7-trees-and-graphs

Contents

        * Height
        * Containment
        * Ancestor
        * Comparison
        * Subtrees
        * Size
        * Random
        * Binary search trees
        * Binary heaps
        * Tries
        * B-Trees
    - Worker&Iterator approach
- Graphs
    - Representations
        * Class & Nodes
        * Adjacency Matrices
        * 2D Flags Array
    - Creation
    - Traversing
        * BFS
        * DFS
        * Dijkstra
        * A-star
- Sorting
    - Topological sort

Introduction

Trees

They are a recursive structure, which can be defined by having a node with n-children. Each of those children can also have n-children and so on. Nodes might have a link to their parent but that is strictly implementation dependent and is not required. Important to note:

- trees have no loops within them, otherwise they would be graphs
- each tree has a root node, which has no ancestors
- each node in the tree including the root might have 0 or more children
- a tree with a max of n-number of children is called n-ary tree, a node with 2 children, for example would be 2-ary tree, or binary tree, with 10, 10-ary tree
- a node without children is called a leaf node, the root can be both a leaf node, and a root in a tree
- referencing the node's parent while iterating through the tree is usually implemented using recursion

A sample representation of a tree node might look like this, note that a tree class is rarely needed, it does not really add any significant value, since passing the root Node around, to represent the tree is simply enough, a Tree class might be useful to attach an interface to the tree, actions, which might be done on a tree, but that is usually just it.

An N-ary tree node could look like this

```
    class Node {
        Integer value;
        Node[] children;
```

```
}
```

A binary tree node could look like this

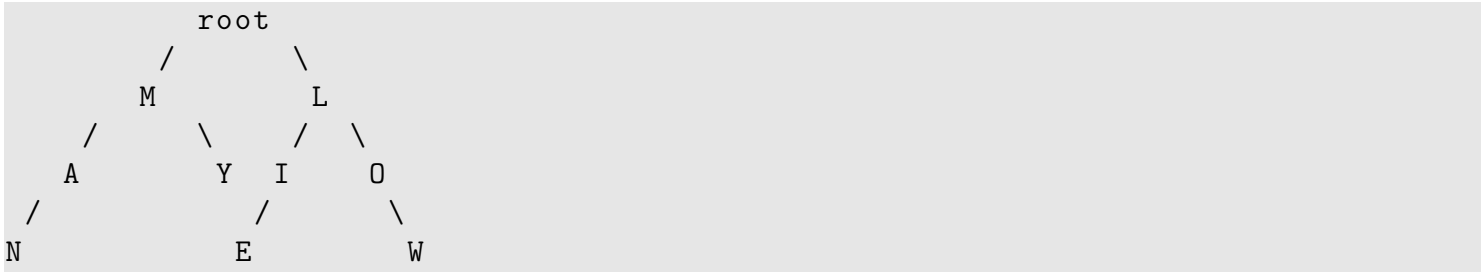
```
class Node {
    Integer value;
    Node left;
    Node right;
}
```

Types

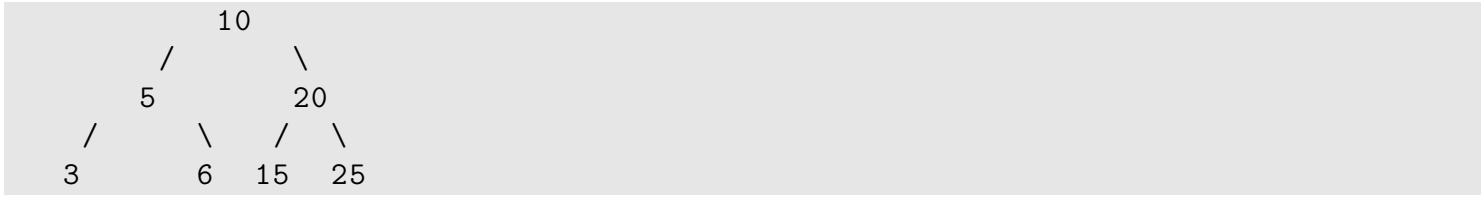
Several common types of trees exist, and it is important to be able to distinguish them, since each have different properties and assumptions that can be made.

Before solving a problem be sure to understand what type of tree is the problem or solution requiring so it can be solved, each tree type has different properties

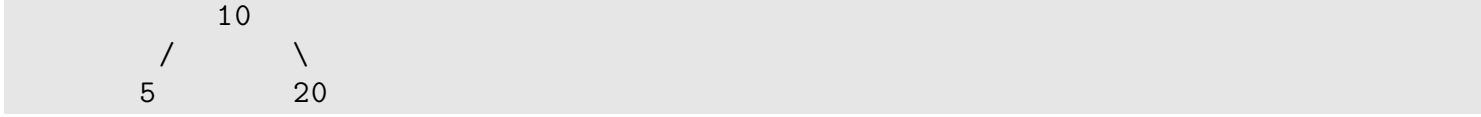
- tries - those are type of n-ary trees, which come up very often, and are usually used to find word prefixes, tries are sometimes used to store the entire English language, each node in the tree has at most 26 children, 1 for each letter of the alphabet. Each path of nodes which forms a valid word, is terminated with special terminating node which denotes that.



- binary trees - these are just trees where each node has at most two children, meaning it can have 0, 1 or 2. Nodes are not arranged or ordered in any meaningful or special way
- binary search trees - these are binary trees which are constructed in a very special way, where the tree nodes are ordered, each node on the left is usually less than or equal to it's parent, each node to the right is strictly bigger than its parent. This condition must be met for all sub-children of a given node, recursively, not just the immediate children of a given node. See in the example below how 10 is not just bigger than 5, but also than 5's children too, 3 and 6



- balanced binary search trees - these are trees where the height of the left and right sub tree do not differ with more than 1 level. Balanced trees might be binary, but any n-ary tree could be really. Example of balanced trees are red black trees, or AVL trees, these are examples of balanced binary search trees. See below, how the height of the left sub tree of the root is 3 and the right is 2, the difference is no more than 1



```
  /   \
 3     6
```

- binary heaps - these are special type of binary trees, similarly to the search trees, but the order of elements is such that either the min element or the max, depending on the type of the tree, is at the top of the tree, in the root node, elements below each node are all bigger (min) or smaller (max) than the root, this is true for each node recursively. See how no matter which child you take below the root, they are all bigger (in this example, for min heap).

```
          1
       /     \
      5       20
   /    \   /   \
  8      9 25   35
```

- complete binary trees - are these trees which where each level of the tree is completely filled, except maybe the last level.

```
          10
       /     \
      5       20
   /    \  /
  3      7 15
```

- full binary trees - are these trees which where each node has either 0 or 2 children, and nothing in between

```
          10
       /     \
      5       20
   /    \
  3      7
```

- perfect binary trees - are these where all interior nodes, have children, the only nodes that have no children are the leaf nodes

```
          10
       /     \
      5       20
   /    \  /    \
  3      7 15    23
```

Creating

A fast way to create a binary tree from a list of items, is to just use the heap approach. What we do is simply start off from the first index, and calculate which elements are children of that index, get that from the array and create a node child subtree, and keep recursing. Keep a base case check for the index, must not exceed the list size.

- left child is (index * 2) + 1
- right child is (index * 2) + 2

```
Node create(Node root, int index, List elements) {
    if (index >= elements.size()) {
```

```
            return null;
        }

        root.value = elements.get(index);
        if (root.value == null) {
            return null;
        }
        root.left = create(new Node(), (index * 2) + 1, elements);
        root.right = create(new Node(), (index * 2) + 2, elements);
        return root;
    }

    Node create(List elements) {
        return create(new Node(), 0, elements);
    }
```

Traversal

There are a few ways to traverse a binary tree, or n-ary tree, usually that involves different visiting permutations of the nodes. For example for a binary tree we can visit the 3 nodes, in 6 different ways, in total

- root, left, right
- root, right, left
- left, root, right
- left, right, root
- right, left, root
- right, root, left

Some of the above have names, since they come in very often, for a binary search tree, we have post, pre and in order traversal

- left, root, right - in order traversal, because the tree nodes would be visited in ascending order.

```
    void order(Node node) {
        if (node == null) {
            return;
        }
        order(node.left);
        visit(node.root);
        order(node.right);
    }
```

- root, left, right - pre order traversal, where root is first then, ordered are visited only the children

```
    void order(Node node) {
        if (node == null) {
            return;
        }
        visit(node.root);
        order(node.left);
        order(node.right);
    }
```

- left, right, root - post order traversal, where children are ordered, then lastly the root is visited

```
    void order(Node node) {
        if (node == null) {
            return;
        }
        order(node.left);
        order(node.right);
        visit(node.root);
    }
```

Height

```
    int height(Node root) {
        if (root == null) {
            // when the node is invalid there is no meaningful height value
               to return for it, but zero
            return 0;
        }
        if (root.left == null && root.right == null) {
            // when the node is a leaf, the height could be only one, the
               height is the node node itself
            return 1;
        }
        // when the node has children, the height is the current height, one,
            plus the max height between the left or right subtrees of
        // the current node
        return 1 + Math.max(height(root.left), height(root.right));
    }
```

Containment

To check if a given node reference, not by value, is contained in a tree, we have to go through all nodes and compare each to the target reference we have. The premise is simple compare by reference, either the left or the right subtrees could contain that node. If we reach a nil node, we can return false, if both paths `check(left)and check(right)` return false, therefore we have not found the node, at least one should return true.

```
    boolean contains(Node root, Node node) {
        if (root == null) {
            // the root does not contain the given node
            return false;
        }

        // the root and the node are the same, yes the initial root contained
            the node
        if (root == node) {
            return true;
        }

        // try go to the left and right of the current root looking for the
           target node
```

```
        return contains(root.left, node) || contains(root.right, node);
    }
```

Ancestor

This is a tricky, but very useful problem to solve, find the first common ancestor of two nodes. Given two nodes at any depth in a tree. To do this we have to realize we have two conditions / states only. The nodes could be in

- both nodes are on the left or right subtree of a given node
- one node is on the right the other on the left or vice versa

We know that if one node is on the right the other on the left, of a given parent node, or vice versa, that this parent node is the common ancestor, otherwise if they are on the same side, we have to go down that side, until they split, and are no longer on the same side

```
        10
       /  \
      3    15
    /  \
  1     4
 /
0
```

If we take a look at the example above, the nodes 0 and 4 have an immediate common ancestor, the node 3. We can see how this is the first node where both 0 and 4 are on both sides of 3, while if we take a look at node 10, 0 an 4 are on the same side of 10 (the left subtree)

What we need to do is simply verify that the given nodes do not lie in the same subtree at the same time of a given node, then we know that node is the common ancestor, note that this works for example if we had the nodes 3 and 0, where the common ancestor would be 10, (not 3)

```
    Node ancestor(Node root, Node first, Node second) {
        if (root == null) {
            // invalid root no common ancestors
            return null;
        }

        // check if the first node is on the left of the root
        boolean firstOnLeft = contains(root.left, first);

        // check if the second node is on the right of the root
        boolean secondOnRight = contains(root.right, second);

        // we know that if both of those variables are equal, then the
           current root is the common ancestor, why ? if we take any two
        // nodes in a binary tree, there exists only one single node, for
           which the two nodes are located on the left and the right,
        // at the same time, and that node is the common ancestor. Note that
           it does not matter if we flip the first and second, so the
        // checks above would return false for both instead, if the check
           above both return false, that simply means the nodes are on
        // the opposite site of what we were checking, but they are on the
           left and right of root still, as long as firstOnLeft ==
```

```
        // secondOnRight, the nodes are on the left and right of current root
           , which (of the nodes first/second) is on which does not
        // matter
        if (firstOnLeft == secondOnRight) {
            return root;
        }

        if (firstOnLeft) {
            // in the case where the first node was on the left, and the
               condition above did not meet, that means first node is on the
            // left but the second one is also on the left, not on the right
               (secondOnRight would be false), so we go to the left
            return ancestor(root.left, first, second);
        } else {
            // the same as above but inversed, if firstOnLeft was false, that
                would automatically mean that secondOnRight was true,
            // therefore the first node is on the right so is the second, go
               to the right subtree
            return ancestor(root.right, first, second);
        }
    }
```

Comparison

There is another problem that comes very often, namely comparing two subtrees, to check if they are equal / the same. To do that we have to traverse both trees at the same time going from the root to the subtrees depth-first or breath-first, does not matter, in either case we have to go through both at the same rate, and keep comparing the nodes, if all nodes are exactly matching and at the same spot in both trees, we can deduce they are equal

```
    boolean compare(Node first, Node second) {
        if (first == null && second == null) {
            // when the two subtrees are null, they are considered equal,
               this base case will also be fulfilled when we reach the bottom
            // of the two trees we compare at the same rate / time / steps,
               knowing we have finished both trees at the same time
            return true;
        }

        if (first == null && second != null) {
            // we have reached the bottom of the first tree, but the second
               did not, meaning that moving at the same pace, the first
            // tree was shorter / smaller than the second one
            return false;
        }

        if (second == null && first != null) {
            // we have reached the bottom of the second tree, but the first
               did not, meaning that moving at the same pace, the second
            // tree was shorter / smaller than the first one
            return false;
```

```
        }

        // compare the values of the roots
        if (first.value.equals(second.value)) {
            // compare the roots, only if the roots have the same value, can
               we then continue down, with the same rate/steps into both
            // subtrees, what we do here is just drill down equally deep and
               on the same side for both input trees, if all roots match
            // up on each side then we can assume they are equal, imagine it
               as if we had compared two strings, we compare current char,
            // then move right one character, until the string ends
            return compare(first.left, second.left) && compare(first.right,
               second.right);
        }
        // the root value did not match, meaning we can terminate, they are
           not equal here, even though there still might be more
        // children to look at
        return false;
    }
```

Subtrees

Another common problem is to find if a subtree exists in a given tree, usually the subtree is much smaller than the main tree itself. This is done by leveraging `compare`, the check function first drills down to the bottom of the main tree, and then starts to run `compare` while unwinding the recursion. Going to the bottom first approach or top first does not matter, all that matters is that we traverse the entire tree, and we find the subtree in either the left or right sub branches. Note the condition which is `check(left)|| check(right) || compare(main, sub)`.

```
    boolean check(Node main, Node subtree) {
        if (subtree == null) {
            // a null subtree, means that it is always a part of the main
               subtree, the main subtree contains many null subtrees, think
            // of the leaves which have 'null' subtrees or in other words the
                children of the leaves
            return true;
        }
        if (main == null) {
            // the main subtree must not be null, otherwise there is nothing
               to check against, and the comparison is invalid
            return false;
        }

        // we first drill down the left and right branches, then the post
           recursive call will compare from the bottom - up approach,
        // either bottom-up or down will work, we have selected to use bottom
           -up in this example solution
        return check(main.left, subtree) || check(main.right, subtree) ||
           compare(main, subtree);
    }
```

Size

To dynamically calculate the size, or in other words the number of nodes a given tree, and to avoid storing that into the node itself as additional information we have to care about and update we can leverage the following approach

```
int size(BinaryNode<Integer> root) {
    // a null node has no size really, so we return 0 by default
    if (root == null) {
        return 0;
    }
    // each valid node, has at least a size of 1 + the size of the left or
       right
    // subchildren, whichever it has, if it has any at all
    return 1 + size(root.left) + size(root.right);
}
```

Random

A common problem, is selecting a random node from a binary tree, where the each node in the tree has an equal chance to be selected. What we have to consider is how to proportionally select a random element, from a tree, which might be imbalanced.

What we need to know is how many nodes, each of our subtrees for the current root has. Let us take the example below, the tree has a total of 7 elements, however the left sub tree is quite a bit heavier in comparison to the right one. To be able to select a random node with equal probability, we have to take this into account

```
        10
       /  \
      3    15
    /  \
  1     4
 / \
0   2
```

We start off with a random number (say 1, that is index 1, elements run from 0 to 6, for total of 7 elements). We can check the size (num of nodes in that subtree) of our left subtree, starting from 10, our left subtree size is 5, the right subtree is 1, the index we have picked is less than the size of the left subtree size, therefore we go to the left, for root 10 element indices span from left[0-4] root[5] right[6-6]

Next is the root 3, its left size is 4, its right size is 1, the index is still less, indices for root 3 span from left[0-2] root[3] right[4-4].

We will keep going down, to the left, to 1, where the size of the left is 1, the right is 1, the indices of the elements for root 1 span from left[0-0] root[1] right[2,2], now here our input random index is equal to the 'index' of our root.

Eventually if the random index is not greater than the total number of elements in the tree - 1 (which is a must) then the function will fall down into this sole base case i.e. where `index == root_index`

```
    Node random(Node root, int index) {
        // this is not a valid case we can simply return nil
        if (root == null) {
            return null;
        }
```

```
        // this must always be true, for each invocation otherwise the index
           is not within the correct boundaries, this is a sanity
        // check, but is actually deduceable, if we know we are given a valid
            index always, the index will always be less than the total
        // number of elements / size of the current subtree and always
           greater or equal to 0
        assert (index >= 0 && index < size(root));

        // find how many elements there are in the left subtree, if we know
           how many elements we have on the left, and we know that the
        // input index will never exceed the total number of elements for the
            current index (which we guarantee below by clamping the
        // index when going to the right) we can know with certainty in which
            direction we have to go, when the index value is within
        // these 3 ranges below
        // - [0 - (leftSize - 1)] - go to the left
        // - [leftSize - leftSize] - exactly at the root
        // - [(leftSize + 1) - (total subtree size)] - go to the right
        int leftSize = size(root.left);

        if (index < leftSize) {
            // if the index is within the number of elements in the left
               subtree go down the left subtree path, the index does not need
            // to be touched for the current left subtree, since the range of
                the index falls exactly in elements within the current
            // left sub tree, we adjust the index only when we have to go to
               the right, and we go to the right only when the index is
            // outside the boundaries of left and root i.e between [(leftSize
                + 1) and (total subtree size)]
            return random(root.left, index);
        } else if (index == leftSize) {
            // if the index is exactly the left size, that means this is the
               root item exactly, left indexed items are only from [0 and
            // leftSize - 1], index equaling leftSize means we have to pick
               the root, at some point at some level this is the base case
            // which we will hit for sure, while going down, the index will
               become equal to the root 'index' and we stop and return
            return root;
        } else {
            // if the index falls in the right subtree, before going down the
                right subtree, we have to normalize the index, while the
            // current index is valid for the current tree, when we go down
               the right one, we have to clamp the index to be valid for the
            // right sub tree, to do that we have to calculate what the
               current index corresponds to in terms of the right sub tree,
               this
            // we can deduce by knowning that (lefSize + 1) is the lower
               bound index for the right sub-tree, therefore we can subtract
               that
            // from the current index. This will give us an index which is in
                the index space of the right subtree for the current root,
```

```
            then
        // we can call the function with the new index and the rigth
            subtree as the new root.

        // For example if the tree at this moment has 10 elements, and
            the target index was 9, the left size was 6, and we account
        // for the 1 element being the current root, then we can deduce
            that the right subtree has 3 elements, therefore the new
        // index must be in range [0-2], and it will be, indeed : 9 - (6
            + 1) = 2
        index = index - (leftSize + 1);
        return random(root.right, index);
    }
}

// initial invocation, note index is between [0, size - 1]
random(create(elements), random(0, elements.size() - 1))
```

Binary search trees

The most simple implementation of a binary search tree, is one that simply conform to the rule, and does not try to keep the tree balanced, which in some cases can lead to linear search time complexity, since the tree can get malformed into a linked list, which can very often happen, when the tree is not kept balanced.

For example, adding a new element which is always bigger or always smaller than the previous added one, would form a linked list of nodes, making only one side of the tree have nodes

Here first 10 was inserted, then 5, then 3, and if we keep adding smaller and smaller elements, 2, 1, 0 etc, the tree would keep forming a linked list of nodes. These elements can be re-structured (called balancing) to make the structure more like a tree, which would have a search of O(log(n)), instead of the O(n) which is what it is going to be if the tree looks like a list.

```
      10
    /
  5
 /
3
```

Creating One way to quickly create a binary search tree given a sorted array of elements, is to simply binary partition the array. What we do is partition the tree in half. What is important to note here is how we control start and end. They are based off of the root index.

- the left subtree is created from the left half of the array, i.e all elements from start to just before the current root's index in the array
- the right subtree is create from the right half of the array, i.e all elements from the current root's index + 1, to the end of the array

```
Node create(List elements, int start, int end) {
    // calculate the diff between start and end, note that start is index
        , and end is a size, therefore the only valid difference
    // between them can be at most 1, for example when start = 0 and end
        = 1, that means the sub-array we have to look at has
```

```
        // only one element, the one at index 0, end tells us the number of
           items in the sub-array, the start is the starting index
        int diff = end - start;

        if (diff <= 0) {
            // if the diff is invalid just return here and stop recursion,
               this would imply that the start (index) overlaps with end
            // (size), which is not a valid state, meaning we can stop,
               nothing more to subdivide
            return null;
        }

        // the current midpoint index is the diff divided by two, offset with
            the start index. The diff / 2 would give us the relative
        // mid point element, for the current subdivided sub-array, but to
           make it absolute in the context of the source input array we
        // have to add the current start index to that.
        int index = start + (diff / 2);

        // create the new root node, the value for that new root node would
           be the one at 'index', or in other words the midpoint
        // element in the array, for the current range of start and end
        Node node = new Node();
        node.value = array[index];

        // the left subtree for the root node start from start up until, but
           not including the current node i.e start -> index, remember
        // the end argument to this function is not an index, it is a size,
           so the range of elements for the left subtree would be
        // the inclusive range [start:index-1]
        node.left = create(array, start, index);

        // the right subtree for the root node start from start up until, but
            not including the current node i.e index + 1 -> end,
        // remember the end argument to this function is not an index, it is
           a size, so the range of elements for the right subtree
        // would be the inclusive range [index+1:end-1]
        node.right = create(array, index + 1, end);

        // return the node
        return node;
    }

    create(elements, 0, elements.size());
```

Validating To validate a tree is a binary search tree, to do this we use an approach where with each path we take to the left or the right we narrow down the starting min and max rnages. Initially they start off as nil, at the very root of the tree.

When we go to the left, we know that all elements to the left must be smaller than our root, therefore we set the max to be the current root, when we go to the right we set the min to be the current root, as elements

to the right cannot be smaller than the current root.

Going down left and right the min and max represent the path we came from, the values of the roots we came from, and if all of those child nodes in the left AND right subtrees up until the very leaves do match the condition `root.value > min && root.value < max`, we know that the BST property is met.

```
boolean check(BinaryNode<Integer> root, Integer min, Integer max) {
    if (root == null) {
        // this is a weird base case, but if we reach this point it means
            we have traversed the entire tree and it has kept the bst
        // property up until the very bottom
        return true;
    }

    // we know that the current root value must not be smaller than min
       and not bigger than max range, if it is either, then the
    // tree does not meet the bst rules, we return false
    if ((min != null && root.value < min) || (max != null && root.value >
        max)) {
        // the current root value was not within the max or min ranges,
           therefore it does not meet the rules of a bst, return false,
        // the and (&&) condition below would guarantee that for this
           tree as a whole the check for bst would return false
        return false;
    }

    // the trick here is to see how we gradually restrict the max/min
       values while we go down. When we start going left/right down
    // the tree, the min/max ranges would keep narrowing down, note that
       both subtrees must evalute to true, i.e must conform to the
    // rules of bst, therefore note the and (&&) sign below
    // - the left sub-tree we know that values there can not be bigger
       than root.value, therefore we set max to be root.value
    // - the right sub-tree we know that values there can not be smaller
       than root.value, therefore we set min to be root.value
    return check(root.left, min, root.value) && check(root.right, root.
       value, max);
}

check(root, null, null);
```

Max Finding the max element in a BST is simply following all right subtrees from the root of the tree, the max element would be contained in the right most child / leaf node starting from the root

```
Node max(Node root) {
    // to find the max node, simply drill down the right subtree of the
       current node, until there is no more right nodes left
    while (root != null && root.right != null) {
        root = root.right;
    }
    return root;
}
```

Min Finding the min element in a BST is simply following all left subtrees from the root of the tree, the min element would be contained in the left most child/leaf node starting from the root

```
Node min(Node root) {
    // to find the min node, simply drill down the left subtree of the
       current node, until there is no more left nodes left
    while (root != null && root.left != null) {
        root = root.left;
    }
    return root;
}
```

Insertion This process is very similar to searching a node with a value, we go down down 3 paths, either the value is bigger, smaller or equal to the current root's value,

- if bigger - go to the right
- if smaller - go to the left
- if equal - terminate, duplicates are not allowed

If we never find an equal element, the recursion will eventually terminate with root == null, meaning we have drilled down the tree into a spot which conforms to the rule of BST, and is empty, create a node, return it up the call stack.

The return is crucial, it would correctly attach the new node to it's immediate parent, since we recursed down, and now return back the same way, due to the call stack unwinding.

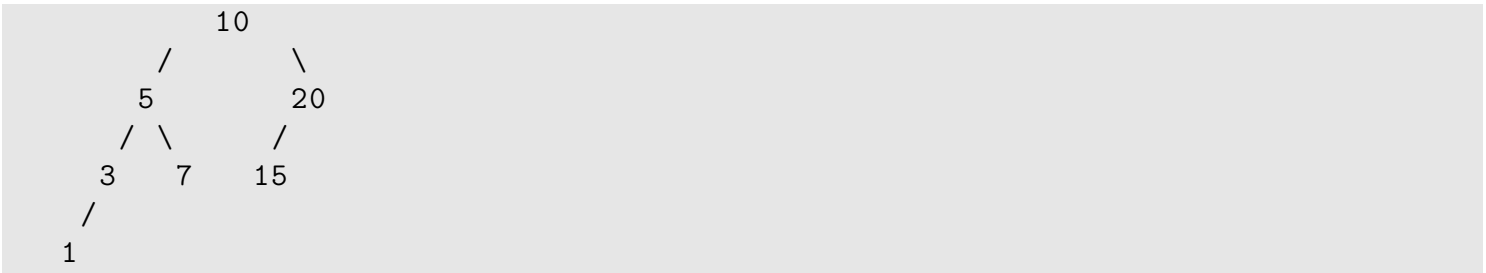
```
Node insert(Node root, int value) {
    if (root == null) {
        // if we drill down till this point it means that the correct
           spot for the new element was found, and we create the node and
        // return it, in the assignments below that node will be assigned
            correctly to it's parent node, in other words, the
        // recursive calls below took right/left paths such that they
           lead us to a null child, which is the spot at which this new
        // node must be linked, returning it here, and assignment below
           will ensure that linkage is made to the parent of this new
        // node, to either the right or the left child.
        Node node = new Node();
        node.value = value;
        return node;
    }

    if (value > root.value) {
        // we have to go down to the right, this recursive call will
           either return the same node, if the value is equal to
        // root.right, or will drill down until there is not root.right
           anymore, at which point the node will be created with the
        // new element and returned back
        root.right = insert(root.right, value);
    } else if (value < root.value) {
        // we have to go down to the left, this recursive call will
           either return the same node, if the value is equal to
```

```
            // root.left, or will drill down until there is not root.left
               anymore, at which point the node will be created with the
            // new element and returned back
            root.left = insert(root.left, value);
        }
        // return the node always, we only recurse inside the if/else, this
           case here also handlers equals, but in all 3 cases we want to
           return
        // the original node anyway
        return root;
    }
```

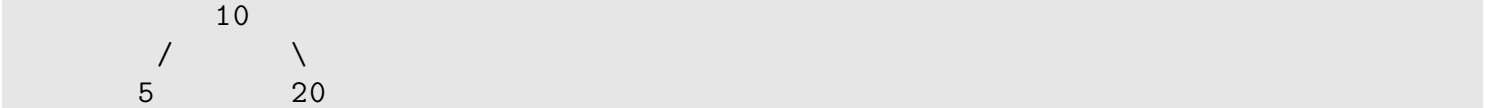
Delete Deletion is a bit more convoluted, but can still be split into a few base cases. However first the element to be deleted has to be located in the tree, only if it exists can we proceed to try to delete it.

- if deleting a node without children, simply return null
- if deleting a node with only a left subtree, return that left subtree
- if deleting a node with only a right subtree, return that right subtree
- if deleting a node with both subtrees existing, two different options
  - drill down the left subtree and find the max element of root.left
  - drill down the right subtree and find the min element of root.right
  - found element will either be the root.left (max case) or root.right (min case), when they have no right (max case) or left (min case) child nodes, or the most right leaf node (max case), or the most left leaf node (min case).
  - the found element's value we swap with the root, the links of the found element have to be maintained, and not lost, we have one of two cases
    - in the max case the found element would not have any right subtrees, only left ones,
    - in the min case the found element would not have any left subtrees only right ones
  - attach the left/right of the found to the parent of the found element

Deleting the node with value 5, we have to first find the node we want to delete, in this case let's assume we have done that. (This example does not cover all cases, it is just a small showpiece)



It has two children, we choose to find the max node from the left subtree of the node to delete (5), which is the node with value 3. It is the max since it has no right children, if it had we would have gone down to the right



```
    / \     /
   3   7   15
  /
 1
```

The found node is immediate child of the node to delete (5). Now we have to detach the found node (3) from the tree, therefore node (5).left = (3).left (Since we went with max case, we know that there could be nothing to the right of node 3, otherwise we would have gone down that route, but there might be to the left)

```
        10
      /    \
     5      20
    / \    /
   /   7  15
  /
 1
```

Finally we set the value of the found node (3) to the value of the node we want to "delete", notice how we did not delete the node which held the value 5, but exchanged the values, and deleted the child of the node we want to delete (5). The BST property is maintained

```
        10
      /    \
     3      20
    / \    /
   /   7  15
  /
 1
```

```
Node delete(Node root, T value) {
    if (root == null) {
        // drilling down did not find the element, simply abort the
           deletion, at this point we have reached the end of the
           possible
        // path in the tree and the element was not present, the
           recursion will unwind the call stack till the top, returning
        // the nodes unmodified
        return null;
    }

    if (value > root.value) {
        // drill down the right subtree until we find equal element, note
            the assignment, this allows us to propagate changes to
        // the correct subtree, allowing us to re-link a non-immediate
           child node with the current root, sub recursive calls can
        // return child nodes deep in the tree, and assign them to the
           current node, making linking very easy, without having to
        // keep track of parent nodes.
        root.right = delete(root.right, value);
    } else if (value < root.value) {
        // drill down the left subtree until we find equal element, note
           the assignment, this allows us to propagate changes to
```

```
        // the correct subtree, allowing us to re-link a non-immediate
           child node with the current root, sub recursive calls can
        // return child nodes deep in the tree, and assign them to the
           current node, making linking very easy, without having to
        // keep track of parent nodes.
        root.left = delete(root.left, value);
    } else {
        // at this point the root would point at an element with value
           which must be equal to the target value, why ?, we will be
        // here only if the value is equal to the root, any other case
           will either exit, if node == null, we have drilled down to
        // the bottom of the path, if value > root.value or value < root.
           value then we will keep recursing, therefore we reach this
        // point only when value equals root.value
        if (root.left == null && root.right == null) {
            // the root with the value has no children, therefore, we
               return null, this return is very crucial, it will return
               to
            // the previous recursive call where the root will reference
               the parent of the current root node, therefore without
            // using any parent pointers in the tree, we will correctly
               assign to the left or right of the parent of this root,
            // null, detaching the current node that was found to equal
               the value
            return null;
        } else if (root.left != null && root.right == null) {
            // the same as above, but instead, we know that we have only
               left subtree, therefore we link the left subtree of the
            // current root, which equals the target value, to the parent
                of the current root, this will essentially attach the
            // current root's left subtree to the left or right subtree,
               based on off of where we came, to the parent of the
               current
            // root / node
            return root.left;
        } else if (root.right != null && root.left == null) {
            // the same as above, but instead, we know that we have only
               right subtree, therefore we link the right subtree of the
            // current root, which equals the target value, to the parent
                of the current root, this will essentially attach the
            // current root's right subtree to the right or right subtree
               , based on off of where we came, to the parent of the
            // current root / node
            return root.right;
        } else {
            // we have both left and right subtrees for the current root
               for which the value is found to be equal, what we do here,
            // two options are available, in that scenario, both equal,
               and either one is fine
            // - find the min element from the left subtree
            // - find the max eleemnt from the right subtree
```

```
            Node curr = root.left;
            Node prev = root;

            // we choose to find the max element from the left subtree,
               that is done by starting off from the root.left, and
            // drilling down to the right, we can choose the other option
                find the min element in the right subtree it does not
            // matter, it has some effect on how we do the checks inside
               the ifs below though if we go that route, but they are
            // symmetric
            while (curr != null && curr.right != null) {
                // prev will always move one step behind curr, and prev.
                   right will always point at curr at any point inside or
                // outside this loop, we need to know prev, to detach
                   curr from it, and attach the curr's children to prev
                   instead
                prev = curr;
                curr = curr.right;
            }

            if (prev == root) {
                // we have not moved at all, the while did not loop,
                   meaning that there is no right nodes to the left or
                   root,
                // meaning that the prev will point to root, (what it was
                    initially set to), therefore we can safely detach
                   curr
                // node, by assigning it's left subtree to the prev (
                   which still points at root) left subtree, effectively
                   detaching
                // it from prev, because root.left == prev.left == curr,
                   so far, no longer after we link root.left == prev.left
                    to
                // curr.left
                prev.left = curr.left;
            } else {
                // we have drilled down to the right, meaning that curr
                   must be the max node, that node can possibly have a
                   left
                // subtree (with nodes smaller than it), what we do to
                   detach it ? assign to it's parent/previous node's
                   right
                // subtree the left subtree of the current's left subtree
                    (this will detach curr itself, without loosing curr's
                // links to it's child nodes, because prev.right == curr)
                    if it has any it will be correctly re-linked (but it
                   has
                // no right one that is for sure, since we drill down to
                   that condition, see while loop above)
                prev.right = curr.left;
            }
```

```
                // copy the value of the curr node over to the root, the curr
                    node will be the one which replaces the root's value,
                // since the root's value is the one we want to delete in the
                    first place, but it had to be replaced with a correct
                // value computed from it's children first. The curr.value
                   will maintain the bst property, since we found the max
                   value
                // from the root.left subtree, and replaced the root.value
                   with it, therefore the property for the subtrees of root,
                   and
                // their values are maintained
                root.value = curr.value;
            }
        }
        return root;
    }
```

Searching This is in any case a case which is covered in the delete and insert operations, when inserting the found case means we can not insert in the tree, in delete operation the found case means we found the node to delete. In the search case the found case would simply return the node it had found

Use iterative method, simply because it is simpler to return a value, from it, in this case we only drill down one single path, without doing any tree modifications or complex operations.

```
Node search(Node root, Integer value) {
    // drill down until either a node with a value equaling value is found,
       or
    // there is no more nodes to look at, while conforming to the BST rules
    while(root != null) {
        if (value > root.value) {
            // drill down the left subtree when the value is bigger than the
               root
            root = root.left;
        } else if (value < root.value) {
            // drill down the right subtree when the value is smaller than
               the root
            root = root.right;
        } else {
            // there was a root node found that equals the target value we
               search for
            return root;
        }
    }
    // at this point no node equaling the target value was found, therefore
       it
    // was not currently in the tree, if the tree is a complete BST.
    return null;
}
```

Balancing As we know there are ways to make a BST tree imbalanced, when inserting elements in specific order, that can be malformed into linked list like structure. A balancing technique we will look at is called subtree rotation used in AVL trees, which are special type of very well balanced trees, unlike Red Black trees, AVL trees always have a height difference at each subtree by no more than 1 level

There are two major rotations that can be done on a branch with a root, left and right. We do the counter rotation when the tree is heavy on one side

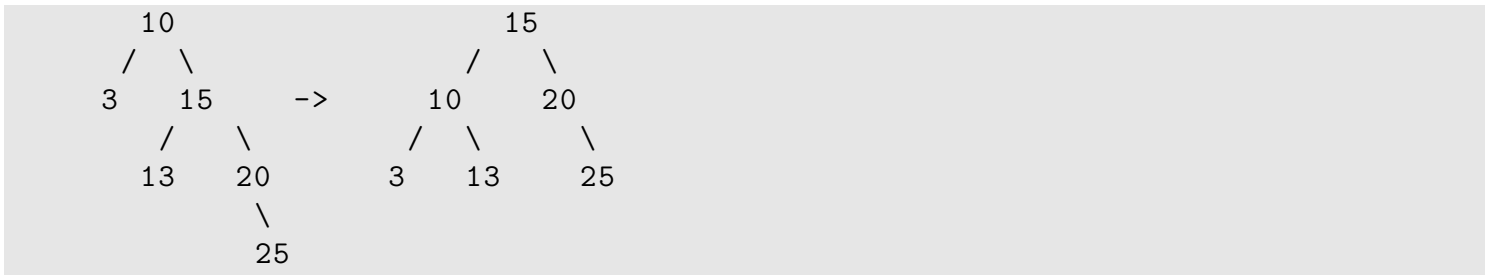
- left rotation - when branch is heavy on the right - i.e left(root)
- right rotation - when branch is heavy on the left - i.e right(root)

There are two other ones, which can be done combining the two major ones, in two separate steps,

- left-right - combination of the two major rotations, first left, then right - i.e root.left = left(root.left); right(root)
- right-left- combination of the two major rotations, first right, then left - i.e root.right = right(root.right); left(root)

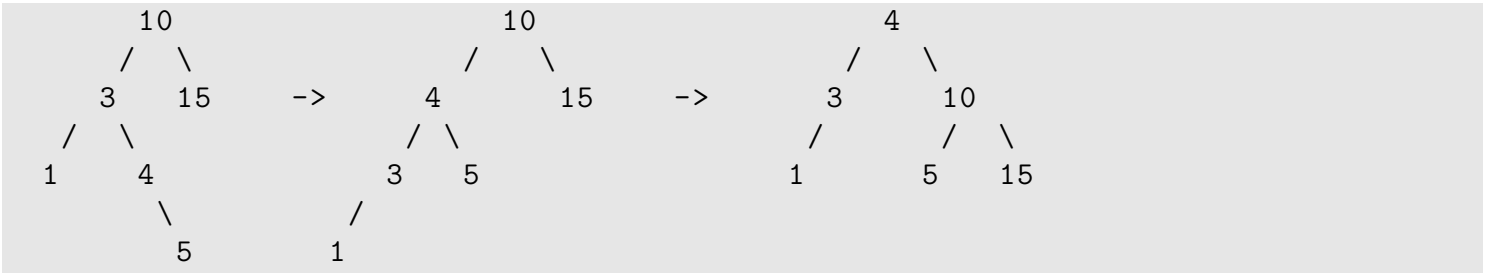
1. Right-Right rotation - we can see that at the level of node 10, the height of the left subtree is 3, the left one is 1, the balance factor is 2 == (3 - 1). We rotate around the node 10, in this case, where the imbalance is found
   - the root will become the node 3 (by reference)
   - the old right of node 3 becomes the new left of node 10
   - the new right of node 3 becomes the old root 10 node

2. Left-Left rotation - we can see that at the level of node 10, the height of the left subtree is 1, the right one is 3, the balance factor is -2 == (1 - 3). We rotate around the node 10, in this case, where the imbalance is found
   - the root will become the node 15 (by reference)
   - the old left of node 15 becomes the new right of node 10
   - the new left of node 15 becomes the old root 10 node

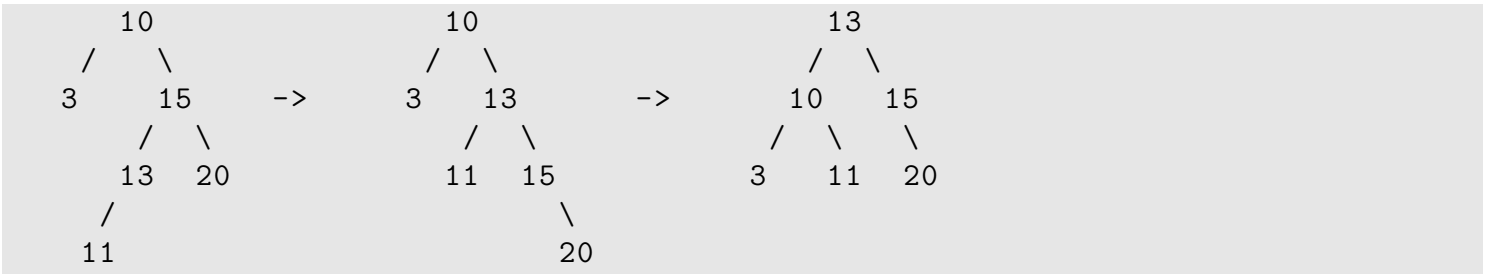


3. Left-right rotation - here we first perform left rotation around node 3, then around the main node 10, where the actual imbalance is. It is again 2 == (3 - 1).
   - First a left rotation around the node 3,
     - the root will become the node 4 (by reference)
     - the old left of node 4 becomes the new right of node 3
     - the new left of node 4 becomes the old root 3 node
   - Second a right rotation around the node 10

- the root will become the node 4 (by reference)
- the old right of node 4 becomes the new left of node 10
- the new right of node 4 becomes the old root 10 node



4. Right-left rotation - here we first perform right rotation around node 15, then around the main node 10, where the actual imbalance is. It is again -2 == (1 - 3)
   - First a right rotation around the node 13
     - the root will become the node 13 (by reference)
     - the old right of node 13 becomes the new left of node 15
     - the new right of node 13 becomes the old root 15 node
   - Second a left rotation around the node 10
     - the root will become the node 13 (by reference)
     - the old left of node 13 becomes the new right of node 10
     - the new left of node 13 becomes the old root 10 node



Below is represented the general interface used to balance a tree, we have the rotations, and the height method, along with the balance calculation and the re-balance action itself.

```
int height(Node root) {
    // height is 0 if the node does not exist
    if (root == null) {
        return 0;
    }
    // return the stored height, for the node
    return root.height;
}

Node right(Node root) {
    // right rotation is done to offset left heavy tree branches, right
       rotation means that left subtree would "move" or "shift" to
    // the right, or in other words the root.left would become the new
       root, and the old root would become the right, of the new
    // root. This way the subtrees of the new node would be balanced

    // remember the left node, it will become our new root at the end of
       the function the new root is returned
    Node left = root.left;
```

```
        // the left of the current root would become the right subtree of the
            new root i.e left.right node
        root.left = left.right;

        // the new root's right subtree now is assigned the old root
        left.right = root;

        // since root's subtrees were modified above, left was assigned a new
            subtree, calculate the height again
        root.height = 1 + Math.max(height(root.right), height(root.left));

        // since left's subtrees were modified above, right was assigned a
           new subtree, calculate the height again
        left.height = 1 + Math.max(height(left.right), height(left.left));

        // this is the new root, after the rotation was done, i.e the old .
           left subtree of root
        return left;
    }

    Node left(Node root) {
        // left rotation is done to offset right heavy tree branches, left
           rotation means that right subtree would "move" or "shift" to
        // the left, or in other words the root.right would become the new
           root, and the old root would become the left of the new
        // root. This way the subtrees of the new node would be balanced

        // remember the right node, it will become our new root at the end of
            the function the new root is returned
        Node right = root.right;

        // the right of the current root would become the left subtree of the
            new root i.e right.left node
        root.right = right.left;

        // the new root's left subtree now is assigned the old root
        right.left = root;

        // since root's subtrees were modified above, right was assigned a
           new subtree, calculate the height again
        root.height = 1 + Math.max(height(root.right), height(root.left));

        // since right's subtrees were modified above, left was assigned a
           new subtree, calculate the height again
        right.height = 1 + Math.max(height(right.right), height(right.left));

        // this is the new root, after the rotation was done, i.e the old .
           right subtree of root
        return right;
    }
```

```
int balance(Node right, Node left) {
    // the balance calculated as signed integer from left to right, when a
       tree is imbalanced, it would tend to malform into a
    // structure very similar to a linked list, it is quite obvious when
       either the left or right subtrees have a chain of links,
    // - node->left->left->left....etc, with no right links, along the path,
       left heavy, positive balance value
    // - node->right->right->right..etc, with no left links, along the path,
       right heavy, negative balance value
    return height(left) - height(right);
}
```

```
    Node rebalance(Node root, T value) {
        if (root == null) {
            return root;
        }
        // first make sure the current height is up to date, if re-balance
           was called, caller must expect that the height of the root
        // might have changed, during a tree action, insert or delete
        root.height = 1 + Math.max(height(root.right), height(root.left));

        // after the update of the height we check what is the current
           balance, between the current level's subtrees, for the current
        // level, i.e if the balance value is abs(balance) > 1, the tree
           needs to be rebalanced around the current root / level
        int balance = balance(root.right, root.left);

        // check if there is imbalance in the tree, it must be a value
           strictly greater, (not equal or greater) than one, does not
        // matter which subtree (left or right) is heavier. This post
           recursive action will find the very first level in the tree which
        // is imbalanced, from bottom to top.
        // - balance will be positive if the left side is heavier than the
           right
        // - balance will be negative if the right side is heavier than the
           left
        if (balance > 1 || balance < -1) {
            // the checks blelow are very neat and cool, why ?, well having
               the value to insert and the property of the binary search
            // tree, we can know in which direction the new node was inserted
               , combined with the sign of the balance we can tell in
            // which direction the new node imbalanced the tree, i.e which
               subtree became more heavy / imbalanced, this helps us to find
            // which of the 4 rotations have to be applied around the current
                root

            // left heavy && value < root.left.value - left-left
            if (balance > 1 && value < root.left.value) {
                // first start off from the balance value, if it is positive,
                    then the imbalance is in root.left, then we check if the
                // inserted value is smaller than the left node's value, if
```

```
                yes, then the new node was inserted to the left of the
                left
            // subtree. meaning the tree is left-left heavy, therefore we
                do the (opposite rotation) right rotation
            return right(root);
        }

        // right heavy && value > root.right.value - right-right
        if (balance < -1 && value > root.right.value) {
            // first start off from the balance value, if it is negative,
                then the imbalance is in root.right, then we check if the
            // inserted value is bigger than the right node's value, if
                yes, then the new node was inserted to the right of the
            // right subtree. meaning the tree is right-right heavy,
                therefore we do the (opposite rotation) left rotation
            return left(root);
        }

        // left heavy && value > root.left.value - left-right
        if (balance > 1 && value > root.left.value) {
            // first start off from the balance value, if it is positive,
                then the imbalance is in root.left, then we check if the
            // inserted value is bigger than the left node's value, if
                yes, then the new node was inserted to the right of the
                left
            // subtree. meaning the tree is left-right heavy, therefore
                first we left rotate, which will morph root.left case into
                a
            // left-left, after than we do a right rotate (same as left-
                left case above)
            root.left = left(root.left);
            return right(root);
        }

        // right heavy && value < root.right.value - right-left
        if (balance < -1 && value < root.right.value) {
            // first start off from the balance value, if it is negative,
                then the imbalance is in root.right, then we check if the
            // inserted value is smaller than the right node's value, if
                yes, then the new node was inserted to the left of the
            // right subtree. meaning the tree is right-left heavy,
                therefore first we right rotate, which will morph root.
                right
            // case into a right-right, after than we do a left rotate (
                same as right-right case above)
            root.right = right(root.right);
            return left(root);
        }
    }
    return root;
}
```

Insertion and deletion also need slight modification to include the re balancing of the tree after the operation, all parents up to the root, have to be visited, and update their heights at the very least, also check for imbalance

```
Node insert(Node root, int value) {
    // generic insert logic for bst
    if (root == null) {
        Node node = new Node();
        node.value = value;
        return node;
    }

    if (value > root.value) {
        root.right = insert(root.right, value);
    } else if (value < root.value) {
        root.left = insert(root.left, value);
    }

    // reaching at this point, we have very likely inserted an element,
       therefore we need to re-balance the tree, the return of
    // re-balance might be the same root, if no re-balance is required,
       or one of it's left or right children depending on what
    // rotation was done during balancing
    return rebalance(root, value);
}
```

```
Node delete(Node root, T value) {
    // generic delete and logic for bst
    if (root == null) {
        return null;
    }

    if (value > root.value) {
        root.right = delete(root.right, value);
    } else if (value < root.value) {
        root.left = delete(root.left, value);
    } else {
        if (root.left == null && root.right == null) {
            root = null;
        } else if (root.left != null && root.right == null) {
            root = root.left;
        } else if (root.right != null && root.left == null) {
            root = root.right;
        } else {
            Node curr = root.left;
            Node prev = root;

            while (curr != null && curr.right != null) {
                prev = curr;
                curr = curr.right;
            }
            if (prev == root) {
```

```
                prev.left = curr.left;
            } else {
                prev.right = curr.left;
            }
            root.value = curr.value;
        }
    }

    // try to check if re-balance is required, all the way, to the root,
       this is outside the else, because we have to post recurse, to
    // the root, to at the very least update the heights of each parent
       along the way, if a deletion has occurred, and also check if
    // the balance is okay.
    return rebalance(root, value);
}
```

Binary heaps

Binary heap tree can be one of two types, a max or a min binary heap tree. Those are special types of binary trees, where the structure of the tree is such that the Min or the Max element in the tree is always at the top, and all elements below it are strictly smaller (max) or bigger (min) heap tree type.

They are usually represent in a dynamic array, but can be done in the usual left/right pointer style nodes, that regular trees use. The reason to use a standard array is due to the operations such as insert and delete, which usually start off from the last element in the tree, and in arrays we have a very fast way to find, which one is the last element i.e. simply `array[array.len - 1]`

A Min heap might look like this.

```
      10
     /  \
   15    20
  /  \
33   17
```

A Max heap might look like this.

```
      10
     /  \
    5    2
  /  \
3     1
```

```
// the type of the heap as simple enum, heap type can not change during
   heap
// operations, it remains constant for the lifetime of the heap
public enum HeapType {
    MIN, MAX
}

// sample heap representation as a dynamic array, due to fast insert and
// look up operations at desired indices, i.e. for child / parent
   elements
List heap = new ArrayList<>();
```

Indexing We have to worry about 3 types of indexing in a heap, finding the parent, the left or the right child of a given node.

- left - the left child of each heap element is always an odd number, it is derived from `(2 \* i)+ 1`
- right - the right child of each heap node is always an even number, it is derived from `(2 \* i)+ 2`
- parent - the parent, using the two equations above, can be derived, by simply reversing them, check if the number is odd/even to know, the child index you have, derived from `(child - (child % 2 == 0 ? 2 : 1))/ 2`
- max/min - the very root of the heap, it's index is simply 0, always, we will find the max/min element to be the first element of the heap

```
Integer peek(List heap) {
    if (heap.isEmpty()) {
        // an empty heap has nothing to peek at
        return null;
    }
    // the element of a heap that can be looked at is always at the top,
       either the smallest, or the biggest in a heap
    return heap.get(0);
}

int parent(int child) {
    // parent index of a heap element can be computed from either the
       left or the right child, we know that left child indices are
    // always not even, and right ones are always even, therefore, first
       check mod 2, and subtract 2, for right child, or 1 for a
    // left child
    child = child % 2 == 0 ? child - 2 : child - 1;

    // finally to get the parent divide by 2, this basically reverses the
        equation for finding a child from a parent index, see the
    // methods below
    return child / 2;
}

private int left(int parent) {
    // to get the left child, multiply the parent index by two, and add
       one, each number multiplied by two becomes an even, adding
    // one would make it odd, all right children are on odd indices
    return (2 * parent) + 1;
}

private int right(int parent) {
    // to get the right child, multiply the parent index by two, and add
       two, each number multiplied by two becomes an even, adding
    // two would keep it even, all left children are on even indices
    return (2 * parent) + 2;
}
```

Swapping Since it is often used operation in a heap, it is good to show what it does, simply put swap the two elements located at the input indices in the heap.

```
    void swap(List<T> heap, int l, int r) {
        // simple method to swap elements located on two indices
        T c = heap.get(r);
        T p = heap.get(l);

        // exchange the elements at the two index locations in the heap array
        heap.set(r, p);
        heap.set(l, c);
    }
```

Inserting To insert in a binary heap tree we have to do two major things.

- the new element to insert goes to the end of the heap
- the new element is bubbled up to the correct position

The bubbling is based on the tree type, but briefly, we check the current element with its parent, and swap if the conditions below are met

- for min heaps = if smaller than the parent we swap them
- for min heaps = it bigger than the parent we swap them

```
    Integer insert(List heap, Integer value) {
        // first off we start by putting the new item at the end of the array
            or heap, and getting that new index location of the last
        // element we have just inserted
        heap.add(value);
        int curr = heap.size() - 1;

        // what is going on here, is looping until the current element is not
            pointing at the root, or in other words at index 0,
        // current starts off from the insertion position, i.e the end of the
            heap, the very last element and bubbles up the new element
        // until the correct spot is found
        while (curr > 0) {
            // calculate the parent of the current index, and extract the
               value of the parent
            int prev = parent(curr);
            Integer parent = heap.get(prev);

            // find if the new element is bigger, smaller or equal to the
               parent
            int diff = value - parent;

            // when equal to the parent, we can abort, duplicates are not
               swapped, for simplicity
            if (diff == 0) {
                break;
            }

            if (type == HeapType.MIN && diff < 0) {
                // when we are in a min heap, the swap is happening only if
                   the new element is smaller than it's parent, this is
```

```
                because
            // the new element being smaller, has to be bubbled up, until
                it is no longer smaller than it's parent
            swap(heap, curr, prev);
        }

        if (type == HeapType.MAX && diff > 0) {
            // when we are in a max heap, the swap is happening only if
               the new element is bigger than it's parent, this is
               because
            // the new element being bigger, has to be bubbled up, until
               it is no longer bigger than it's parent
            swap(heap, curr, prev);
        }

        // move up to the parent of the current element, eventually prev
           here would become 0, index 0, and the while loop will
        // finish, meaning that the element was bubbled to the top, for
           extra peformance one might just break after we find the
        // first element where diff does not perform a swap, meaning the
           element is already at the correct spot.
        curr = prev;
    }
    return value;
}
```

Deleting Deleting an element from the tree is a bit more convoluted, what we do here is to take the last element from the heap, put it at the very root / top of the heap and then gradually try to drop it down until a correct spot for it is found. We can also remember the old root element and return it (optional).

What is important here to take note of is how we decide to move down the element. On each step the last element we choose would either go down the tree or stop, if it stops then it has found the correct place. To go down we can follow these rules for the two types of trees

- if we have min heap - for min heap we know that smaller elements would be closer to the top than bigger ones, therefore we have to find the smallest element between the root, left or right if the root happens to be the smallest, it is at the correct place, otherwise swap the root with the smaller one between left or right. The new root becomes the index of the child with which we swapped
- if we have max heap - for min heap we know that bigger elements would be closer to the top than smaller ones, therefore we have to find the biggest element between the root, left or right if the root happens to be the biggest, it is at the correct place, otherwise swap the root with the bigger one between left or right. The new root becomes the index of the child with which we swapped

```
Integer delete(List heap) {
    // there is nothing to delete from an emtpy heap in the first place
    if (heap.isEmpty()) {
        return null;
    }

    // first get the last element from the heap and swap with with the
       root, the last element will then be bubbled down, and the
```

```
        // root returned from this function as result
        int last = heap.size() - 1;
        swap(heap, last, 0);

        // the head or root of the heap is now the last one, after the swap,
           just remember it, and remove it from the heap
        Integer head = heap.get(last);
        heap.remove(last);

        // we start off from the top or head of the heap, going down locating
            the left and right indices of the current element and
        // check if the root needs to stay in it's place, or go left or right
        int next = 0;
        while (next < heap.size()) {
            // get the indices of the left and right children of the current
               element
            int left = left(next);
            int right = right(next);

            // start off by assuming the smallest / biggest (whichever heap
               type we have) element between the root and it's children is
            // the root, set the 'base' index to point at the root initially,
                this base index will either change, or remain the same,
            // based on this we would know to continue or not to swap
               elements
            int base = next;

            if (type == HeapType.MIN) {
                // we first compare the value of the 'base' index with it's
                   left child, if the left child is smaller than the root, we
                // update the 'base' index to point to the left child index
                if (left < heap.size() && heap.get(left) < heap.get(base)) {
                    // left becomes the smaller between base and left
                    base = left;
                }

                // we first compare the value of the 'base' index with it's
                   right child, if the right child is smaller than the 'base
                   ',
                // we update the 'base' index to point to the right child
                   index
                if (right < heap.size() && heap.get(right) < heap.get(base))
                   {
                    // right becomes the smaller between base and right
                    base = right;
                }
            } else if (type == HeapType.MAX) {
                // we first compare the value of the 'base' index with it's
                   left child, if the left child is bigger than the root, we
                // update the 'base' index to point to the left child index
                if (left < heap.size() && heap.get(left) > heap.get(base)) {
```

```
                    // left becomes the bigger between base and left
                    base = left;
                }

                // we first compare the value of the 'base' index with it's
                   right child, if the right child is bigger than the 'base',
                // we update the 'base' index to point to the right child
                   index
                if (right < heap.size() && heap.get(right) > heap.get(base))
                   {
                    // right becomes the bigger between base and right
                    base = right;
                }
            }

            if (base != next) {
                // if the base index actually changed, then we can swap the
                   base with next, meaning that either the left or right
                // children of the root were smaller / bigger (based on the
                   heap type we have)
                swap(heap, base, next);
                // now we move to the next index with which we swapped,
                   either the left or right, to continue to drop down
                // the element until it finds its place
                next = base;
            } else {
                // at this point the base did not change, meaning the root
                   was the smallest / biggest of the 3 (root,left,right),
                // therefore the element is at the correct position, there is
                    nowhere for it to go further, it conforms to the rules
                // of a heap
                break;
            }
        }
        return head;
    }
```

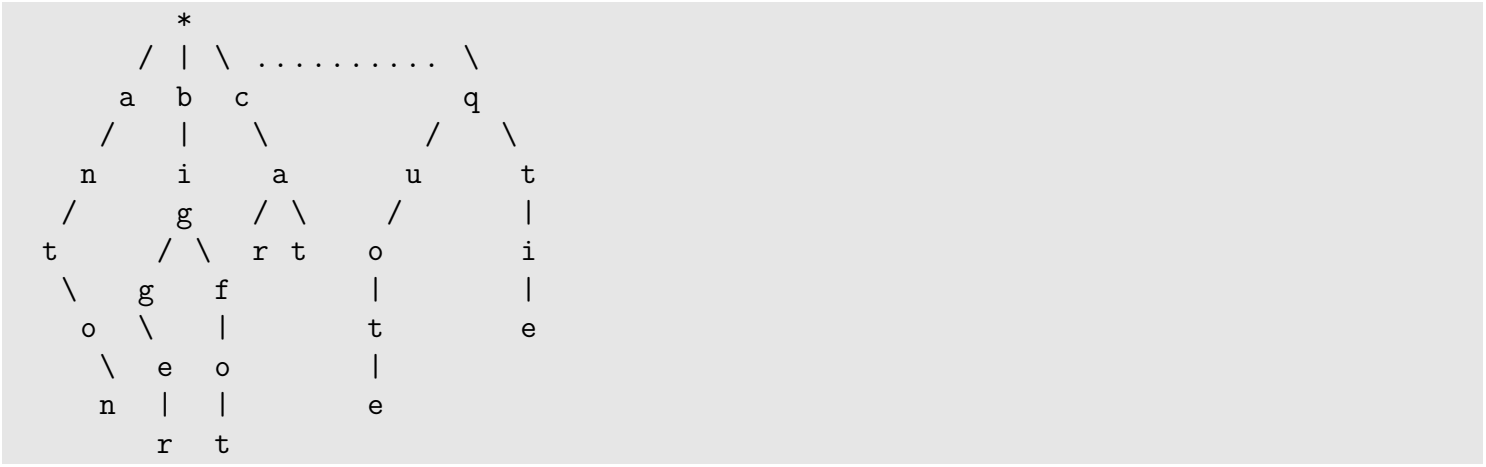
Tries

Also called prefix trees, the word trie comes from the word Retrieval, they are used to store words, and look up if a word is present in O(M) time, where M is the length of the word, unlike regular trees that store each entire words in a each node, tries are more efficient. Words are represented as with their prefixes

These are a type of n-ary trees, where the n in this case is the number of characters in an alphabet, for the English one that would be 26. Each node BESIDES the very root of the Trie represents not only a node with children, but a character as well, even though that is implicitly done.

Each root will have up to the n-children, but not all children are going to exist at the same time, see example below. However to denote a word trie nodes usually have a boolean flag, which tells us if the path to that node forms a word. (e.g. if we have the word bigger, but the word big was not added, to the trie, the path big would exist but it would not form the word big, until we insert explicitly the word big)

A trie that represents a few words, might look like that, note how the root contains a lot of nodes, but the children might just have handful, it is not mandated that each level is fully filled (with in 26 nodes, for the English example)



This trie example might contain words such as (remember to denote a word actually is part of the tree, not just a path to another word, it has to have its terminating flag set to true, at the node, which represents the end of the word, e.g. 'big' would have the terminating flag in the g node set to true)

- an, ant, anton
- big, bigger, bigfoot,
- cat, car
- quote, qtie

```
class TriePrefixNode {
    TriePrefixNode[] children;
    boolean terminating = false;
}
```

Inserting Inserting in a trie, is simply following the existing path, if one does not exist create the new nodes, along those path using the characters from the input word / string. At the end mark the final node as terminating, as it would indicate that the specific unique path that was followed/created terminates with a word that is part of the trie, it has been inserted

```
TriePrefixNode insert(TriePrefixNode root, String string, Integer value)
   {
    // check if the input values first make any sense at all, the string
       can not be nil, and the value can not be nil either
    if (string == null || value == null) {
        return null;
    }
    // for simplicity reasons normalize all strings to lower case, this
       is not really a part of an usual trie implementation but it
    // makes things easier to understand
    string = string.toLowerCase();

    // either start off from the provided root, or make on if the insert
       was called without a valid root
    TriePrefixNode base = root;
    if (root == null) {
        root = new TriePrefixNode();
```

```
        base = root;
    }

    // go over each character in the input string.
    for (int i = 0; i < string.length(); i++) {
        // fetch the character and convert it to a number, in the ascii
           table the english alphabet lower case letters start at 97,
        // up until 97 + 26. This is used to index the character in node
           in the children list of a trie node
        Character c = string.charAt(i);
        int cval = c.charValue();

        if (root.children == null) {
            // the current root has no children, therefore we have to
               create an array of 26 children first, this is needed,
            // to make an empty array of null pointers, which are going
               to be filled in with node instances
            root.children = new TriePrefixNode[26];
        }

        // find which child index the current character corresponds to,
           it can either already exist, or not, if it exists we just
        // take that path, and continue down, setting the next root to
           the found node
        TriePrefixNode node = root.children[cval - 'a'];
        if (node == null) {
            // the node at that char position does not exist, create a
               new node instance, and assign it to that position in the
            // root's children
            node = new TriePrefixNode();
            root.children[cval - 'a'] = node;
        }

        // the next root would now be the last character node we inserted
           , continue until the string is completely looped over
        root = node;
    }

    // after the string has been looped over the root node here would
       point at the last character node, we mark it as terminating,
    // meaning that it represents an end of the path, or in other words
       one complete word, and we set a value mapping to it. The
    // value here is not a usual part of trie implementation, it is done
       to demonstrate how a value can be mapped to a word/string
    // in a trie, similarly to a hash-map
    root.terminating = true;
    root.value = value;

    // return the base root node, which represents the passed in or
       created root
    return base;
```

```
    }
```

Searching Searching is very similar to inserting, but instead of creating missing nodes, we strictly try follow existing paths, if one does not exit, or the last node in the path is not `terminating = true`, then we can deduce this particular word being searched for is not part of the tree (the path might exist, but as part of another bigger / longer word)

```
    boolean search(TriePrefixNode root, String string) {
        // make some validation on the input, nil strings are not valid,
           neither are non existent roots
        if (string == null || root == null) {
            return false;
        }

        // for simplicity reasons normalize all strings to lower case, this
           is not really a part of an usual trie implementation but it
        // makes things easier to understand
        string = string.toLowerCase();

        // go over each character in the input string.
        for (int i = 0; i < string.length(); i++) {
            // fetch the character and convert it to a number, in the ascii
               table the english alphabet lower case letters start at 97,
            // up until 97 + 26. This is used to index the character in node
               in the children list of a trie node
            Character c = string.charAt(i);
            int cval = c.charValue();

            // if the current root is invalid or has no children array
               initialized, there is nowhere to go really, therefore the
               passed
            // in string 'word' is not part of the trie. there is no path in
               the tree which would describe this word
            if (root == null || root.children == null) {
                // nothing was found, return false
                return false;
            }
            // set the next root to the child which corresponds to the char
               index, from the root.children nodes, that node can either be
            // null, or be initialized, when we re-enter the next iteration,
               it will either terminate on root == null / or root.children
            // == null or keep going
            root = root.children[cval - 'a'];
        }
        // reaching this point does not mean we have found the string 'word'
           still, the path might exist, as a subpath of another longer
        // word, we have to see if the node at the end of the path for the
           current string 'word' was marked as terminating, meaning it
        // represent a word in the trie.
        return root.terminating;
    }
```

Deleting todo: this is not yet finalized, finish it

B-Trees

Worker&Iterator approach

Solving tree problems very often involves a very common approach technique, which as to be taken extra note of. Very often the problems for trees involve developing two recursive solutions. One might be called the main worker solution, and the other is the iterating function. This approach comes very often in tree problems. Two functions with dedicated roles / actions.

```
void worker(Node root, Params... params) {
    // is a recursive function that does the actual work on a node basis
    // it is meant to do the actual work, produce result and use that
       result
    // in the iterating function, to accumulate it, to store it, to
       return it
}

void iterator(Node root) {
    // iterator solution which basically goes through the nodes, collects
    // information for the worker function to do it's job, the worker
    // execution might happen as pre, post recursive call or we might
       call it
    // multiple times in the iterator function , but it is the main
       solver
    worker(root)
    iterator(root.left)
    iterator(root.right)
}
```

We can notice how many times the worker & iterator approach was observed above, there are multitude of solutions, which make use of it like

- Ancestor - makes use of the contains, which is the main function providing the result based on which the iterator function decides how to proceed further down the tree, or stop
- Subtrees - makes use of the compare, which is the main function providing the result to validate subtree equality, the iterator function simply goes in both left and right direction

Graphs

Graphs are a collection of nodes with edges between some of them. It is a known fact that all trees are graphs, but not all graphs are trees. A graph, which has cycles cannot be called a tree. Trees can only be valid when there are no cycles between nodes.

A graph could contain multiple other sub-graphs, such that there are no connections between different sub-graphs, if there are connections between all nodes in a graph, we call this graph a connected one

Below we have an example of a graph, which contains two sub-graphs, there are no connections between the two sub-graphs

```
          k    o      m -- j -- w
         /     /      \     |
```

```
a -- c -- f       y --
 \       /
  b --- d
```

In practice, the most used graph we would encounter is a directed graph, where the edges / connections have a direction, meaning connection from `a -> b` does not mean that `b -> a` exists automatically

Representations

Depending on the task at hand there might be better or worse representations to use, what is important is that as much information is contained in the representation as possible (incoming edges, parent nodes etc.)

Class & Nodes

There are multiple ways to represent a graph, the most common one is with a graph node, and a graph class, which is just enough to implement most all graph features we would ever need. We need the class on top of a node since in a graph it is possible to not have path from one single root node to all others, that is actually usually the case, a graph might have many root nodes (ones with no incoming edges), or it might have no roots (nodes with 0 incoming edges), in that case if all nodes have incoming edges this graph has cycles, there is no obvious graph root to speak of

```
class Node {
    int incoming;
    Integer value;
    Node[] children;
}

class Graph {
    Node[] nodes;
}
```

In the structure above take note of the `incoming` property, which is important, and tells us how many incoming edges link to the current node, while `children` tell us how many outbound connections we have coming out of the current node

Adjacency Matrices

One popular way to represent a graph is by simply using a hash map where the keys are the values of the nodes, and the nodes themselves, and each value / node has a connection to a list of the nodes it connects to. Instead of a normal hash map we could also use a multi map, which is a more convenient way to add new node connections

```
HashMap<Integer, List<Integer>> matrix;
matrix.put(1, Arrays.asList(5, 3, 1));
matrix.put(2, Arrays.asList(3, 1));
matrix.put(3, Arrays.asList(2, 5, 1));
matrix.put(4, Arrays.asList(3));
matrix.put(5, Arrays.asList(3, 4));
```

In the example above each key represents a unique node value, while the arrays represent the links that node has to other nodes. This is for directed graphs

2D Flags Array

Another variant of the Adjacency matrix is a simple 2d array where each position where an edge / connection exists in the 2d array is marked with 1, and where no connection exists is marked with 0. In this representation either the rows or the cols will represent from and to. They are not interchangeable.

Below the `to` is presented by the columns, and the `from` is presented by the rows. Meaning we can read this matrix like that - `col to row` tells us the outgoing edges for the node and `row to col` tells us the incoming edges for the node.

- Take col-node `2` which in the matrix has 1 incoming edge - `3`
- Take row-node `2` which in the matrix has 2 outgoing edges - `1, 3`

That information tells us that node `2 and 3` are bidirectionally connected in the graph example given below. While node `2 and 1` are only connected from node `2 to 1` but not the other way, in other words this is a singly directed edge.

```
  | 1 2 3 4 5
-------------
1 | 1 0 1 0 1
2 | 1 0 1 0 0
3 | 1 1 0 0 1
4 | 0 0 1 0 0
5 | 0 0 1 1 0
```

Creation

A very simple example, which creates a graph from a list of edges, where the edges represent a simple `from - to` relation between nodes. In the example below there are two main functions

- edges - creates an array / list of edges, the input here is a simple list of values, which represents the `unique` graph node values, and the links they form to other nodes. e.g. `a, b, a, c, a, d, c, b, c,d`. In simple terms node `a` has outgoing / child connections to nodes `b, c and d`, then node `c` has outgoing / child connections to nodes `b and d` and so on.
- create - creates the graph node representation, in this case a list of nodes, representing the graph, from a list of edges or connections. While the edge `from` property must be present, the `to` might be null, signaling that this node has no connection to another node for example, or in other words it has no outgoing connection, the input array / list of edges might look something like that - `a, nil, a, c, a , d`.

```
class Node {
    Integer value;
    int incoming = 0;
    List<Node> children;
}

class Edge {
    Integer from;
    Integer to;
}

List<Edge> edges(List<T> elements) {
    // list has to be an even number since connections are from - to,
       note
```

```
        // that to could be a null element but still has to be present
        if (elements.size() % 2 != 0) {
            return Collections.emptyList();
        }

        // hold the list of edges, representing the graph connections
        List<Edge> edges = new ArrayList<>();

        // elements is a plain array which contains pairs of links,
           connections
        // between the nodes, we use that to construct the connections or
           edges
        for (int i = 0; i < elements.size(); i += 2) {
            edges.add(edge(elements.get(i), elements.get(i + 1)));
        }

        // return the list of edges constructed from the elements array
        return edges;
    }

    List<Node> create(List<Edge> edges) {
        // the cache holds each unique node based on it's primary value, and
           a
        // link to the node instance / reference itself
        Map<T, Node> cache = new HashMap<>();

        // simple constructor producer which initializes a graph node with
        // correct value, like incoming edges, and the children
        Function<T, Node> constructor = (T value) -> {
            Node node = new GraphNode<>();
            node.children = new ArrayList<>();
            node.value = value;
            node.incoming = 0;
            return node;
        };

        // go through all the edges, and create and link the node instances
        for (Edge edge : edges) {
            if (edge.from != null && !cache.containsKey(edge.from)) {
                // construct the node instance for the `from` edge
                cache.put(edge.from, constructor.apply(edge.from));
            }

            if (edge.to != null && !cache.containsKey(edge.to)) {
                // construct the node instance for the `to` edge
                cache.put(edge.to, constructor.apply(edge.to));
            }

            // extract the node `from`, from the cache, `from` link is
               mandatory
            // and will always be present, since it represents the source
```

```
                node,
            // while `to` is optional since an outgoing connection might or
            // might not be present.
            Node current = cache.get(edge.from);

            if (edge.to != null) {
                // when the edge connection to is valid, we link it to the
                // instance / reference of the node which represents the edge
                    .to
                Node child = cache.get(edge.to);

                // add the child node to the current edge.from children list,
                // this represents the forward, outbound connections of edge.
                    from
                current.children.add(child);

                // make sure to increment the incoming edges for the child,
                    as
                // we have added a new incoming edge
                child.incoming++;
            }
        }

        // get the values from the cache map, which are all the nodes in the
        // graph, return this as the final result of the constructed graph
        return cache.values().stream().collect(Collectors.toList());
    }
```

Traversing

Graphs have two major traversal approaches, similarly to trees, we can either use:

- Breadth first - where each level of the graph is visited, before the next one is, then drill to the next immediate level and repeat, algorithm is based off of storing the nodes in a queue
- Depth first - where each level is visited in depth, down to the very last link, before neighbor levels are considered, algorithm is based off of storing the nodes in a stack

In the examples below the input list of graph nodes `List<Node>` represent a set of input nodes from which to start the traversal, that list could simply be the entire list of nodes, which belong to the graph, or simply a small set of pre-selected ones, for example, select only the ones without incoming edges, and traverse from them in depth or breadth

BFS

Similarly to trees, however not as widely used with tree traversal, BFS is very much a cornerstone traversal approach for graphs, this approach inspects each immediate graph neighbors before going down to each of their children, inspecting a level of depth at a time

Note that is quite important when a node is marked as visited in either BFS or DFS traversals, this is due to how the traversal works, while in the recursive approach we have more freedom, in the iterative BFS approach the visited mark must be done when the child node is added to the queue.

This detail, of when the node is marked as visited comes in again when we discuss the Dijkstra algorithm, where the visited mark is done in a different position in the iterative process, and it is very important to take a note of that. The early or late marking of a node as visited, could drastically change the algorithm, the problem and the solution.

```
    void breadth(Queue<Node> queue, List<Integer> path, Set<Node> visited) {
        // remove the current element from the queue
        Node node = queue.remove();

        if (node == null) {
            // reaching this point should not really be possible, since the
            // queue would normally not get any nil elements pushed into it
            return;
        }

        // add the value of the current node to the path, before adding the
        // children to the queue
        path.add(node.value);

        // for each child of the current node
        for (Node child : node.children) {
            // before trying to go down, make sure the child is not visited
               already
            // could be, in case other nodes have links to this child, and
               have
            // been processed before
            if (!visited.contains(child)) {
                // mark the current node as visited, we want to make sure
                   that if the
                // graph has cycles, we never pass through this again,
                   otherwise it
                // would be added to the queue repeatedly
                visited.add(child);

                // append child to the end of the queue
                queue.add(child);
            }
        }
    }

    List<Integer> breadth(List<Node> graph) {
        // the input list graph, would contain a list of starting nodes, from
        // which we would like to start traversing the graph in breadth

        // construct the queue firstly initialized with the input graph nodes
        Queue<Node> queue = new LinkedList<>(graph);

        // hold state for visited nodes, already having been traversed over
        Set<Node> visited = new HashSet<>();

        // hold the nodes, in order of the way they were visited
```

```
        List<Integer> result = new LinkedList<>();

        while (!queue.isEmpty()) {
            // pull from the queue the current element, add its children and
            // keep doing that until the queue is emtpy, meaning all nodes
               that
            // were reachable from the starting ones were visited at least
               once
            breadth(queue, result, visited);
        }

        // return the path formed by traversing the nodes starting from the
        // input list of graph nodes
        return result;
    }
```

DFS

Works very much like a DFS in trees, with the added caveat that we do not have 2 children at most, but N children, each node is fully inspected, down to its last leaf children, before the next one is picked up.

```
    void depth(Node node, List<Integer> path, Set<Node> visited) {
        if (node == null) {
            // reaching this point should not really be possible, since
               children
            // array should never contain nil nodes, but add this guard case
            return;
        }

        // add the value of the current node to the path, before adding the
        // children to the queue
        path.add(node.value);

        // for each child of the current node
        for (Node child : node.children) {
            // before trying to go down, make sure the child is not visited
               already
            // could be, in case other nodes have links to this child, and
               have
            // been processed before
            if (!visited.contains(child)) {
                // mark the current node as visited, we want to make sure
                   that if the
                // node has cycles, we never pass through this again,
                   otherwise infinite
                // recursion would occur
                visited.add(child);

                // dive in depth for the current child
                depth(child, path, visited);
            }
```

```
        }
    }

    List<Integer> depth(List<Node> graph) {
        // the input list graph, would contain a list of starting nodes, from
        // which we would like to start traversing the graph in depth

        // hold state for visited nodes, already having been traversed over
        Set<Node> visited = new HashSet<>();

        // hold the nodes, in order of the way they were visited
        List<Integer> result = new LinkedList<>();

        for (Node node : graph) {
            // go through all nodes of the graph, dive depth first for each
               node
            depth(node, result, visited);
        }
        return result;
    }
```

Dijkstra

A way to find the shortest path between two points in a weighted directed graph (which might have cycles). Where all edges must have positive values. Note that a side effect of Dijkstra is that it finds the shortest paths from the `start` node to any other node that can be reached by start, and one of them being `end` (if start and end have path or link between each other of course).

The way the Dijkstra works to find the shortest path, is by maintaining 4 separate structures

- priority queue / min heap - this is the core of the Dijkstra algorithm, we push nodes, and their absolute cost / path in that min heap, by the nature of the min heap, the nodes with the least cost / weight will be at the very top of the heap, pulling from it means we always pull the best cost node
- distances - maps a node, actually a unique `node.value,` and a min path to that node, in the end of the algorithm we would have the shortest paths to all nodes in the graph from the starting position, in the distances map
- previous - mirrored structure of distance, where for each min distance for a given node the previous node from where this min distance was achieved is stored in, we use previous to backtrack from where we came to achieve the shortest path, and thus can generate all the pahts from all nodes to the starting node (if the starting node had a path to them of course)
- visited - nodes, which are pulled from the priority queue are marked as visited, to avoid re-visiting them, the first time a node is visited, is also the first time we extract it from the min heap, meaning it is going to be the one with the min weight

What is crucial to note is that the min heap can contain the same node i.e `node.value` more than once, but with different weights, since we can reach that node from many places, each time we do find a better cost weight (compared to the current `distance[node.value]` weight) we then push to the heap.

Naturally because it is a min heap, the best cost `WeightNode` would be found near the top of the heap, when we pull a node that is not visited for the first time, we always pull the best `WeightNode` for that node (even

if there are duplicates, the others would have worse cost, and will be deeper in the heap), we then mark the node by `node.value` as visited.

Note that unlike DFS and BFS, we do not mark the nodes when we first encounter them as children, but we mark them as visited only when we pull them from the queue, this guarantees that by the time we pull the node, all paths to that node would have been investigated, best cost weight would have been found. And naturally we would pull the best cost from the min heap for that node / `node.value`. After we pull a node we have to check if it has been visited, having been visited already means that the best cost for that node was pulled, and processed, we can simply `continue` (this handles the duplicates with worse costs / weights, by simply removing them from the queue without taking any action, and just skips to the next iteration in the algorithm)

The two main loop ending conditions for the Dijkstra are 2, these do not tell us if we have reached the end node target, however they are used to know when to stop traversing the graph

- all nodes - either the visited structure contains the same number of nodes as the entire graph, meaning we have visited all possible nodes, this might not obtain if we have sub-graphs with no connections, and the start node is in one sub-graph, the end-node in another, we would never reach the 2-nd sub-graph nodes, meaning `visitedGraphNodes.size` would never equal `totalGraphNodes.size`
- min heap empty - priority queue is empty, this would happen in case we pop every possible node that we could have reached from the starting position node, by that time we still do not know if we have actually reached end, but at least we have gone over all possible paths that could have been traversed starting from the `starting` node

The final stage, we try to reconstruct the path from end to start backwards, using the previous structure, we can check if we have mappings following the previous, if we reach `start` then we had a link from `end` to `start`, otherwise no path between the two targets exists.

```
    // base utility weight node class which extends from the base graph node
       class,
    // used to maintain and track the absolute weight of the graph nodes
       while
    // executing Dijkstra, in a priority queue
    class WeightNode extends Node implements Comparable<WeightNode> {
        int weight = 0;

        // create a weight node from a graph node, by cloning over the
        // properties of the base node
        public WeightNode(Node node, int weight) {
            this.weight = weight;
            this.value = node.value;
            this.children = node.children;
            this.incoming = node.incoming;
        }

        @Override
        public int compareTo(WeightNode o) {
            // required for the priority queue, we care only about comparing
               the
            // total weight, or cost of each node.
            return this.weight - o.weight;
        }
```

```
    @Override
    public boolean equals(Object obj) {
        WeightNode node = (WeightNode) obj;
        return this.value.equals(node.value);
    }
}

List<String> computeShortestPath(List<Node> nodes, Node start, Node end)
   {
    // this is the cornerstone of this algorithm, this is a min heap,
       which orders the nodes such that the ones with smallest cost
    // / weight are at the front of the queue, remember that the same
       node i.e. node.value can be added in this heap more than once,
    // but with different costs, since we might reach that node from
       multiple other nodes, but since we work with a min heap, these
    // WeightNodes would be ordered by cost, so we will always pull the
       smallest node by cost first.
    PriorityQueue<WeightNode> heap = new PriorityQueue<>();

    // visited holds nodes by identifier, that is important, since the
       heap might contain the same 'node' multiple times with
    // different costs
    Set<String> visited = new HashSet<>();

    // distances from a given node / node.value so far, these are the
       total accumulated distances, and always represent the minimum
    // distance to the specific node, or more precisely the shortest
       distance to each node id <-> node.value
    Map<String, Integer> distances = new HashMap<>();

    // a mirror of the distances map, it tells us from where we came to
       achieve that min distance for a given node, they are both
    // updated together always, this way later on we can use previous to
       backtrack the shortest path
    Map<String, Node> previous = new HashMap<>();

    // nodes are identified by their value, that is like an id for a node
       , so update the initial distances, and previous, as well as
    // the heap, which at the start would contain the 'start' node we
       want to start from. The cost to the current start node is
    // obviously zero, add this as a 0 weight node to the heap
    distances.put(start.value, 0);
    previous.put(start.value, null);
    heap.add(new WeightNode(start, 0));

    // loop until we see all nodes in the graph, this is the very basic
       exit condition, it might not obtain, if we have sub-graphs
    // that have no connections between each other in the main graph, but
        the heap -> break would be our backup exit strategy
    while (visited.size() < nodes.size()) {
        // it is possible for the queue to become empty, before all nodes
```

```
            in the graph are visited, one way this could happen is if
        // we have two sub-graphs, part of the main graph, which have no
           connections between each other
        if (heap.isEmpty()) {
            break;
        }
        // pull the current shortest path node, from the priority queue /
            min heap, in this case, the heap is sorted based on the
        // cost of the weight node
        WeightNode node = heap.remove();

        // we can check if it has been visited before first, based on the
            value of the node, if it has it means that it was present
        // already in the heap with a smaller total cost / weight, so any
            further presence of the same node would be with higher
        // cost / weight we do not want to consider it
        if (visited.contains(node.value)) {
            continue;
        }
        // add / mark the current node as visited, reaching here means
           the current node was not visited so far, meaning this is the
        // first time we pull it from the heap, meaning we are pulling
           the one with the smallest cost, remember the heap might
        // contain multiple nodes with the same value, but different '
           costs', due to the fact that in a graph we might reach the
        // same 'node' from multiple parent nodes. By saying 'node' we
           really mean 'node.value' - that is the identifier for a node
        visited.add(node.value);

        for (Node child : node.children) {
            // extract the current path to the current child so far, or
               default to some big value, careful, we are not using
            // INT_MAX, due to the fact that it might overflow if we add
               anything more than 0 to it
            Integer current = distances.getOrDefault(child.value, 9999);

            // compute the 'would be' distance from the current node to
               the current child, this is the cost of the current node so
            // far + the weight cost of the current child, care with
               overflow, if using INT_MAX. The method `getWeightCost`,
               simply
            // returns the cost / edge value from the current `node` to
               the `child`
            Integer possible = distances.getOrDefault(node.value, 9999) +
                node.getWeightCost(child);

            // in case the possible new distance is actually smaller than
                the distance stored so far, we update the distance stored
            // so far, update the previous node we come from, and add the
                child to the heap
            if (possible < current) {
```

```
                    // update from which node we came to the current child,
                       since we found better / lower cost path value
                    previous.put(child.value, node);

                    // update the distances to the current child node as we
                       have found shorter path to it than what we had so far
                    distances.put(child.value, possible);

                    // every time we find path to a given node which is
                       shorter (so far) we add it to the heap, it would be '
                       sorted'
                    // based on it's weight into the heap, note we add the
                       total 'weight' not just the 'child.weight'
                    heap.add(new WeightNode(child, possible));
                }
            }
        }

        // collect the path from start - end
        List<String> path = new LinkedList<>();
        Node target = end;

        do {
            // starting off from the end node, track back using the previous
               map, which tells us from where we came to achieve the
            // current path, remember previous and distances were updated
               together meaning previous contains the shortest cost / path to
            // a given node / node.value
            path.add(target.value);

            // move backwards to previous node
            target = previous.get(target.value);
        } while (target != null && target != start);

        // return the path, we might want to validate if the start / end
           correspond to the
        return path;
    }
```

A-star

Similarly to Dijkstra, the implementation is actually the same, a way to find the shortest path between two nodes in a weighted graph, the only meaningful difference is the way the cost is computed, in Dijkstra we simply compare the cost / weight of the shortest path to a given node so far, to the current weight or cost of the path to that node at the moment. However A-star takes us a step further, it defines an additional function `h`, a heuristic, which modifies the calculation such that it tries to guess how much cheaper would the cost be if we go through the current `node` to the end goal

The heuristic function is only computed when we add the node to the priority queue, it biases the priority queue order such that nodes with less cost `IN RELATION TO THE TARGET GOAL ONLY`, would be put to the front of the heap, which means that unlike Dijkstra in the distances array, we do not have all shortest paths

from the start to every other node, we only have the shortest distance from start to end, since the children we took from the heap were the ones with cost biased by the heuristic, and the heuristic function is a function of the `current` and `end` node strictly, i.e. `h = heuristic(node, end)`.

The value of the heuristic function could be positive, negative or 0, depends on how it is defined. Imagine a negative value of the heuristic, would imply that the cost to the goal node is actually very low, since we would subtract it from the current `weight` cost, a very big positive heuristic value would imply a very costly path to the goal node if we took that route.

A heuristic is defined very much differently based on the use case, if our graph was instead a grid, the heuristic is usually a coordinate computation of the `current node`, in relation to the `end` goal. There are several famous ones:

- Manhattan distance - on a grid where we can move in 4 directions, up/down/right/left `d = abs(x1 - x2)+ abs(y1 - y2)`. The simplest one, we can move only in right angles

```
  - 1 -
  1 x 1
  - 1 -
```

- Euclidean distance - on a grid where we can move in 8 directions, including the 4 cardinal ones, plus, moving in diagonals, it is simply the Pythagorean theorem - `d = sqrt(pow(x1 - x2, 2)+ pow(y1 - y2, 2))`. Here moving in diagonals is simply the length of hypotenuse of the right triangle formed.

```
  5 3 5
  4 x 4
  5 3 5
```

- Chebyshev distance - on a grid where we can still move in the 8 directions, as described above, `d = max(y2 - y1, x2 - x1)`. Here moving in diagonals is just as expensive as moving in the costliest direction

```
  3 3 3
  2 x 2
  3 3 3
```

In the examples above we have to simply substitute the coordinates of the current node for `x1 and y1` and then the coordinates for the target goal node `x2 and y2`, the resulting value is our heuristic function. Notice that a lower value - meaning lower distance, meaning less cost, is better, higher means - more distance between the child and goal node, meaning it is worse to take that path through that node (to the goal node) overall.

The implementation pretty much follows the base Dijkstra one, with the addition of having a special function, which calculates the heuristic cost of each node. And take a good look at how it is used, the heuristic is calculated, at the very moment the node is added to the queue, by that time we know that this distance coming from the parent is smaller than any current distance for that node, we further bias that node's cost based on the heuristic, if the heuristic is very big the cost would be big, the node would be pushed down the min heap, if the heuristic is low (maybe negative) the cost would be low, and the node would be pushed to the front of the min heap

A-star with heuristic, which always evaluates to 0, is essentially just the Dijkstra algorithm, since no node is biased towards the end goal, in other words, there is no one or more special nodes in the graph we can take to make our cost extra cheaper, we only consider the cost of the weights, there is no additional feature / property, which could further reduce our costs.

```
. . . . . .
```

```
    while (visited.size() < nodes.size()) {

        ....................................

        for (Node child : node.children) {
            // current min cost so far, to reach the child node, or default
               to some high value if none
            Integer current = distances.getOrDefault(child.value, 9999);

            // the possible new cost, note that the heuristic is still not
               calculated here, see below
            Integer possible = distances.getOrDefault(node.value, 9999) +
               node.getWeightCost(child);

            // in case the possible new distance is actually smaller than the
                distance stored so far
            if (possible < current) {
                // these remain the same as they do in Dijkstra, we update
                   both previous and
                // distances for the specific node, they must be kept in sync
                   , to use later
                previous.put(child.value, node);
                distances.put(child.value, possible);

                // take a very good note of how and where the heuristic is
                   calculated, and
                // that it is based on the child node, and the cost it would
                   take
                // to get to the goal node, and that is the value we put in
                   the
                // queue, the lower the heuristic the better, means less cost
                   .
                heap.add(new WeightNode(child, possible + heuristic(child,
                   end));
            }
        }
    }

    ....................................
```

Sorting

Topological sort

This type of sort / ordering of graph nodes works only for directed graphs, without cycles. There are many ways to detect cycles in a graph, the implementation below assumes valid graph.

Topological sorting is a way to order nodes in a graph in such a way that the nodes with the least amount of incoming edges come first. The nodes are basically ordered / sorted in increasing order of the number of incoming edges.

A very common use case would be to represent a build system, where each module is a node, each dependency between the modules is an edge. We would like to see, which modules need to be built first, second, third and so on. We would like to print the modules in the correct build order. This is where Topological sort comes in handy.

What we do is select the modules with no incoming edges, meaning they have no dependencies, then we breadth traverse their immediate children, decrease the children's incoming edge count, if a child's incoming edge count becomes 0, add that to the path, and to the processing queue, repeat until processing queue is empty

```
    List<String> sort(List<Node> graph) {
        // the queue here will receive the next node in the graph which has
           no more incoming edges
        Queue<Node> queue = new LinkedList<>();

        // the result holds the list of nodes in topological order, ordered
           front to back, at the front the nodes with the least number
        // of incoming edges - zero, and increasing
        List<String> result = new LinkedList<>();

        // put in the queue all the nodes with no incoming edges, these would
            be the ones with which we would have to start, if there
        // are none, then there is no topological order, meaning the graph
           probably has cycles
        for (Node node : graph) {
            if (node.incoming == 0) {
                // add node with no incoming edges
                queue.add(node);
            }
        }

        // while the queue is not empty, meaning we have nodes to go through,
        while (!queue.isEmpty()) {
            // pop the current node, remember, they are only added to the
               queue if they have no incoming edges
            Node node = queue.remove();

            // add the node to the result
            result.add(node.value);

            // for each child of the current node, go and decrement the
               incoming edge, child is linked to the current node,
            // therefore we have to 'remove' this incoming edge from the
               current node to the parent.
            for (Node child : node.children) {
                // 'remove' the current connection from node - child
                child.incoming--;

                // when incoming becomes zero or less, add the child node to
                   the queue, the child might have more than one incoming
                // edges, it will be added only when the last incoming edge
                   for it was removed
```

```
                if (child.incoming <= 0) {
                    // add to the processing queue
                    queue.add(child);
                }
            }
        }

        // the topological order of the graph's nodes are contained in the
           list result
        return result;
    }
```