

COMP20012

3: Trees

Trees

Seen lots of examples.

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

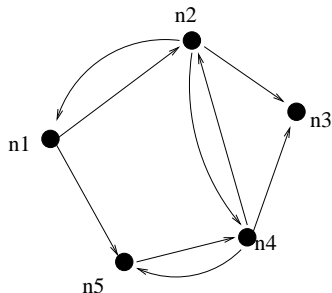
Trees have natural recursive structure

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

Mathematical descriptions of Graphs and Trees

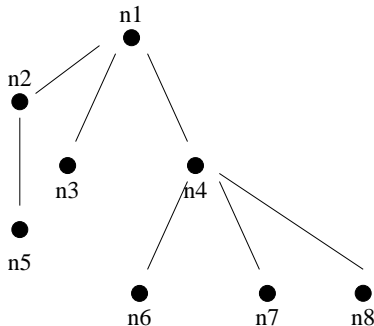
A *directed graph* is a pair (N, E) consisting of a set of nodes N , together with a relation E on N . There is no restriction on the relation E .

aEb 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

A (rooted) tree also has the property that it is a directed graph such that

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

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 *Java* and *C* representations of *binary trees*.

```
public class BinaryNode
{
    private Object    element;
    private BinaryNode left;
    private BinaryNode right;
    ....
}
```

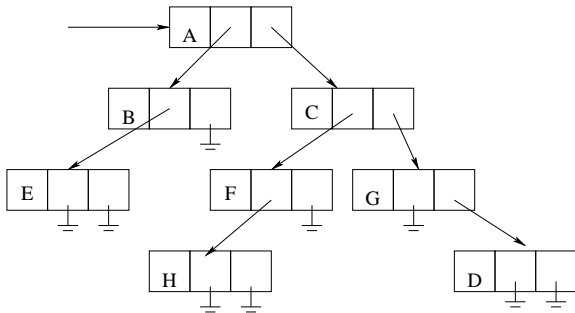
```
public class BinaryTree
{
    private BinaryNode root;
    ...
}
```

and in C

```
typedef struct TreeNode *PtrToNode;
```

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

```
typedef PtrToNode Tree;
```



Non-binary trees are almost as simple

```
public class NaryNode
{
    private Object    element;
    private NaryNode firstChild;
    private NaryNode firstSibling;
    ....
}
```

```
public class NaryTree
{
    private NaryNode root;
    ...
}
```

and

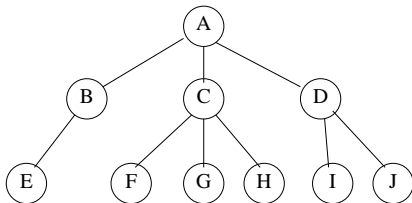
```
typedef struct TreeNode *PtrToNode;
```

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

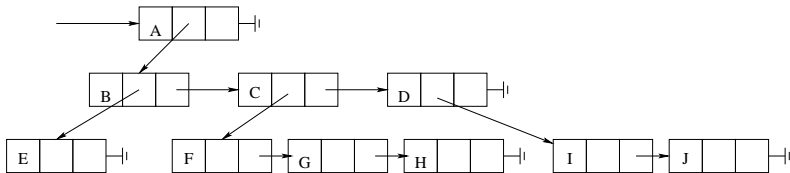
```
typedef PtrToNode Tree;
```

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

The nary tree



is represented as



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
- Level order: list all the nodes at each level followed by all nodes at the next level etc.

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

How would you implement a level-order traversal?

Other representations

- All the above representations use explicit pointers to represent the tree structure
- Now look at some *implicit* representations which do not use pointers.

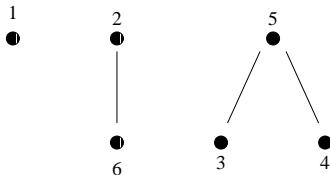
Array based representations

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

- Could also have data associated with each node
- This representation is not particularly useful if tree traversal is required
- How would we find the *first child* of a node or *next sibling* in this context?

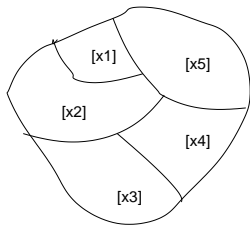
This representation is very useful for representing a *partition* of the set $1 \dots n$.

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. $\forall i, j, i \neq j \cdot X_i \cap X_j = \emptyset$

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

A partition of a set is just a way of breaking it up into disjoint chunks.



Partitions are used to represent *equivalence relations* on some set (see COMP10020 notes)

Equivalence Relations

- Often elements of a set are regarded as equivalent because they have some set of properties in common even though they are *not identical*.
- For example two electrical components might be regarded as equivalent, even if they are produced by different manufacturers.
- Two different Java functions might be regarded as equivalent if they always produced the same results, even if they were implemented in completely different ways.

It seems 'obvious' that any notion of equivalence should satisfy the following:

- 1 a is equivalent to a
- 2 If a is equivalent to b then b is equivalent to a
- 3 If a is equivalent to b and b is equivalent to c then a is equivalent to c

For any set A there will be many possible equivalence relations.

Equivalence Classes and Partitions

- The idea of an equivalence relation R on a set A is that if aRb then in some sense a and b are the same.
- Any equivalence relation on a set A divides A in a natural way into disjoint parts.
- Each part contains elements that are equivalent to each other, so two elements a, b are put into the same part if and only if aRb .

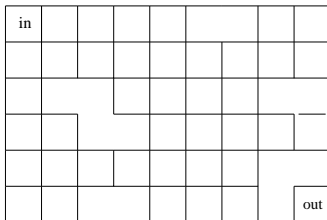
- One example of their use is in generation of mazes.
- Start with a rectangular grid of cells, with walls between every cell.
- Choose a desired entry square and exit square

in								
								out

- Construct a partition by using the equivalence relation, a is related to b if there is a path from a to b .
- Initially this partition will consist of a set of sets of single elements, since with all the walls in place the only thing that is related to a is a itself.
- So the initial state of the partition will be

$$\{\{1\}, \{2\}, \dots, \{n\}\}$$

- Now choose a wall at random, and if the two cells separated by that wall are not already connected, remove the wall, and merge the partition members containing the two cells.



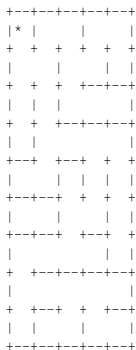
- For example

$$\{\{1\}, \{2\} \dots \{6,7\} \dots \{n\}\}$$

- Continue this process until the entry and exit cells are in the same partition member.

$\{\{25, 26\}$

$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,$
 $21, 22, 23, 24, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,$
 $41, 42, 43, 44\}\}$



```

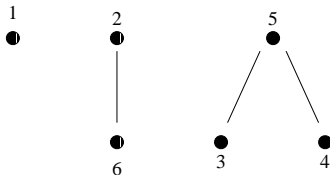
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|*|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+ +---+ + +---+---+---+ +---+ +---+---+ + +---+ +---+---+---+ + +---+
| | | | | | | | | | | | | | | | | | | | | |
+ + +---+ + + +---+---+ +---+ + + + +---+ +---+ + +---+---+ + +
| | | | | | | | | | | | | | | | | | | | | |
+ +---+ + +---+ +---+ +---+ +---+---+---+ +---+ +---+---+ + + +
| | | | | | | | | | | | | | | | | | | | | |
+ + + + + +---+---+ +---+ +---+ + +---+---+ +---+---+ + +---+ + +
| | | | | | | | | | | | | | | | | | | | | |
+ +---+---+ +---+ +---+ +---+---+---+ + +---+---+ +---+---+ + +---+
| | | | | | | | | | | | | | | | | | | | | |
+---+ +---+ +---+---+ + +---+---+ + +---+ +---+---+---+ + +---+ + +
| | | | | | | | | | | | | | | | | | | | | |
+ + +---+---+ +---+---+ +---+ +---+ + + + +---+ + + +---+
| | | | | | | | | | | | | | | | | | | | | |
+---+---+ + +---+---+ +---+---+ +---+---+---+ +---+ +---+ + + +---+---+
| | | | | | | | | | | | | | | | | | | | | |
+ +---+---+ + +---+---+ + +---+ +---+ +---+ +---+---+ + +---+ + +
| | | | | | | | | | | | | | | | | | | | | |
+---+---+ +---+---+---+---+ + +---+---+ +---+ +---+ +---+ + + + +
| | | | | | | | | | | | | | | | | | | | *|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

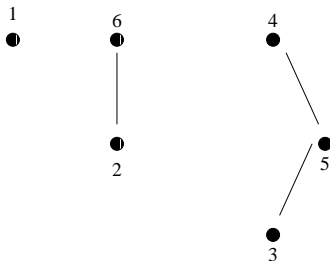
A simple way of representing a partition is by using a *forest of trees*
For example

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

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

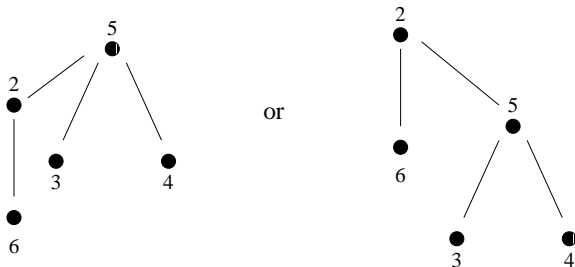


The same partition could also be represented by the forest

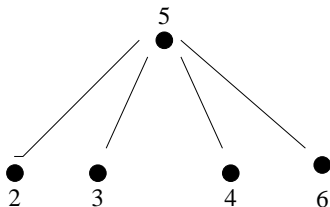


To determine whether 2 elements are in the same member of the partition, just find the *root* of the trees they are in.

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 $\{3, 4, 5\}$ from the first representation of the above partition



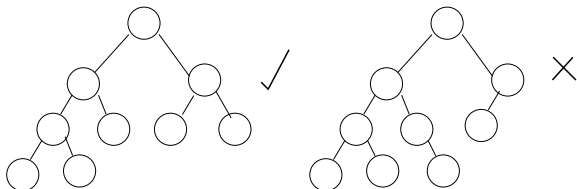
Can optimise the tree by flattening, by making the root of each node its parent



We will see other applications of this technique later in the course

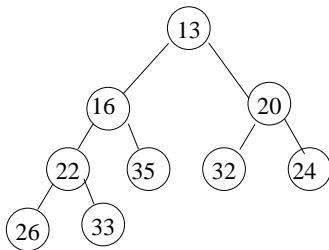
Binary Heaps

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



- How many leaves does such a tree have?

- 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.
- Note difference between this and an ordered binary tree

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

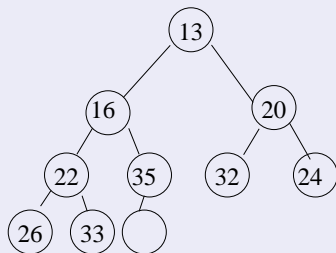
index	0	1	2	3	4	5	6	7	8	9	10
value		13	16	20	22	35	32	24	26	33	

- 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.

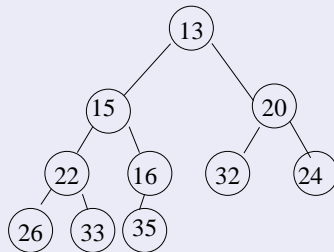
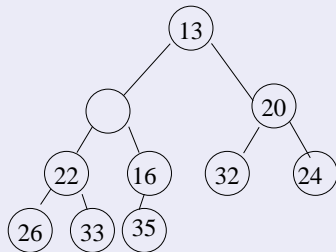
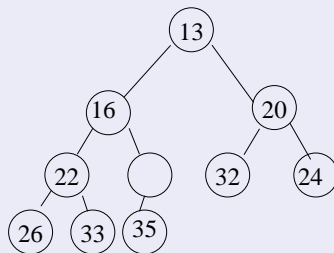
- Increase size of heap by one to open up a hole at the last element
- If the item to be inserted is *less than* the parent of the hole, move the parent into the hole and create a new hole
- Repeat this until the parent of the hole is less than the element to be inserted and insert the element into the hole

Inserting 15 into the heap

1



2



- 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

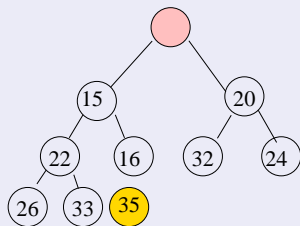
Delete Minimum

This is performed as follows:-

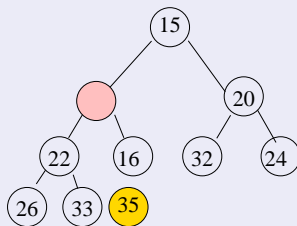
- Remove the minimum element, which will be at the root, creating a hole, and decreasing the size of the heap by 1, remembering the last element in the heap
- If last element is smaller than the children of the hole, insert it, and end
- Move the hole down by moving up the smaller of the two children until last element can be placed in the hole (which may be at a leaf)
- Move the old last element into the hole

Delete Minimum Example

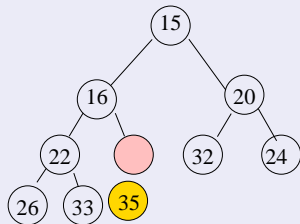
1



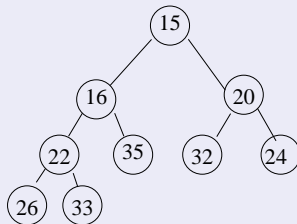
2



3

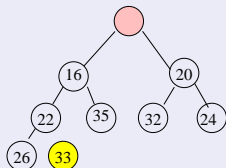


4

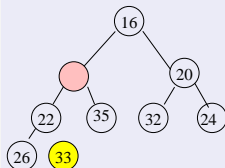


Delete next lowest

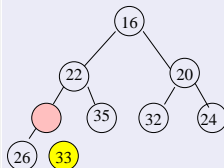
1



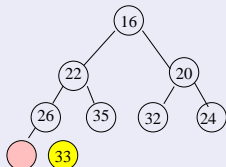
2



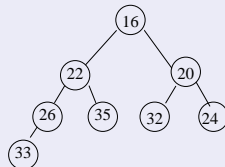
3



4

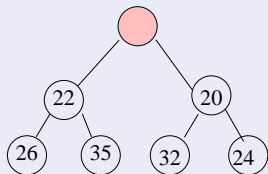


5



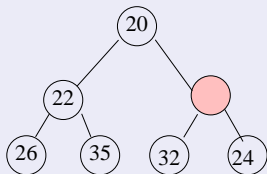
...and the next lowest

1



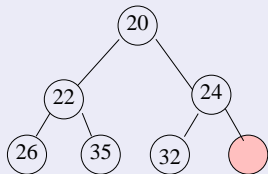
33

2



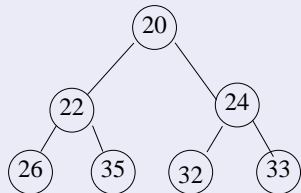
33

3



33

4



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.

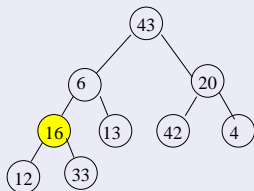
Building Heap example

For example:

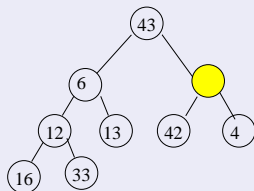
index	0	1	2	3	4	5	6	7	8	9	10
value		43	6	20	16	13	42	4	12	33	

Building Heap example

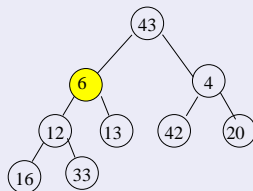
1



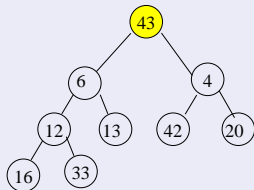
2



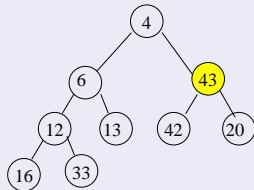
3



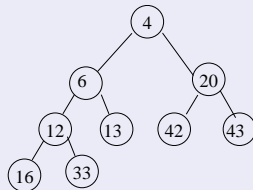
4



5



6



Using this technique followed by removing the smallest element from the heap until it is empty, gives an $O(n \log n)$ 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.

An interface for Priority Queues

```
public interface PriorityQueue
{
    boolean insert( Comparable x );
    Comparable findMin( );
    Comparable deleteMin( );
    boolean isEmpty( );
    void makeEmpty( );
    int size( );
}
```

Binary Heaps in Java

This code is largely taken from Weiss, with some minor modifications

```
public class BinaryHeap
{
    private static final int DEFAULT_CAPACITY = 100;

    private int currentSize;      // Number of elements
    private Comparable [ ] array; // The heap array
```

```
/**
 * Construct the binary heap.
 */
public BinaryHeap( )
{
    currentSize = 0;
    array = new Comparable[ DEFAULT_CAPACITY + 1 ];
}
```

```

/**
 * Construct the binary heap from an array.
 * @param items the initial items in the binary heap.
 */
public BinaryHeap( Comparable [ ] items )
{
    currentSize = items.length;
    array = new Comparable[ items.length + 1 ];

    for( int i = 0; i < items.length; i++ )
        array[ i + 1 ] = items[ i ];
    buildHeap( );
}

```

```

/**
 * Insert into the priority queue.
 * Duplicates are allowed.
 * @param x the item to insert.
 */
public boolean insert( Comparable x ) {
    if( currentSize + 1 == array.length )
        doubleArray( );
    // Percolate up
    int hole = ++currentSize;
    array[ 0 ] = x;

    for( ; x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
    return true;
}

```



```
/**
 * Find the smallest item in the priority queue.
 * @return the smallest item.
 */
public Comparable findMin( )
{
    if( isEmpty( ) )
        throw new RuntimeException( "Empty binary heap" );
    return array[ 1 ];
}
```

```
/**
 * Remove the smallest item from the priority queue.
 * @return the smallest item.
 */
public Comparable deleteMin( )
{
    Comparable minItem = findMin( );
    array[ 1 ] = array[ currentSize-- ];
    percolateDown( 1 );

    return minItem;
}
```

```
/**
 * Establish heap order property from an arbitrary
 * arrangement of items. Runs in linear time.
 */
private void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

```
/**
 * Test if the priority queue is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return currentSize == 0;
}

/**
 * Returns size.
 * @return current size.
 */
public int size( )
{
    return currentSize;
}
```

```
/**  
 * Make the priority queue logically empty.  
 */  
public void makeEmpty( )  
{  
    currentSize = 0;  
}
```

```

/**
 * Internal method to percolate down in the heap.
 * @param hole the index at which the percolate begins.
 */
private void percolateDown( int hole )
{
    int child;
    Comparable tmp = array[ hole ];
    for( ; hole * 2 <= currentSize; hole = child ) {
        child = hole * 2;
        if( child != currentSize &&
            array[ child + 1 ].compareTo( array[ child ] ) < 0 )
            child++;
        if( array[ child ].compareTo( tmp ) < 0 )
            array[ hole ] = array[ child ];
        else
            break;
    }
    array[ hole ] = tmp;
}

```

```
/**
 * Internal method to extend array.
 */
private void doubleArray( )
{
    Comparable [ ] newArray;

    newArray = new Comparable[ array.length * 2 ];
    for( int i = 0; i < array.length; i++ )
        newArray[ i ] = array[ i ];
    array = newArray;
}
```

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* than 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.

Seen implementations of these before

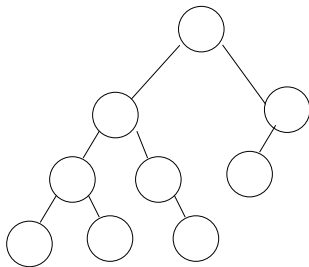
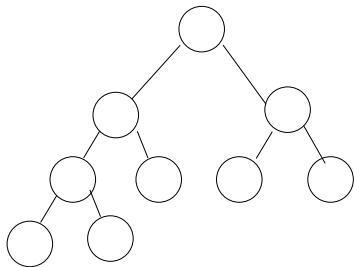
- If we adopt a naïve approach to insertion and the data is pre-sorted, the cost of building a tree becomes *quadratic* in the size of the data (Why?)
- One solution is to attempt to keep the tree *balanced*
- There are several possible definitions of the term *balanced*, one is

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

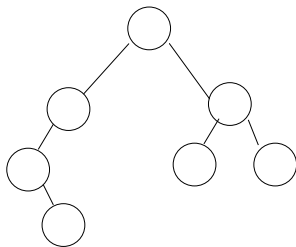
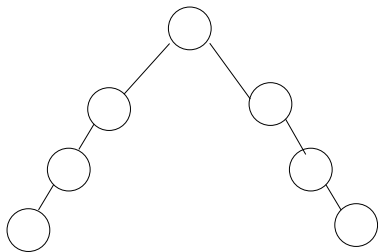
This is another type invariant

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

For example
These trees are balanced



These trees are not



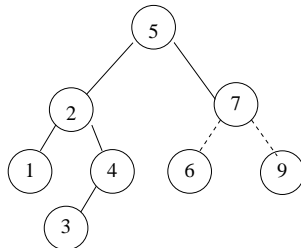
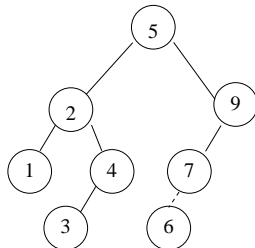
- The problem is that when a new node is inserted, or an old one deleted, the tree can become unbalanced
- When inserting a new node in an AVL tree, find a place to put it in the usual way
- Then check the tree to be sure that it is still balanced
- If tree has lost the AVL property, only nodes between inserted element and root can have balance destroyed. (Why?)

AVL trees: restoring the balance

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

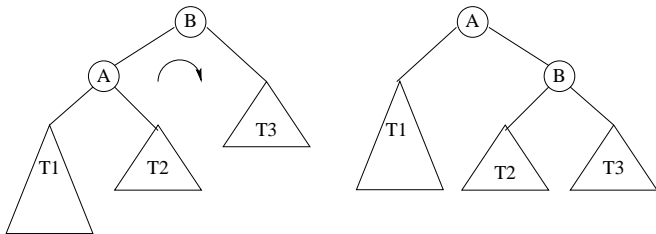
First unbalanced ancestor of new node is node containing 9

Can restore balance by *rotating* unbalanced subtree.

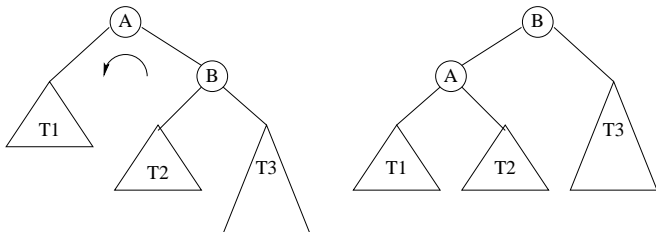


- Suppose n is the first node (going up) which is unbalanced.
- Four different possibilities
 - Imbalance due to insertion in left subtree of left child
 - Imbalance due to insertion in right subtree of right child
 - Imbalance due to insertion in right subtree of left child
 - Imbalance due to insertion in left subtree of right child

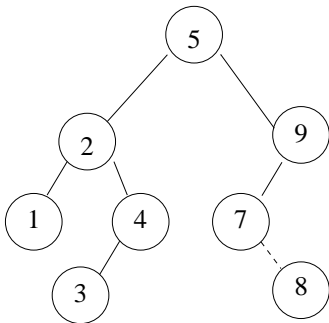
Single right rotation, imbalance in left subtree of left child

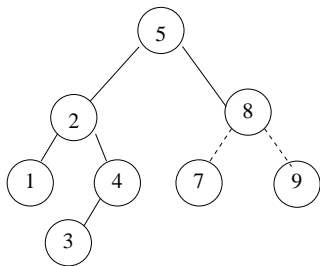
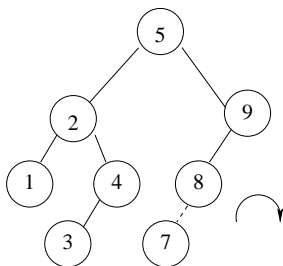
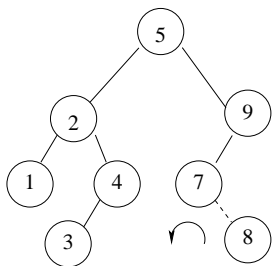


Single left rotation, imbalance in right subtree of right child

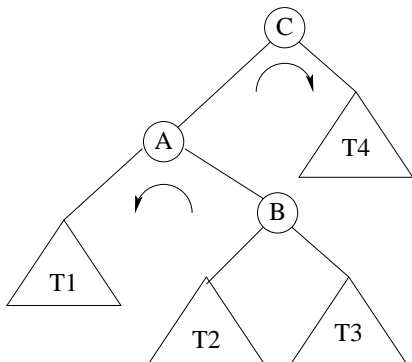


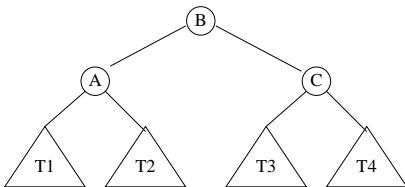
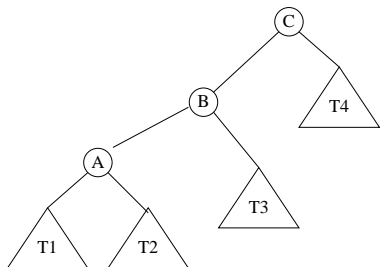
Now if we insert 8 into the tree below, unbalances tree
First unbalanced ancestor of new node is again node containing 9
Need *two* rotations to restore balance



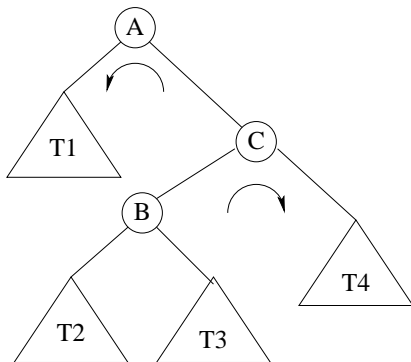


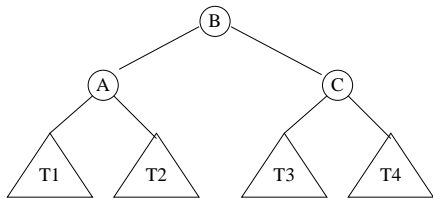
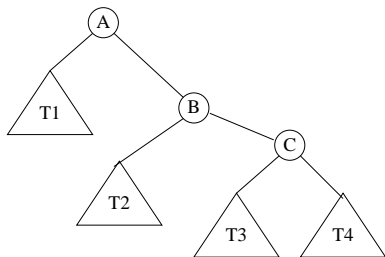
Double left-right rotation, imbalance on right subtree of left child





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

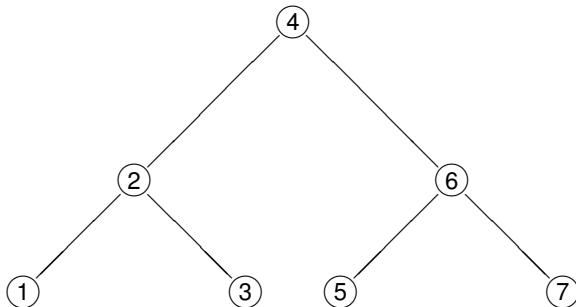




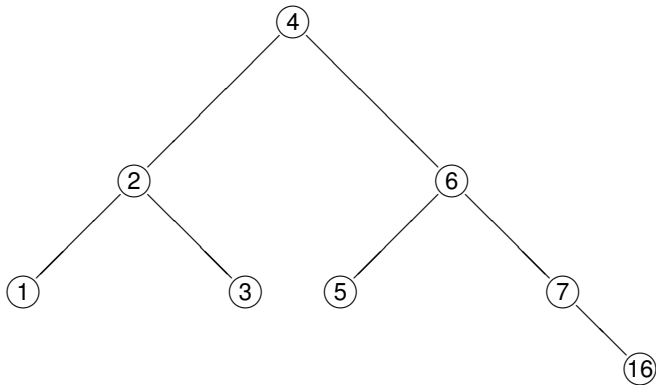
- Balancing algorithm performs at most 1 constant time operation on each of the ancestors of the unbalanced node.
- So, restoring the AVL property to the tree is an $O(\log N)$ operation.

AVL Trees: An example

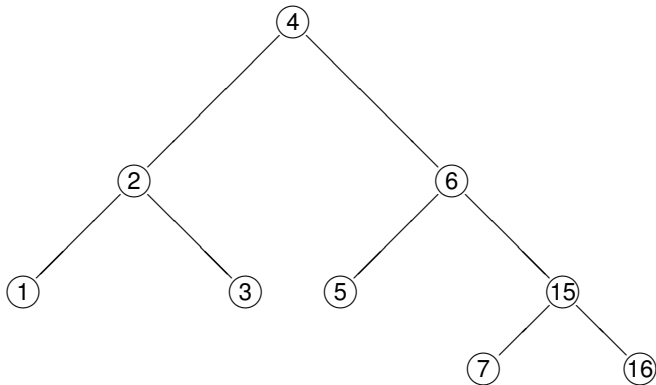
Starting tree



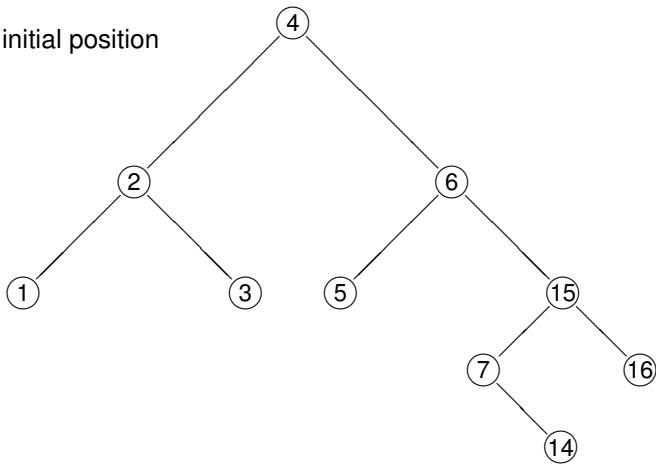
Insert 16



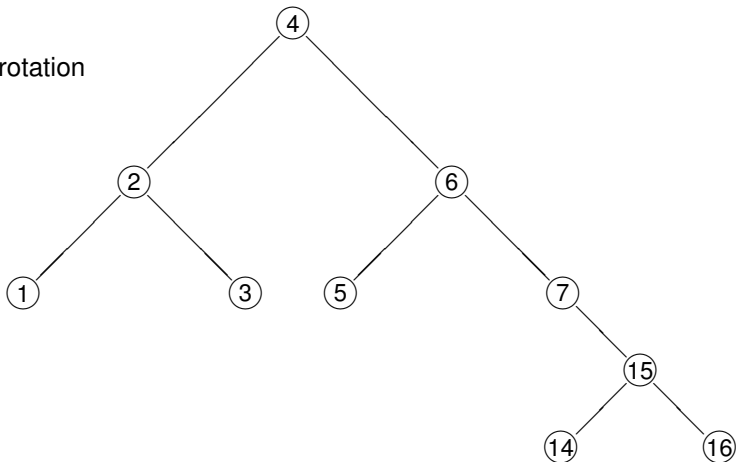
Insert 15



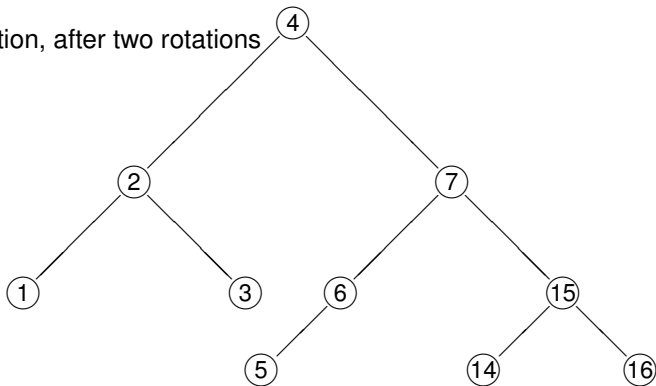
Insert 14, initial position



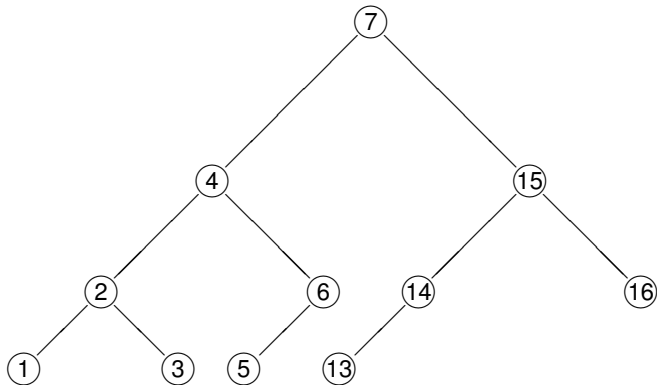
After one rotation



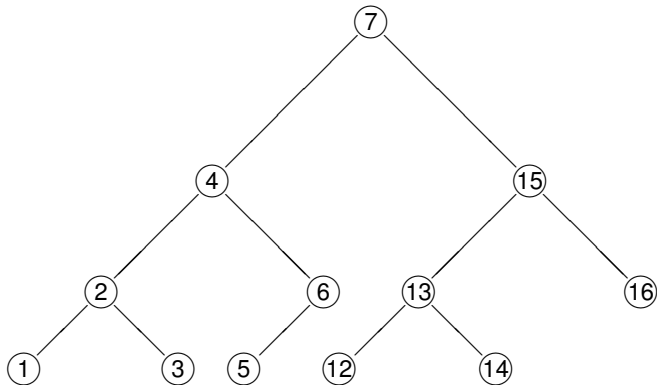
Final position, after two rotations



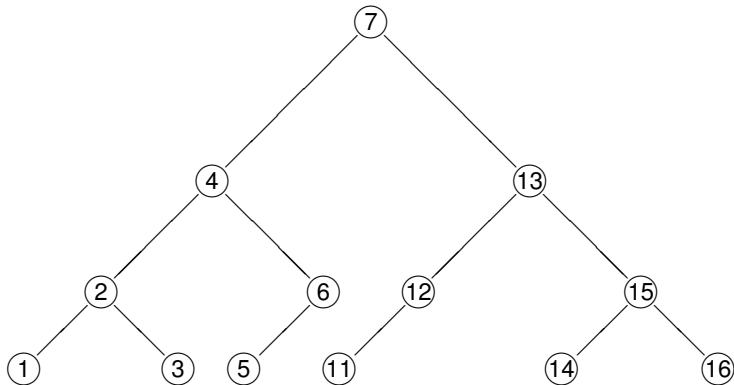
Insert 13



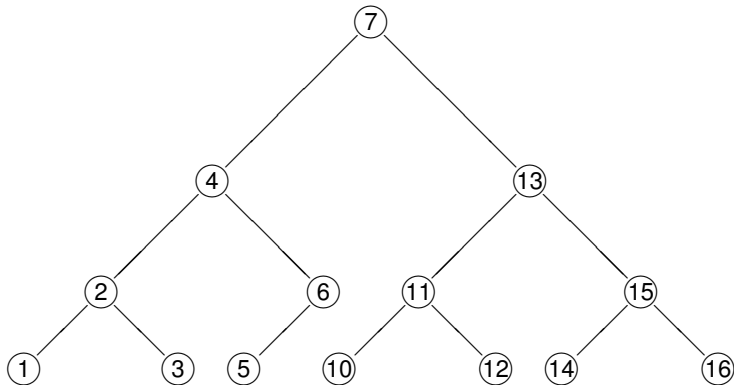
Insert 12



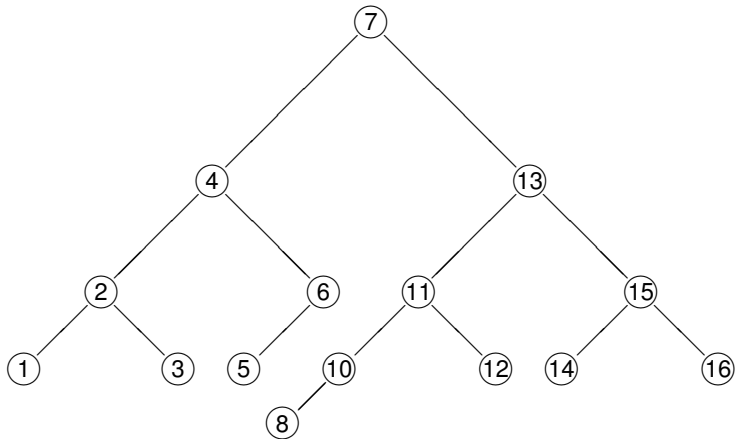
Insert 11



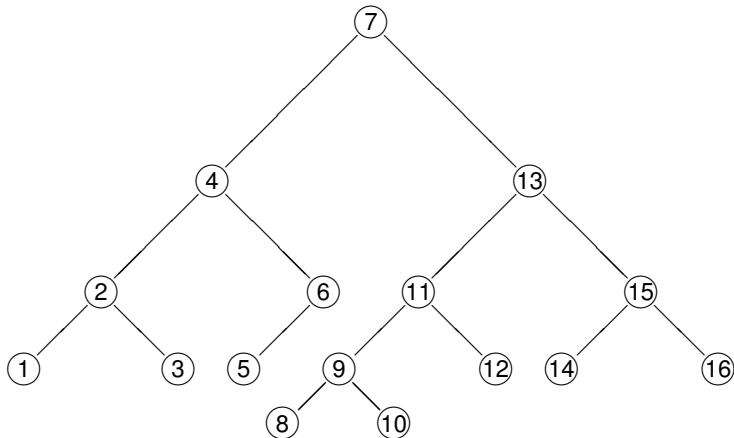
Insert 10



Insert 8



Insert 9



2-3 Trees

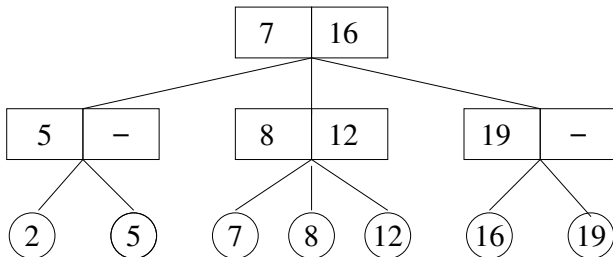
- 2-3 trees are search trees that maintain their balance by relaxing the *structural* constraint of being a *binary* tree.
- Some nodes have 2 children and some have 3, hence the name

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

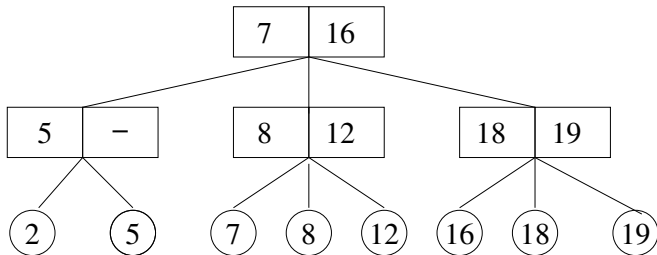
- The root is either a leaf or has either 2 or 3 children
- All data is stored at the leaves.
- All leaves are at the same depth

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

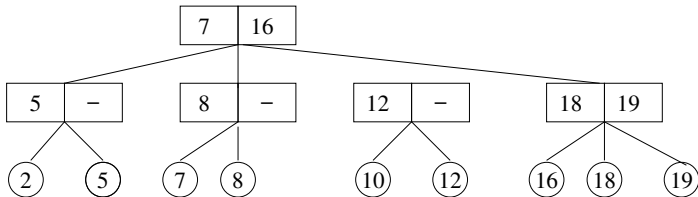
The type invariant is completed with the following constraint
In each node, we store the value of smallest leaf in tree of second child, and value of smallest leaf in tree of third child, if there is one



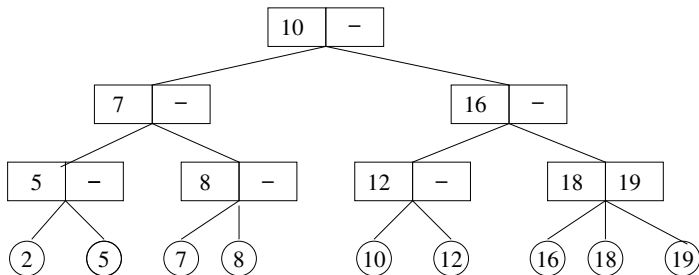
Insert 18



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



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