2: Trees

Trees

A tree is a directed graph with the property

> There is one node (the root) from which *all* other nodes can be reached by *exactly one path*.
>
> Seen lots of examples.

- Parse Trees
- Decision Trees
- Search Trees
- Family Trees
- Hierarchical Structures
  - Management
  - Directories

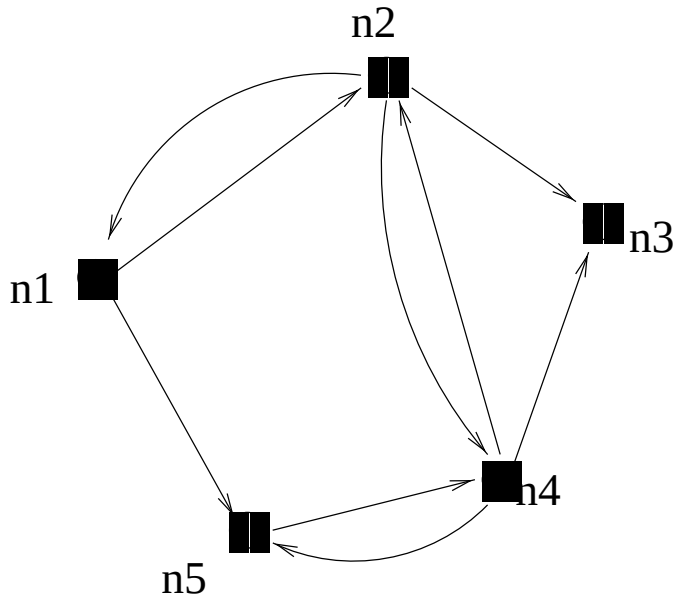
Trees have natural recursive structure

Any node in tree has number of *children* each of which is a tree.

Descriptions of Graphs and Trees

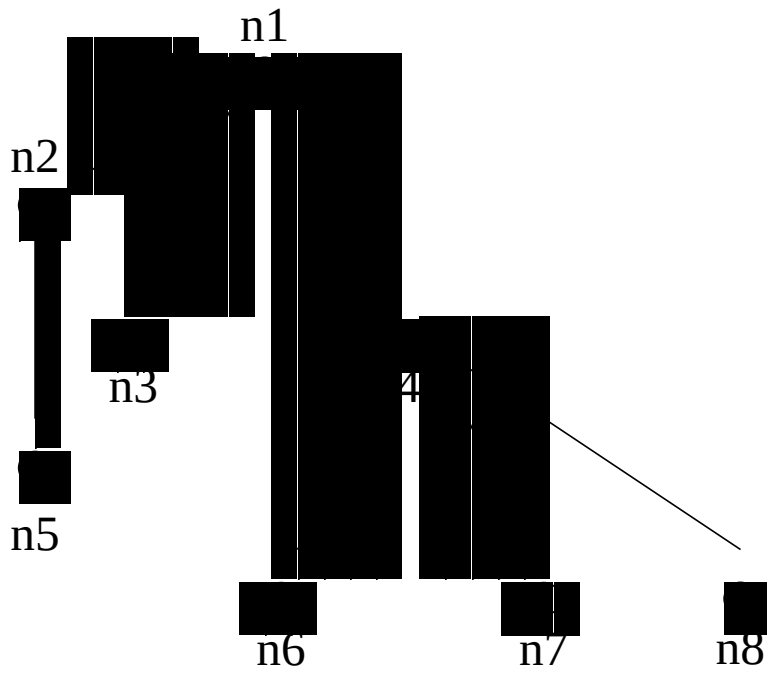
A *directed graph* is a pair (N, E) consisting of a set of nodes N, together with a relation E on N. There is no restriction on the relation E.

$a\,E\,b$ iff there is an edge from a to b



A tree is a graph (N, P) (where the relation P is called *has parent*), with the following property

For any node n, there is at most one node n' with $n\,P\,n'$.



- If there is no node n' with $n\,P\,n'$, n is called a *root* node.
- A tree with a single root node is called a *rooted tree*. Often the word tree is used to mean rooted tree, and the more general collection is known as a *forest* of trees.
- For any node n, the set $\{n' \mid n'\,P\,n\}$ is called the set of *children* of n.
- If a node n has no children it is called a *leaf*

Not difficult to see that this is equivalent to the more normal recursive definition of a rooted tree

Height and Depth

For any node n in a tree the *depth* of n is the length of the path from the root to n (so the root has depth 0)

For any node n in a tree the *height* of n is the length of the *longest* path from n to a leaf (so all leaves have height 0)

The height of a tree is the height of its root.

Representations of Trees

In this section we look at different ways in which rooted trees can be represented in a programming language

Have seen both *SML* and *C* representations of *binary trees*.

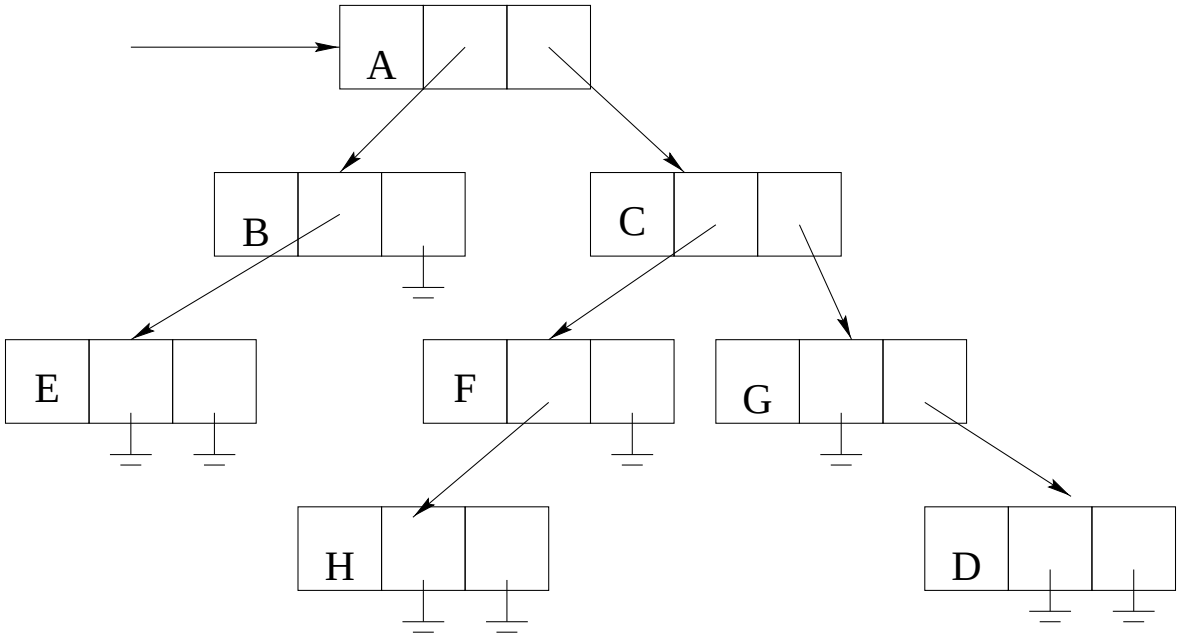
```
datatype 'a TREE = Empty
    | Node of ('a TREE * 'a * 'a TREE)
```

and in C

```
typedef struct TreeNode *PtrToNode;

struct TreeNode {
  ElementType element;
  PtrToNode left;
  PtrToNode right;
};

typedef PtrToNode Tree;
```


A
B
C
E
F
G
H
D

Non-binary trees are almost as simple

```
datatype 'a TREE = Empty
    | Node of ('a * 'a TREE list)
```

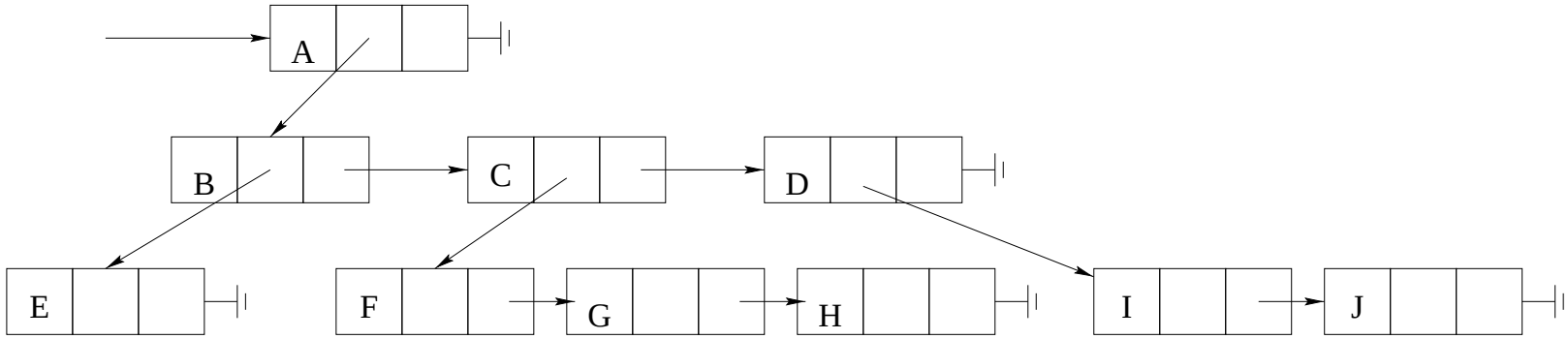
and

```
typedef struct TreeNode *PtrToNode;

struct TreeNode {
  ElementType element;
  PtrToNode FirstChild;
  PtrToNode NextSibling;
};

typedef PtrToNode Tree;
```

This looks almost the same as the binary tree representation, but is interpreted quite differently



Traversing Trees

Can list the nodes of a tree in one of several orders

- Preorder: List the node, then recursively list all children subtrees
- Postorder: Recursively list all children subtrees, then list the node
- Inorder: Only suitable for binary trees. List left subtree, node, then right subtree

Exercise: Write preorder and postorder listing functions for both the binary and n-ary trees.

A Pointer-Free Representation

Suppose the nodes of a tree have names $1 \ldots n$ (or something that we can conveniently map to $1 \ldots n$).

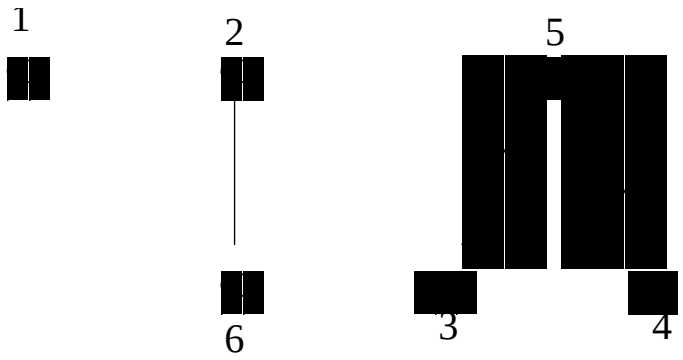
We can represent a tree (or even a forest of trees) with these nodes by use of single array.

The array element `a[i]` should contain the parent of the node `i`, or if `i` is a root node, `i` itself.

So the array

index	1	2	3	4	5	6
contents	1	2	5	5	5	2

Represents the forest of trees



Note that there is no restriction here on the amount of branching in the tree, since we give the parent relation directly, and not the children.

How would we find the *first child* of a node or *next sibling* in this context?

This representation is very useful for representing a *partition* of the set $1 \ldots n$.

A *partition* of a set X is a set of subsets X_i with the properties

- The union of all the sets X_i is X i.e. $\bigcup X_i = X$
- All the sets X_i are pairwise disjoint i.e. $\forall i, j \cdot X_i \cap X_j = \emptyset$

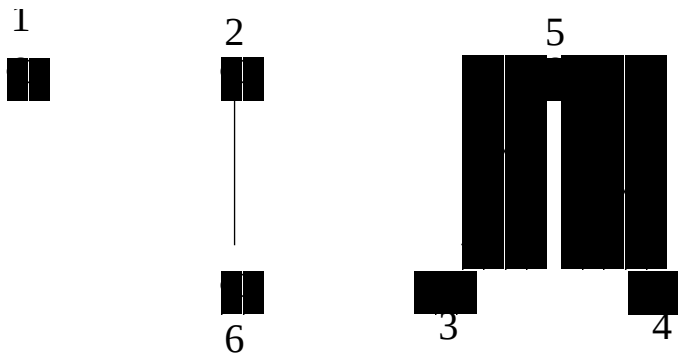
So $\{\{1,3\},\{2\},\{4\}\}$ is a partition of the set $\{1,2,3,4\}$.

A simple way of representing a partition is by using a *forest of trees*

For example

$$X = \{1, 2, 3, 4, 5, 6\}$$

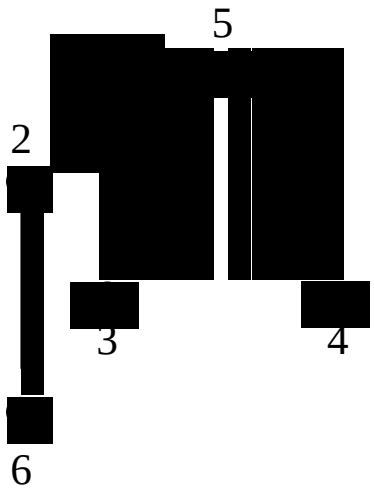
The partition $\{\{1\}, \{2, 6\}, \{3, 4, 5\}\}$ can be represented by the forest



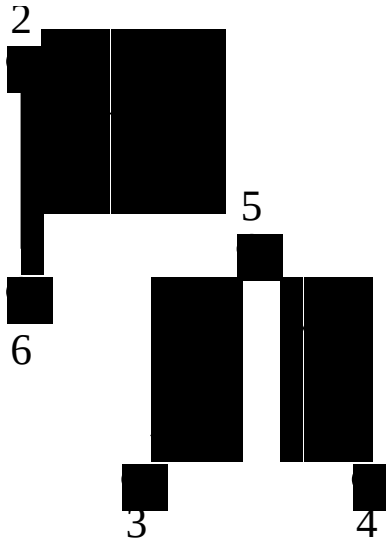
To determine whether 2 elements are in the same member of the partition, just find the *root* of the trees they are in.

To combine two elements of the partition, just ‘graft’ the trees together, by making the root of one tree the parent of the root of the other.

e.g. to combine $\{2,6\}$ and $\{3,4,5\}$



or

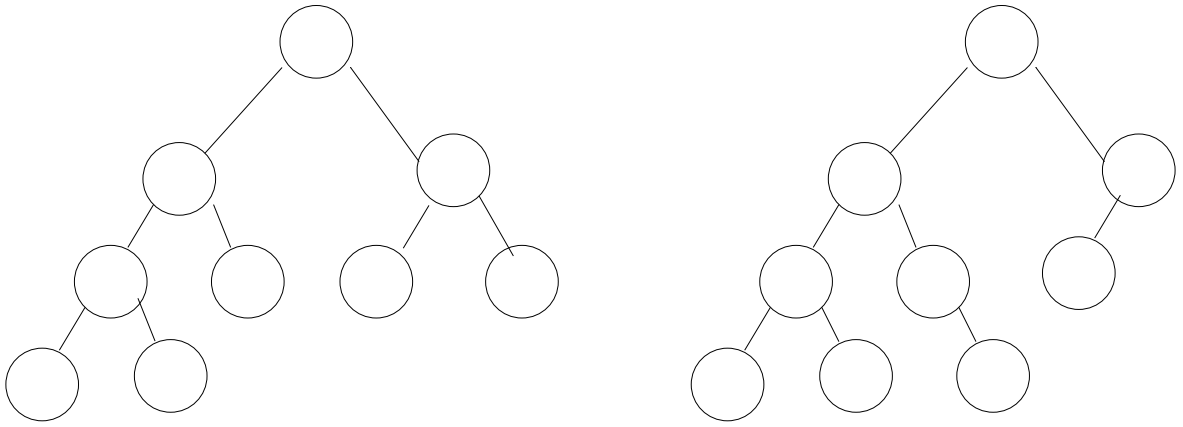


Can optimise the tree by flattening

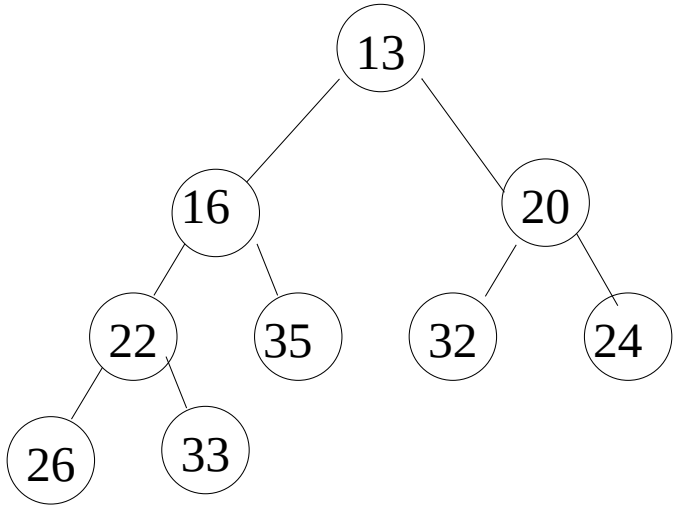
Binary Heaps

This is another type of tree with a pointer-free representation.

Recall from tutorial 3 that an *essentially complete* binary tree is one in which all nodes have *exactly* two children, except possibly those at the lowest level, which is filled from left to right.



A binary tree has the *heap* property if the value of the key at *any* node is *less than or equal to* the values of all the keys of its children.



Sometimes ‘greater than or equal to’ is used instead.

A *binary heap* is a binary tree which has the *type invariant*

- it is essentially complete
- it has the heap property

The smallest element in a heap is always at its root.

All operations on a heap *must* preserve the type invariant

The above heap (or any complete binary tree) can be stored in an array as follows

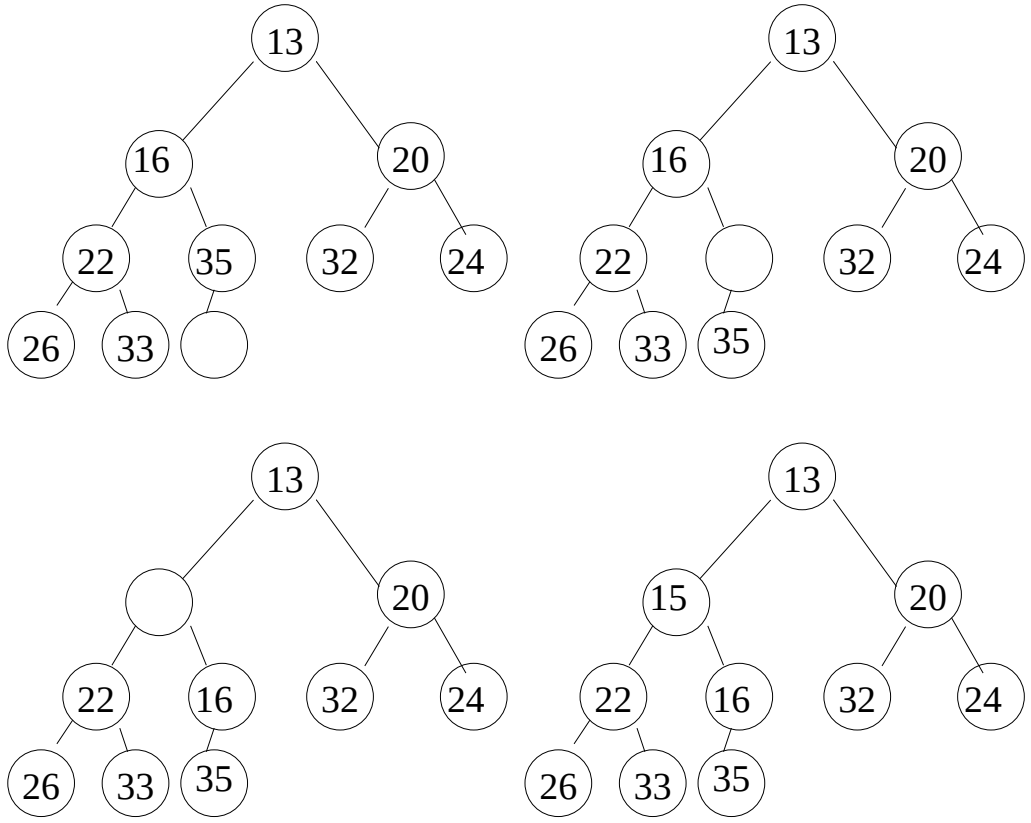
index	0	1	2	3	4	5	6	7	8	9	10
value		13	16	20	22	35	32	24	26	33	

In general, the root is stored in slot 1, and the children of the element at position `i` can be found at positions `2i` and `2i + 1`

To insert into a heap just insert after the last element (as long as there is room.)

This can destroy the heap property, so then need to repair the heap to reestablish the invariant.

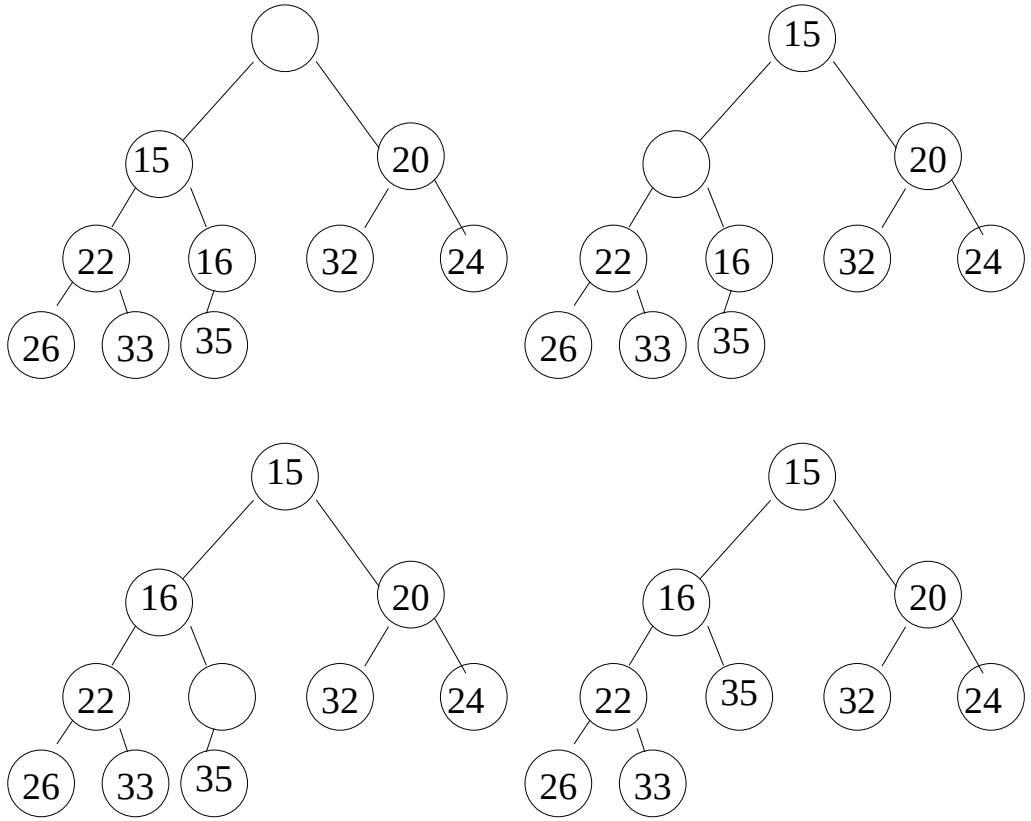
For example to insert 15 into the heap above



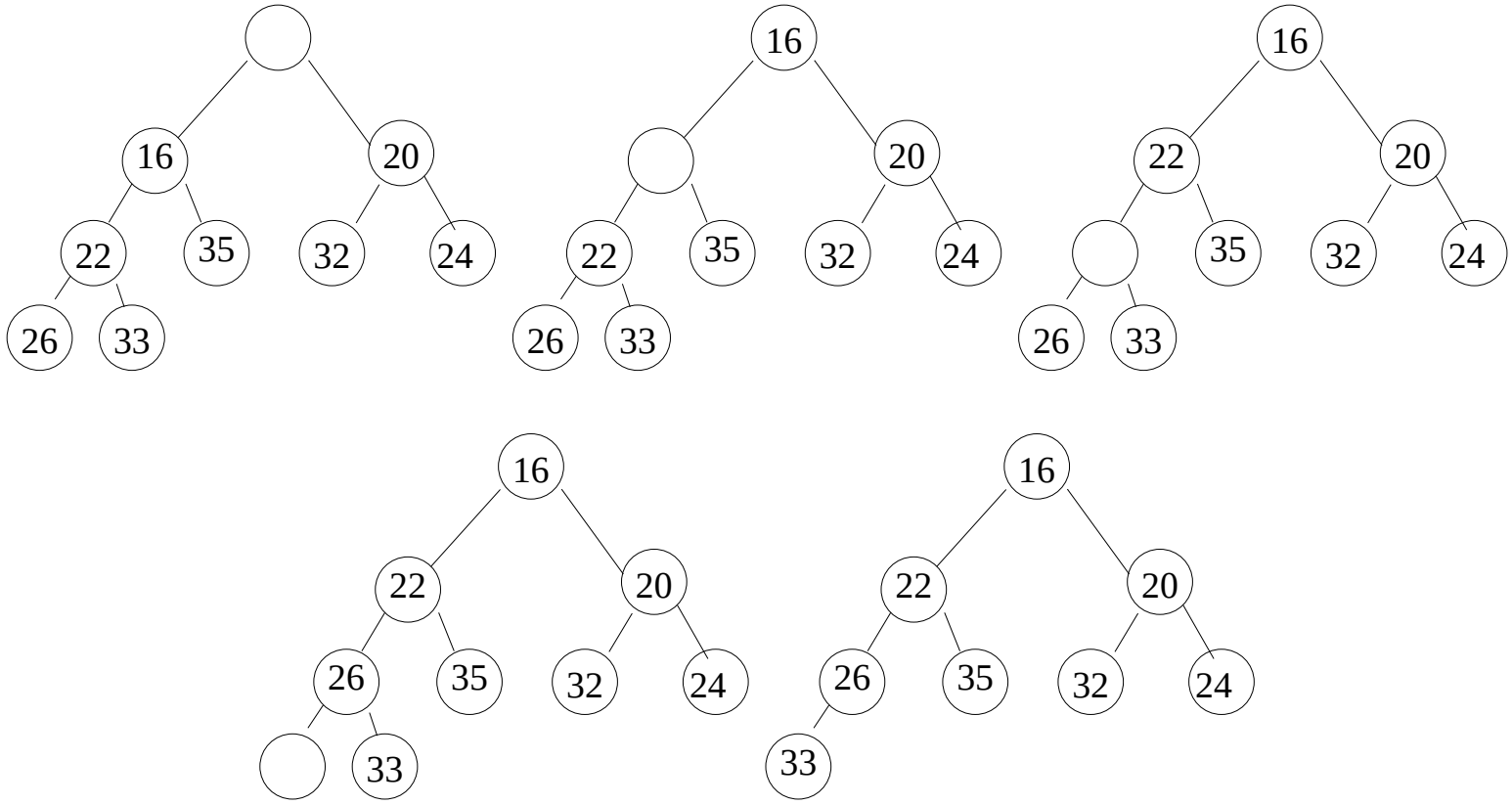
The major use of heaps is as *priority queues*, so a general delete is not usually needed.

Usual form of delete is `deleteMin`, which removes the smallest element from the heap

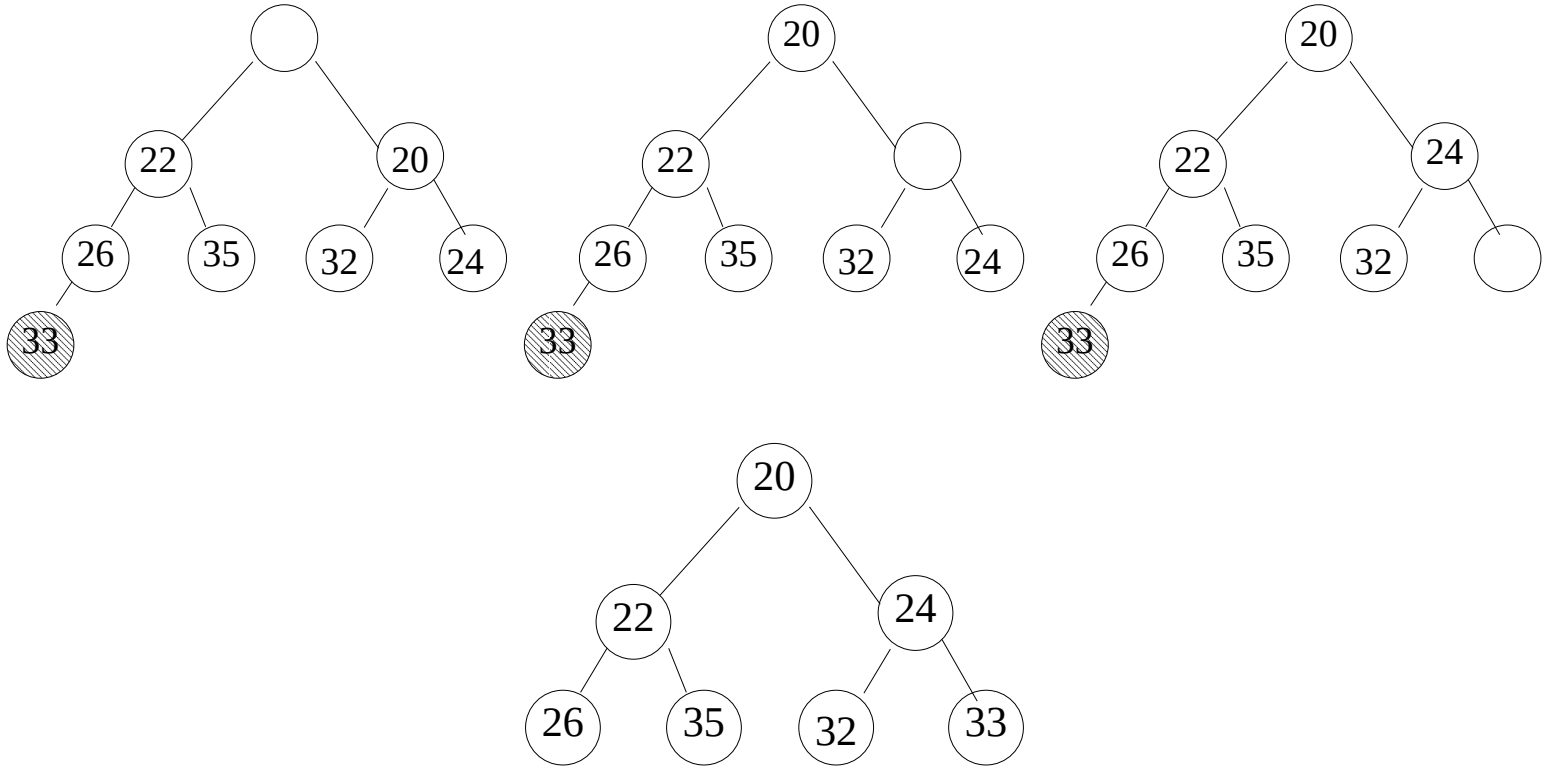
This is performed as follows



If we delete the next lowest



and the next lowest



etc etc

Building Heaps

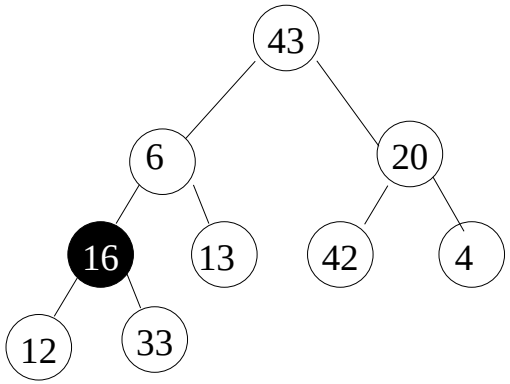
Given a list of keys, a heap can be built simply by using the `insert` function described above.

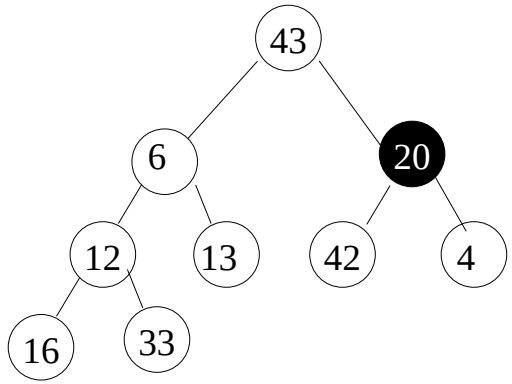
A more efficient $O(n)$ technique is the following:

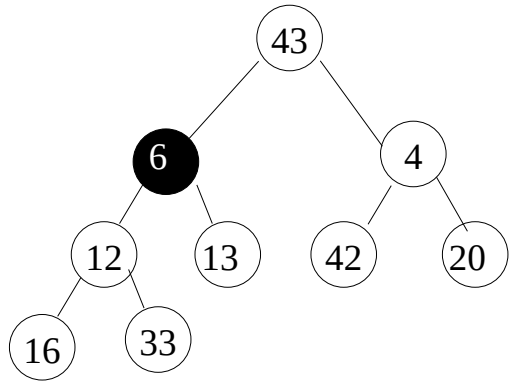
- Put the list elements into the heap array in any order, without worrying about heap invariant
- Turn the array into a heap as follows:
  - Starting at the rightmost, deepest node with a child, swap its contents with that of one of its children to ensure the heap property for the tree below, then percolate that element down if necessary
  - Work leftwards and upwards to the root.

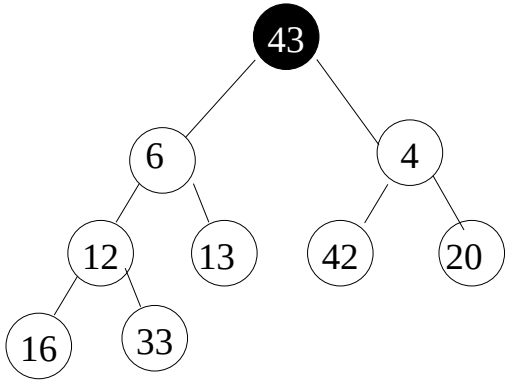
For example:

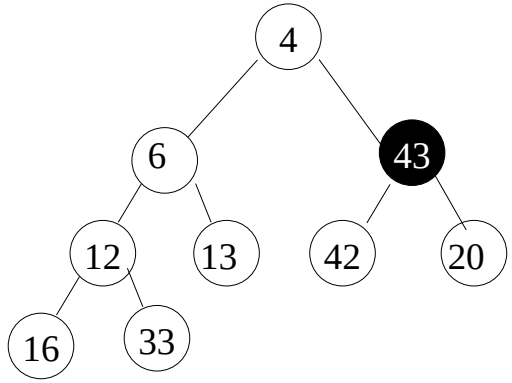
index	0	1	2	3	4	5	6	7	8	9	10
value		43	6	20	16	13	42	4	12	33	

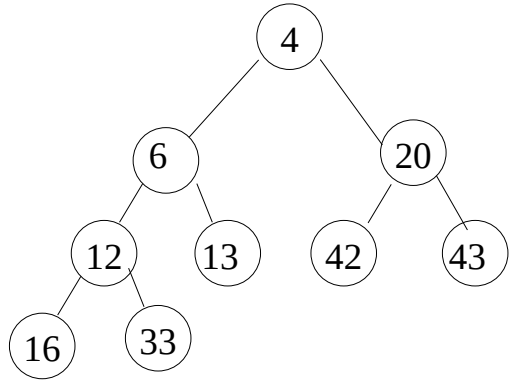












Using this technique followed by removing the smallest element from the heap until it is empty, gives an $O(n \lg n)$ sort technique called *heapsort*

Priority Queues

Normal queues are FIFO devices

In a Priority Queue each element entering a queue is assigned a *value*, usually a number, and the first element to leave the queue is that with the *lowest value*.

Used in Operating Systems etc

Most common implementation of Priority Queues is the heap.

Data Types for Priority Queues

```
struct HeapStruct {
  unsigned int Capacity;     /* Capacity of heap */
  unsigned int Size;          /* Current # of elements in heap */
  ElementType *Elements;
};

typedef struct HeapStruct *PriorityQueue;
```

Initialising a Priority Queue

```
PriorityQueue
CreatePq( unsigned int MaxElements ) {
  PriorityQueue H;

  H = (PriorityQueue) malloc( sizeof( struct HeapStruct ) );
  if( H == NULL )
    FatalError("Out of space!!!");

  /* Allocate the array + one extra for sentinel */
  H->Elements= (ElementType *)
               malloc((MaxElements+1)*sizeof(ElementType));
  if( H->Elements == NULL )
    FatalError("Out of space!!!");
```

```
  H->Capacity = MaxElements;
  H->Size = 0;
  H->Elements[0] = MINDATA;      /* MINDATA less than ALL  */
                                 /* possible data elements */

  return H;
}
```

Insertion in a Priority Queue

```
/* H->element[0] is a sentinel */

void
insert( ElementType x, PriorityQueue H ) {
  unsigned int i;

  if( IsFull ( H ) )
    error("Priority queue is full");
  else {
    i = ++H->Size;
```

```
    while( H->Elements[i/2] > x ) {
      H->Elements[i] = H->Elements[i/2];
      i /= 2;
    }

    H->Elements[i] = x;
  }
}
```

Deletion from a Priority Queue

```
ElementType
DeleteMin( PriorityQueue H ) {
  unsigned int i, child;
  ElementType MinElement, LastElement;

  if( IsEmpty( H ) ) {
    error("Priority queue is empty");
    return H->Elements[0];
  }

  MinElement = H->Elements[1];
  LastElement = H->Elements[ H->Size-- ];
```

```
  for (i=1; i*2<=H->Size; i=child ) {
    child = i*2;          /* find smaller child */
    if((child!=H->Size) &&
       (H->Elements[child+1] < H->Elements[child]))
      child++;

    if( LastElement > H->Elements[child] )     /*percolate */
      H->Elements[i] = H->Elements[child];
    else
      break;
  }
  H->Elements[i] = LastElement;

  return MinElement;
}
```

Ordered Binary Trees

A binary tree is said to be *ordered* if, for every node n in the tree, the values of the keys in its left subtree are *smaller* that the key at n, and those in the right subtree are *greater* than the key at n.

The operations required on an ordered binary tree are

- Initialise a tree
- Find the location (if any) of a given key in a tree.
- Insert a given key in a tree.
- Delete a given key from a tree.
- List the contents of a tree.

For implementations of these see the lab exercise

Another common operation is

- Find the largest/smallest element in a tree

If we adopt a naïve approach to insertion and the data is pre-sorted, the cost of building a tree becomes *quadratic* in the size of the data (Why?)

One solution is to attempt to keep the tree *balanced*

There are several possible definitions of the term *balanced*, one is

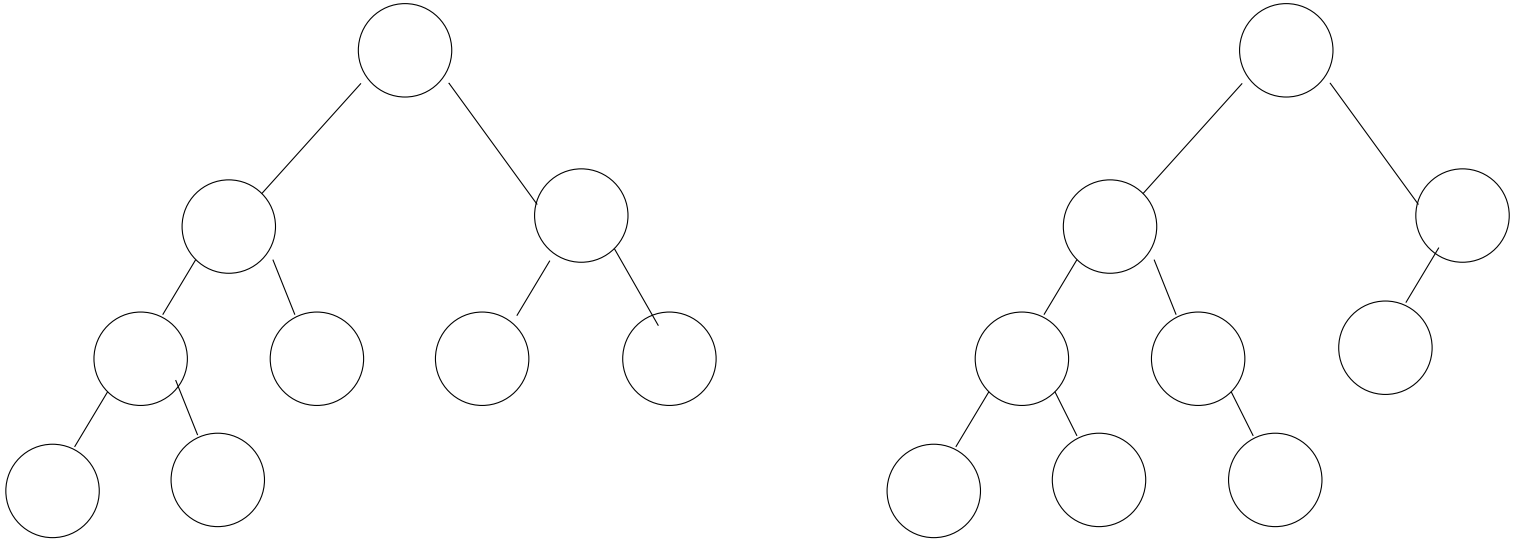
> A tree is balanced if every node has left and right subtrees whose heights differ by at most 1

This is another type invariant

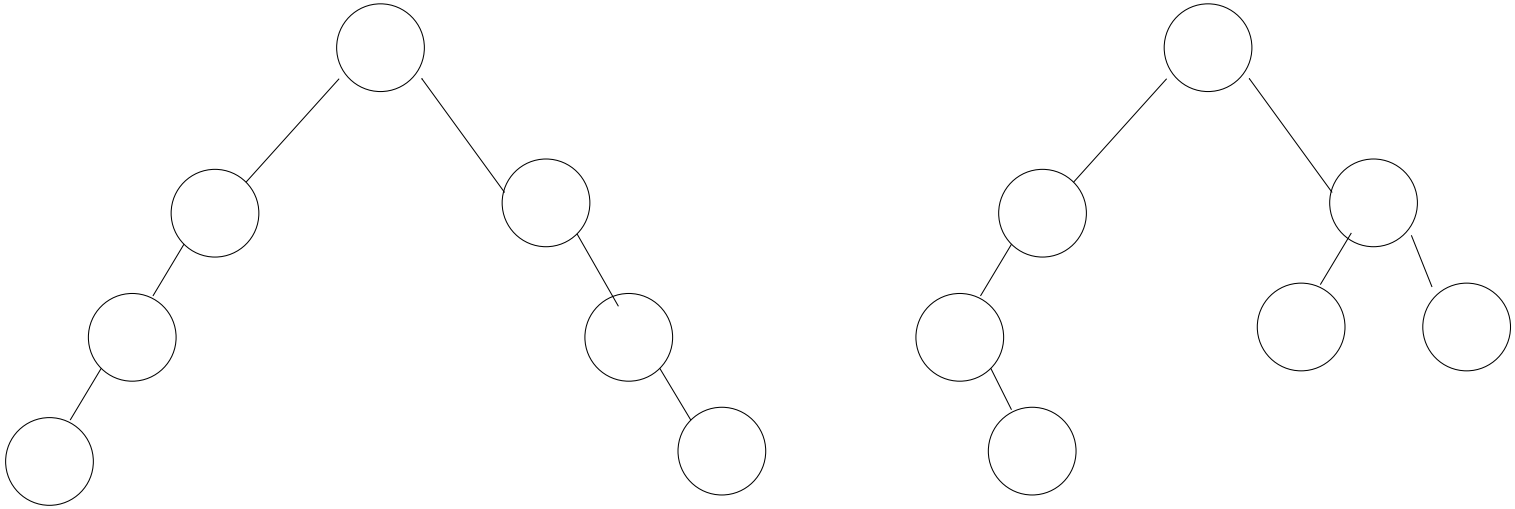
A tree with this property is called an *AVL tree*. (Adelson, Velski and Landis)

For example

These trees are balanced



These trees are not



The problem is that when a new node is inserted, or an old one deleted, the tree can become unbalanced

When inserting a new node in an AVL tree, find a place to put it in the usual way

Then check the tree to be sure that it is still balanced

If tree has lost the AVL property, only nodes between inserted element and root can have balance destroyed. (Why?)

Balancing algorithm performs at most 1 constant time operation on each of the ancestors of the unbalanced node.

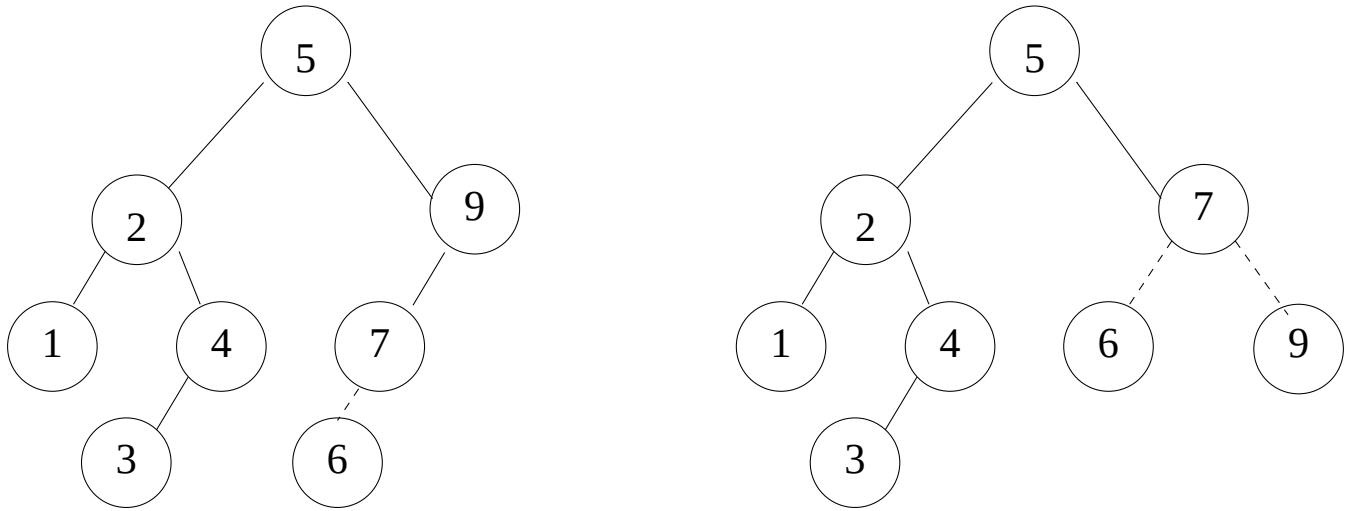
So, restoring the AVL property to the tree is an O(log N) operation.

AVL trees: restoring the balance

For example if we insert 6 into the tree below, unbalances tree

First unbalanced ancestor of new node is node containing 9

Can restore balance by *rotating* unbalanced subtree.



Suppose n is the first node (going up) which is unbalanced.

Four different possibilities

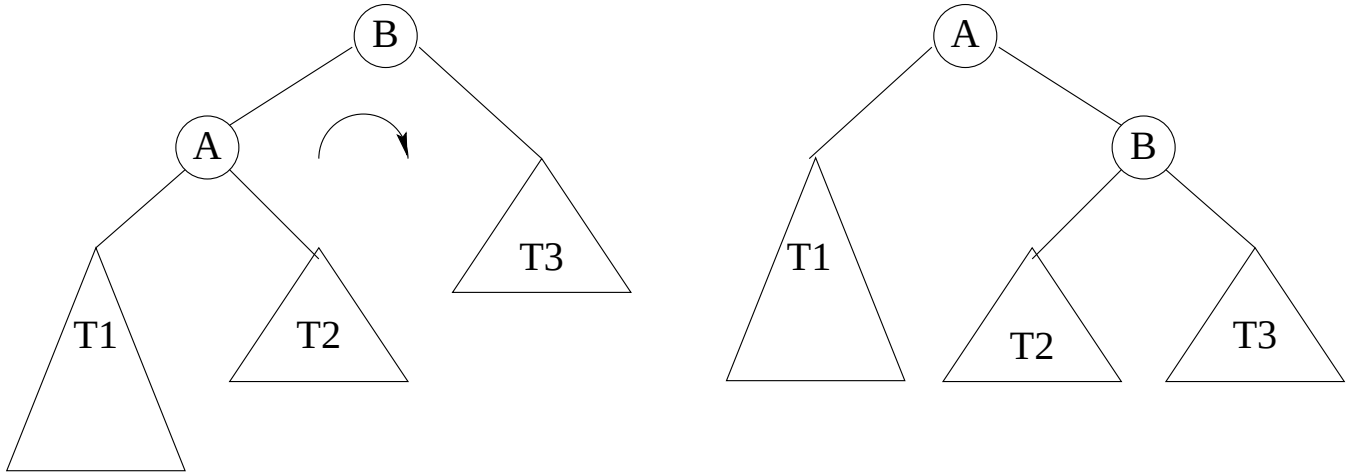
Imbalance due to insertion in left subtree of left child

Imbalance due to insertion in right subtree of right child

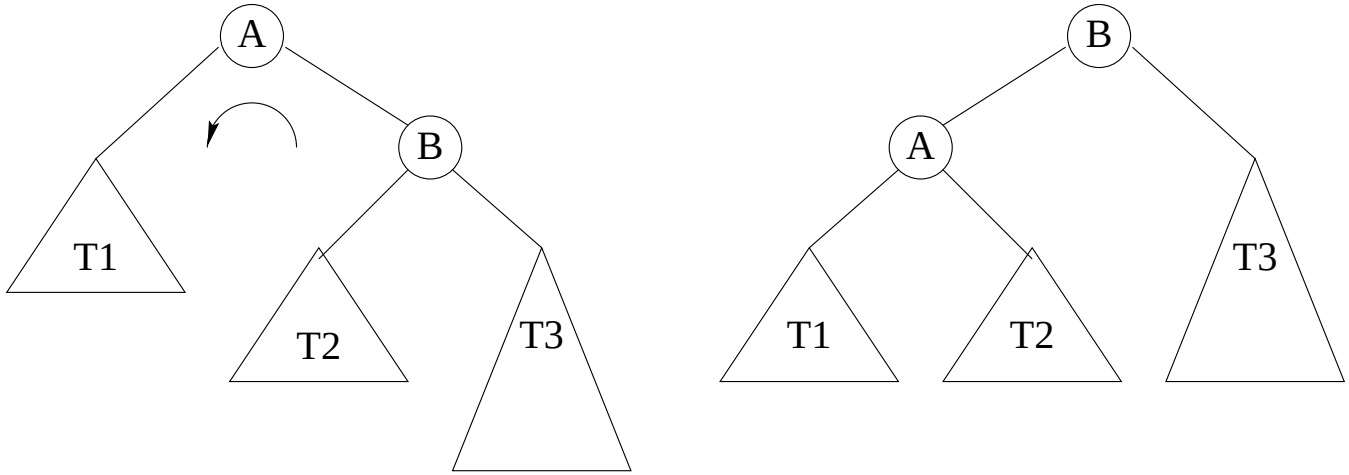
Imbalance due to insertion in right subtree of left child

Imbalance due to insertion in left subtree of right child

Single right rotation, imbalance in left subtree of left child



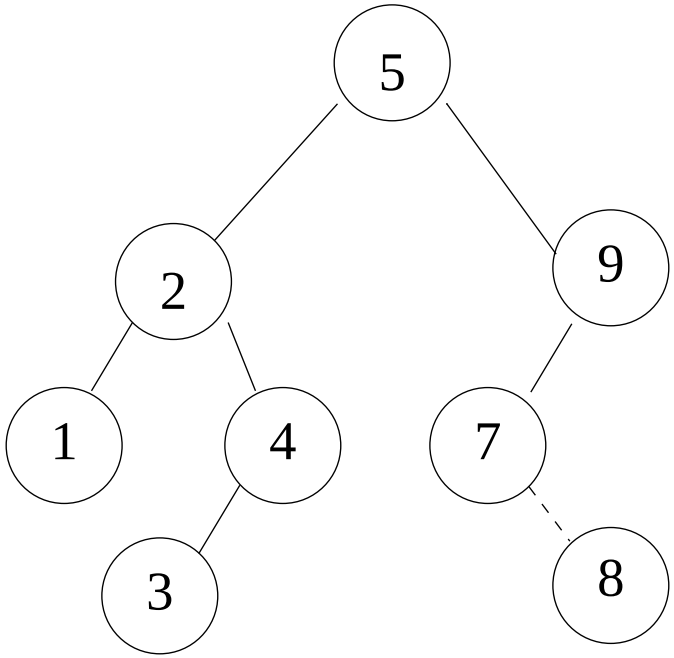
Single left rotation, imbalance in right subtree of right child

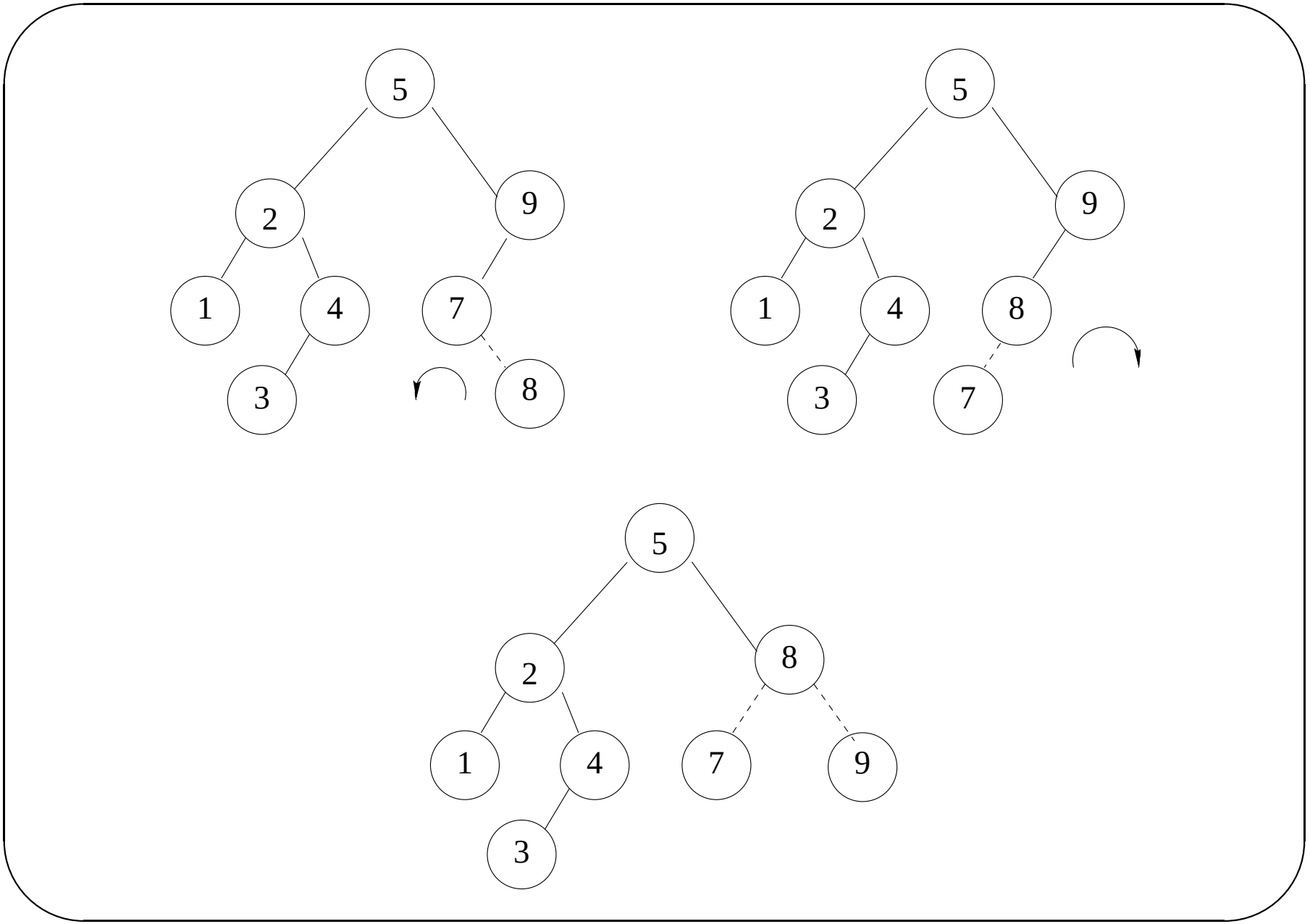


Now if we insert 8 into the tree below, unbalances tree

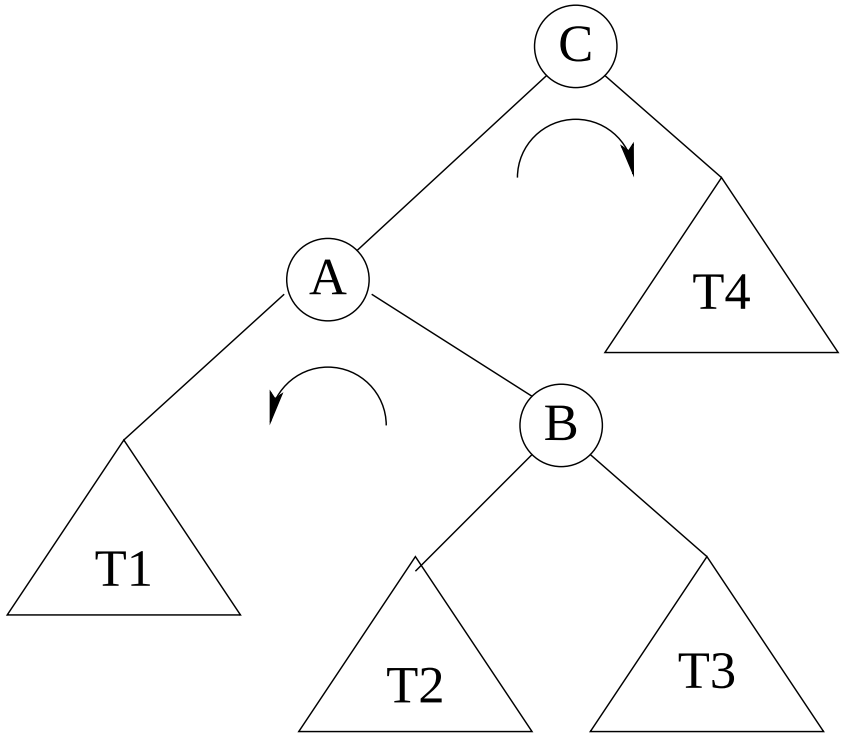
First unbalanced ancestor of new node is again node containing 9

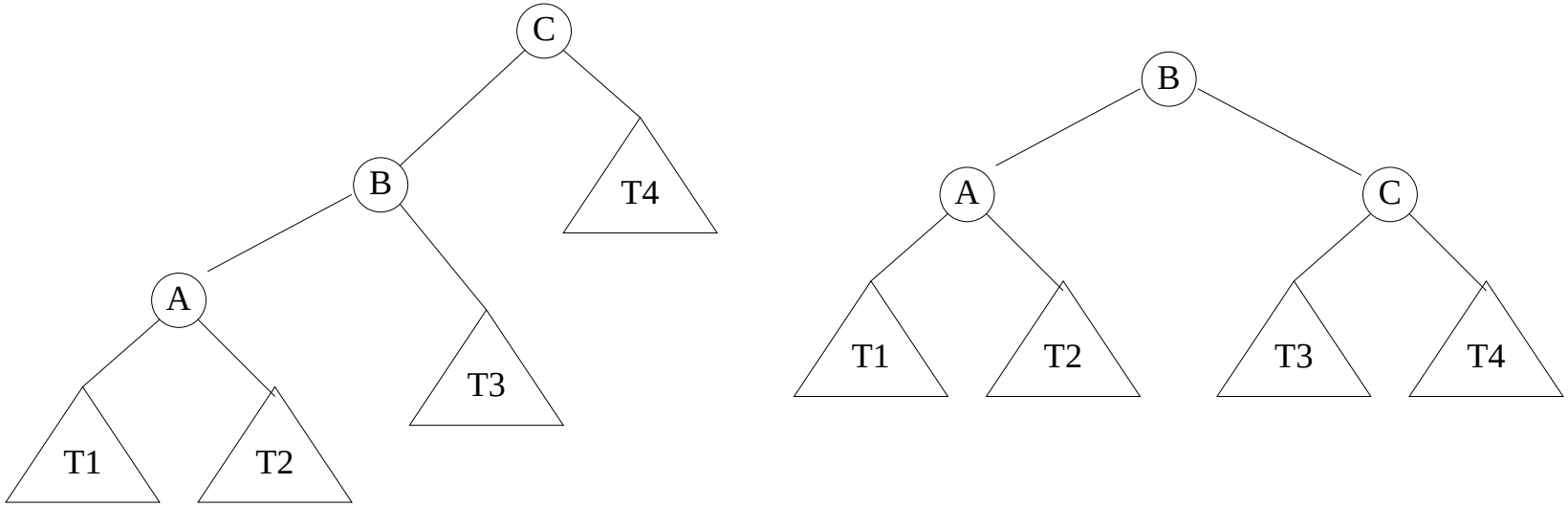
Need *two* rotations to restore balance



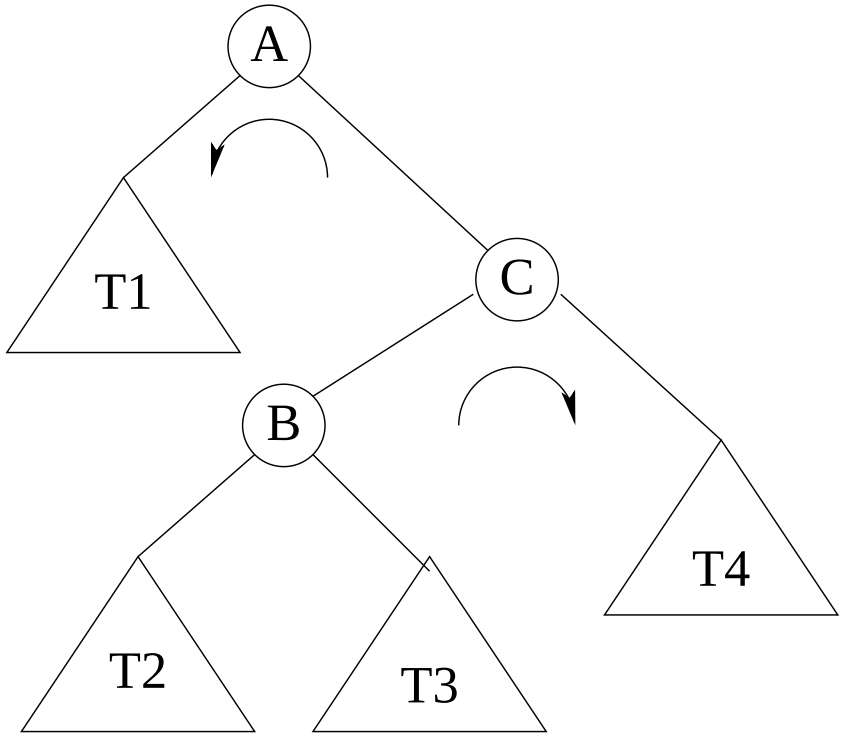

5
2
9
1
4
7
3
8
5
2
9
1
4
8
3
7
5
2
8
1
4
7
9
3

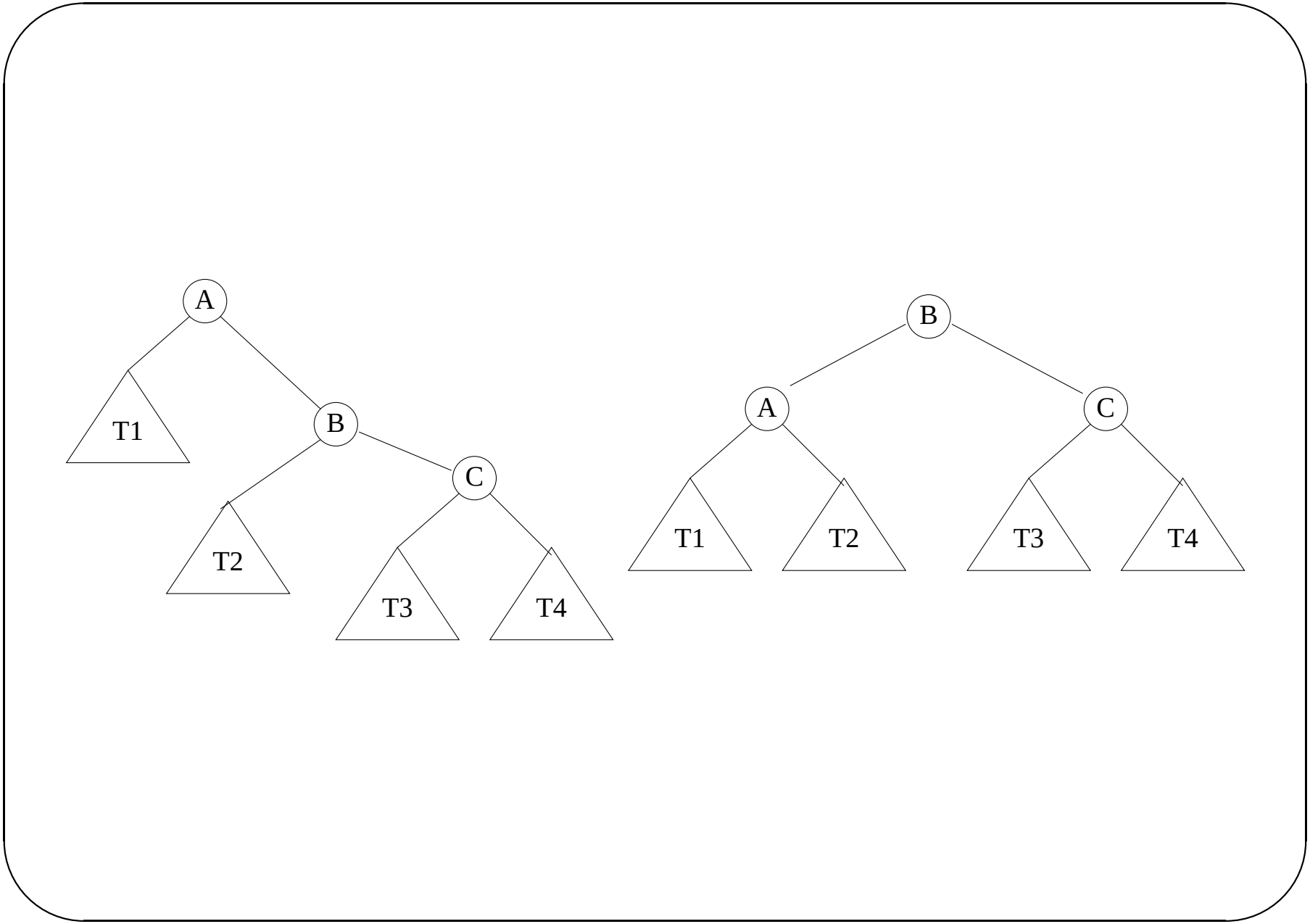
Double left-right rotation, imbalance on right subtree of left child




C
B
T4
A
T3
T1
T2
B
A
C
T1
T2
T3
T4

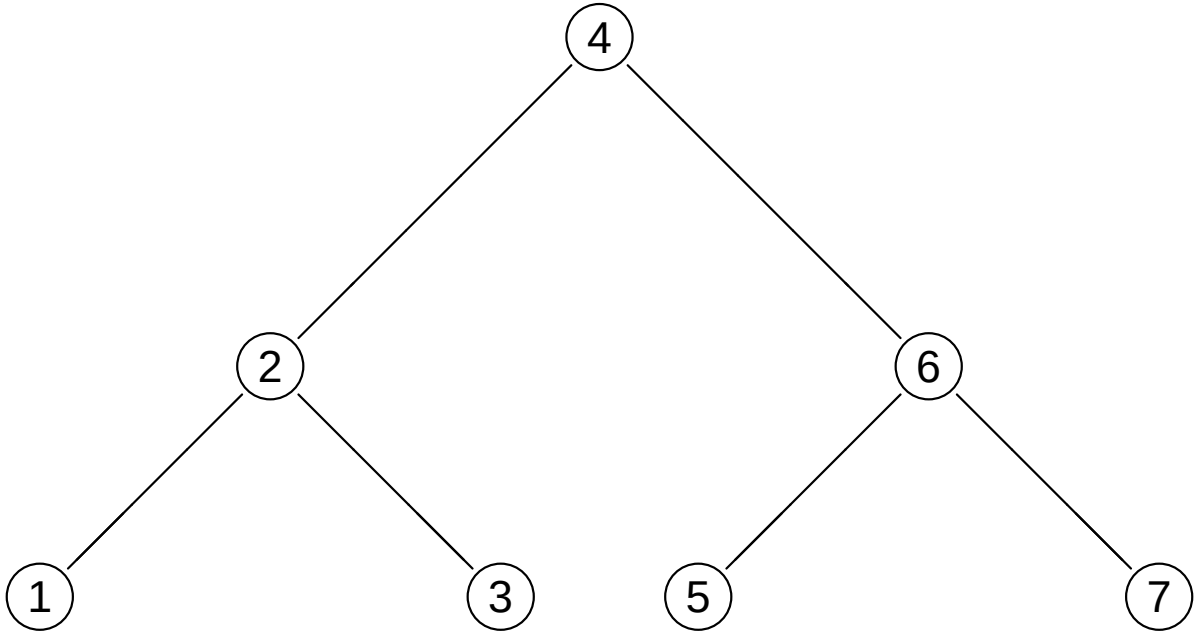
Double right-left rotation, imbalance on left subtree of right child



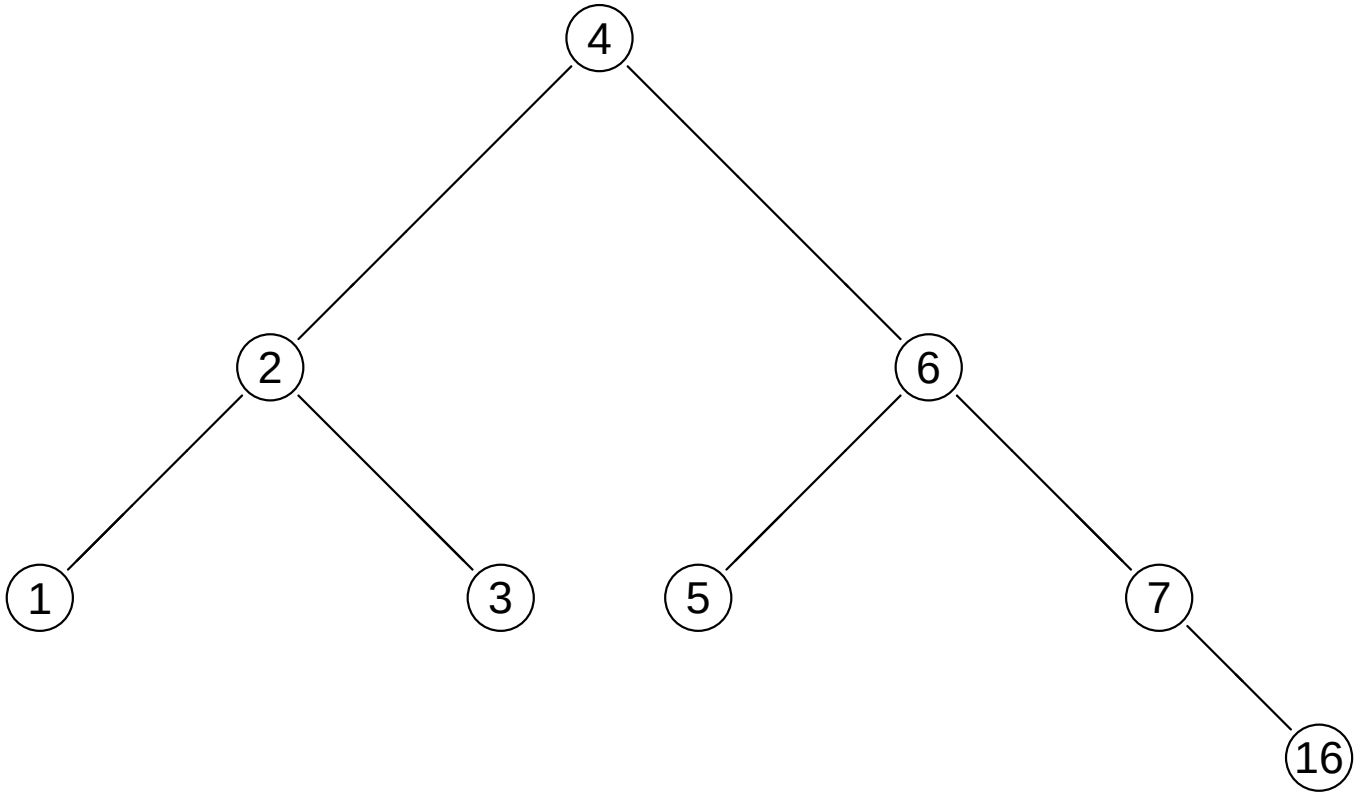

A
T1
B
T2
C
T3
T4
B
A
C
T1
T2
T3
T4

AVL Trees: An example

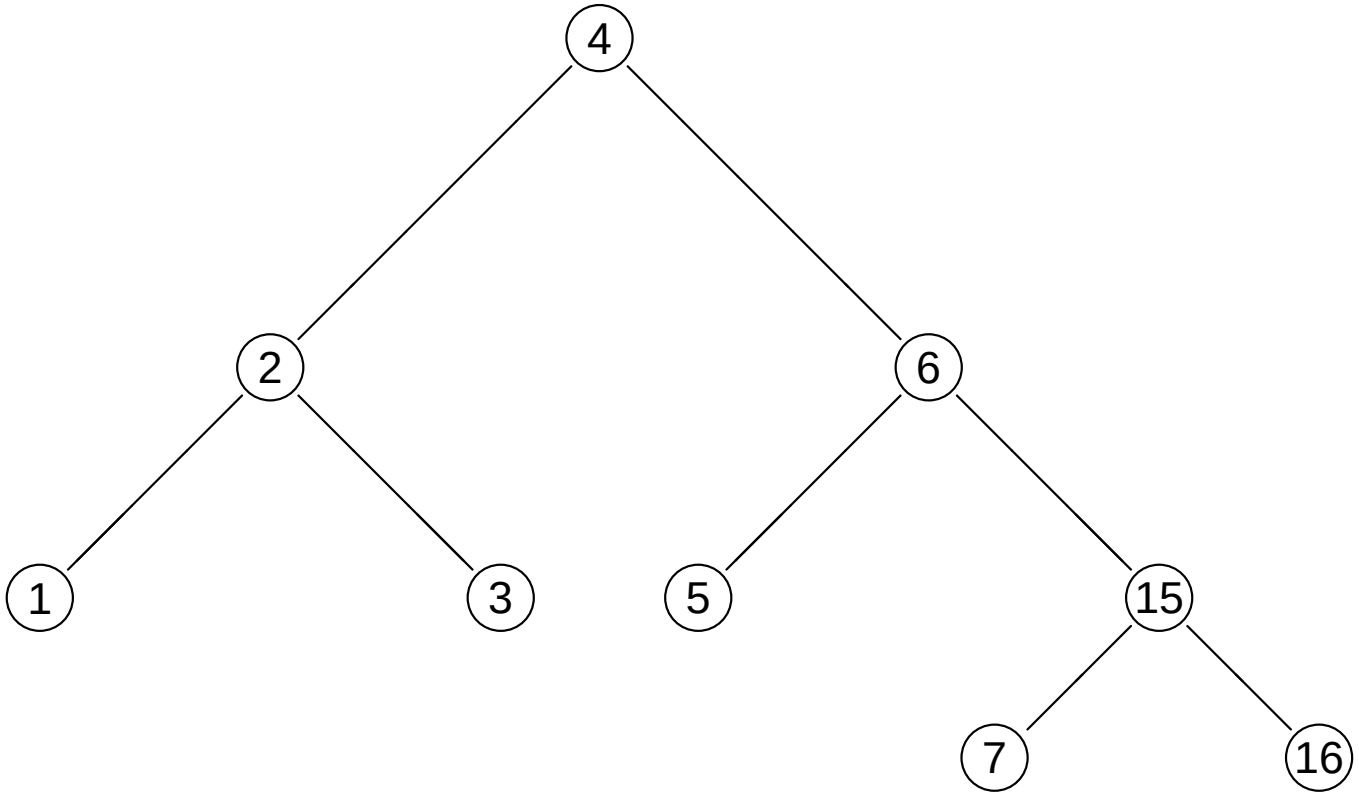
Starting tree



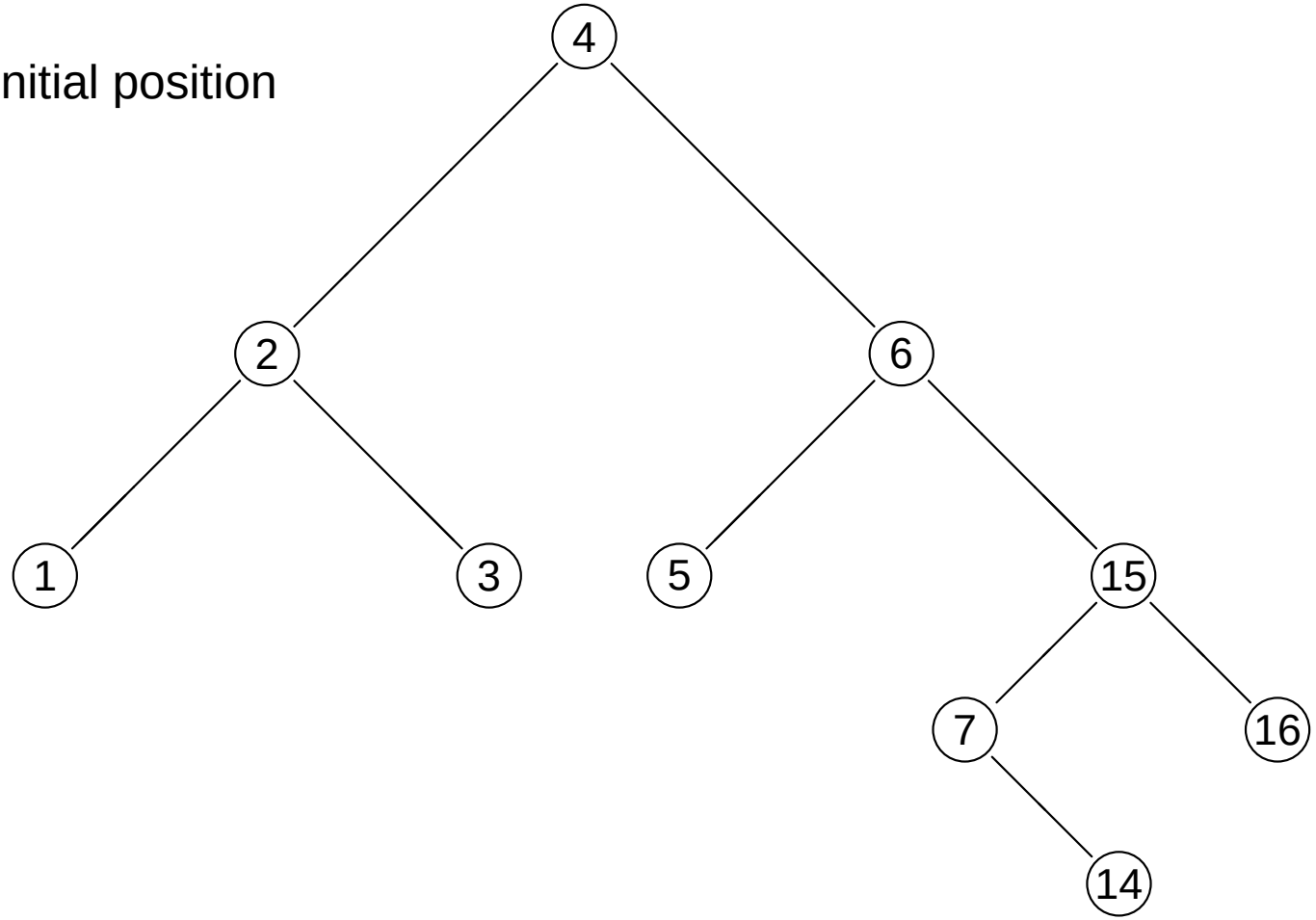
Insert 16



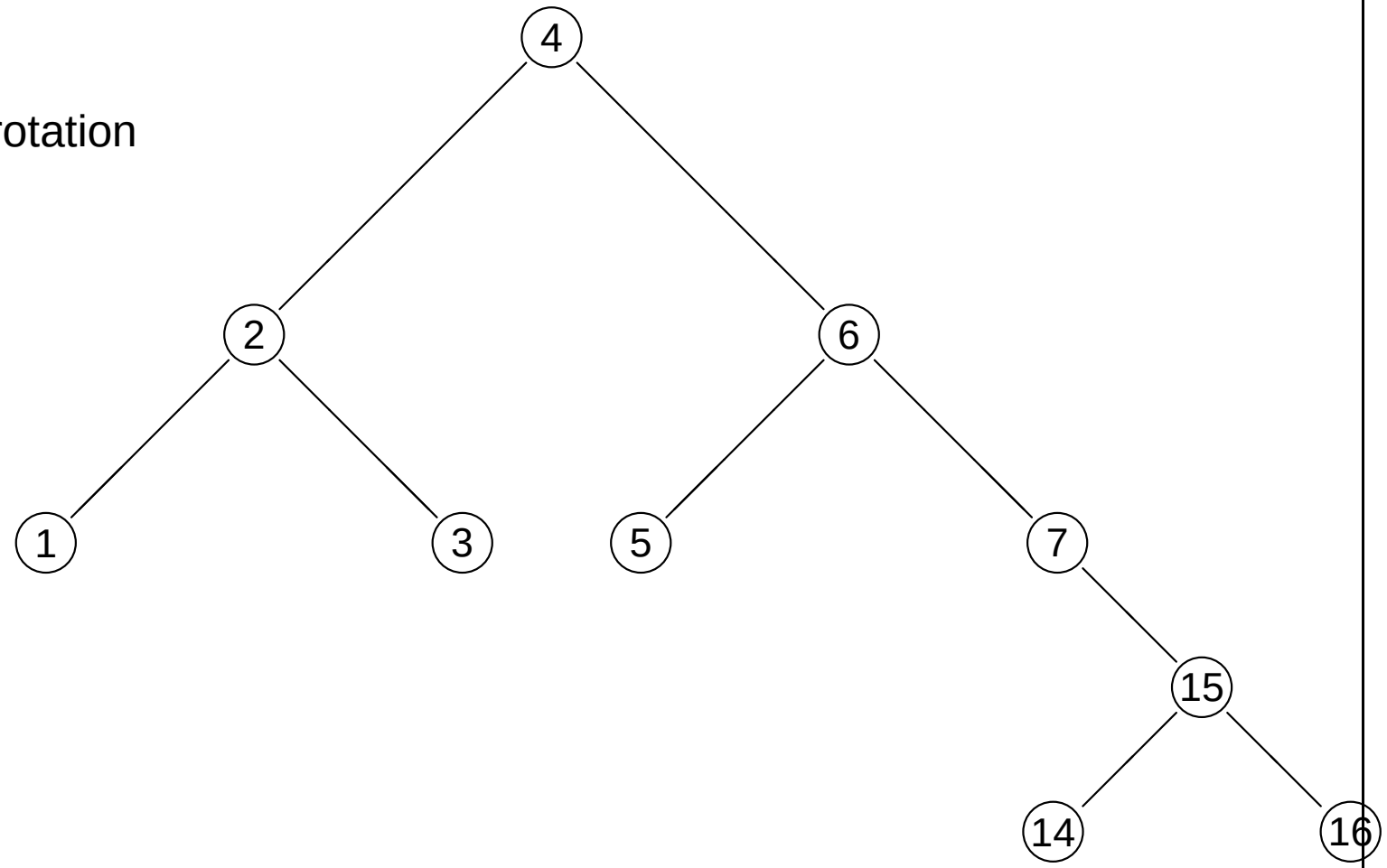
Insert 15



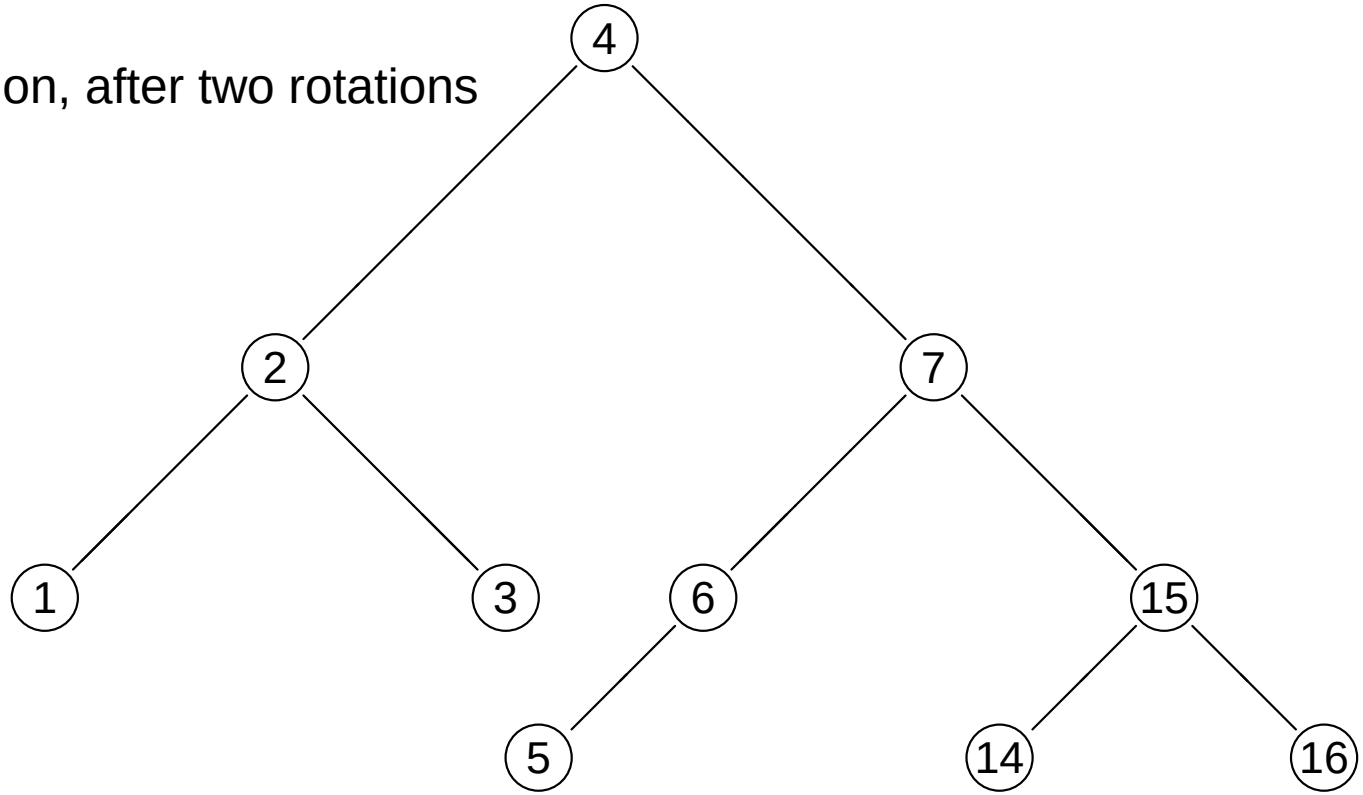
Insert 14, initial position



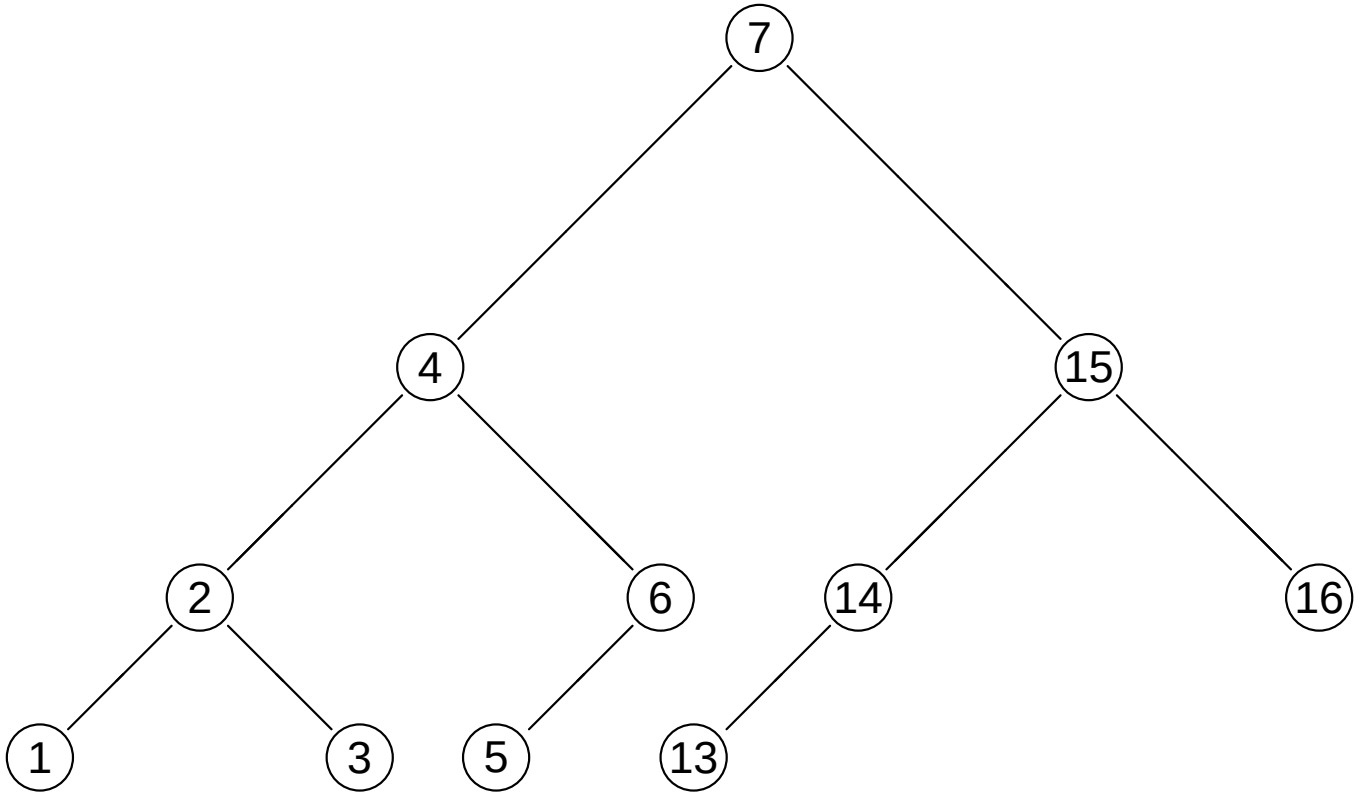
After one rotation



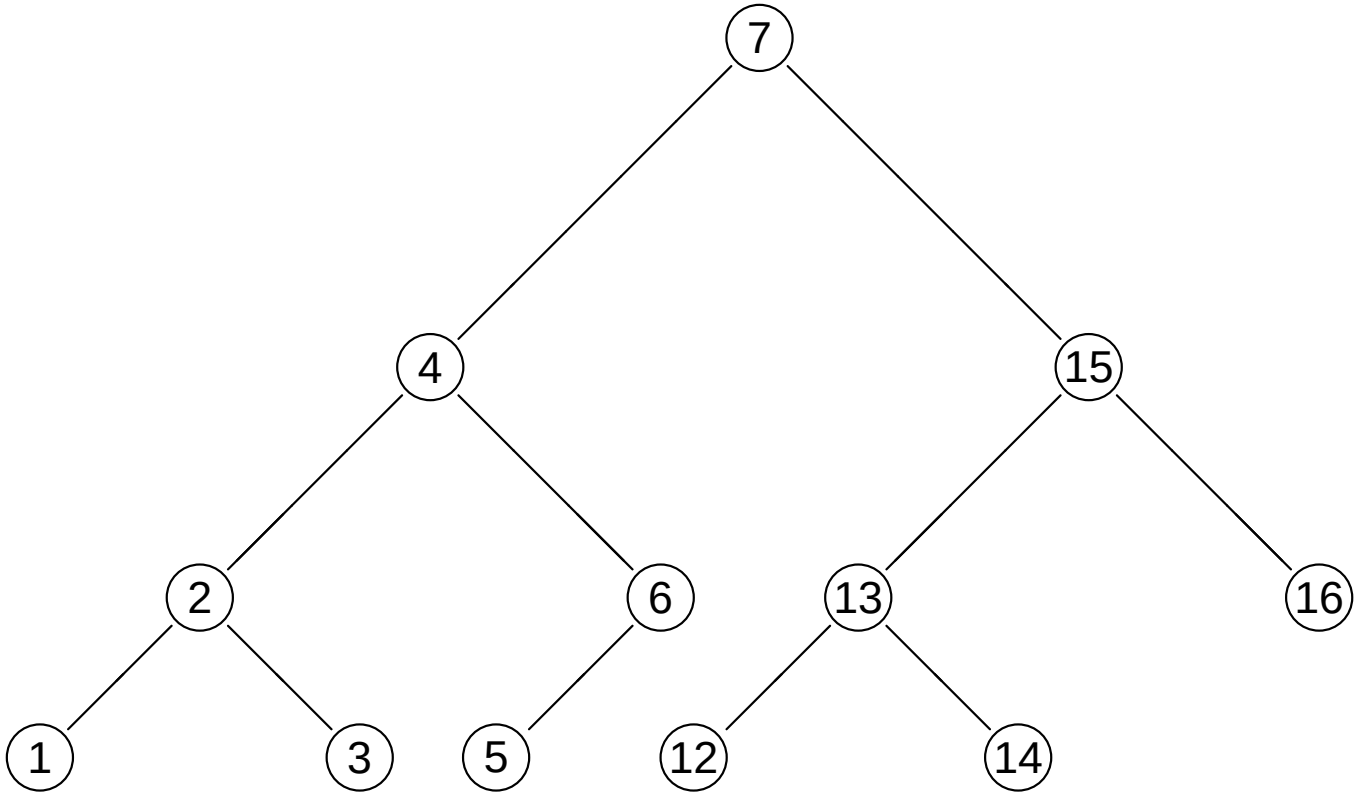
Final position, after two rotations



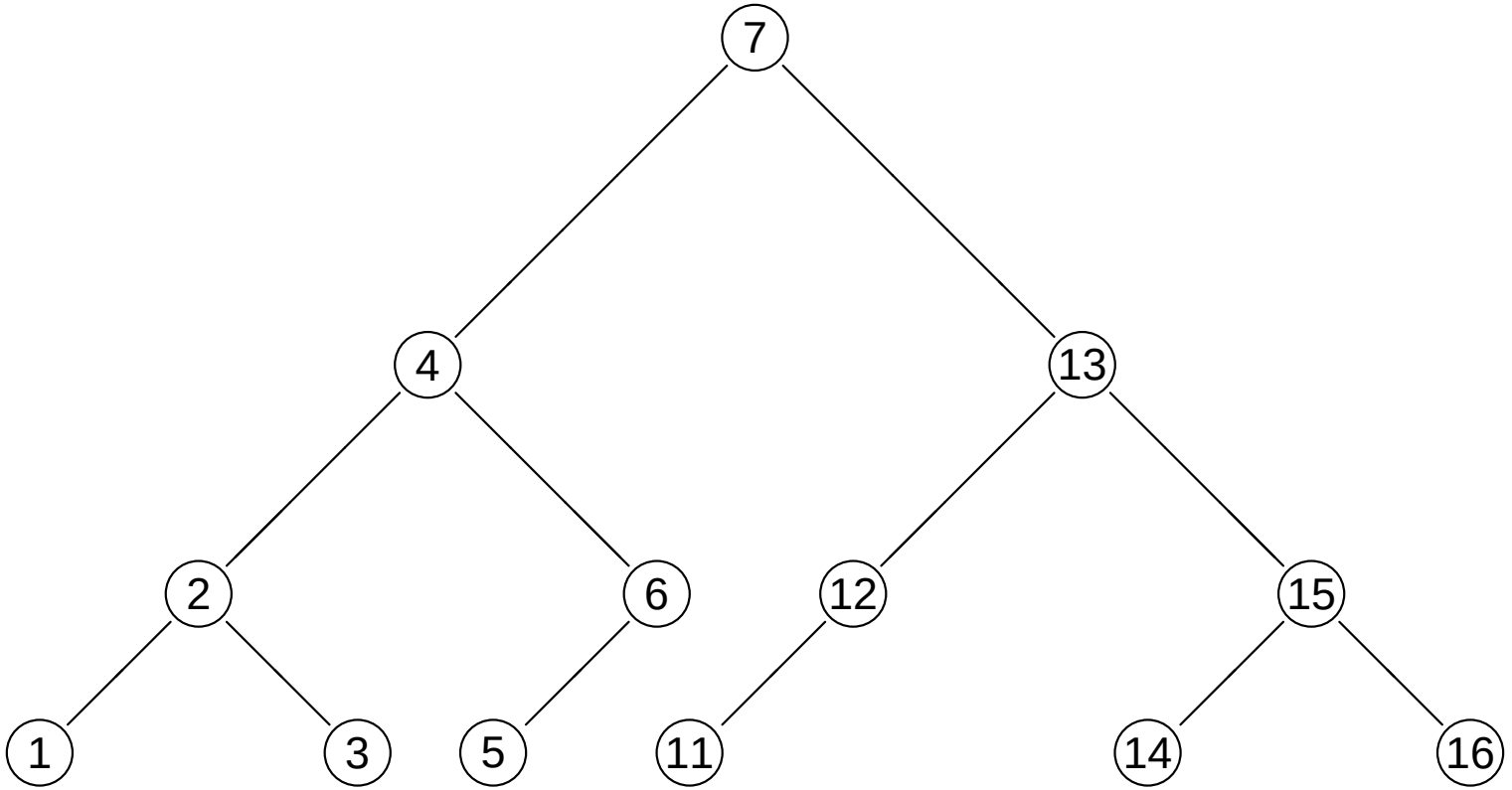
Insert 13



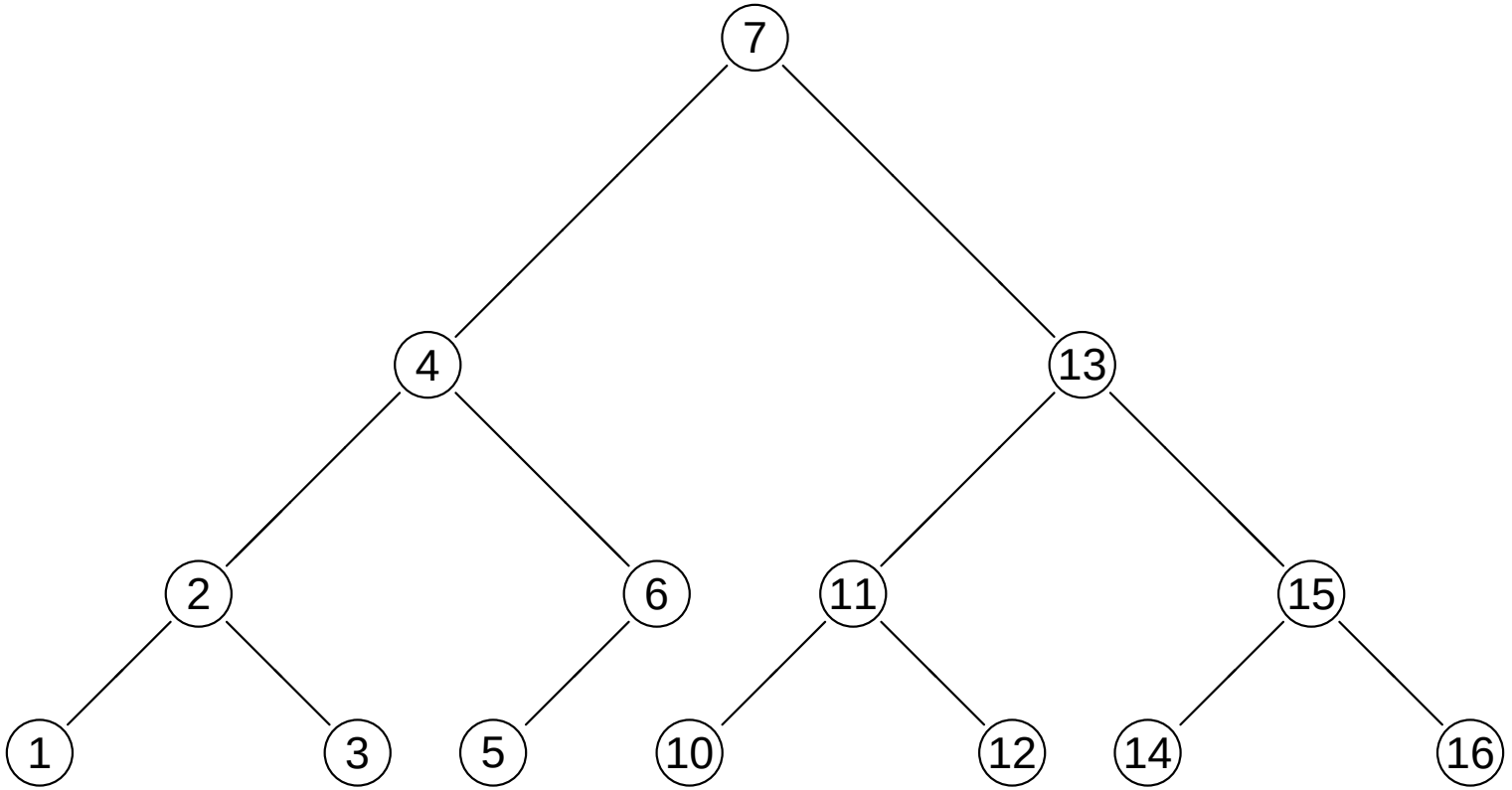
Insert 12



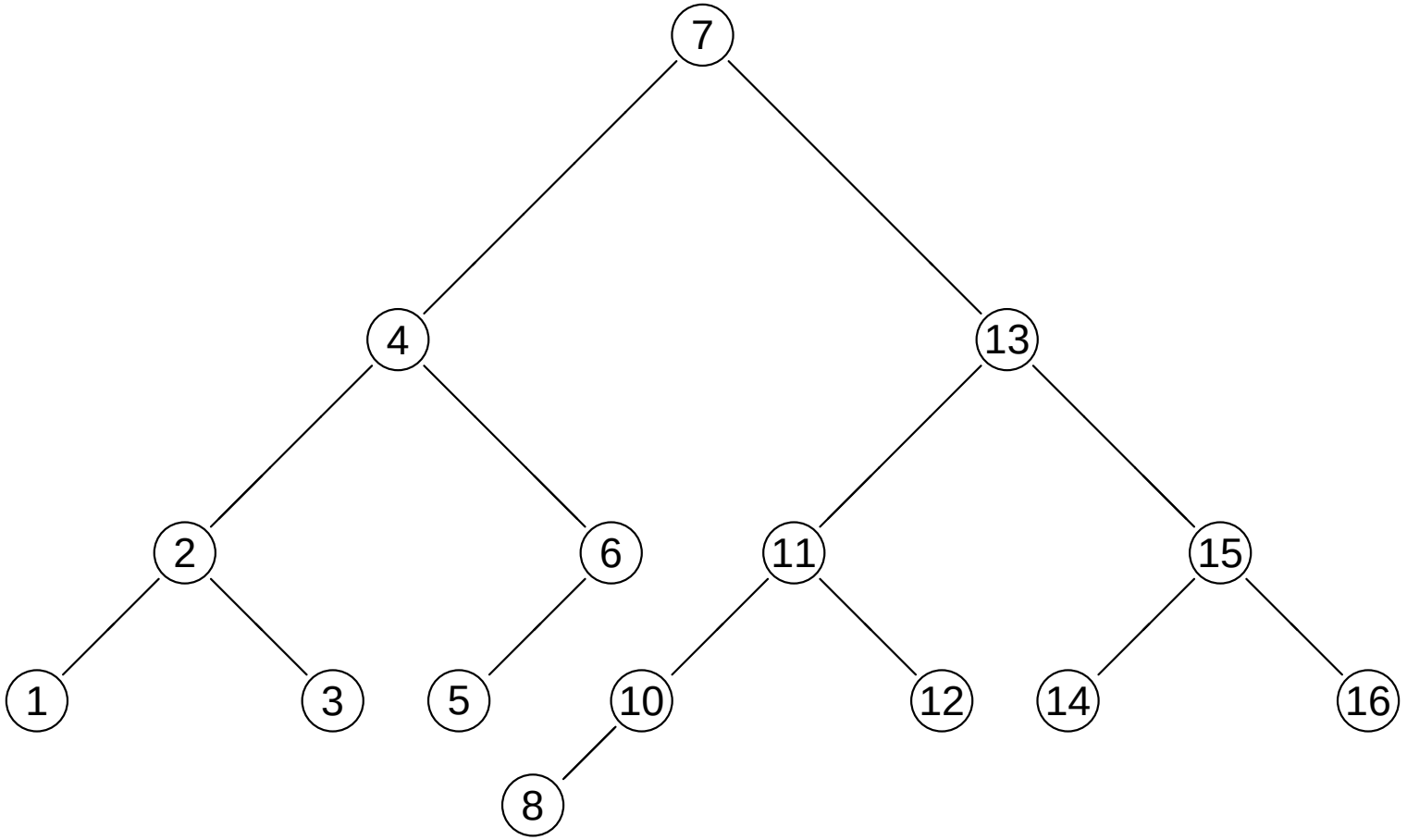
Insert 11



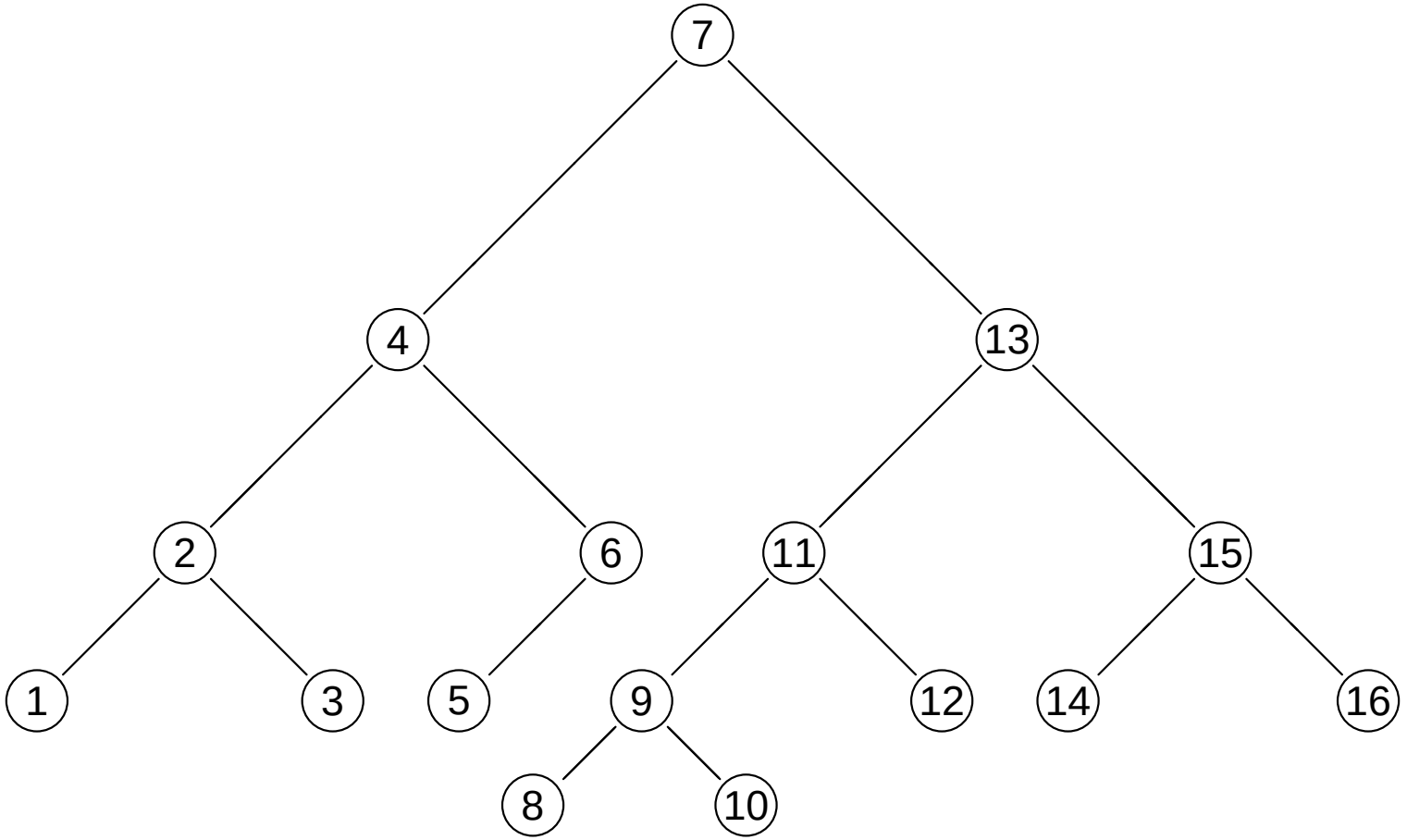
Insert 10



Insert 8



Insert 9



2-3 Trees

2-3 trees are search trees that maintain their balance by relaxing the *structural* constraint of being a *binary* tree.

Some nodes have 2 children and some have 3, hence the name

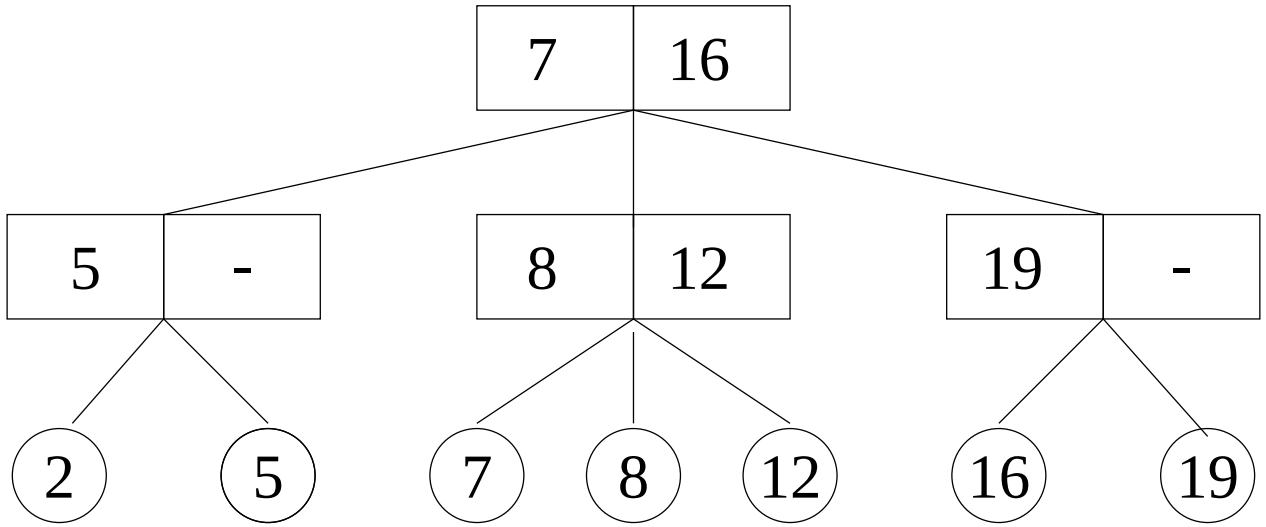
A *2-3 tree* is a tree with the following properties

- The root is either a leaf or has either 2 or 3 children
- All data is stored at the leaves.
- All leaves are at the same depth

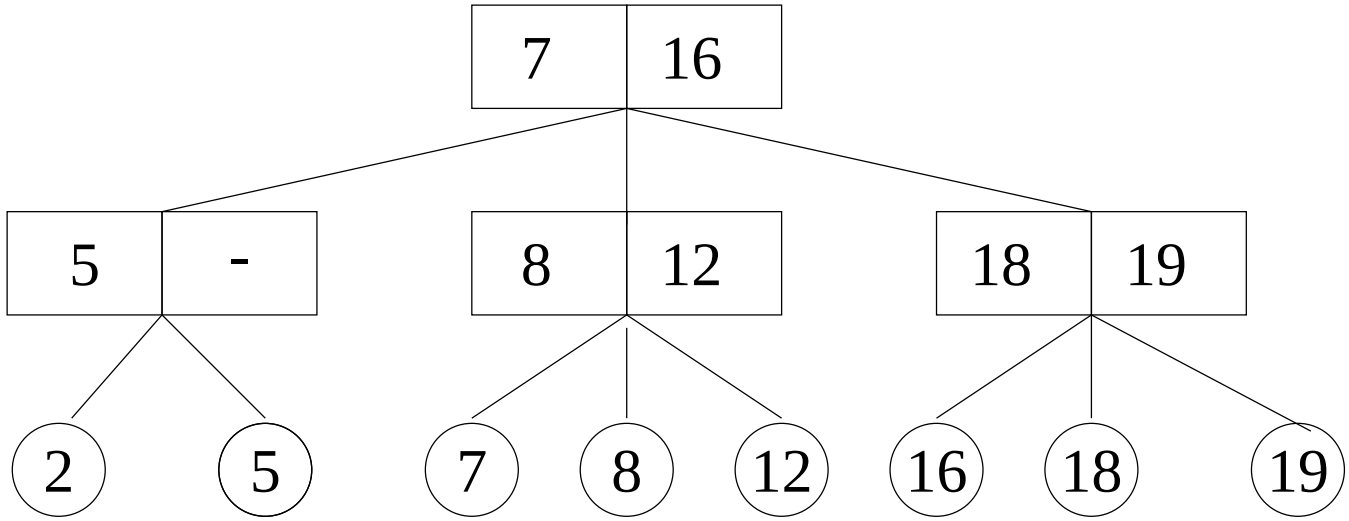
The first two of these constraints are *structural* and the third is part of the *type invariant*

The type invariant is completed with the following constraint

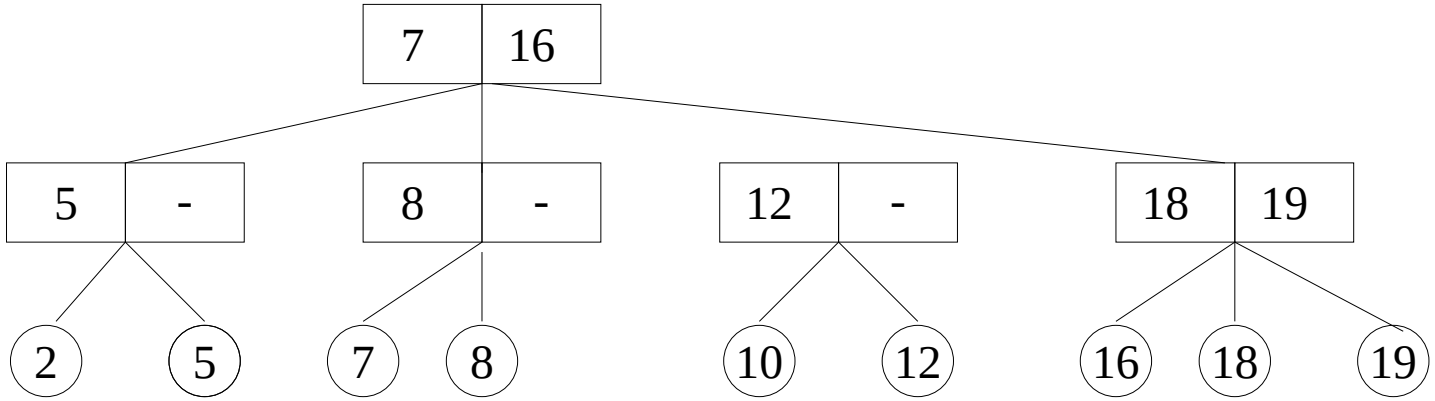
In each node, we store the value of smallest leaf in tree of second child, and value of smallest leaf in tree of third child, if there is one



Insert 18

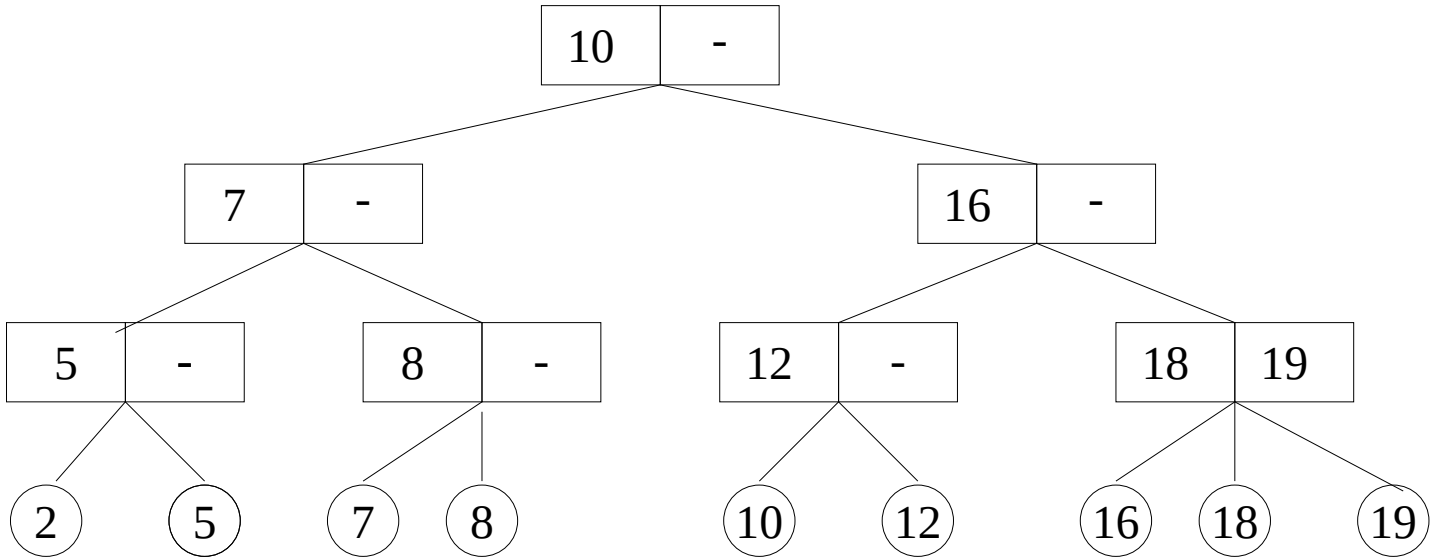


Insert 10



Wants to go between 8 and 12, so need to split 3 node into 2 2's

But root now appears to have 4 children, so need to split and make new root


10
-
7
-
16
-
5
-
8
-
12
-
18
19
2
5
7
8
10
12
16
18
19

B trees

2-3 trees can be generalised to trees that have the following properties, for some natural number M

- The root is either a leaf or has between 2 and M children
- All non-leaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children
- All leaves have the same depth.

Such trees are called *B-trees* of order M.

Main use is in database systems, where tree is on disk rather than in memory

Want to minimise the number of *disk accesses*

B-trees are in general very shallow