Binary SearchTrees

Outline

1. Review on Fundamental Data Structures
2. Binary Search Trees

Dynamic Sets

- A dynamic set is a *collection* of data that can *change* over time both in size and contents
- Basic operations on dynamic sets:
  - Insert (S, x), Delete (S, x), Search (S, x)
- There are *various types* of Dynamic Sets
  - These are referred to as ***Fundamental Data Structures***
  - A type of a Dynamic Set (a Fundamental Data Structure) ***serves a specific purpose*** and is designed to ***address a particular usage context*** determined by:
    - the **subsets of operations** they frequently use:
      - List, Array
    - the **special semantics** of the basic operations:
      - Stack, Queue

Fundamental Data Structures Review

- **Stacks**:
  - *Insert (Push), Delete (Pop)*
  - Implemented by ***Lists****; O(1), O(1)*
- **Queues:**
  - *Insert (Enqueue), Delete (Dequeue)*
  - Implemented by ***Lists****; O(1), O(1)*
- **Priority Queues**: see [Sedgewick 2.4] [CLRS chap 6.5]
  - *Insert, Maximum, ExtractMax, IncreaseKey*
  - Implemented by ***Heaps****; O(log n), O(1), O(log n), O(log n)*
- **Dictionaries:** data structure for key-value pairs
  - *Insert, Search, Delete*
  - Implemented by ***Hashtables***: see [Sedgewick chap 3.4][CLRS chap 11]
  - Generalizes the simple arrays, allows direct addressing of arbitrary positions in O(1)
  - *Hashing with chaining*; *Hashing with open addressing*

Libraries Implementing Fundamental Data Structures

- There are standard libraries implementing the fundamental data structures for most languages
- In industrial practice is recommended to use library implementations instead of redoing your own
- *Sometimes* you may need a very *customized i*mplementation
- BUT: *in order to **understand** them and to use them **correct** and **appropriate**, don't forget anything that you learned about fundamental data structures !*

- Example: Java: Collections
  - https://docs.oracle.com/javase/tutorial/collections/TOC.html

Fundamental Data Structures - Java

- Class **`java.util.Stack<E>`**
- Interface **`java.util.Queue<E>`**
  - General purpose queue implementation:
    class **`java.util.LinkedList<E>`**
- Class **`java.util.PriorityQueue<E>`**
  - Implements an unbounded priority queue based on a *Heap*.
- Dictionaries: Interface **`java.util.Map<K,V>`**
  - General purpose implementation with Hashing:
    Class **`java.util.HashMap<K,V>`**

Binary SearchTrees

[Sedgewick] – Chap 3.2

[CLRS] – Chap 12

Context and Problem

- We need a dynamic set that g*eneralizes both the Dictionary structure as well as the Priority Queue structure*
  - A **Sorted Dictionary:** a Dictionary that provides a *total ordering of the keys* (keys can be sorted, have min, max values, etc)
  - *all operations are equally important*: insert, delete, search, maximum, minimum, ordered list

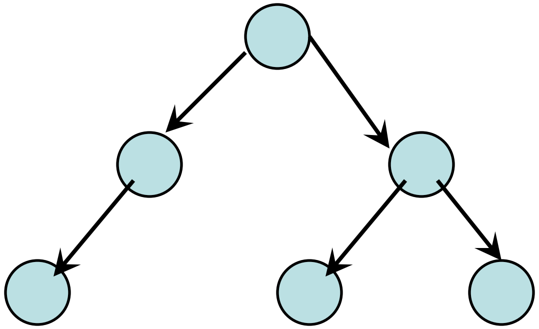
- Such an abstract data type exists in Java collections:
- Interface `java.util.SortedMap<K,V>`
  - `Extends interface Map<K,V>`
- General purpose implementation: Class `java.util.TreeMap<K,V>`
  - `TreeMap` is implemented by *Balanced Binary Search Trees*

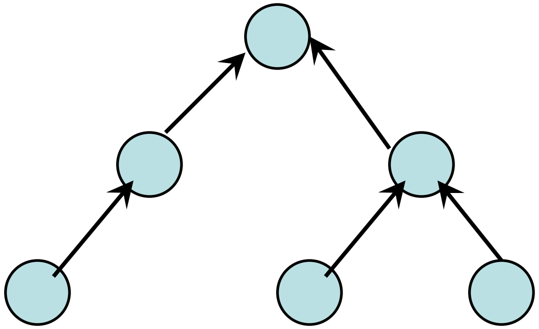
Kinds of Trees

- Binary Trees
  - Min-heaps and Max-heaps
  - Trees for arithmetic expressions
- Binary Search Trees
- Balanced Binary Search Trees

What is a binary tree ?

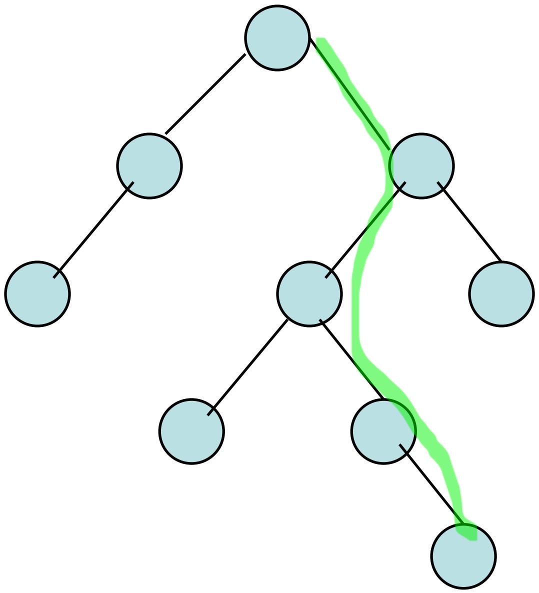
- A binary tree is *a linked data structure in* which *each node* is an object that contains following *attributes*:
  - a *key* and satellite data
  - *left*, pointing to its left *child*
  - *right*, pointing to its right *child*
  - *p*, pointing to its parent
- An implementation of a binary tree as a linked data structure may chose to represent it with help of links to children only or links to parent only.





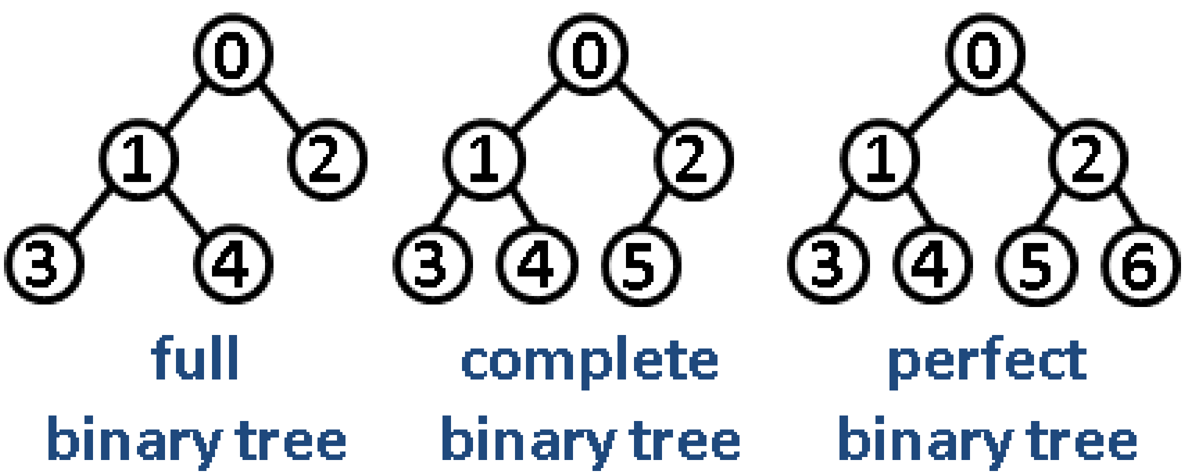
Concepts of a binary tree

- Particular kinds of nodes:
  - ***Root***: node that has no parent
  - ***Leaves:*** nodes that have no children
- ***Height*** of a tree: longest path from the root to one of the leaves; max(heights of subtrees) + 1



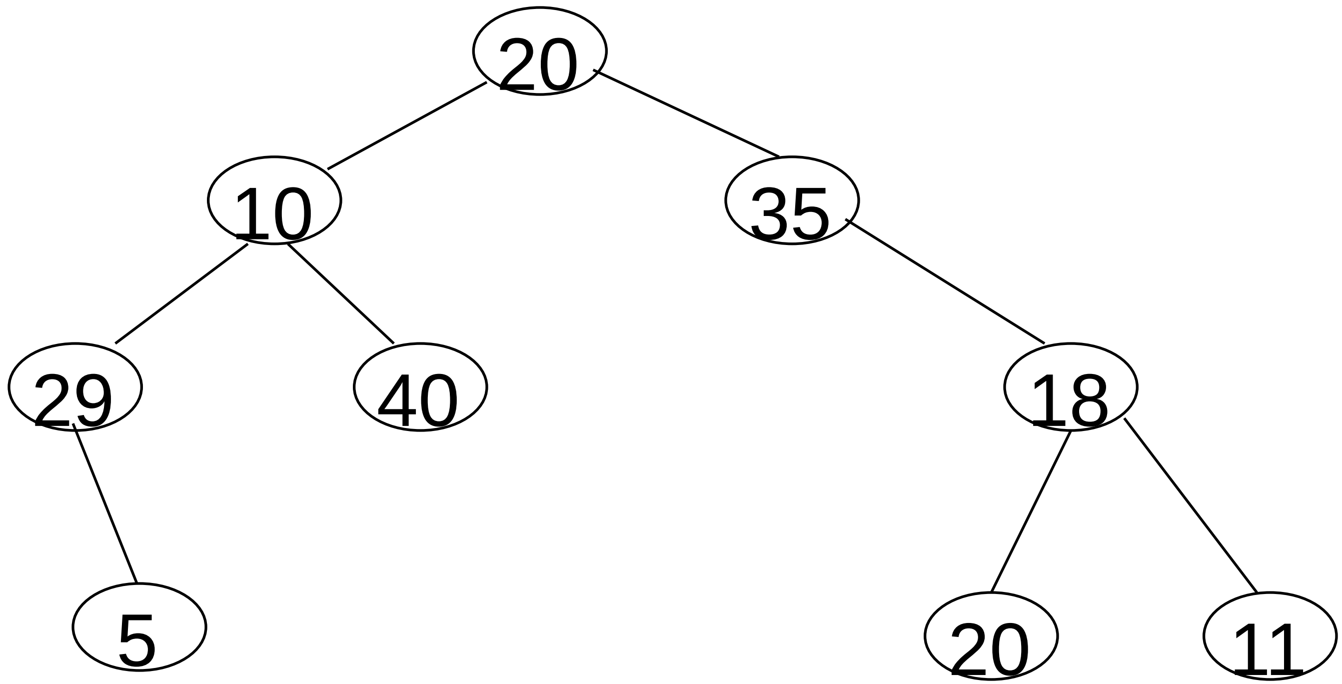
Shapes of binary trees

- A **full binary tree** is a binary tree in which every node other than the leaves has two children.
- A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- A **perfect binary tree** is a binary tree in which all interior nodes have two children *and* all leaves have the same *level.*
- Degenerated tree – if every node has only 1 child, the tree is like a list



Kinds of binary trees

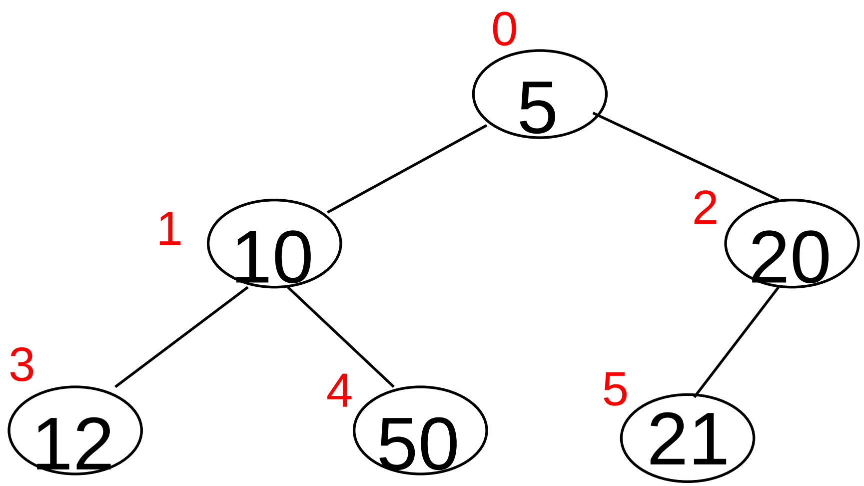
- **General binary tree: no conditions regarding key values and tree shape.** Example:



Kinds of binary trees

- **Min-heap tree**: is a binary tree such that:
  - the key contained in each node is *less* than (or equal to) the key in that node’s children.
  - the binary tree is *complete*
- **Max-heap tree**: is a binary tree such that:
  - the key contained in each node is bigger than (or equal to) the key in that node’s children.
  - the binary tree is *complete*
- Because Max-heap and Min-heap are *complete* trees they can be *represented as arrays* (position in array corresponds to position in complete binary tree) not as linked structures
- Review: Min-heap and Max-heap were used for Priority Queues and Sorting

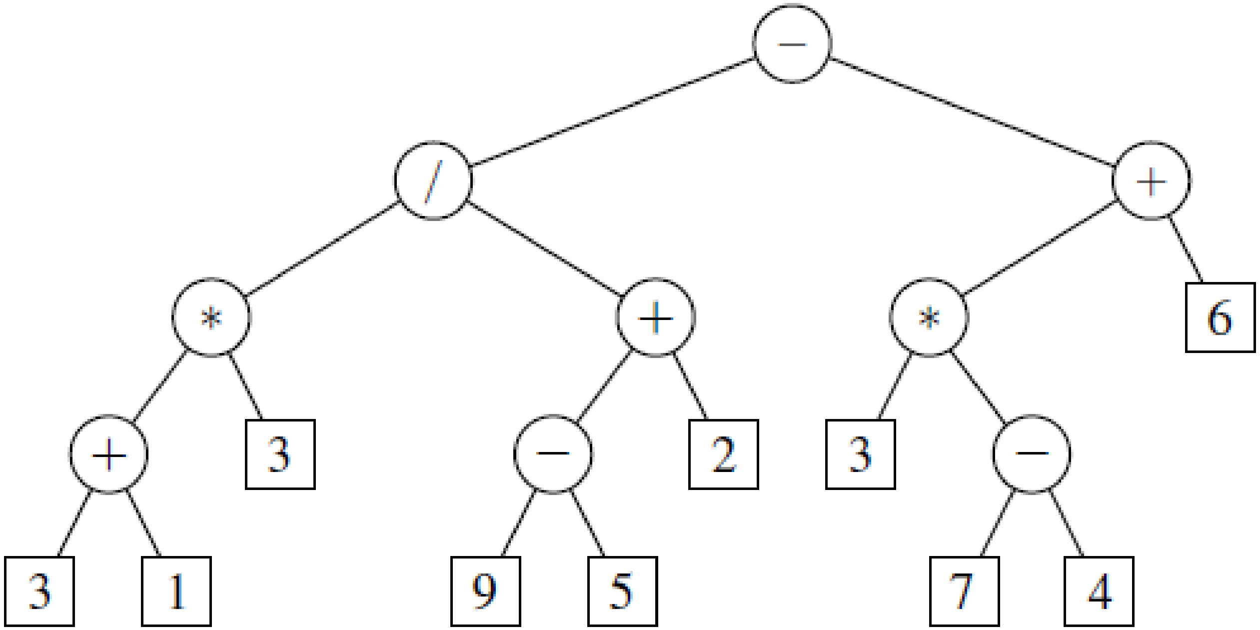
Min-heap kind of Binary Tree Example



0	1	2	3	4	5	
5	10	20	12	50	21	

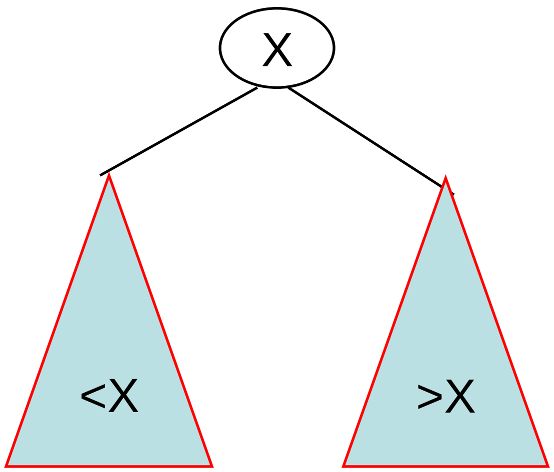
Binary Tree Representing Arithmetic Expression

- Leaves contain numbers, and internal nodes contain binary operators
- Each node has a value associated with it:
  - For non-leaves the value is defined by applying its operation to the values of its children
- Example: ((3+1)*3)/((9-5)+2)-((3*(7-4))+6)

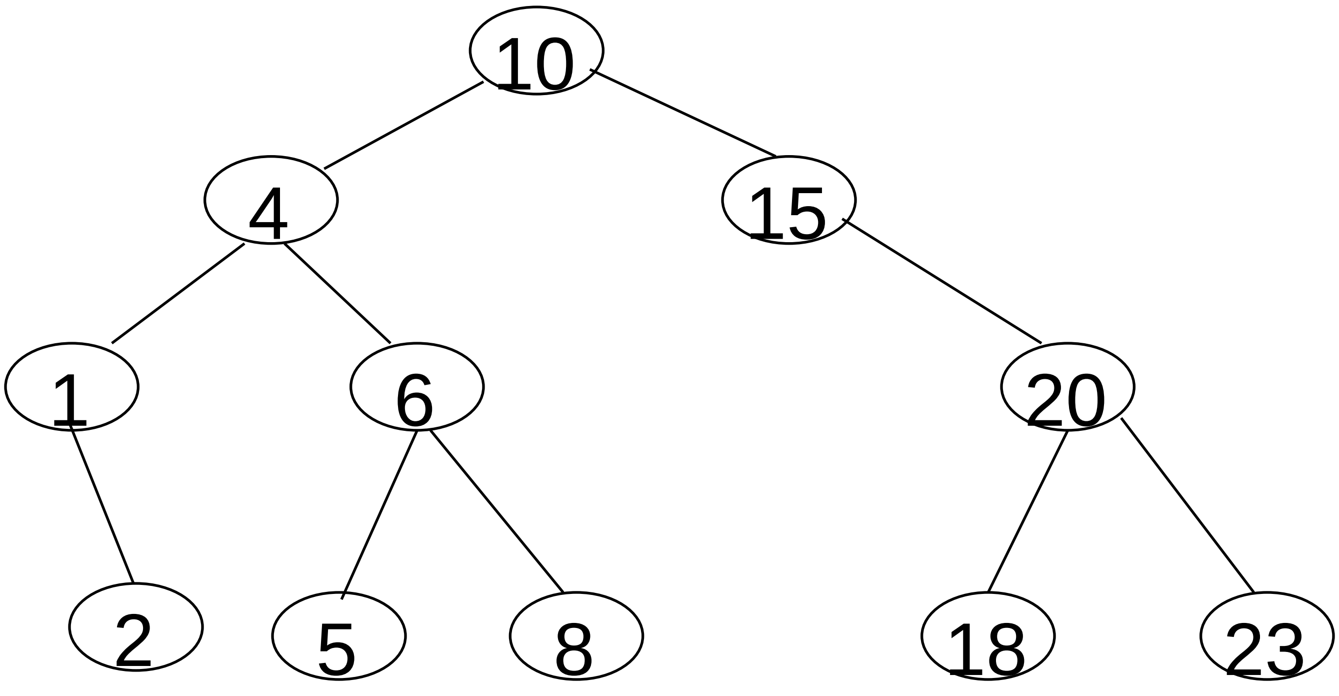


What is a Binary Search Tree (BST)

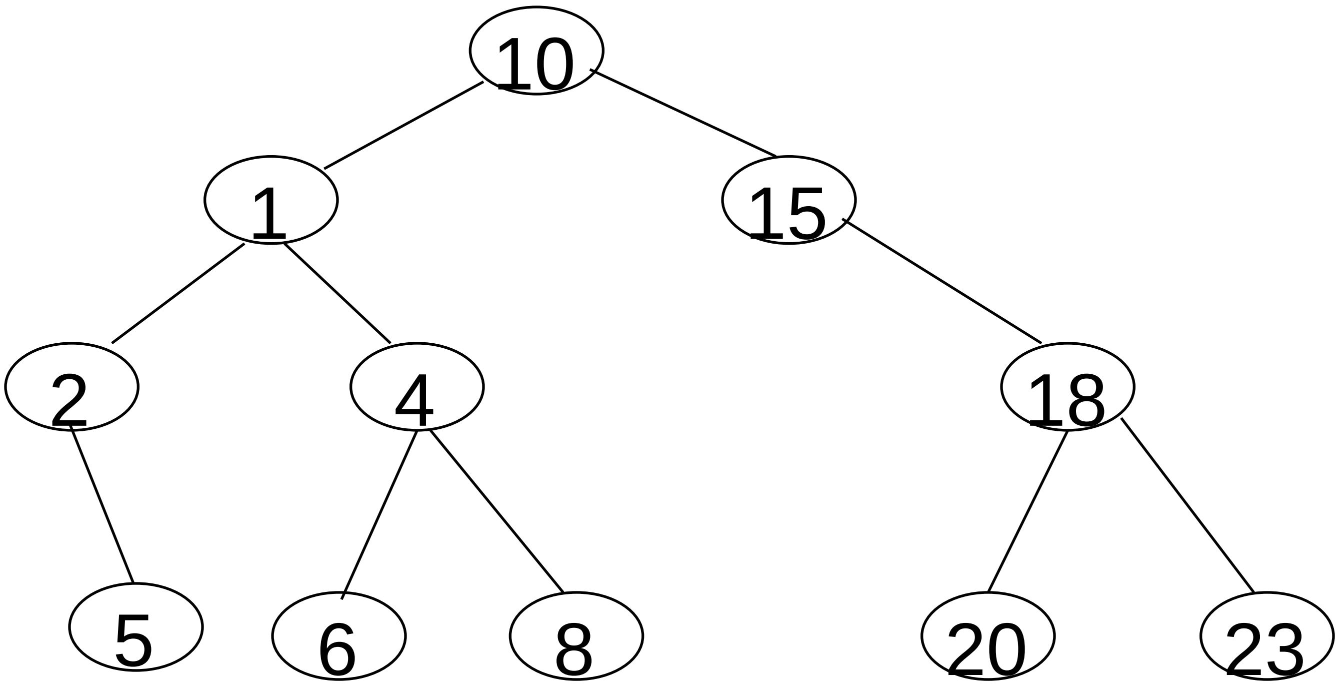
- The ***binary-search-tree property***:
- A *binary search tree* (BST) is a binary tree where each node has a Comparable key and satisfies the restriction that the key in ***any node*** is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.



Example – BST



Counterexample – NOT a BST !



Simple BST in Java

```java
public class IntBST {

    private class Node {   // BST Node = nested inner class
        int key;           // sorted by Key
        int val;         // associated data
        Node left, right;  // left and right subtrees

        Node(int key, int val) {
            this.key = key;
            this.val = val;
        }
    }

    private Node root;           // root of BST

    public IntBST() {
        root = null;       // initializes empty BST
    }
   // …
}
```

As a first example, we consider a **simple BST,** having `int` keys and values

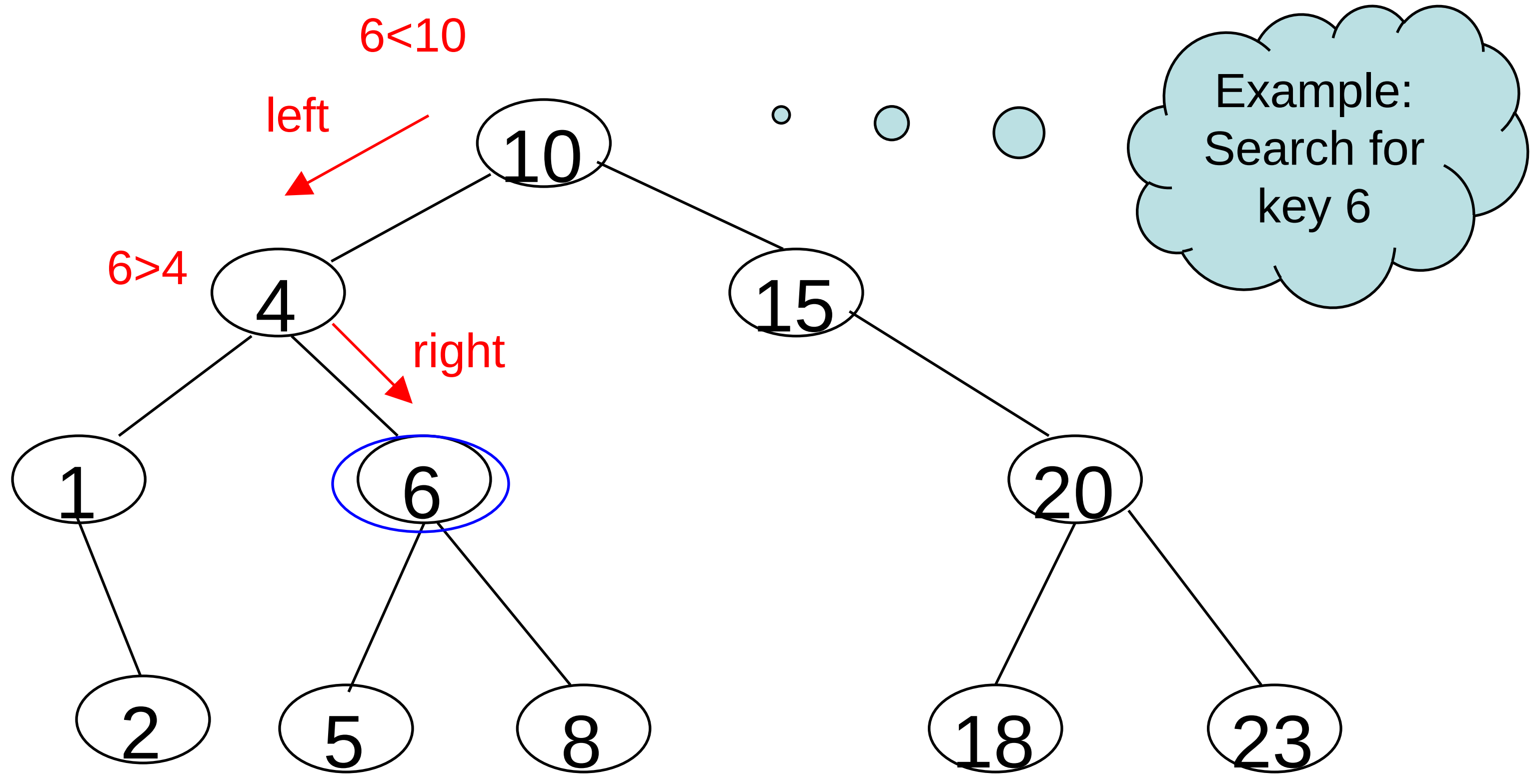
If Node is an inner class, then only class IntBST can see it!
Public methods of IntBST cannot have Node in their signature !

What can be done on a BST ?

- Queries
  - Search: search if there is a node with the given key
    - `boolean contains(int key);`
  - Minimum: returns smallest key
    - `int min();`
  - Maximum: returns biggest key
    - `int max();`
- Tree walks
  - List in order: prints all keys in increasing order
    - **`void`** `inorder();`
- Modifying
  - Insert: if key is not in tree, add node, else update the value
    - `void put(int key, int value);`
  - Delete: delete the node containing key
    - `void delete(int key);`

```
public class IntBST {

    private class Node {
     …
    }

    private Node root;

    public IntBST() {  root = null;    }


    public boolean contains(int key) { … }

    public int min() {  … }

     public int max() { … }

     public void inorder() { …   }

     public void put(int key, int val) { …  }


     public void delete(int key) { … }

}
```

Search - Example

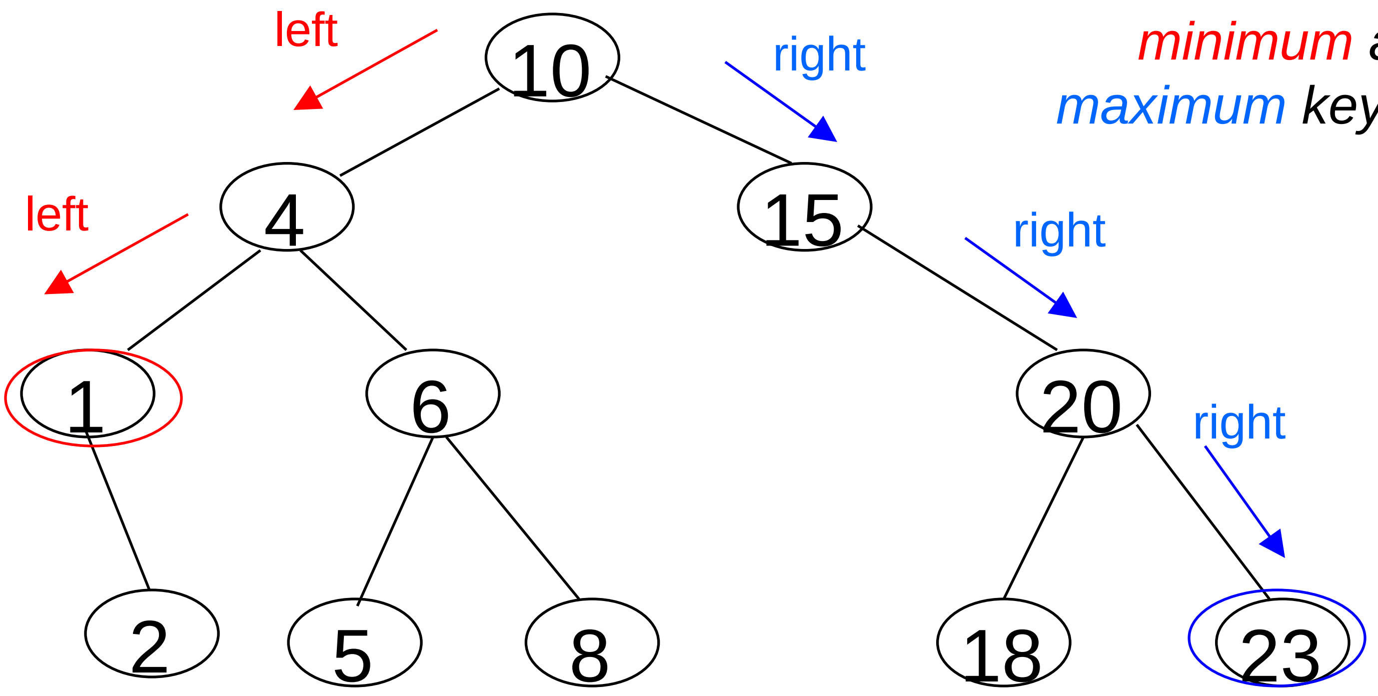


Search – recursive implementation

```
public boolean contains(int key) {
   return contains(root, key);
}

private boolean contains(Node x, int key) {
   if (x == null) return false;
   if (key < x.key) return contains(x.left, key);
   else if (key > x.key) return contains(x.right, key);
   else return true;
}
```

Overloaded method: Public method **hides** the detail of Node

Minimum and Maximum - Example



Search nodes with minimum and maximum key values

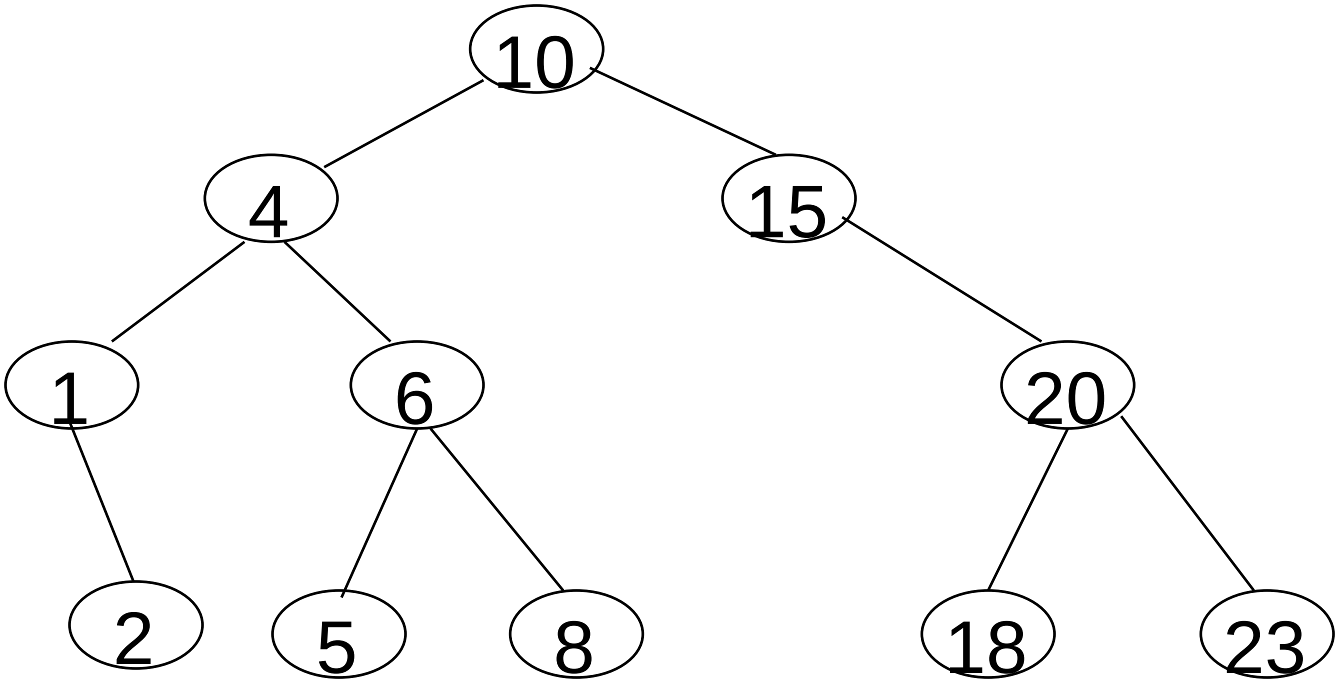
Minimum - Implementation

```
public int min() {
    if (root == null) throw
            new NoSuchElementException("calls min() with empty BST");
    return min(root).key;
}

private Node min(Node x) {
    if (x.left == null) return x;
    else return min(x.left);

}
```

Complexity of queries

- Search: we count the number of nodes inspected during the search
- The maximum number of nodes that are inspected is equal with the height of the tree (the longest path from root to a leaf)
  - This is the worst case for search
  - O(h), h=the height of the BST

Tree walks- List in order - Example



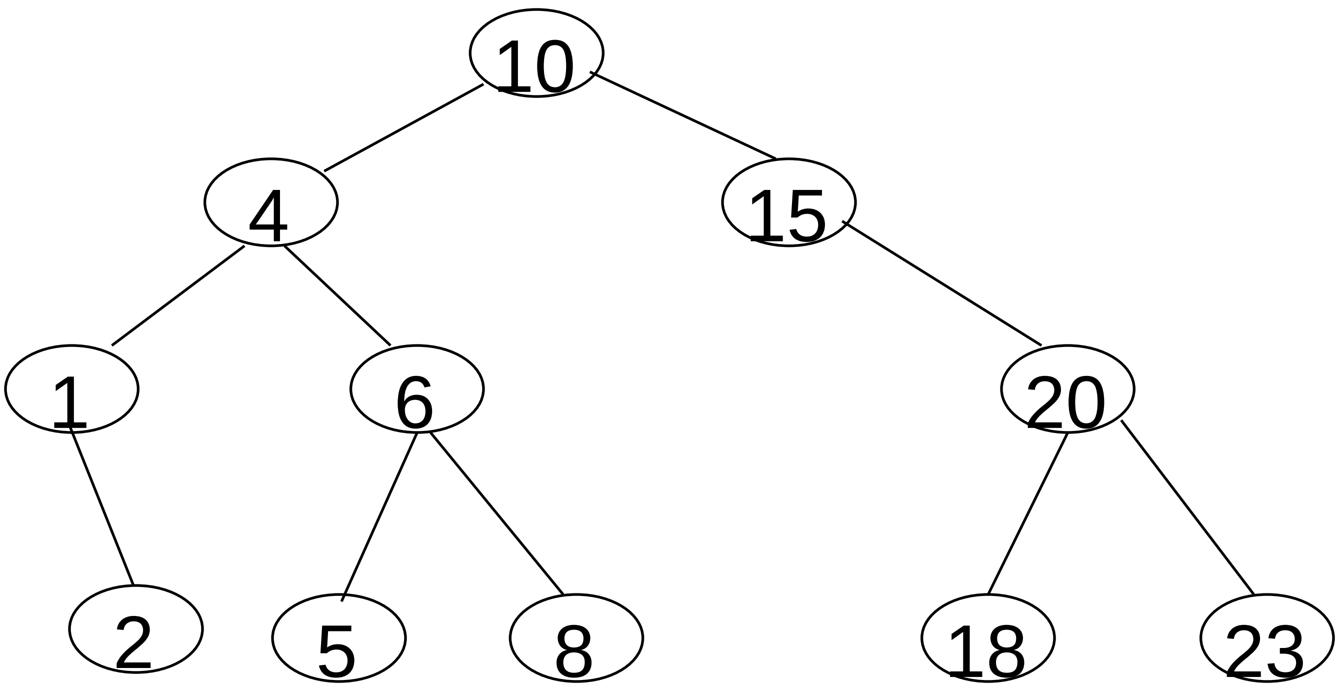
List in order: 1 2 4 5 6 8 10 15 18 20 23

Tree walks

- The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an ***inorder tree walk***.
- This algorithm is so named because it prints the key of the *root in between* printing the values in its left subtree and printing those in its right subtree.

- ***Postorder***: print subtrees, *after* this print root
- ***Preorder:*** print *root first* and then print subtrees

Tree walks - Example



Inorder: 1 2 4 5 6 8 10 15 18 20 23

Postorder: 2 1 5 8 6 4 18 23 20 15 10

Preorder: 10 4 1 2 6 5 8 15 20 18 23

Inorder - Implementation

```
public void inorder() {
    inorder(root);
}

private void inorder(Node x) {
    if (x == null) return;
    inorder(x.left);
    System.out.print(" "+x.key);
    inorder(x.right);

}
```

Complexity of tree walks

- We have a BST with n nodes
- Intuitively: count how many nodes are visited: every node is visited exactly once and a constant time operation (print) is done on it => $\Theta(n)$
- Formal:
- Suppose that the BST with n nodes has k nodes in the left subtree and n-k-1 in the right subtree
- $T(n)=c$, if $n=0$
- $T(n)=T(k)+T(n-k-1)+d$, if $n>0$
- We assume that $T(n)=(c+d)*n+c$ => easy proof by *induction* (verify basecase for n=0, assume true for all smaller than n, replace in T(n) and prove that it is according to the assumed formula)

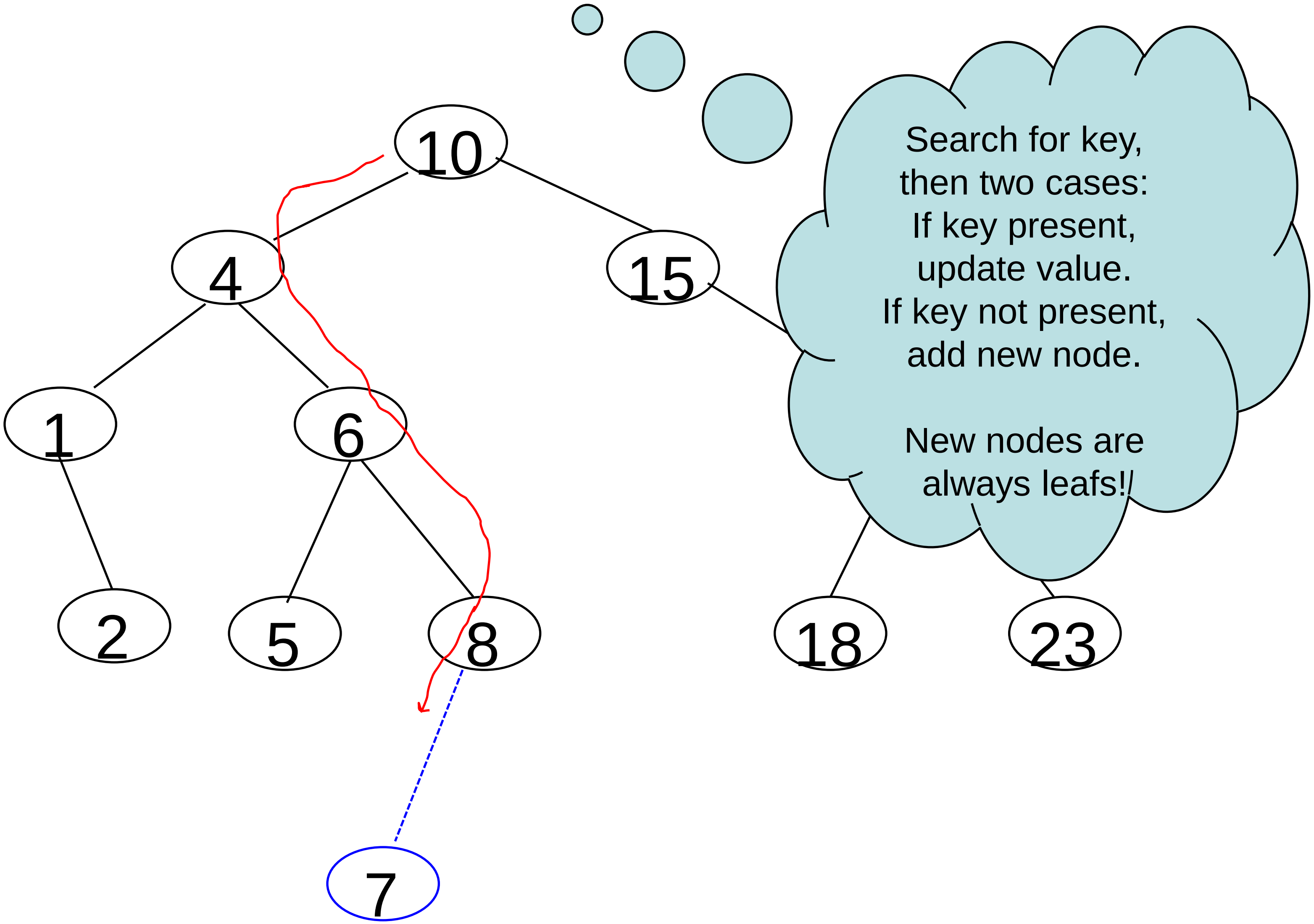
Modifying a binary search tree

- The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change.
- The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold !

Operation: Insert

- put(key, value) creates and inserts a new node in BST, if the BST does not already contain the key
- If BST already contains the key, no new node is created but the value is updated
- Key must be non null
- After insertion, the tree must remain a binary search tree

Example – Insert key 7



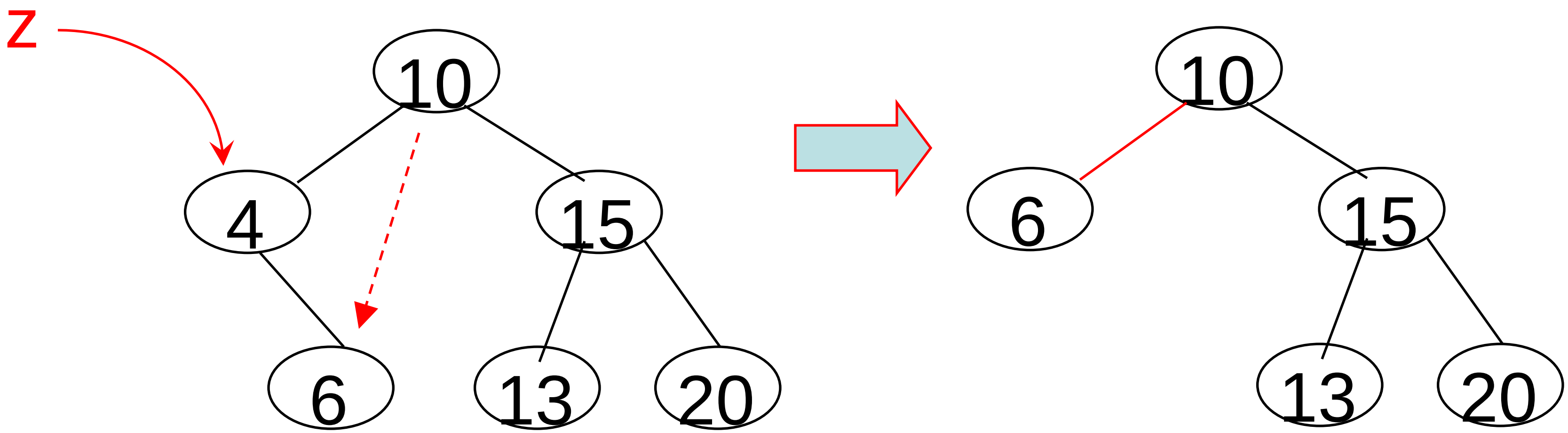
Insert – recursive implementation

```
public void put(int key, int val) {
    root=put(root, key, val);
}

private Node put(Node x, int key, int val) {
    if (x == null) return new Node(key, val);
    if (key < x.key) x.left = put(x.left, key, val);
    else if (key > x.key) x.right = put(x.right, key, val);
    else x.val = val;
    return x;

}
```

Operation: Delete

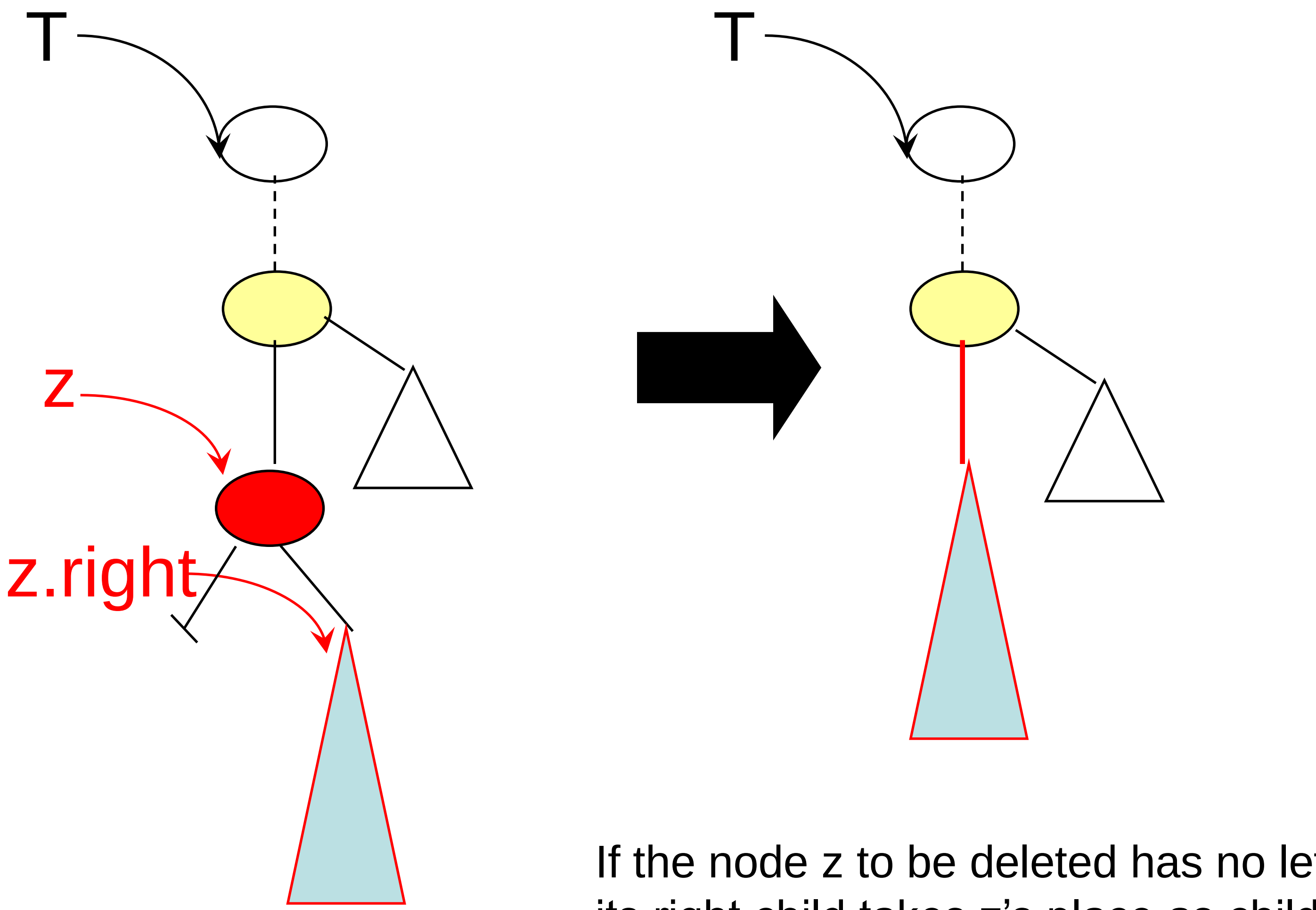
- Delete a node z from a binary search tree T
- After delete, T must remain a binary search tree
  - Important requirement for a good delete algorithm: do a minimum number of tree link reassignments!
- Cases:
  - Cases 1A+1B: node z has only 1 child -> the only child takes the place of its deleted parent z
  - Case 2A+2B: node z has 2 children. The *successor* of z will take its place.
    - In **this case** (node **z has two children**), the successor of z is the minimum value in the right subtree of z
    - In a general case, the successor of a node with no right child is among its ancestors!

Example - delete



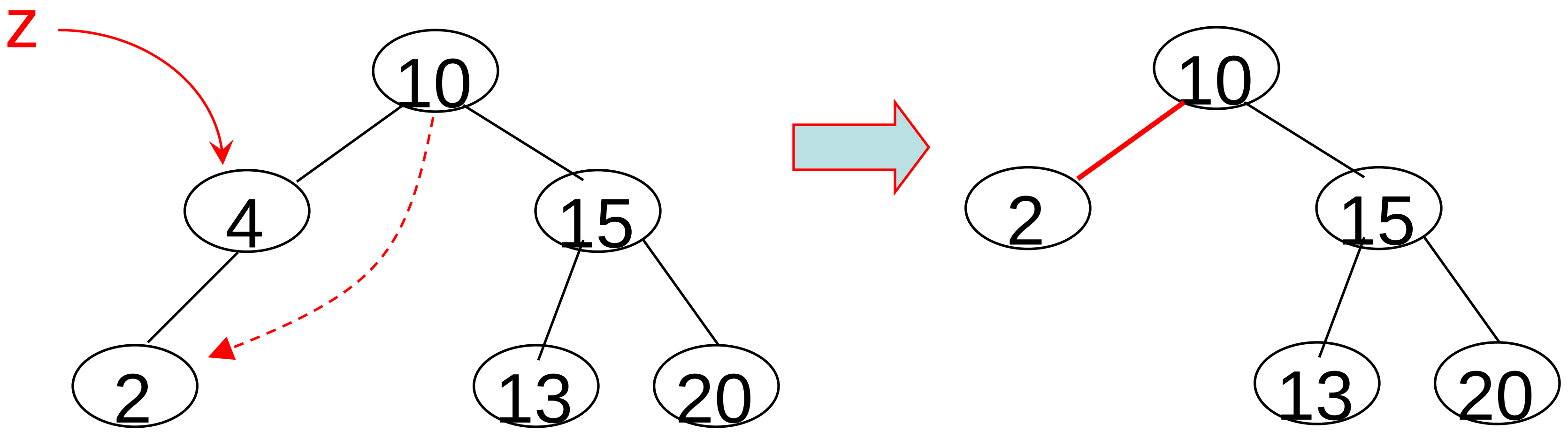
Delete z. Case 1A: z has no left child

Tree delete – case 1A



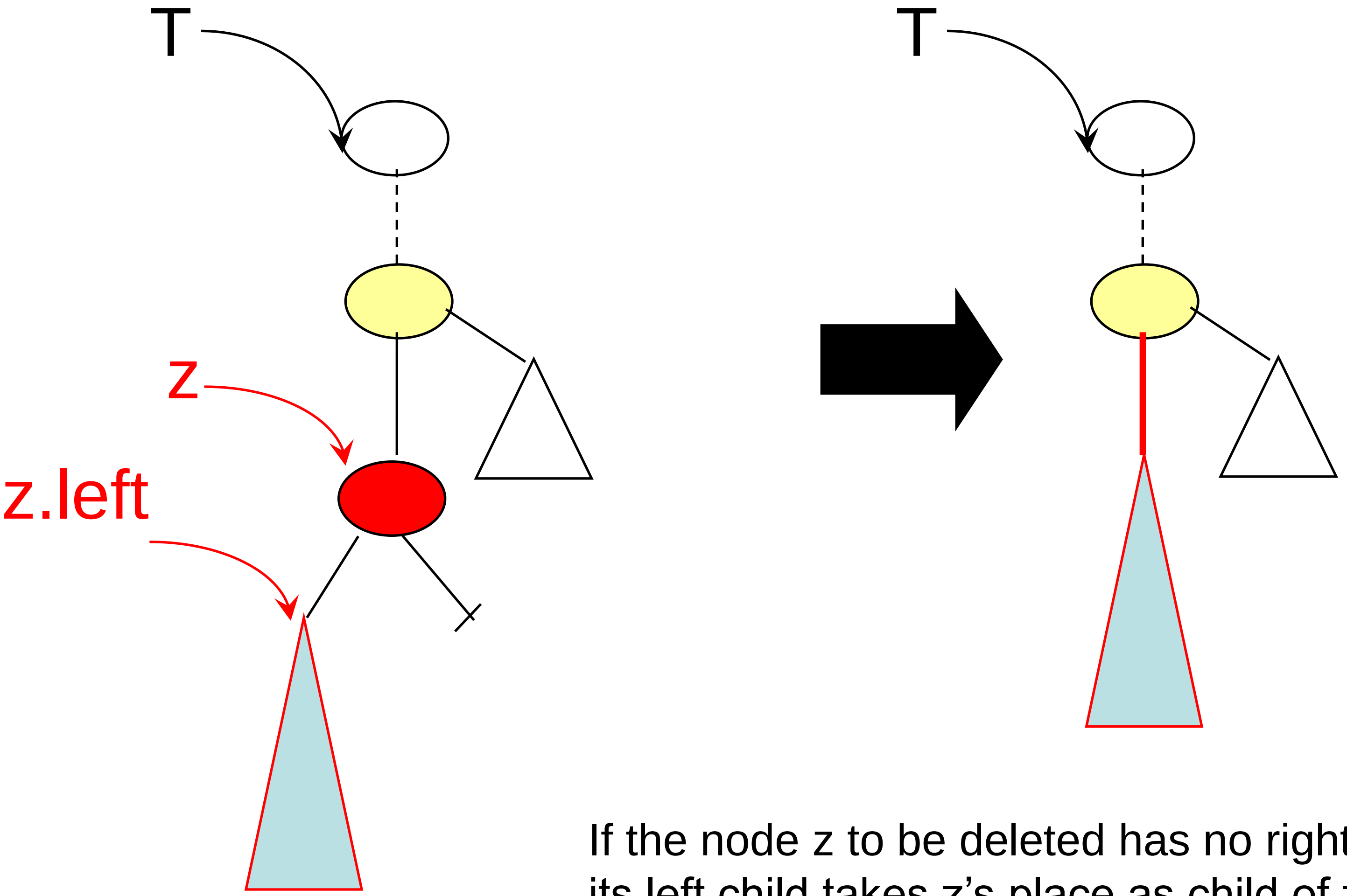
If the node z to be deleted has no left child,
its right child takes z's place as child of z's parent

Example - delete



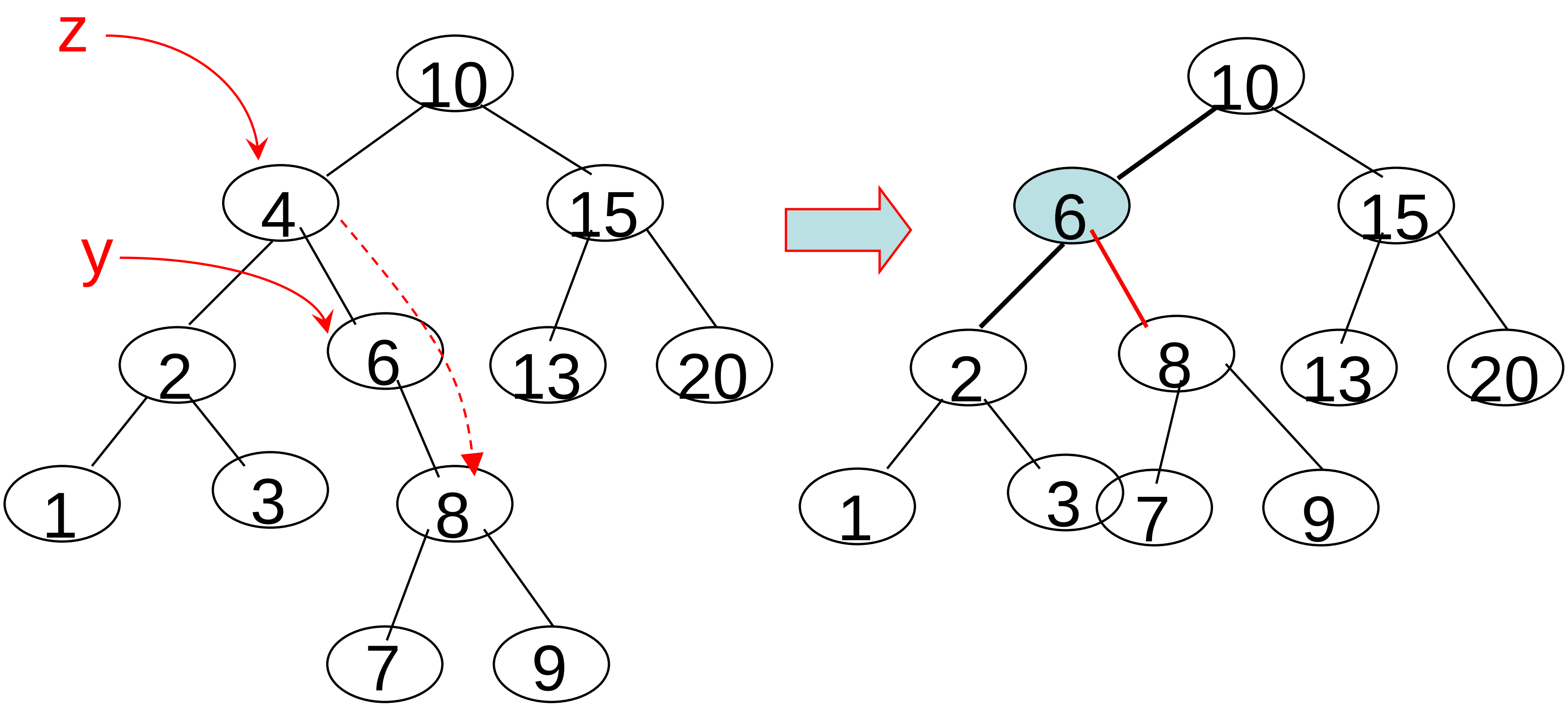
Delete z. Case 1B: z has no right child

Tree delete – case 1B



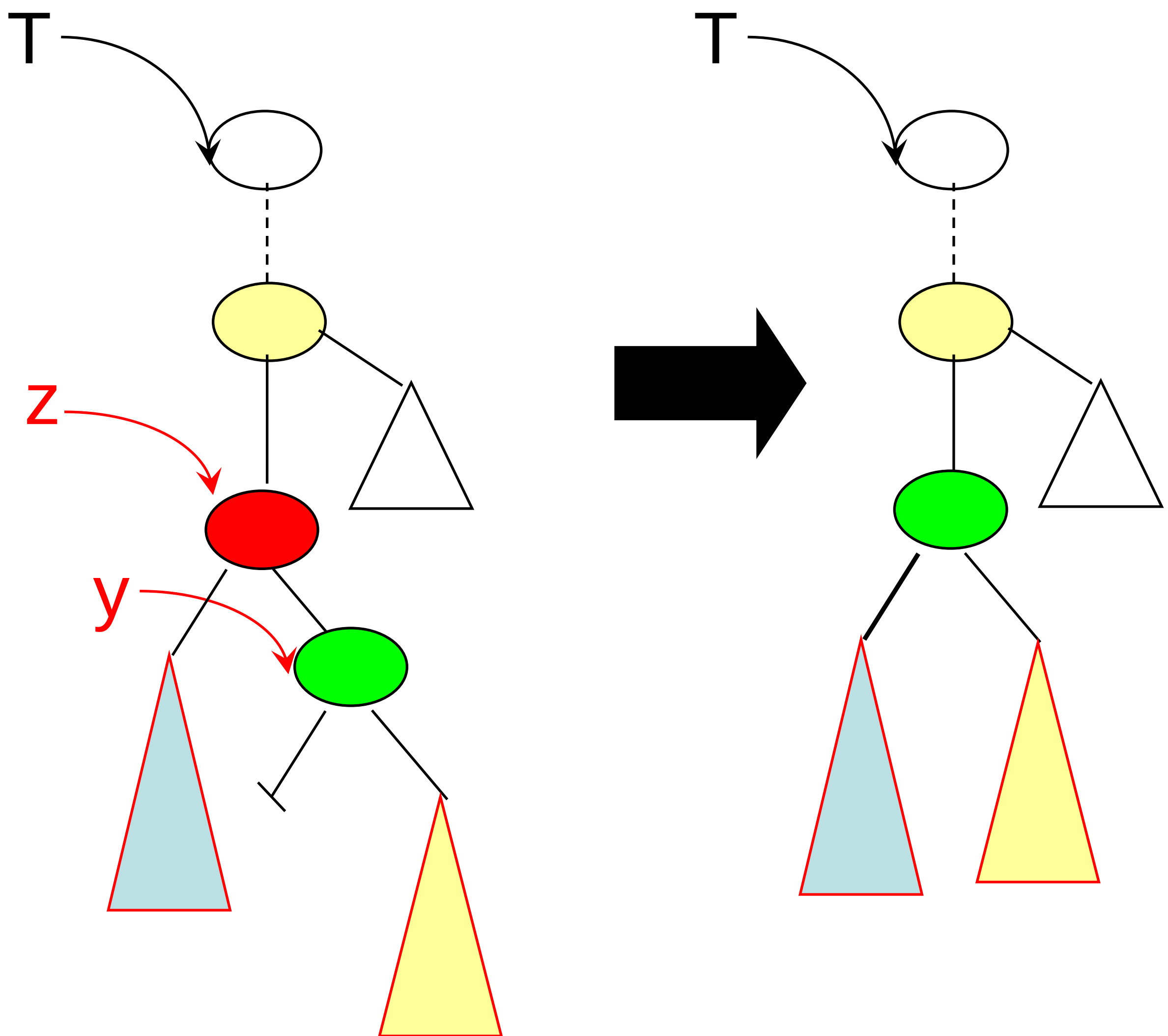
If the node z to be deleted has no right child,
its left child takes z's place as child of z's parent

Example - delete



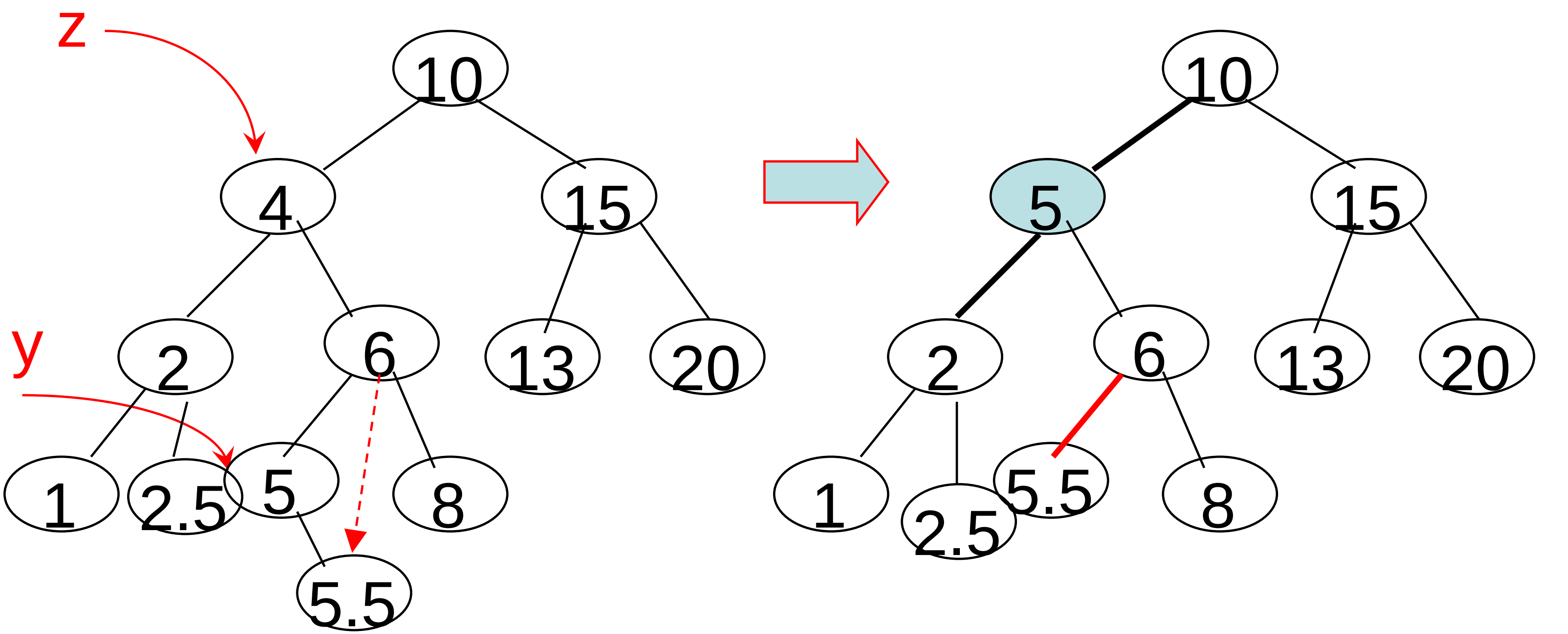
Delete z. Case 2A: z has 2 children and z's successor, y, is the right child of z

Tree delete – case 2A



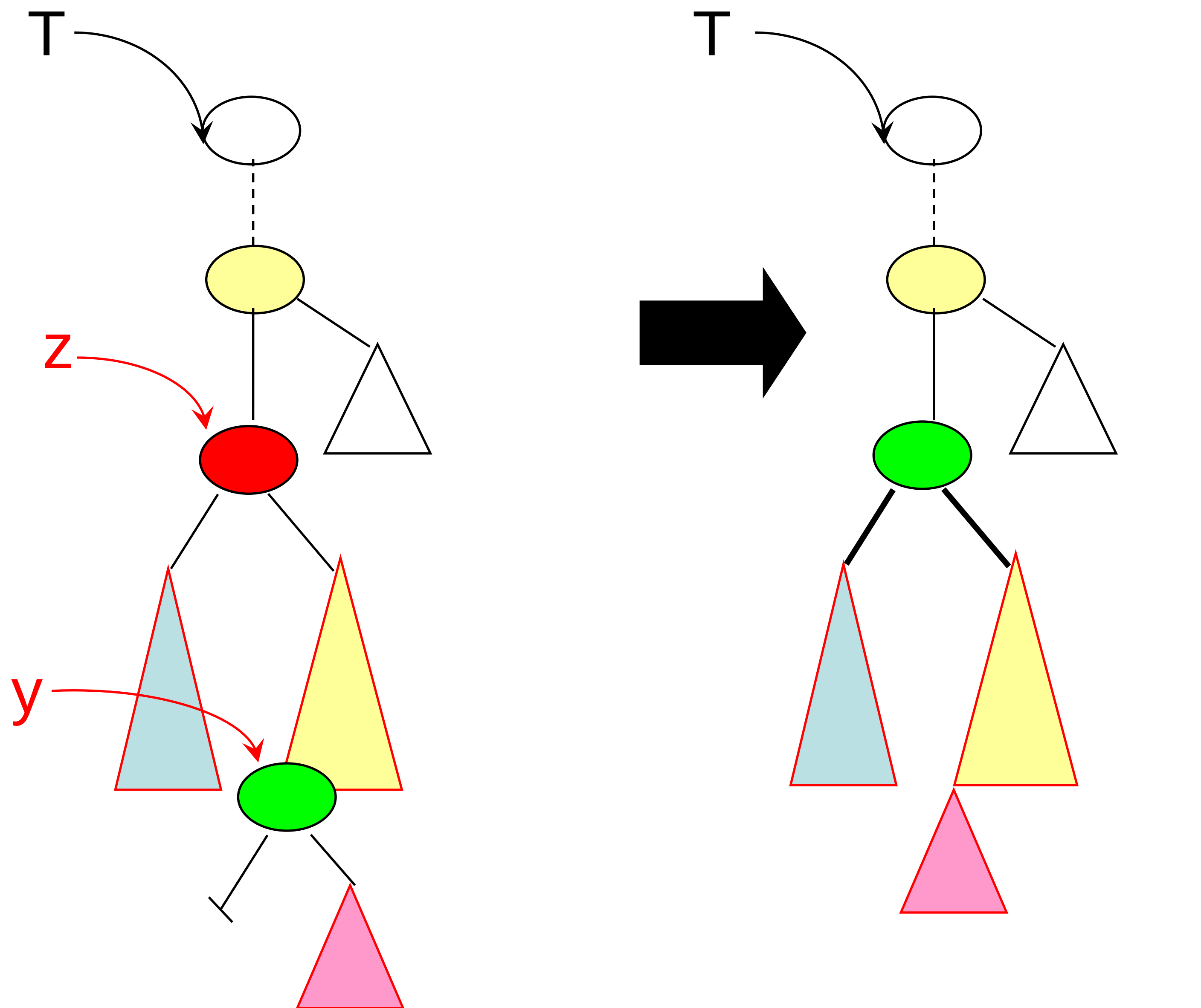
If the node z to be deleted has two children, find minimum y in right subtree of z, delete y from tree, and replace value of z by value of y

Example - delete



Delete z. Case 2B: z has 2 children and z’s successor, y, is not the right child of z

Tree delete – case 2B



If the node z to be deleted has two children, find minimum y in right subtree of z, delete y from tree, and replace value of z by value of y

Delete – recursive implementation

```
public void delete(int key) {
   root = deleteRecursive(root, key);
}

private Node deleteRecursive(Node z, int key) {
   if (z == null) return null;
   if (key < z.key) z.left = deleteRecursive(z.left, key);
   else if (key > z.key) z.right = deleteRecursive(z.right, key);
   else {
      // node z contains the key to be deleted
      if (z.right == null) return z.left;  // case 1: only 1 child left
      if (z.left == null) return z.right;  // case 1: only 1 child right
      //case 2: node z to be deleted has 2 children
      Node y = min(z.right); // find minimum in its right subtree (successor of z)
      z.right = deleteRecursive(z.right, y.key); //delete minimum node - we KNOW it
has max 1 child
      z.key = y.key; //replace current key with minimum  key
   }
   return z;
}
```

BST Delete

- The previous implementation of BST Delete considered replacing the deleted node z with its successor
- Another similar solution is to replace the deleted node z with its predecessor

Exercise

- Draw the Binary Search Tree that results from following sequence of operations:
- Insert 29, 37, 1, 3, 7, 20, 89, 75, 4, 2, 6, 30, 35
- Delete 30, 3

Analysis

- Tree walks $\Theta(n)$
- Queries
  - Search $O(h)$
  - Minimum $O(h)$
  - Maximum $O(h)$
- Modifying
  - Insert $O(h)$
  - Delete $O(h)$

n = number of nodes in BST

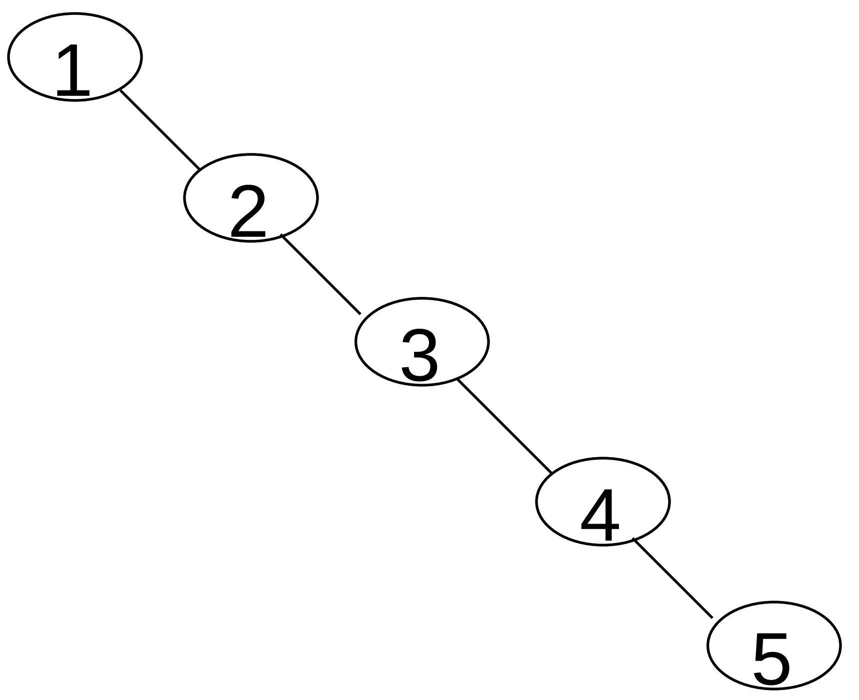
h = height of BST. But what is the value of the height of a BST?

The Height of Binary Search Trees

- Each of the basic operations on a binary search tree runs in O(h) time, where h is the ***height*** of the tree => It is desirable that h is small
- The shape and height of a binary search tree of n nodes depends on the order in which the keys are inserted (insert in order: 1,2,3 vs insert in order 2,1,3)
- The height of a BST with n nodes:
  - Worst case: O(n) => **BST operations are also O(n) !!!**
  - Best case: O(log n)
  - Average case O(log n)

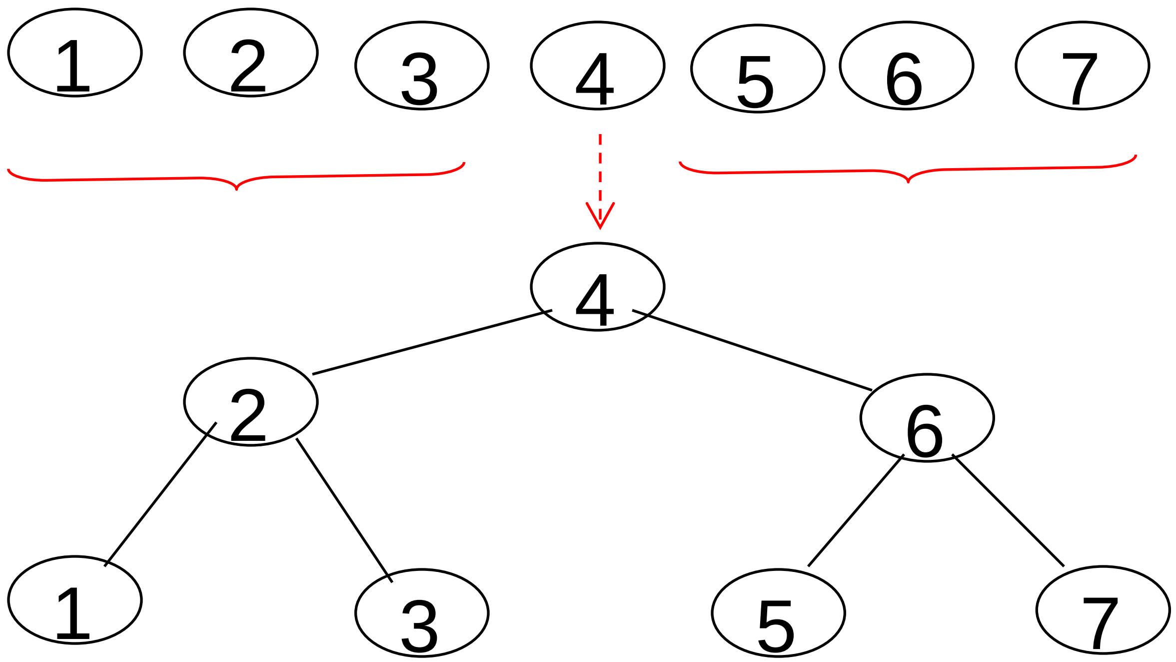
Height of a BST – worst case

- If the n keys are inserted in *strictly increasing* order, the tree will be a chain with height n-1.



Height of a BST – best case

- The best case corresponds to a *balanced* tree
- In this case the height is log n



Height of a BST – random case

- It can be proved that: The *expected* height of a *randomly* built binary search tree on n distinct keys is O (lg n)

Keeping the height of BST small

- Different techniques are used in order to keep the height of BST small – after an insertion or deletion some operations are done in order to redo the balance:
  - AVL trees (Adelson-Velskii and Landis)
  - Red-black trees (symmetric binary B-trees, 2-3 trees)

Implementing a Generic BST

- A Binary Search Tree can hold **any** Key and Value types – it should be a ***generic type (with Key and Value as type parameters)***
- ***Condition:*** The Key type must be a ***Comparable*** !

```java
public class GenericBST<K extends Comparable<K>, V> {

    private class Node {
        K key;          // sorted by Key
        V val;        // associated data
        Node left, right;  // left and right subtrees

        Node(K key, V val) {
            this.key = key;
            this.val = val;
        }
    }

   private Node root;           // root of BST

    public GenericBST() {
        root = null;       // initializes empty BST
    }

// … etc
```

```
public boolean contains(K key) {
    return contains(root, key);
}

private boolean contains(Node x, K key) {
    if (x == null) return false;
    if (key.compareTo(x.key)<0) return contains(x.left, key);
    else if (key.compareTo(x.key)<0)  return contains(x.right, key);
    else return true;
}
```

```java
public class GenericBSTClient {
   public static void main(String[] args) {
      GenericBST<Integer, Integer> bst1 = new GenericBST<>();
      GenericBST<Integer, String> bst2 = new GenericBST<>();
      GenericBST<String, City> bst3 = new GenericBST<>();

      bst1.put(5, 10);
      bst1.put(2, 4);
      bst1.inorder();
      bst2.put(4, “four”);
      bst2.put(2, “two”);
      bst2.put(3, “three”);
      bst2.inorder();
      bst3.put("Timisoara", new City("Timisoara", 230, 300));
      bst3.put("Arad", new City("Arad", 180, 210));
      bst3.put("Brasov", new City("Brasov", 200, 280));
      bst3.inorder();
   }

}
```

Source Code

- IntBST.java
- GenericBST.java, GenericBSTClient.java