Lancaster University

# SCC.121: Fundamentals of Computer Science
# Sorting, Trees and Graphs

## Self-balancing Binary Search Trees

# Today's Lecture

Lancaster University

## Aim:

- Introduce:
  - **Dictionary (Map)** ADT
  - Binary Search Trees (**BST**)
  - **Self-balancing** Search Trees (2-3 trees)

# Searching Problem/Index System

Lancaster University

- Set of **entries**: $\{e_1, \ldots, e_n\}$
- Each entry $e_i$ is a pair $(k_i, v_i)$, where:
  - $k_i$ is a **key value** from some totally ordered domain, say integers or strings,
  - $v_i$ is an associated **data value.**

- We assume keys are **unique.**
- We **do not use data values** to solve the searching problem, but they will be important to the applications that use our data structure.

- **Searching problem:** Given an arbitrary search key $k$, determine if there exists an entry matching this key value and if so, what the associated data value is.

# Example of Index System

Lancaster University

- **DNS lookup**
  - Insert domain name with specified IP address to obtain the set of (domain name, IP address) entries
  - Given domain name, find the corresponding IP address

| Domain name | IP address |
|---|---|
| www.google.com | 142.250.189.164 |
| www.lancs.ac.uk | 148.88.65.80 |
| www.gov.uk | 151.101.40.144 |
| www.wikipedia.com | 198.35.26.98 |

# Examples of Index System

Lancaster University

| application | Purpose of search | key | value |
|---|---|---|---|
| dictionary | Find definition | Word | definition |
| Web search | Find relevant web pages | keyword | List of page names |
| Compiler | Find properties of variables | Variable name | Type and value |
| Routing table | Route internet packets | Destination | Best route |
| DNS | Find IP address | Domain name | IP address |
| Reverse DNS | Find domain name | IP address | Domain name |
| Genomics | Find markers | DNA string | Known positions |
| File system | Find file on disk | Filename | Location on disk |

# Dictionary (Map) ADT

Lancaster University

- **Dictionary** (or map) is an ADT with **operations insert**, **delete** and **find**:
  - **void insert(Key $k$, Value $v$):** Stores an entry with the key-value pair $(k, v)$. (If $k$ is already present in the dictionary, error.)
  - **void delete(Key k):** Deletes the entry associated with key $k$ from the dictionary. (If it does not exist, error.)
  - **Value find(Key k):** Determine whether there is an entry in the dictionary associated with key $k$. If there is, return the associated value. Otherwise, return null reference.

- You can also support **iteration** over the entries, **range queries** (enumerate or count all entries between some minimum and maximum value), **returning $i$th smallest value** as well as **union** and **intersection** (of dictionaries).

# Dictionary (Map) ADT

Lancaster University

- Dictionary (or map) is an ADT that supports operations **insert**, **delete** and **find**:
  - **void insert(Key $k$, Value $v$):** Stores an entry with the key-value pair $(k, v)$. If $k$ is already present in the dictionary, error.
  - **void delete(Key k):** Deletes the entry associated with key $k$ from the dictionary. If it does not exist, error.
  - **Value find(Key k):** Determine whether there is an entry in the dictionary associated with key $k$. If there is, return the associated value. Otherwise, return null reference.

- Common implementations of dictionary ADT use:
  1. Sorted arrays
  2. Search trees
  3. Hashing

# Sequential allocation for Dictionary ADT

Lancaster University

- Most naïve implementation: Store the entries in a linear array without sorting.
  - To find key value, linear search -> $O(n)$ time
  - Insertion only takes $O(1)$ time, if we still have enough space in the array.
    But checking for duplicates -> $O(n)$ time.

- Otherwise, use a sorted (by key value) array.
  - To find key value, binary search -> $O(\log n)$ time.

For $n = 10^9$ entries,
$\log_2 n \leq 30$

# Sequential allocation for Dictionary ADT

Lancaster University

- Most naïve implementation: Store the entries in a linear array without sorting.
  - To find key value, linear search -> $O(n)$ time
  - Insertion only takes $O(1)$ time, if we still have enough space in the array.
    But checking for duplicates -> $O(n)$ time.

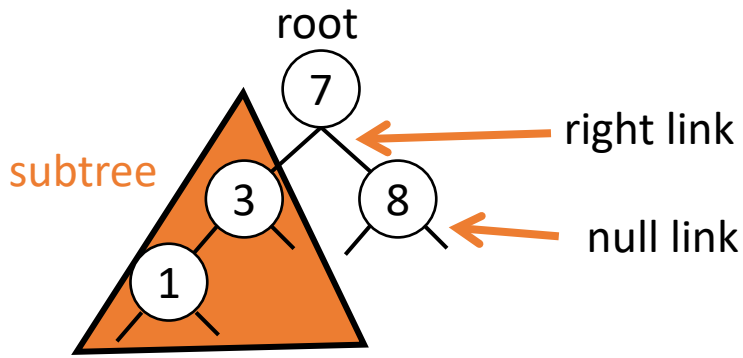
- Otherwise, use a sorted (by key value) array.
  - To find key value, binary search -> $O(\log n)$ time.
  - **But updates (insertion and deletion) take $O(n)$ time as elements in array must be moved around to make space.**

For $n = 10^9$ entries, $\log_2 n \leq 30$

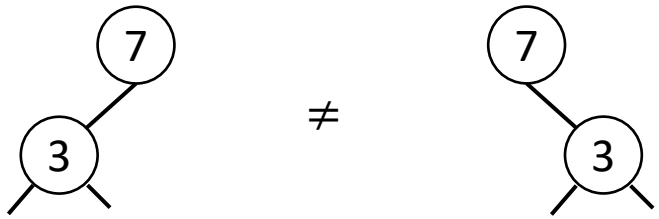
# Binary Search Tree

Lancaster University

- Binary tree is a tree with at most two children per node: **left** child and **right** child




- Unlike the Tree ADT in the previous lecture, the following two binary trees are ≠

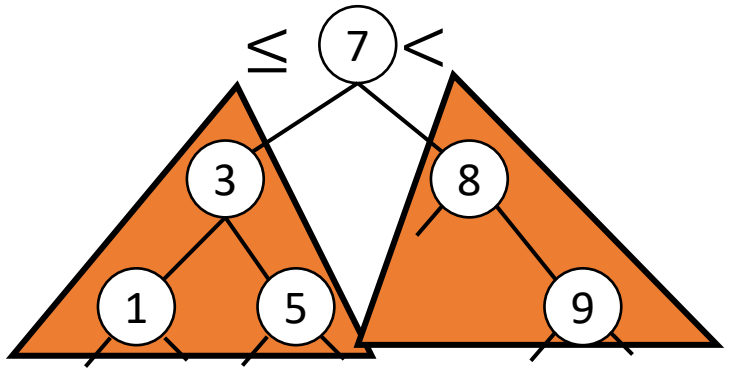


# Binary Search Tree

Lancaster University

- Binary search tree =
  1. Binary tree,
  2. Each node has a key-value pair,
  3. Nodes are “sorted” by their keys.

- **Sorted =** For every node, its key is:
  - Greater or equal to all keys in its left subtree,
  - Strictly smaller than all keys in its right subtree



# BST ADT: Setting up

Lancaster University

```
public class BST<Key extends Comparable<Key>,Value>{
    private Node root;

    private class Node{
        private Key key;
        private Value value;
        private Node leftChild, rightChild;

        public Node(Key key, Value value){
            this.key = key;
            this.value = value;
        }
    }
    …
}
```

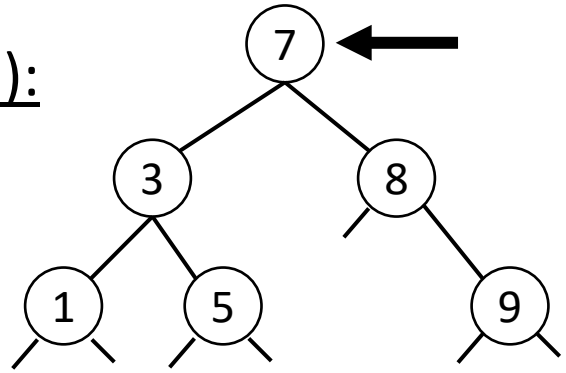
# BST ADT: Implementing Find

Lancaster University

**Leverage the sortedness property:**

1. If root has key, return root,
2. Else if key smaller, go left,
3. Else, go right.

Find(9):

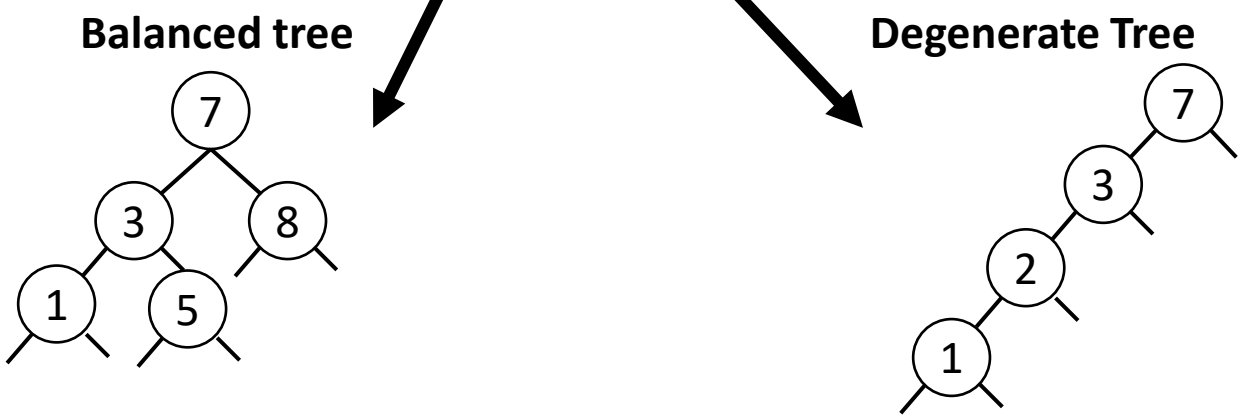



```
public Value find(Key key){
    Node x = root;
    while (x != null){
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x = x.leftChild;
        // cmp < 0 is equivalent to key < x.key
        else if (cmp > 0) x = x.rightChild;
        else return x.value; //cmp == 0
    }
    return null;
}
```

# BST ADT: Time Complexity of Find

Lancaster University

- For a BST of height $h$, the worst-case complexity of find() is $O(h)$.
- Height of BST = $\Omega(\log n)$ and $O(n)$




- BST is very efficient on balanced trees, but inefficient on degenerate trees

# BST ADT: Implementing Insertion

Lancaster University

**Need to maintain sortedness:**

1. If x is null, insert new node
2. If x has key, give error,
3. Else if key < x's key, recurse left
4. Else recurse right

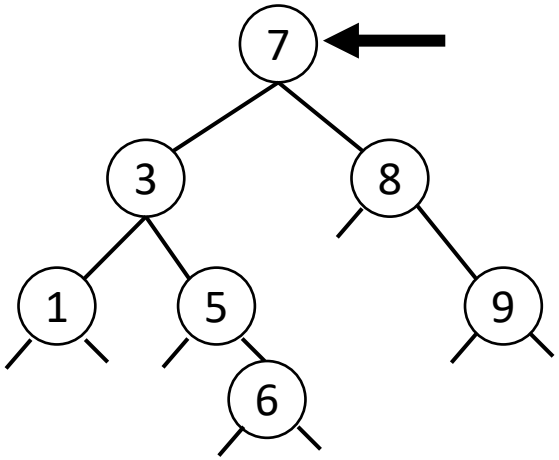
```
public void insert(Key key, Value value){
    root = insert(root, key, value);
}

private Node insert(Node x, Key key, Value value){
    if (x == null) return new Node(key, value);
    int cmp = key.compareTo(x.key);
    if (cmp < 0){
        x.leftChild = insert(x.leftChild, key, value);
    }
    else if (cmp > 0){
        x.rightChild = insert(x.rightChild, key, value);
    }
    else System.out.println("Insert Exception");
        //x.value = value;
    return x;
}
```

# BST ADT: Implementing Insertion

Lancaster University

- Example: **insert(6, 'Jane')**

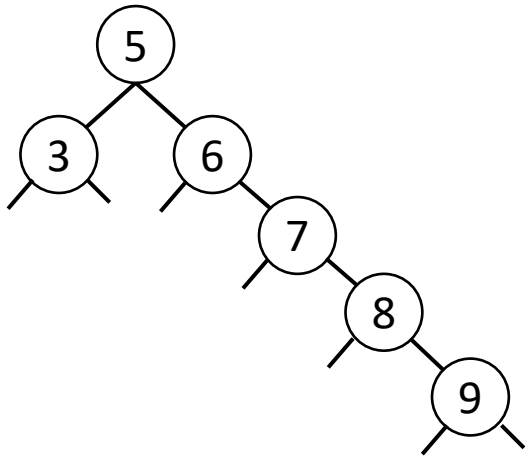


# BST ADT: Insertion Order Matters

Lancaster University

You can have multiple BSTs with the same key-value pairs:

1. insert(5,*)
2. insert(3,*)
3. insert(6,*)
4. insert(7,*)
5. insert(8,*)
6. insert(9,*)



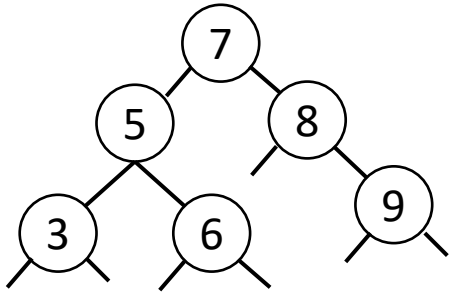

**Degenerate Tree**

# BST ADT: Insertion Order Matters

Lancaster University

You can have multiple BSTs with the same key-value pairs:

1. insert(7,*)
2. insert(5,*)
3. insert(3,*)
4. insert(6,*)
5. insert(8,*)
6. insert(9,*)




**Balanced Tree**

# BST ADT: Time Complexity of Insertion

Lancaster University

Time Complexity:

- For a BST of height $h$, worst-case time complexity = $O(h)$
- Depends on the insertion order

How can we avoid inefficient search?

1. If **insertion order is (key) random**, then (expected) BST height is $O(\log n)$,

   **Average-case time complexity: $\boldsymbol{O(\log n)}$**

2. Otherwise, implement more complex insertion operations to "**balance**" BST height at $O(\log n)$ **for all possible insertion orders**

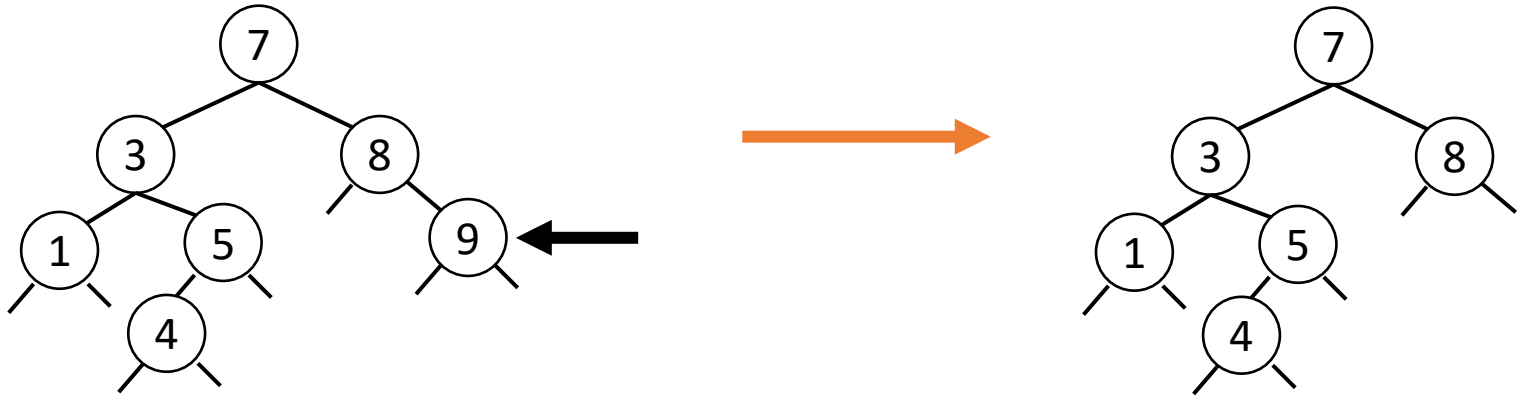
   **Self-Balancing BSTs**

# BST ADT: Implementing Deletion

Lancaster University

Intuition and key steps:

1. **“Find”** the node with the value you want to delete
2. Then, there are **three possible cases**:
   a) Node is **leaf**

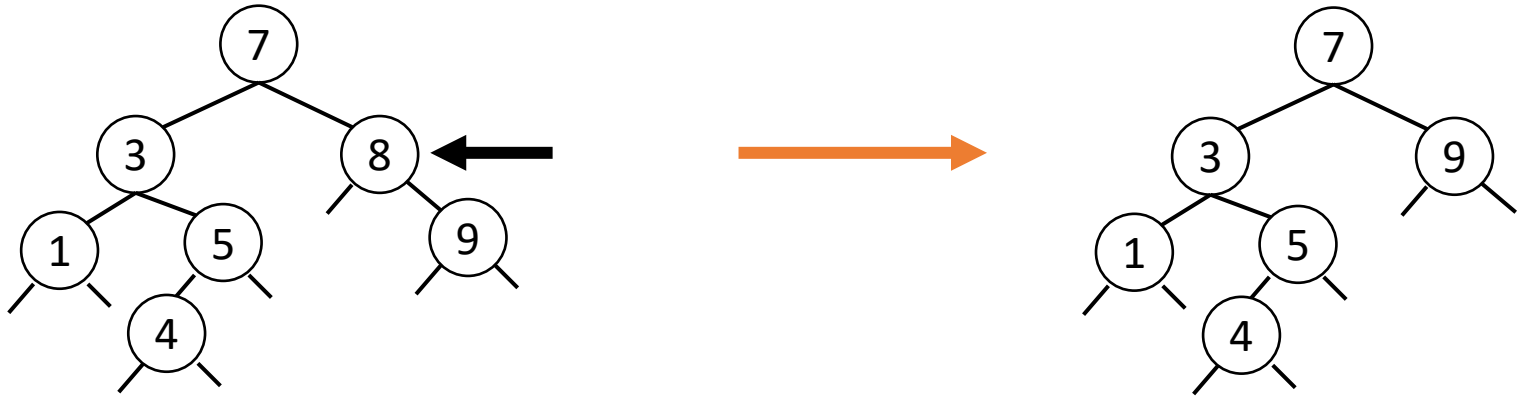


# BST ADT: Implementing Deletion

Lancaster University

Intuition and key steps:

1. **“Find”** the node with the value you want to delete
2. Then, there are **three possible cases**:
   a) Node is **leaf**
   b) Node has a **single child**



# BST ADT: Implementing Deletion

Lancaster University

Intuition and key steps:

1. **"Find"** the node with the value you want to delete
2. Then, there are **three possible cases**:
   a) Node is **leaf**
   b) Node has a **single child**
   c) Node has **both children**

- Find "successor" of 3,
- Use it to replace
- Ensures:
  - keys in left subtree remain smaller,
  - keys in right subtree greater.

# Binary Search Tree: Code for delete()

Lancaster University

Successor of a node with value = node in tree with value just above that

```
private Node successor(Node x){
    if (x == null || x.rightChild == null) return null;
    else{
        x = x.rightChild;
        while(x.leftChild != null) x = x.leftChild;
        return x;
    }
}
```

# Binary Search Tree: Code for delete()

Lancaster University

```
public void deleteMin(){
    root = deleteMin(root);
}

private Node deleteMin(Node x){
    if (x.leftChild == null) return x.rightChild;
    else {
        x.leftChild = deleteMin(x.leftChild);
        return x;
    }
}

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

# Binary Search Tree: Code for delete()

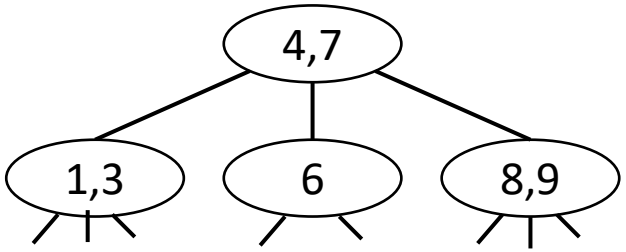
Lancaster University

```
private Node delete(Node x, Key key){
    if (x == null) return null;
    int cmp = key.compareTo(x.key);  // cmp = sign(key - x.key) for integers
    if (cmp < 0) x.leftChild = delete(x.leftChild, key);
    else if (cmp > 0) x.rightChild = delete(x.rightChild, key);
    else{ // if we need to delete node x
        if (x.rightChild == null) return x.leftChild;
        else if (x.leftChild == null) return x.rightChild;
        else{
            Node t = x;
            x = successor(t);
            x.rightChild = deleteMin(t.rightChild);
            x.leftChild = t.leftChild;
        }
    }
    return x;
}
```

# Beyond Simple Binary Search Trees

Lancaster University

- All operations on a BST of height $h$ have a worst-case time complexity of $O(h)$.
- But that will also hold for non-binary search trees.
  - Where you have $i \leq max$ keys per tree node and $i + 1$ children.
  - The first two subtrees contain, respectively, nodes whose keys are smaller and greater than the first key.
  - The second and third subtree …. whose keys are smaller and greater than the second key.




- To make these operations more efficient, use **self-balancing search trees**

# Self-Balancing Search Trees

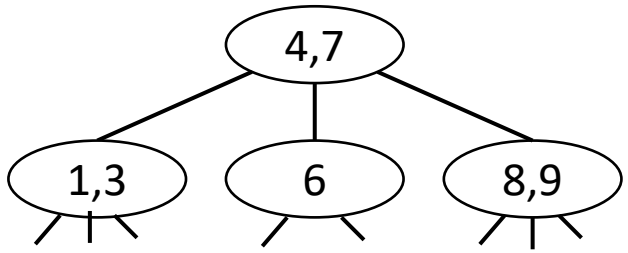
Lancaster University

- There are many options, with diverse tradeoffs
  - **2-3 trees**: tertiary self-balancing search trees
  - AVL trees: self-balancing BST
  - Red-black trees are a (BST) generalization of 2-3 trees
  - B-trees: generalization of 2-3 trees to more keys per node
- Some are **self-balancing but not height-balanced**:
  - Splay Trees
  - Treaps
- For example, you may want your search tree to self-balance for **common letter (key) patterns** rather than for any letter (key) sequence.

# 2-3 Trees

Lancaster University

- 2-nodes: 1 key, 2 children
- 3-nodes: 2 keys, 3 children




- (Natural variant of) Inorder traversal gives keys in ascending order

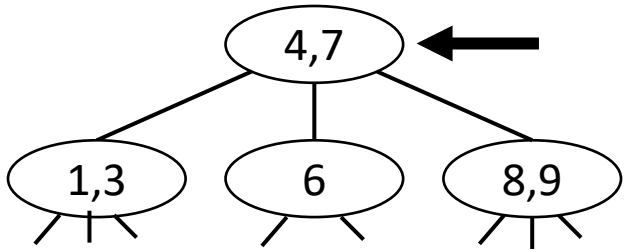
- **Perfectly balanced tree:**
  Every path from the root to a leaf has the same length

- How do we maintain this property despite insertions and deletions?

# 2-3 Trees: Find

Lancaster University

- Works (almost) just as in BSTs,
- Example: **find(8)**

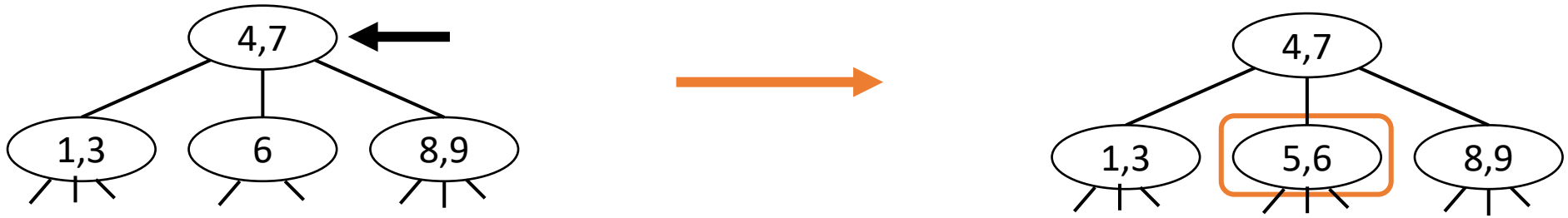


# 2-3 Trees: Insertion

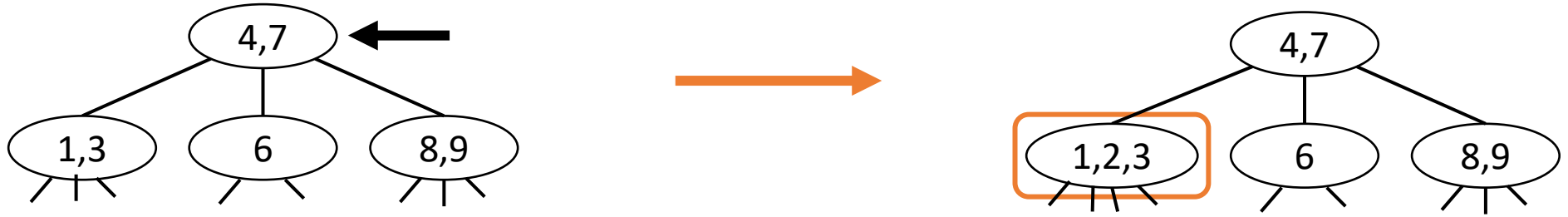
Lancaster University

Slightly more complex than in BSTs. **Three cases:**

1. **insert(5)**



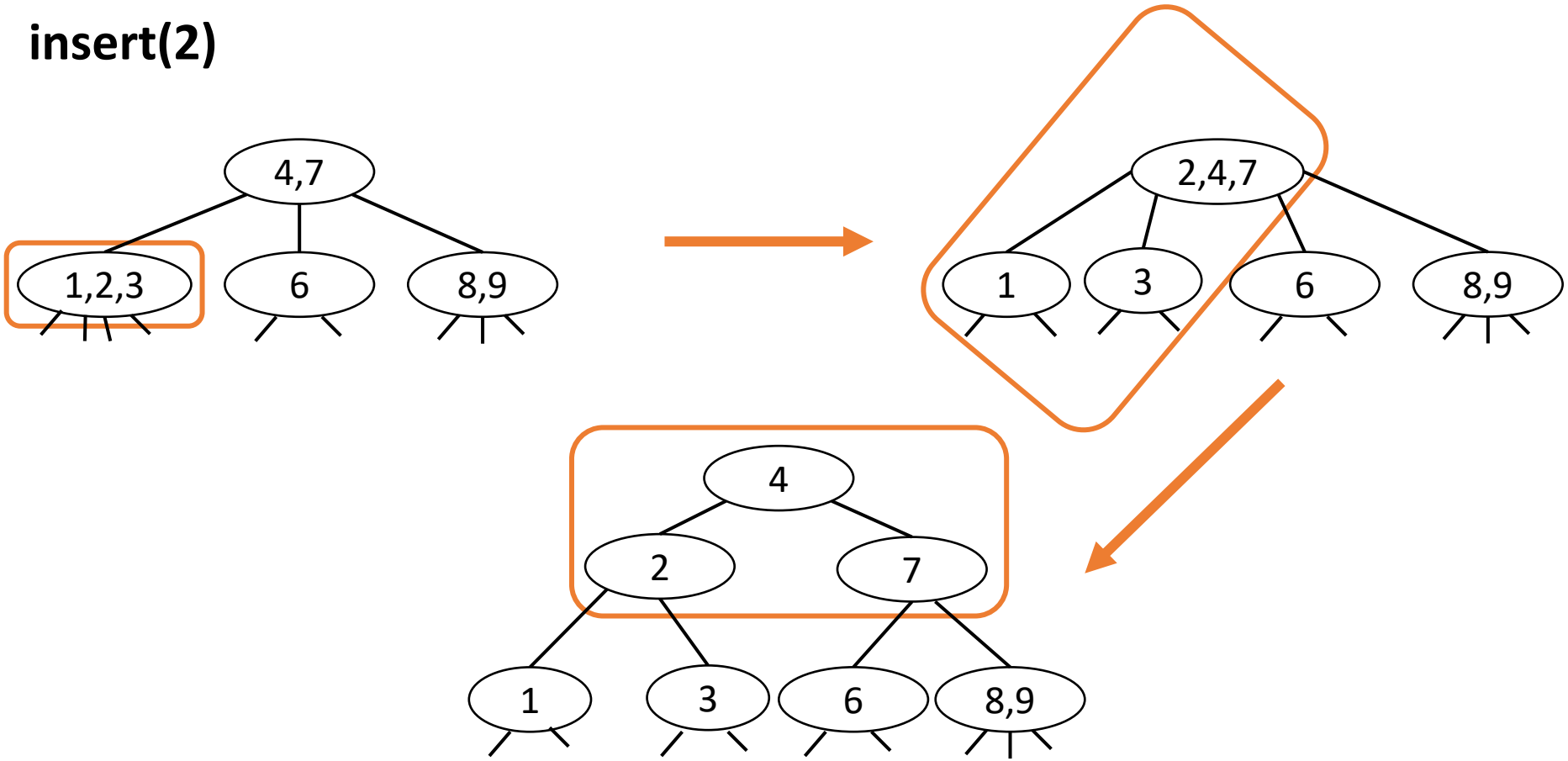

2. **insert(2)**



# 2-3 Trees: Insertion

Lancaster University

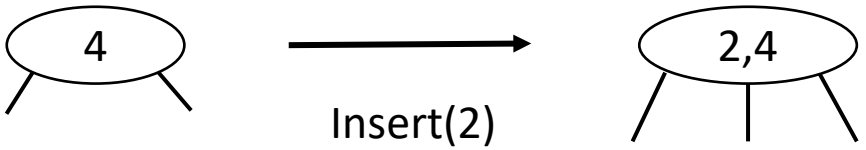
3. **insert(2)**



# 2-3 Trees: Insertion

Lancaster University

- Why does insertion maintain perfect balance?
  - Either a 2-node is transformed into a 3-node,



# 2-3 Trees: Insertion

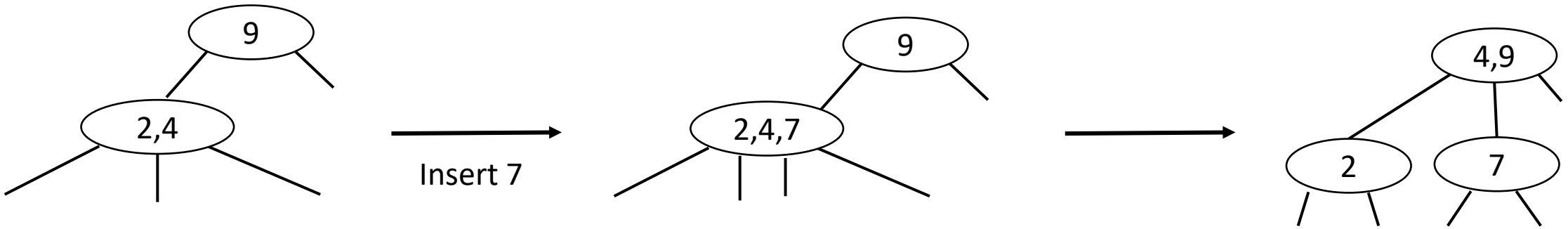
Lancaster University

- Why does insertion maintain perfect balance?
  - Either a 2-node is transformed into a 3-node,
  - Or a 3-node is split into three 2-nodes,

# 2-3 Trees: Insertion

Lancaster University

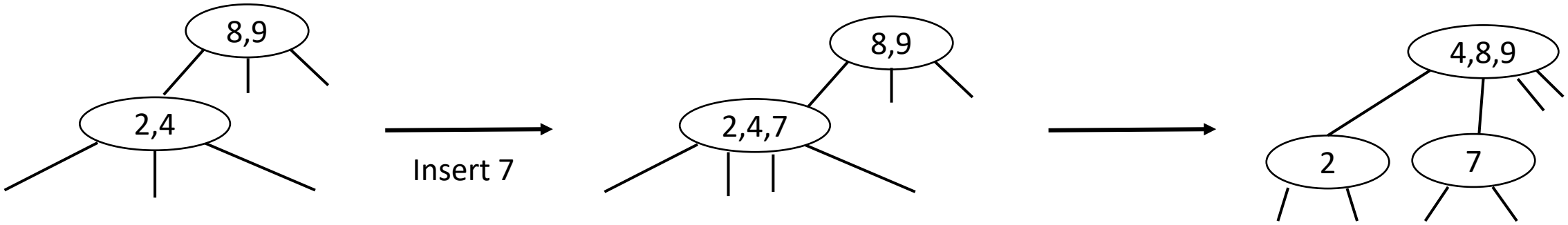
- Why does insertion maintain perfect balance?
  - Either a 2-node is transformed into a 3-node,
  - Or a 3-node is split into three 2-nodes (root),
  - Or a (non-root) 3-node with a 2-node parent transforms into a temporary 4-node,



# 2-3 Trees: Insertion

Lancaster University

- Why does insertion maintain perfect balance?
  - Either a 2-node is transformed into a 3-node,
  - Or a 3-node is split into three 2-nodes (root),
  - Or a (non-root) 3-node with a 2-node parent transforms into a temporary 4-node,
  - Or a (non-root) 3-node with a 3-node parent transforms into a temporary 4-node,

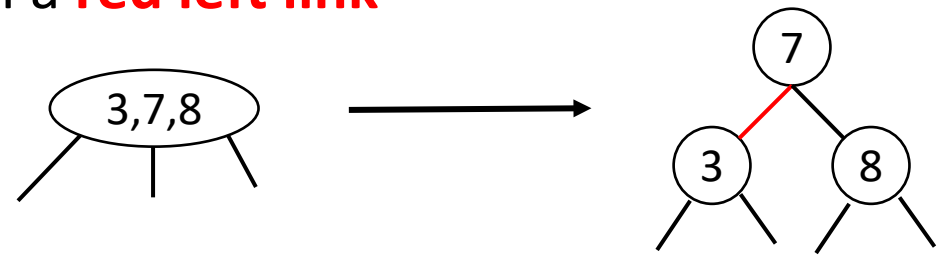


# 2-3 Trees: Cons

Lancaster University

- However, 2-3 trees are quite inconvenient to implement.
  1. Different types of nodes (2-nodes and 3-nodes)
  2. We need to do multiple compares at each node,
  3. We need to move back up the tree to split 4-nodes
  4. Large number of cases for that splitting.

- Red-black trees are BSTs that build upon 2-3 trees, representing 3-node as a binary tree node with a **red left link**



# Summary

Lancaster University

**Today's lecture:**

Introduced:

- Dictionary (Map) ADT,
- Binary Search Trees
- And Self-balancing Search Trees (2-3 trees)

- **Next Lecture:** Graphs and Shortest Paths
- **Any questions?**