delete(Node h, Key key) {
    // assert contains(h, key);

    if (key.compareTo(h.key) < 0)  {
        if (!isRed(h.left) && !isRed(h.left.left))
            h = moveRedLeft(h);
        h.left = delete(h.left, key);
    }
    else {
        if (isRed(h.left))
            h = rotateRight(h);
        if (key.compareTo(h.key) == 0 && (h.right == null))
            return null;
        if (!isRed(h.right) && !isRed(h.right.left))
            h = moveRedRight(h);
        if (key.compareTo(h.key) == 0) {
            Node x = min(h.right);
            h.key = x.key;
            h.val = x.val;
         }
        else h.right = delete(h.right, key);
    }
    return balance(h);
}
```

File Edit Format View

Thank You