



# Searching

Frank Walsh

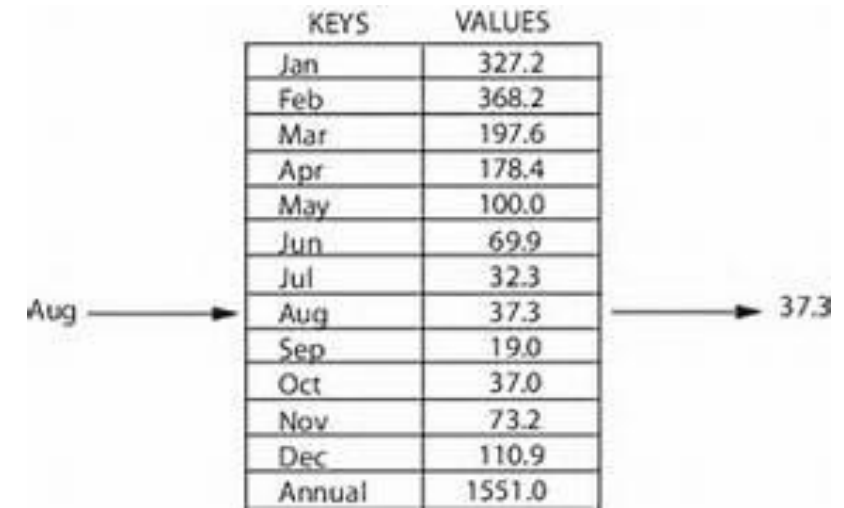
Reference: <http://algs4.cs.princeton.edu/33balanced/>

# Agenda

- Symbol Tables
- Linked Lists
- Binary Tree
- Balanced Binary Tree
- Hash Tables

# Symbol Tables

- A symbol table is a data structure of key – value pairs
- Supports two operations:
  - Get
  - Put
- Get: gets a value associated with a given key.
- Put: puts a new key/value pair into the table.
- E.g. Dictionaries you've been using previously.



The diagram illustrates a symbol table used for storing monthly temperature data. It consists of a table with two columns: 'KEYS' and 'VALUES'. The 'KEYS' column lists the months from Jan to Dec, followed by an 'Annual' entry. The 'VALUES' column contains the corresponding temperature values. An arrow labeled 'Aug' points to the 'Aug' row in the 'KEYS' column, and another arrow points from the '37.3' value in the 'VALUES' column to the right, demonstrating the 'Get' operation.

KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

# Symbol Table Applications

application	purpose of search	key	value
<i>dictionary</i>	find definition	word	definition
<i>book index</i>	find relevant pages	term	list of page numbers
<i>file share</i>	find song to download	name of song	computer ID
<i>account management</i>	process transactions	account number	transaction details
<i>web search</i>	find relevant web pages	keyword	list of page names
<i>compiler</i>	find type and value	variable name	type and value

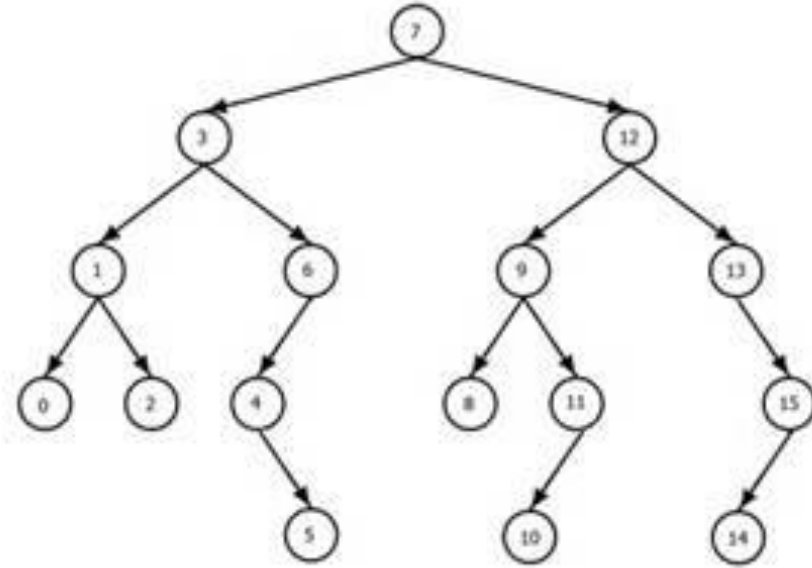
**Typical symbol-table applications**

# Symbol Tables API

<code>void put(Key key, Value val)</code>	<i>put key-value pair into the table (remove key from table if value is null)</i>
<code>Value get(Key key)</code>	<i>value paired with key (null if key is absent)</i>
<code>void delete(Key key)</code>	<i>remove key (and its value) from table</i>
<code>boolean contains(Key key)</code>	<i>is there a value paired with key?</i>
<code>boolean isEmpty()</code>	<i>is the table empty?</i>
<code>int size()</code>	<i>number of key-value pairs in the table</i>
<code>Iterable&lt;Key&gt; keys()</code>	<i>all the keys in the table</i>

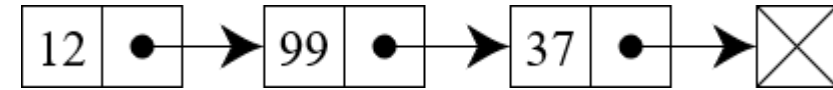
**API for a generic basic symbol table**

# Binary Trees



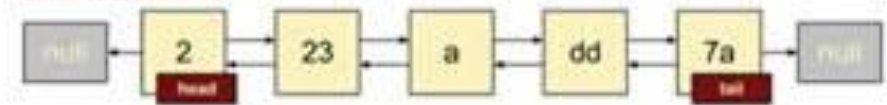
# Aside - Linked Lists

- Linear collection of data elements, called nodes.
- Each node is composed of data and reference to the next node in the sequence.
- Allows for efficient insertion or removal of elements.
- Implementations in Java SE:
  - [LinkedList](#)
- Implements List interface from Collections API



## Array vs. Linked List

### Linked List

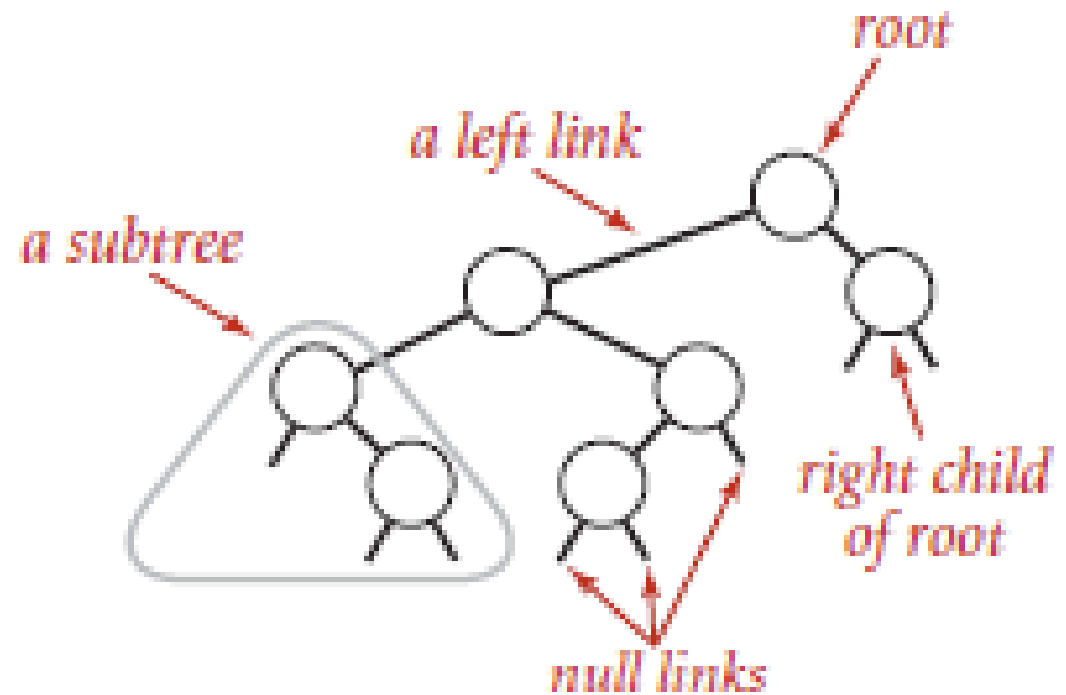


### Array



# Binary Tree

- Combines insertion of Linked List with efficiency of searching a sorted Array
- One of the most fundamental algorithms in computer science

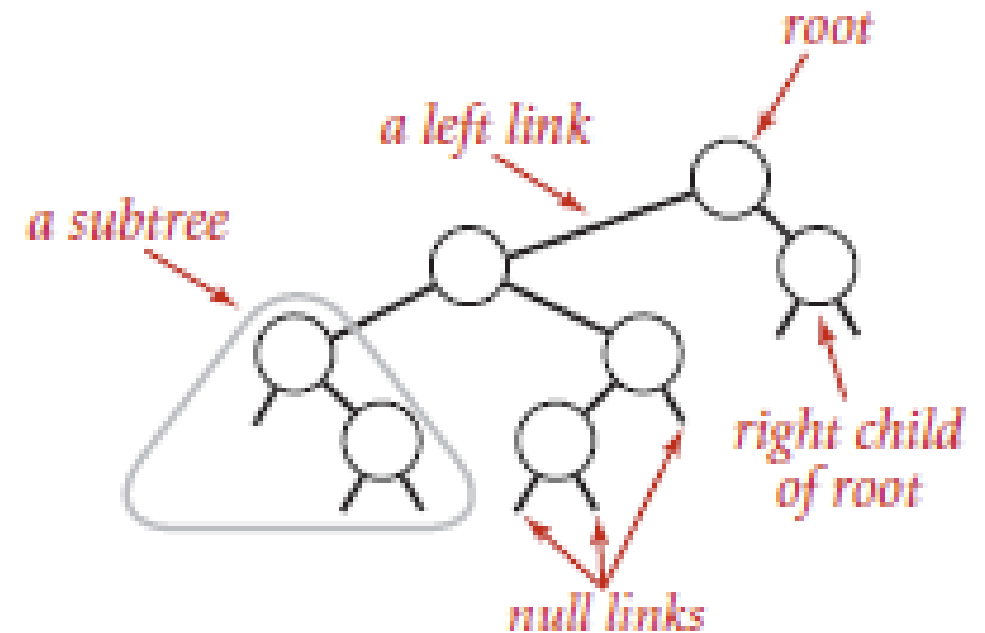


**Anatomy of a binary tree**



# Binary Tree Anatomy

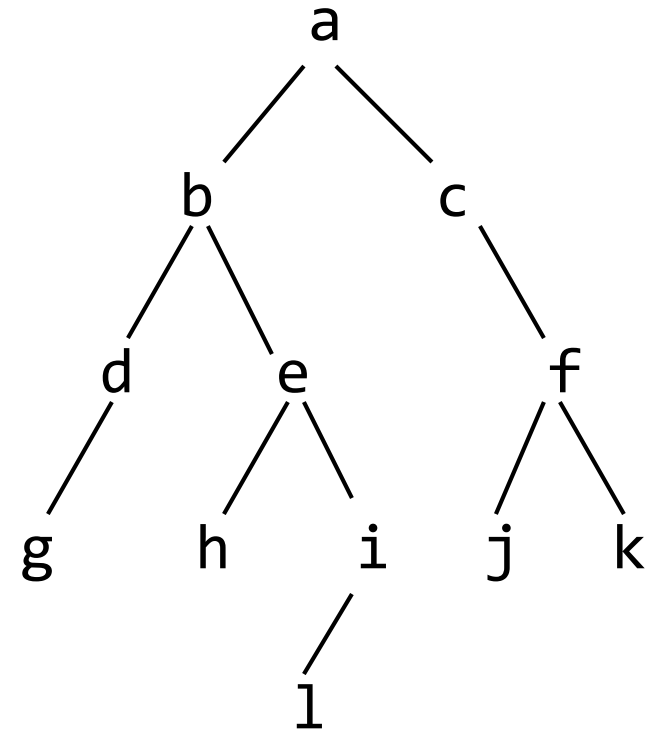
- A binary tree is composed of zero or more **nodes**
  - In Java, a reference to a binary tree may be **null**
- Each node contains:
  - A **value** (some sort of data item)
  - A reference to a **left child** (may be **null**), and
  - A reference to a **right child** (may be **null**)
- A binary tree may be *empty* (contain no nodes)
- If not empty, a binary tree has a **root node**
  - Every node in the binary tree is reachable from the root node by a *unique* path
- A node with no left child and no right child is called a **leaf**
  - In some binary trees, only the leaves contain a value



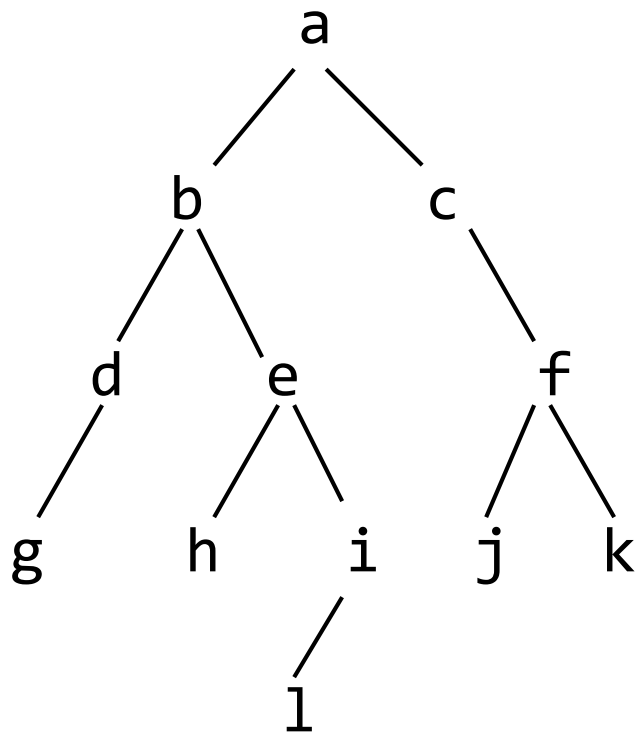
**Anatomy of a binary tree**

# More terminology

- Node A is the **parent** of node B if node B is a child of A
- Node A is an **ancestor** of node B if A is a parent of B, or if some child of A is an ancestor of B
  - In less formal terms, A is an ancestor of B if B is a child of A, or a child of a child of A, or a child of a child of a child of A, etc.
- Node B is a **descendant** of A if A is an ancestor of B
- Nodes A and B are **siblings** if they have the same parent

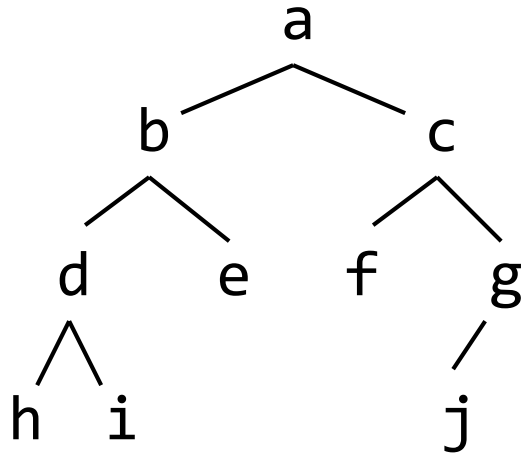


# Size and depth

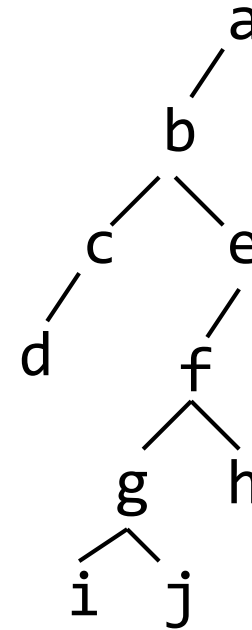


- The **size** of a binary tree is the number of nodes in it
  - This tree has size 12
- The **depth** of a node is its distance from the root
  - **a** is at depth zero
  - **e** is at depth 2
- The **depth** of a binary tree is the depth of its deepest node
  - This tree has depth 4

# Balance



A balanced binary tree

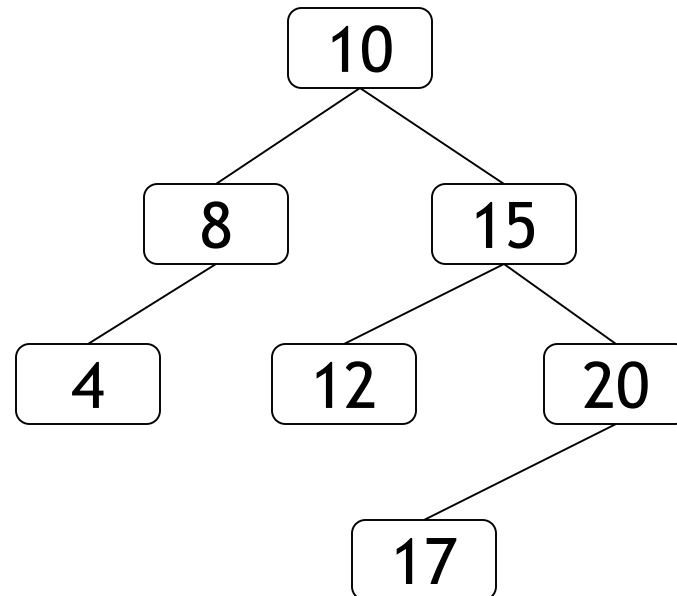


An unbalanced binary tree

- A binary tree is balanced if every level above the lowest is “full” (contains  $2^n$  nodes)
- In most applications, a reasonably balanced binary tree is desirable

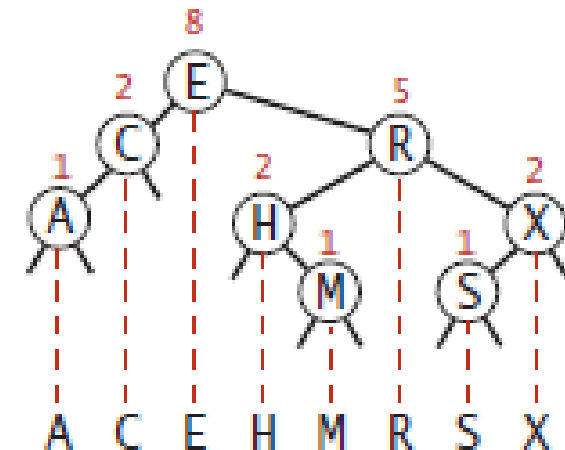
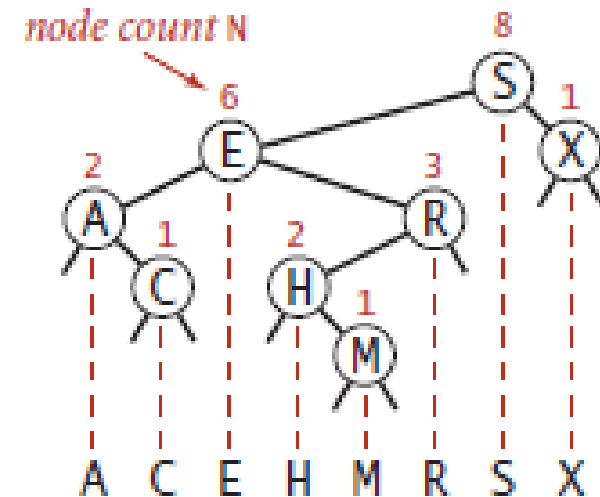
# Binary Search Trees

- A binary tree is sorted if every node in the tree is larger than (or equal to) its left descendants, and smaller than (or equal to) its right descendants
- Equal nodes can go either on the left or the right (but it has to be consistent)



# Binary Search Trees (BST)

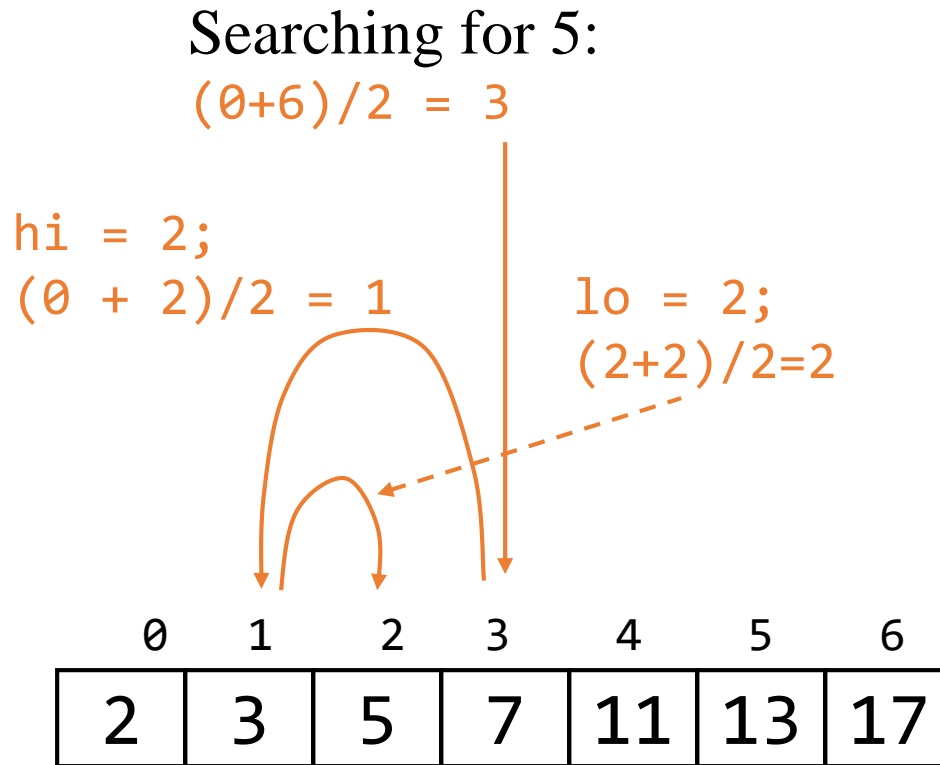
- Examine the following list of keys (sorted).  
A,C,E,H,M,R,S,X
- Opposite diagram represents keys as two different Sorted Binary Trees.
- Each left link references a Binary tree with smaller keys.
- Each right link references a Binary tree with larger Keys



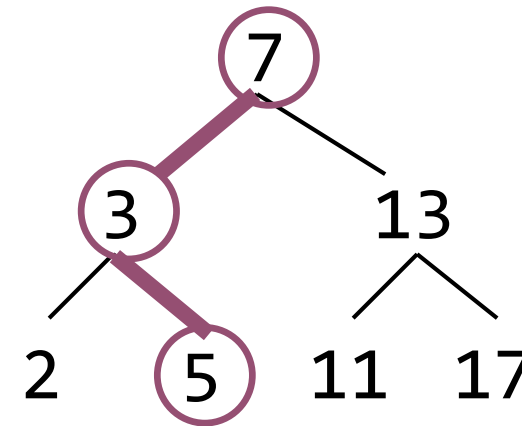
Two BSTs that represent  
the same set of keys

# Searching a Binary Search Tree.

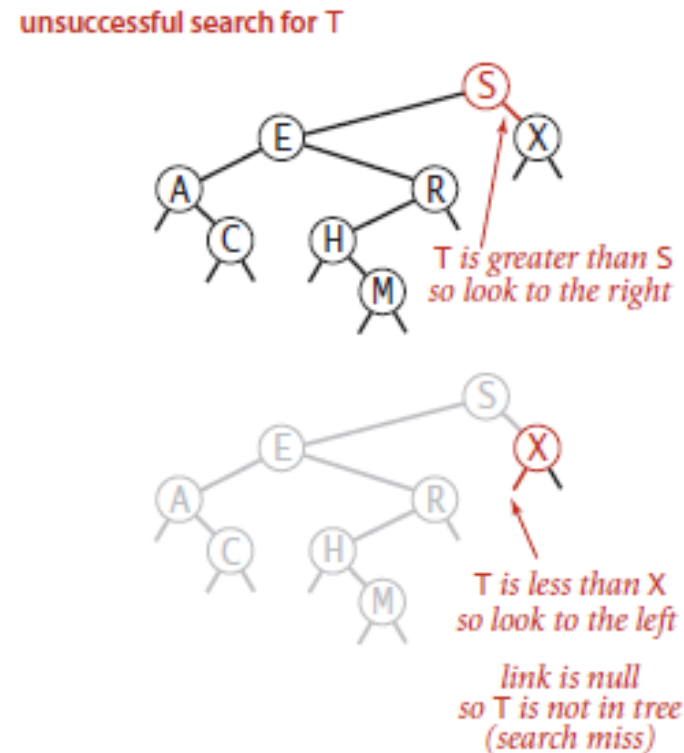
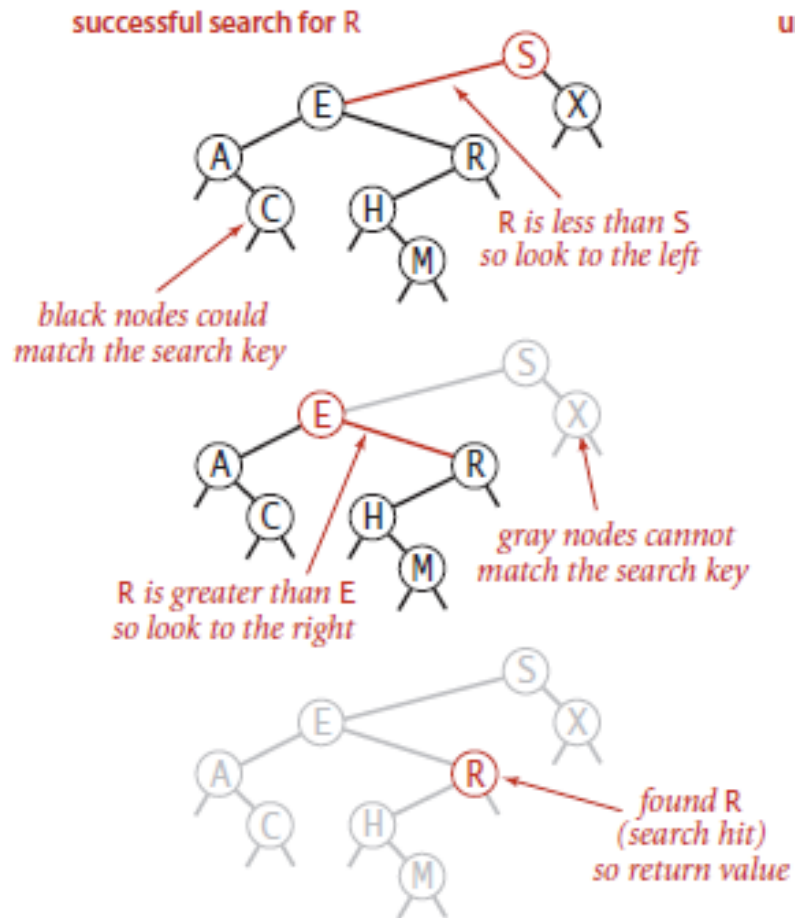
- Look at array location  $(lo + hi)/2$



Using a binary search tree



# Searching a Binary Search Tree





# Tree traversals

- A binary tree is defined recursively: it consists of a **root**, a **left subtree**, and a **right subtree**
- To **traverse** (or **walk**) the binary tree is to visit each node in the binary tree exactly once
- Tree traversals are naturally recursive
- Since a binary tree has three “parts,” there are six possible ways to traverse the binary tree:
  - root, left, right
  - left, root, right
  - left, right, root
  - root, right, left
  - right, root, left
  - right, left, root

# BinaryTree class

```
class BinaryTree<V> {  
    V value;  
    BinaryTree<V> leftChild;  
    BinaryTree<V> rightChild;  
  
    // Assorted methods...  
}
```

- A constructor for a binary tree should have three parameters, corresponding to the three fields
- An “empty” binary tree is just a value of null
  - Therefore, we can’t have an `isEmpty()` method (why not?)

# Preorder traversal

- In **preorder**, the root is visited *first*
- Here's a preorder traversal to print out all the elements in the binary tree:

```
public void preorderPrint(BinaryTree bt) {  
    if (bt == null) return;  
    System.out.println(bt.value);  
    preorderPrint(bt.leftChild);  
    preorderPrint(bt.rightChild);  
}
```

# Inorder traversal

- In *inorder*, the root is visited *in the middle*
- Here's an inorder traversal to print out all the elements in the binary tree:

```
public void inorderPrint(BinaryTree bt) {  
    if (bt == null) return;  
    inorderPrint(bt.leftChild);  
    System.out.println(bt.value);  
    inorderPrint(bt.rightChild);  
}
```

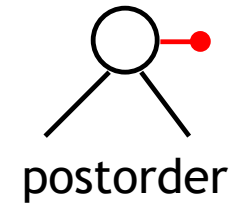
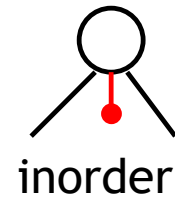
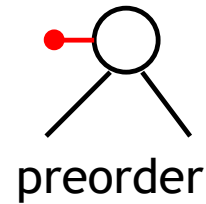
# Postorder traversal

- In `postorder`, the root is visited *last*
- Here's a postorder traversal to print out all the elements in the binary tree:

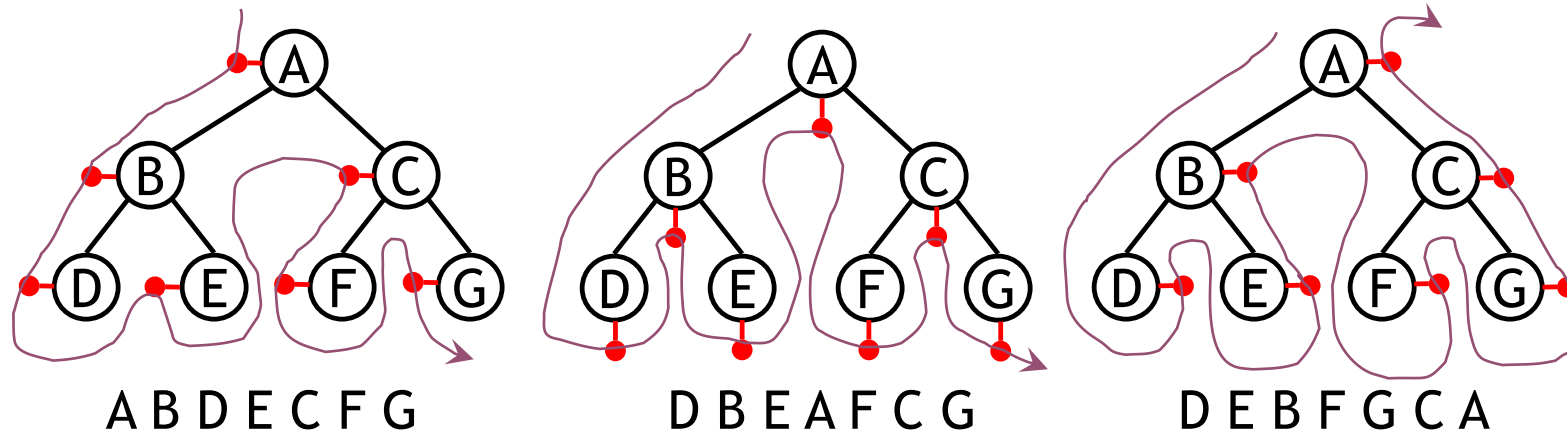
```
public void postorderPrint(BinaryTree bt) {  
    if (bt == null) return;  
    postorderPrint(bt.leftChild);  
    postorderPrint(bt.rightChild);  
    System.out.println(bt.value);  
}
```

# Tree traversals using “flags”

- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a “flag” attached to each node, as follows:

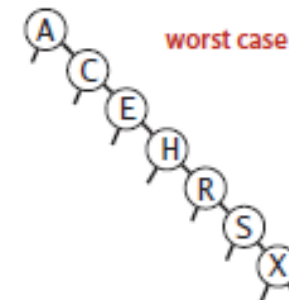
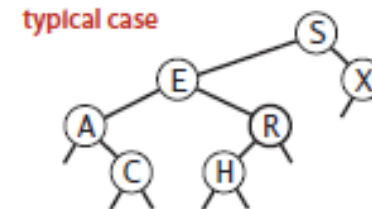
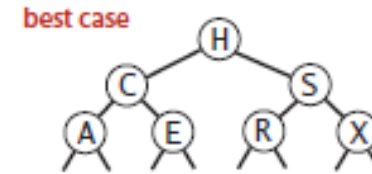


- To traverse the tree, collect the flags:



# Analysis of BST

- Running times of algorithms on BSTs depend on the shapes of the trees
- Depends on order in which keys are inserted
- Best Case: perfectly balanced, with  $\sim \lg N$  nodes between the root and each null link
- Worst Case:  $N$  nodes on the search path
- Good to get Balance...



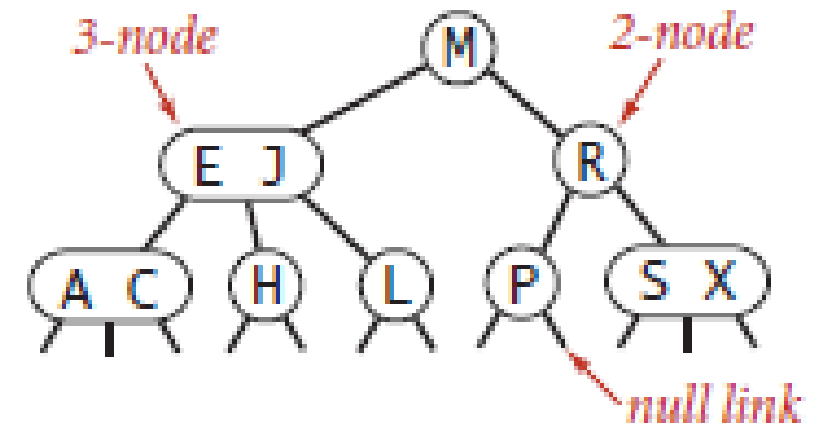
BST possibilities

Balanced Tree



# 2-3 search trees

- In an  $N$ -node tree, we would like the height to be  $\sim \lg N$
- Guarantees searches can be completed in  $\sim \lg N$  compares
- A *2-3 search tree* is a tree that has
  - A *2-node*, with one key, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys
  - A *3-node*, with two keys and *three* links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node's keys, and a right link to a 2-3 search tree with larger keys

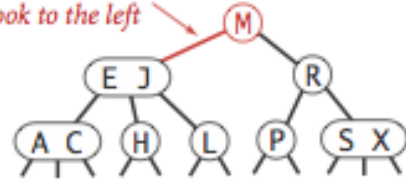


Anatomy of a 2-3 search tree

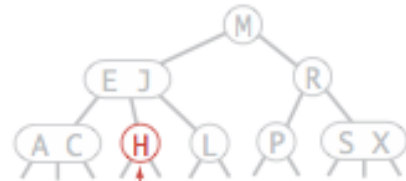
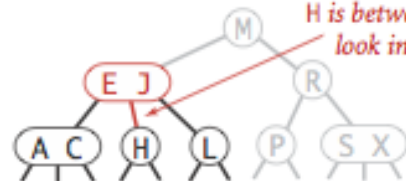
# Searching 2-3 node tree

successful search for H

*H is less than M so  
look to the left*



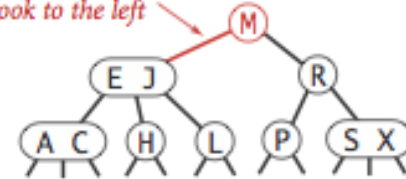
*H is between E and J so  
look in the middle*



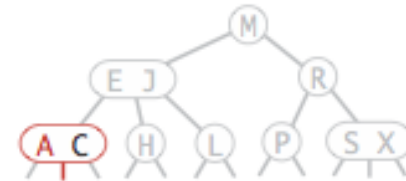
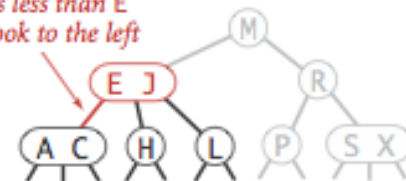
*found H so return value (search hit)*

unsuccessful search for B

*B is less than M so  
look to the left*



*B is less than E  
so look to the left*

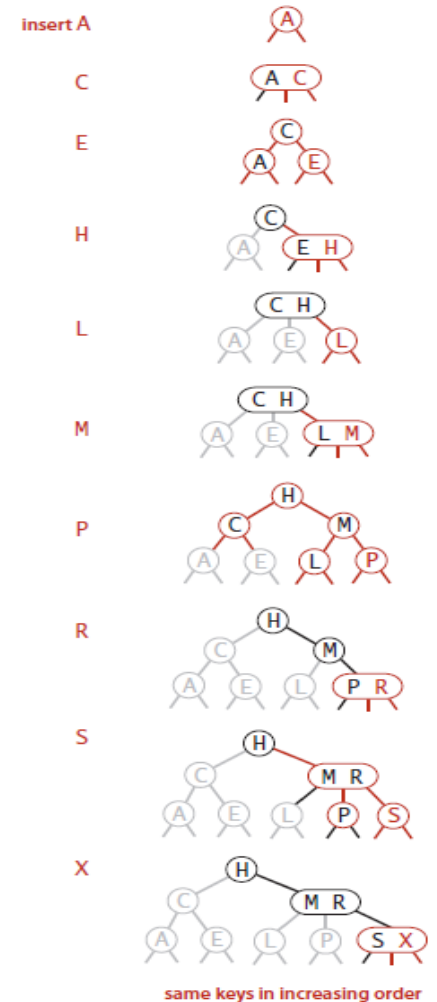
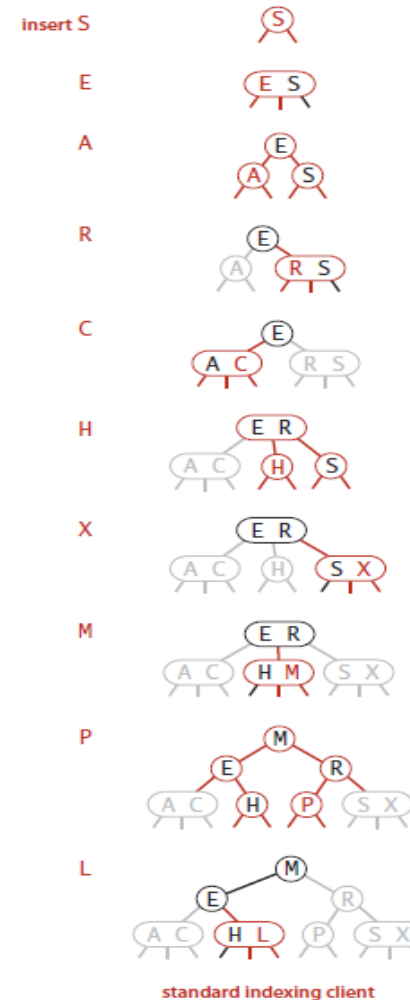


*B is between A and C so look in the middle  
link is null so B is not in the tree (search miss)*

Search hit (left) and search miss (right) in a 2-3 tree

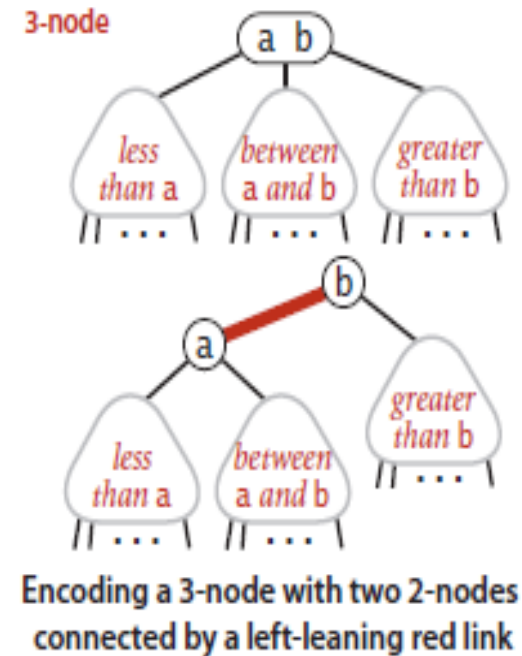
# Creation trace & Implementation

- Implementation of 2-3 tree is difficult:
  - need to maintain two different types of nodes
  - compare search keys against key
  - copy links and other information from one type of node
  - convert nodes from one type to another.
- Simplified using Red-Black BST
  - Reuses simple BST structure

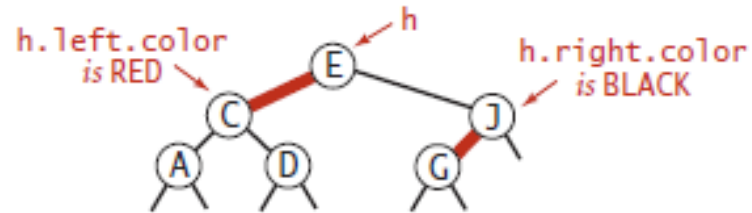


# Red-Black BST

- Basic idea:
  - use standard BSTs (which are made up of 2-nodes) and add extra information to encode 3-nodes.
- Use two different link types:
  - *red* links, which bind together two 2-nodes to represent 3-nodes
  - *Black* links, which bind together the 2-3 tree.
- No node has two red links connected to it.
- The tree has *perfect black balance* : every path from the root to a null link has the same number of black links.



# Node implementation



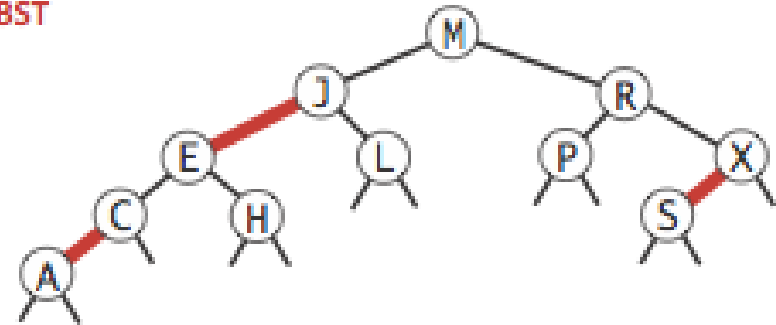
```
private static final boolean RED = true;
private static final boolean BLACK = false;

private class Node
{
    Key key;           // key
    Value val;         // associated data
    Node left, right;  // subtrees
    int N;             // # nodes in this subtree
    boolean color;     // color of link from
                      // parent to this node

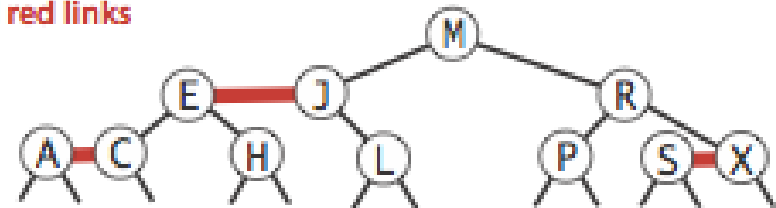
    Node(Key key, Value val, int N, boolean color)
    {
        this.key = key;
        this.val = val;
        this.N = N;
        this.color = color;
    }
}

private boolean isRed(Node x)
{
    if (x == null) return false;
    return x.color == RED;
}
```

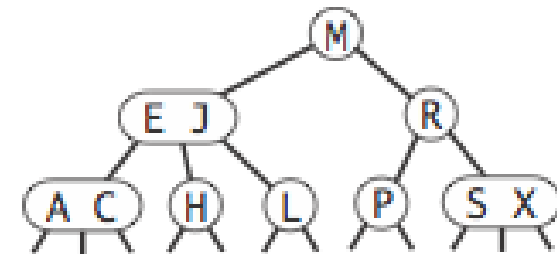
red-black BST



horizontal red links



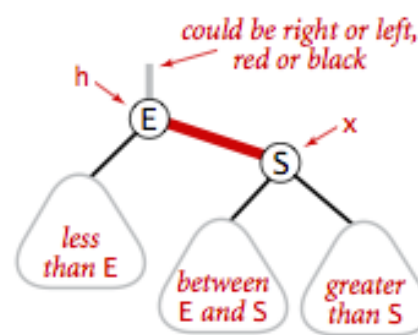
2-3 tree



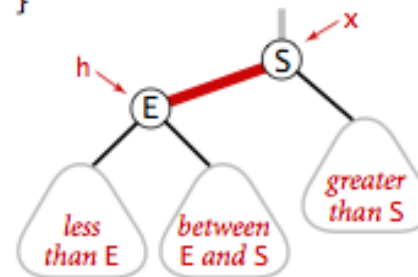
1-1 correspondence between red-black BSTs and 2-3 trees

# Rotating and Flipping

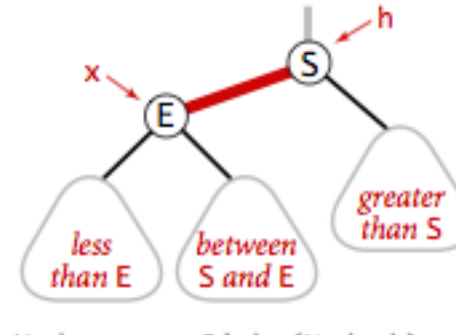
- Right-leaning red links or two red-links in a row can occur during operations (add, delete etc)
- This is corrected using a left or right rotate operation.
- May need to also “flip” colours



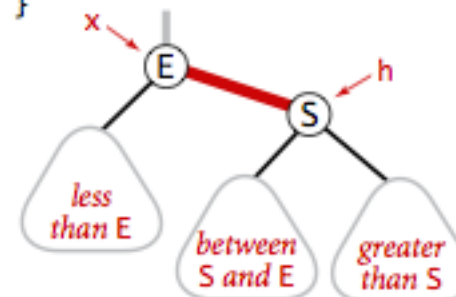
```
Node rotateLeft(Node h)
{
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left)
        + size(h.right);
    return x;
}
```



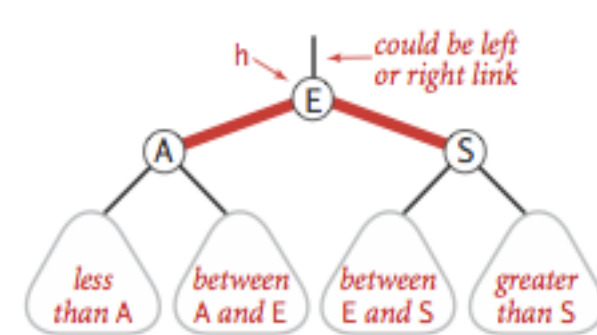
Left rotate (right link of h)



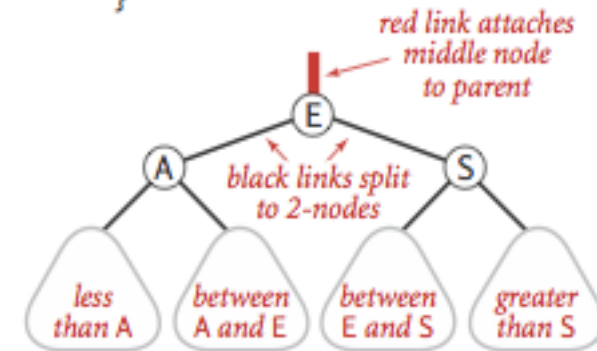
```
Node rotateRight(Node h)
{
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    x.N = h.N;
    h.N = 1 + size(h.left)
        + size(h.right);
    return x;
}
```



Right rotate (left link of h)



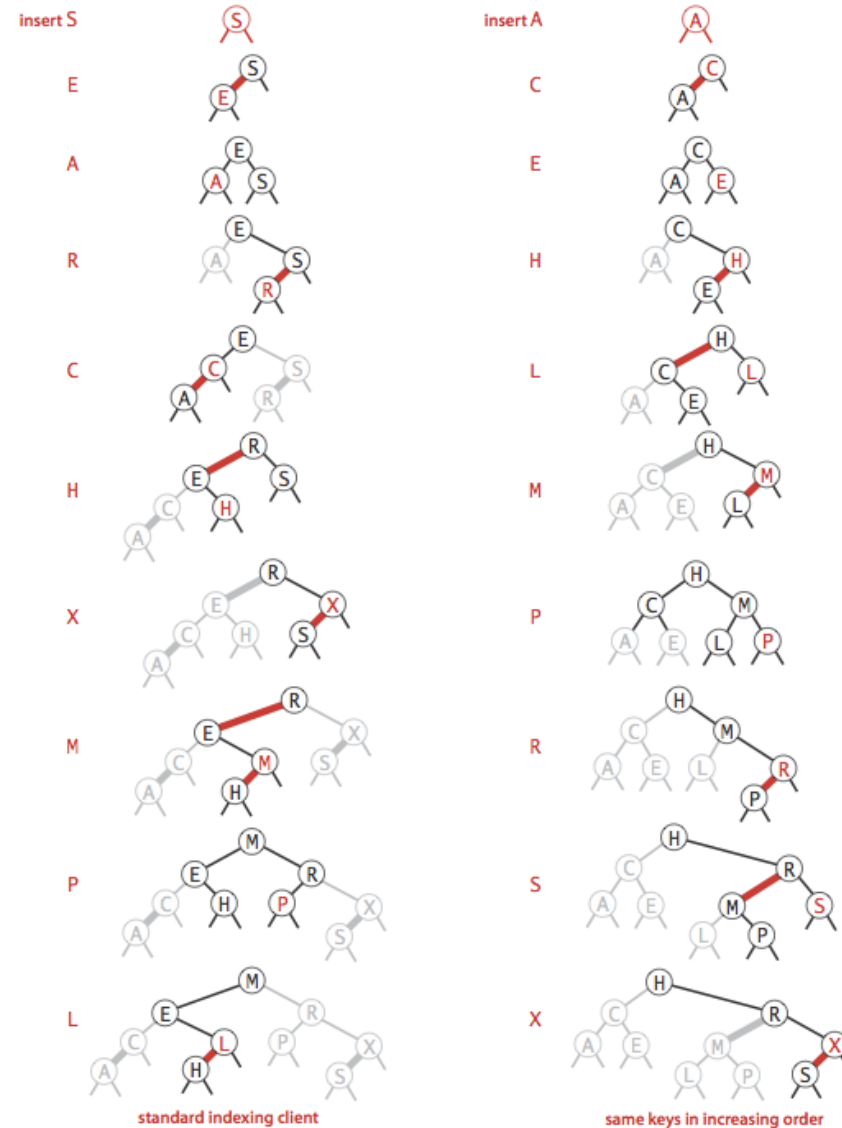
```
void flipColors(Node h)
{
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```



Flipping colors to split a 4-node

# Implementation

- Get source code at:  
<http://algs4.cs.princeton.edu/33balanced/RedBlackLiteBST.java.html>



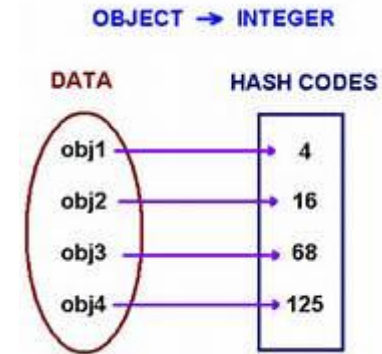
Red-black BST construction traces

# Hash Tables



# Hash Tables

- Using Key, Value pairs for data, if the key values are integers then we can interpret the key as the position in an Array.
  - For key  $i$ , store the value at location  $i$  in an array
  - Example(student Ids)
- Where keys are more complicated (e.g. string, phone number), consider Hashing.
- An example of Hashing is performing arithmetic operations to transform keys into array indices.
  - Store value at the corresponding index.

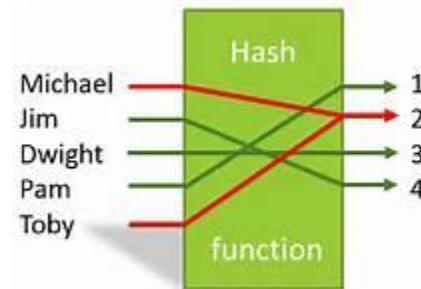


# Searching using Hash Tables

- Search algorithms that use hashing consist of two separate parts.
  - a *hash function* that transforms the search key into an array index.
  - *collision-resolution* process two or more different keys hash to the same array index.

$$f(A) = f(B)$$
$$A \neq B$$

This can  
happen



# Hash Functions

- Consider an array that can hold  $M$  key-value pairs.
- This requires a Hash function that can transform a key to an integer range 0 to  $M-1$ ,  $[0, M-1]$ .
- Hash function should compute index efficiently and evenly distribute values from 0 to  $M-1$  (uniform distribution).

# Hashing Positive integers

- Modular Hashing:
  1. Choose array of size  $M$  where  $M$  is a prime number.
  2. For an integer key  $k$ , compute  $k$  modulo  $M$ . This gives the remainder of  $k$  divided by  $M$ .
  3. Produces uniform distribution from 0 to  $M-1$

key	hash ( $M=100$ )	hash ( $M=97$ )
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25
701	1	22
418	18	30
601	1	19

Modular hashing

# Hashing Floats

- Key is real number,  $k$ , between 0 and 1.
  1. Choose array of size  $M$  where  $M$  is a prime number.
  2. Multiply  $k$  by  $M$  and round to the nearest integer to get an index between 0 and  $M-1$ .
  3. Better way: Use modular hashing on the binary representation of the key

# Hashing Strings

- Treat Strings as huge integers.
  1. Choose array of size M where M is a prime number.
  2. Choose small prime number R.
  3. The following computes a hash.

```
int hash = 0;  
for (int i = 0; i < s.length(); i++)  
    hash = (R * hash + s.charAt(i)) % M;
```

# Multiple Fields

- If the key type has multiple integer fields:
  - Mix together as follows for Student object with studentId, yearOfBirth, and CourseId fields.

```
int hash = (((studentId * R + yearOfBirth) % M) * R + CourseId) % M;
```

# Example of User Defined Hash Code

```
public class Transaction
{
...
    private final String who;
    private final Date when;
    private final double amount;
    public int hashCode(){
        int hash = 17;
        hash = 31 * hash + who.hashCode();
        hash = 31 * hash + when.hashCode();
        hash = 31 * hash + ((Double) amount).hashCode();
        return hash;
    }
...
}
```



# References

<http://algs4.cs.princeton.edu/home/>

[https://en.wikipedia.org/wiki/Binary\\_tree](https://en.wikipedia.org/wiki/Binary_tree)