Data structures and algorithms
Tutorial guide

Authors:

Stanislav Protasov

Adil Mehmood Khan

Innopolis, 2016

Contents

Abstract data types & data structures

Data

Consider and discuss “knowledge => data => information” scheme. Assume you drive the car. What information do you get to manage your car (speedometer value, signs values, …)? What data does this information create (speed limits, actual speed, …)? What knowledge except this information do you have (local policies, road quality, …)?

Think about importance of choosing the right data structure for the task. Start with the problem, go to the data involved in this task and speak about this data in term of operations: insertion, deletion, updates, merges or other operations. Consider the order: is it important, should the items be unique? Discuss few examples from real life: accounting system, selecting speed by self-driven car, school students database, control version system and so on. Think about data in these systems, what will be the data, what ADTs and data structures can be used. Use whiteboard to collect the answers.

In most cases we can rely on declarations (ADTs, interfaces). But we should think of internal (physical) representation (while dealing with computations: overflows, floating point sums, and so on) especially when we use primitive data types. E.g. "year 2000 problem", "Unix end of time", floating point measurements.

Algorithms

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task and must satisfy the following criteria:

1. Input: There are zero or more quantities that are externally supplied.
2. Output: At least one quantity is produced.
3. Definiteness: Each instruction is clear and unambiguous.
4. Finiteness: The algorithm terminates after a finite number of steps/instructions.
5. Effectiveness: Every instruction must be basic enough to be carried out. It must be definite and also be feasible.

You will need 5-7 sheets of paper and 5-7 pens. Each group should write and algorithm on paper of making one move in tic-tac-toe (XO, 3x3) in pseudocode. Students should provide exact instructions that should for each input field return either the address of the cell to make a move, or "return null" if move is impossible. As an input they get a field as array [1..3, 1..3], where each value is from the set {null, me, enemy}. This pseudocode will be executed by a human. Assume each algorithm is a player – build play-off table for them. Ask 2 random students to play a game for each pair of algorithms according to the rules specified on the paper. This game will most probably reveal violation of algorithm criteria 2-5 (output, finitness, definitness, effectiveness (understandable)). Discuss the problems and ways for solving them.

Interface

Consider interfaces in computer science: where they can be met (from power to USB, from programming to web-services) and that implementation is always standing behind the interface (any metal for USB, any language for web service).

Collections

Discuss collections. Refresh the ideas covered on the lecture: different interfaces to access collections, iterators, loops. How will you organize access to all elements of collection, list, and dictionary? Imagine and create (in groups) an interface, that will allow to access all collection

elements one-by-one, but it will be not a list, dictionary, array. Let them present their ideas in front of others.

Complexity estimation

Time measurement

Time is continuous, machine clock is discrete. We should design programs to work in "real time" but rely on machine time for measurement.

Machine measures time in two ways: for itself and for human. For itself in just counts ticks, for human it converts ticks to date/time. Converting involves leap year, leap seconds, UTC shifts (e.g. Kazan UTC +3) and NTP (network time protocol) synchronization (for auto correction).

How machine measures time? Clock is a microchip that generates frequency. It counts in **ticks** that are usually about nanosecond long (usual CPU frequency is 1-3 GHz). Clock generates just ticks, but there's also TSC (Time Stamp Counter) - mechanism that allow us to get the number of ticks from some moment (maybe machine last start, maybe first start depending on hardware implementation). It is very precise and it can be used to measure relatively short time intervals. ASM instruction for this – RDTSC ("Read TSC"). In Java you can access it using `System.nanoTime()` – that returns nanoseconds. TSC is sensible to sleep mode, so you cannot use it for long measurements that can involve hibernation.

Another way is to rely on "human" time representation – in this case we (or operating system) time and consider shifts (UTC, summer time, …). Sensible to clock changes, so you should not rely on it for short interval measurement. What happens if a person moves from one time zone to another? Use `System.currentTimeMillis()` to get current milliseconds.

Also talk about that scheduler affects your program (it can pause the execution of any thread), so different program runs take different time (even with the same parameters). So you should measure execution multiple times to get reliable results.

Limitations and complexity

Consider the idea of importance for memory limitations (time is unlimited, memory is limited) and problems that can be produced by bad memory design: XML bombs for XML DOM vs push/pull SAX for embedded systems. Stack overflow. Discuss memory leaks effect.

There are also other types of "complexity". Cyclomatic complexity measures if the code is "understandable" according to static code analysis. This allows to get metrics on number of tests for the method.

Practice

Implement "counting sort" algorithms that sorts an array of integers. Use `Math.random() % K` to fill array where K is data value upper limit. **Let K = 10 000**.

a. Calculate the time complexity for implemented algorithm using exact technique, as it is shown on lecture. Assume **reading single value takes log(K)** for each array item. Discuss it and re-estimate complexity.
b. Implement time measurement for the algorithm. Measure time using `System.nanoTime()` for array size of 100, 1000, 10000, 100000, 1000000 elements in array. Build a graph (Excel, gnuplot, ...) to see the dependency.
c. Use `Runtime.getRuntime().totalMemory()` to measure **memory consumption** for this algorithm. What is the space complexity of the algorithm?

d. Vary **K from 10 000 to 10 000 000** to see how it affects time consumption. Discuss the result. See also how it affects **memory** consumption.

Complexity and real life

If algorithm is run on predictable data set it is sometimes better to use worse algorithm (in terms of O-notation).Refresh them intersecting graphs and **example from lecture** ($0.01*n^2$ *vs* $8n*\log(n)$). Provide more them think of examples.
Also, in theory some algorithms can be similar, but practice forces you to select a good one. E.g. on embedded system you will most probably not use count sort: even better use bubble or other simple sort that does not consume memory if data size is relatively small.
Usually difference is on **initial stage**. You prefer to run algorithm in parallel if it works with large data, but for small data thread initiation will cost more than execution.

Practice

Implement "**counting**" part of counting sort **in parallel**. Compare results. Find balance point for data size where approach should be switched.

Lists

Theoretical part

1) Why list, not sets/maps:

- Sometimes **order** is very important (ask students. Examples: instructions consisting of non-commuting operations; you want to have easy "**next**" operation; representing polygons).
- Sometimes you want to keep you set **always sorted** (e.g. to get max of min element easily, or for priority queue).
- This is the simplest and the least **space-consuming structure**. Best choice for immutable sets. (Constant overhead for array-based, 4(8)bytes*N overhead for linked list).

2) Which implementation to select?

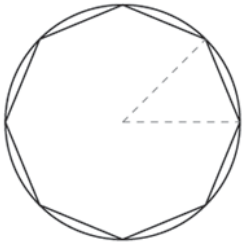
Draw this table without filling it (first 6 rows). Ask students to help you to fill complexity values in this table. Decide which implementation is better. Tell them about additional (*, **, ***) problems of array-based implementation.

Operation	Array-based	List-based
`size()`	O(1)	O(1)
`isEmpty()`	O(1)	O(1)
`get(i)`	O(1)	O(n)
`set(i, v)`	O(1)	O(n)
`add(i, v)`	O(n)*	O(n)
`remove(i)`	O(n)**	O(n)
Draw lower part later		
`addFirst(v)`	O(n)	O(1)
`a.concat(b)`	O(m) ... O(n+m) ***	O(1)

** - add() in array-based sometimes may also involve array resizing;*
*** - remove operation in array-based implementation blocks the list for all the time. It is really bad for parallel implementation.*
**** - depending on if we need to resize an array.*

After filling main part of the table. Add 2 remaining rows. Discuss in which situations will they choose first or second.

Practice



1) We need to create a polygon representation in Java. We will use **Cartesian coordinates** to represent the point, and **list** to aggregate them. Each point stands for **vertex**. Each two neighbor points represent the **edge** (+ one edge is between first and last points). Your task is to **generate polygon** approximating some complex curve. **Write a loop** that will calculate coordinates of polygon vertices. You may generate this data using any well know equation: circle, Lemniscate of Bernoulli or cardioid. Good way is to use parameter t = 0..N-1 and polar coordinates equation for each Θ = 2*PI * t/N. Then find (x, y) = (ρ*cos Θ, ρ*sin Θ).
2) Implement **single-linked list as a class,** where you will store polygon points. Your class should be able at least to **insert**, **delete**, **get** and **update** value **by index**.
3) Implement *Iterable* interface for your structure and *Iterator* class. Use foreach to iterate through list.
4) **Implement crossing number algorithms** for "point in polygon" problem (some kind of `static boolean inside(YourListType polygon, Point2D point)`) Short explanation: for point of interest we build a random ray. If this ray intersects polygon even number of times (0, 2, 4, ...) – point is outside, else point is inside. Use Kramer rule to detect if lines do intersec: checks if **ab** intersects with **cd** using `javafx.geometry.Point2D`).

```
public static boolean intersects(Point2D a, Point2D b, Point2D c, Point2D d) {
        // We describe the section AB as A+(B-A)*u and CD as C+(D-C)*v
        // then we solve A + (B-A)*u = C + (D-C)*v
        // let's use Kramer's rule to solve the task (Ax = B) were x = (u, v)^T
        // build a matrix for the equation
        double[][] A = new double[2][2];
        A[0][0] = b.getX() - a.getX();
        A[1][0] = b.getY() - a.getY();
        A[0][1] = c.getX() - d.getX();
        A[1][1] = c.getY() - d.getY();
        // calculate determinant
        double det0 = A[0][0] * A[1][1] - A[1][0] * A[0][1];
        // substitute columns and calculate determinants
        double detU = (c.getX() - a.getX()) * A[1][1] - (c.getY() - a.getY()) * A[0][1];
        double detV = A[0][0] * (c.getY() - a.getY()) - A[1][0] * (c.getX() - a.getX());
        // calculate the solution
        // even if det0 == 0 (they are parallel) this will return NaN and comparison will fail -> false
        double u = detU / det0;
        double v = detV / det0;
        return u > 0 && u < 1 && v > 0 && v < 1;
}
```

   a) Test your method for sections
      i. (0,0)-(10, 10); (10, 0)-(0,10) - true
      ii. (0, 0)-(10, 10); (1, 0)-(11, 10) - false
      iii. (0, 0)-(10, 10); (1, 0)-(50, 10) - false
   b) Use Monte Carlo method of integration to calculate the area of the polygon.

Discuss skip list data structure.

Recursion and search

Theoretical part

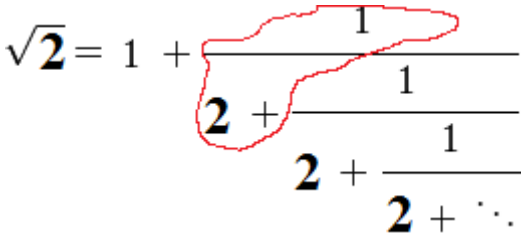
Recursion is representing a function that calls itself, maybe, with different parameters. It applies to math, philology, programming etc.

Stop rule

There are **two types** of recursion: finite and infinite. For finite one you have "**stop rule**" – condition in which we know the result of the function ("**base case**" in lecture). Infinite recursion is useless in

programming but useful in math – (representing different recurring sums and continued fractions). For calculating infinite math formulas, we can use stop rule for the sake of accuracy.

Example:



```
public static double squareRoot2(int deep) {
	if (deep == 22) return 1;
	// deep == 10 - diff in 10th sign after point
	// deep == 20 - diff in 16th sign after point
	// deep == 22 - equal in double
	double value = 1 / (2 + squareRoot2(deep + 1));
	if (deep == 0) value += 1;
	return value;
}
```

Recursion and stack

Inside recursion there always lives a Stack. This is a part of programming paradigm that allows us to call subprograms. When we call one method from another, Java **push()** our **return address** and local data (maybe call parameters in some implementations) into stack. When this invoked methods ends, it **pop()** this data to know, what to do next (where to return). In Java we cannot access this stack directly, but we can view it:

```
public static int binarySearchRecursive(int[] array, int value, int low, int high) {
	System.err.println("Current range [" + low + ", " + high + "]");
	/* this method dumps stack of current thread to console */
	Thread.dumpStack();
	if (low > high) return -1;
	else {
	  int mid = (low + high) / 2;
	  if (value == array[mid]) return mid;
	  else if (value < array[mid])
	    return binarySearchRecursive(array, value, low, mid - 1);
	  else
	    return binarySearchRecursive(array, value, mid + 1, high);
	}
}
```

When you run this code, you are able to see line numbers, (return addresses in fact). As you can see, most of them contain the same method names.

Recursion == stack?

There is a type of recursion called “tail recursion”, because recursive calls are situated in the very end of the method, and their results are just passed up as a result of the method (no assignments are conversions). Theoretically stated you can convert any tail-recursive algorithm to a **loop+stack** algorithms.
Let’s try to do it with binary search:

```
public static int binarySearchNaive(int[] array, int value) {
	int low = 0, high = array.length - 1;
	Stack<int[]> stack = new Stack<>();
	stack.push(new int[] { low, high });
	do {
		// we "call a function" with parameters on the top of stack
		int[] borders = stack.peek(); // NB: never pop!
		low = borders[0];
		high = borders[1];
		if (low > high) return -1;
		else {
			int mid = (low + high) / 2;
			if (value == array[mid]) return mid;
			else if (value < array[mid])
				// return binarySearchR(array, value, low, mid - 1);
				stack.push(new int[] { low, mid - 1 });
			else
				// return binarySearchR(array, value, mid + 1, high);
				stack.push(new int[] { mid + 1, high });
		}
	} while (true);
}
```

It works! Can we avoid the stack? We never **pop()** (the same idea – we just return the value from the deepest method call), so it just grows!

```
	public static int binarySearchLinear(int[] array, int value) {
		int low = 0, high = array.length - 1;
		// Stack<int[]> stack = new Stack<>();
		// stack.push(new int[] { low, high });
		do {
			if (low > high) return -1;
			else {
				int mid = (low + high) / 2;
				if (value == array[mid]) return mid;
				else if (value < array[mid])
					high = mid - 1;
					// stack.push(new int[] { low, mid - 1 });
				else
					low = mid + 1;
					// stack.push(new int[] { mid + 1, high });
			}
		} while (true);
	}
```

Practice

Implement **recursive binary search** (reference lecture slides). Use **Thread.dumpStack**() to see the stack state on each iteration. It should return the **index of the element** or -1 otherwise.

Sorting

Theoretical part

Sorting allows us to do multiple tasks:

- Sorted elements can be found in O(log(n)) for search using binary/interpolation search (or even a place they should be inserted)
- They allow us to select linked areas of data in O(n) time (SELECT * FROM T where C1 > V1).

- They allow us to make cool math (Principal Component Analysis include sorts; create a polygon out of points; compression with loss algorithms involve sorting)
- They allow us to represent data for humans.

Faster sorts

There's a class of sorts that allow us to do it in *n-log-n* time: QuickSort (involves recursive strategy), heap sort and merge sort. Some of them provide O(n* log(n)), some – Ω(n*log(n))

O(n)-sorts

Unlikely bubble and insertion sort, there's a class of algorithms that does not involve comparison. All O(n) sorts are non-comparative, because it is proven, that comparative sorts cannot perform better then Ω(n*log(n)). There are also well-known non-comparative sorts:

- **Counting sort.** O(n+k).

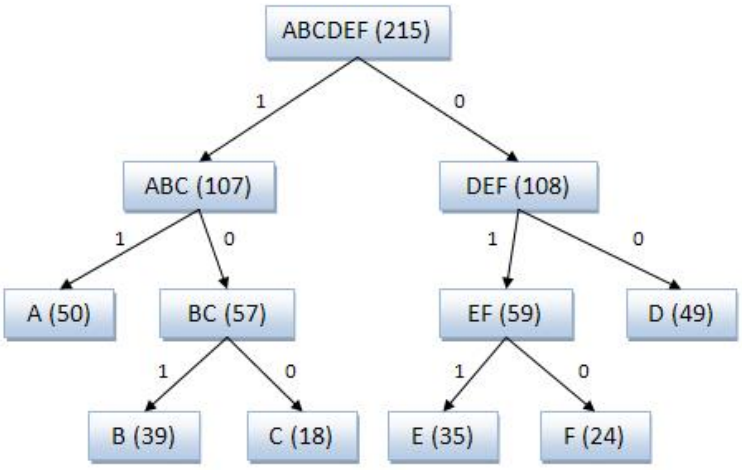
```
public static void countSort(int[] array) {
	int max = Integer.MIN_VALUE, min = Integer.MAX_VALUE, idx;
	for (int i = 0; i < array.length; i++) {
		max = Math.max(max, array[i]);
		min = Math.min(min, array[i]);
	}
	int[] counts = new int[max - min + 1];
	for (int i = 0; i < array.length; i++) {
		idx = array[i] - min;
		counts[idx]++;
	}
	idx = 0;
	for (int j = 0; j < counts.length; j++)
		for (int k = counts[j]; k > 0; k--)
			array[idx++] = j + min;
}
```

- **Bucket sort**. Idea is to split the range of values into buckets [1-50, 51-100, …]. Then **Scatter**: place items in the buckets according to the value (value / Nbuckets). Sort each non-empty bucket. Then **Gather**: collect values back into array. Worst case: **$O(n^2)$**, best/average case **T(n+k).** Study this sort and try to implement it for integers.
- **Radix sort**. Similar idea: use digits as buckets.

Binary search tree

Binary trees in common

One of the old and very efficient methods of compression for natural language is **prefix code**. This is a code of variable length, in which code for a char will never start with the code for other char. E.g. [0, 11, 101] alphabet is ok, [0, 1, 10] is not ok. **Code [binary] trees –** graphical representation of such code. Assign 0/1 to left edge and 1/0 to right. Chars – are always leaves.



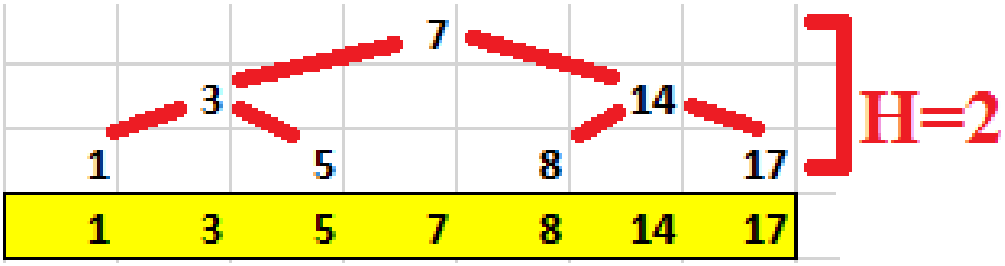
Building such a tree is a purpose of an algorithms, that base on probability. Most popular codes used even now are Shannon's code and Huffman's code.

Binary search tree

BST is a binary tree with some constraints **on data**: all the properties and operations remain the same. We still have tree height, root, and so on. The only constraint on data – is to preserve tree order: everything on the left is smaller; everything on the right is greater. Why should we use tree? This is because of the balance of CRUD operations. Let's consider these operations in the following order.

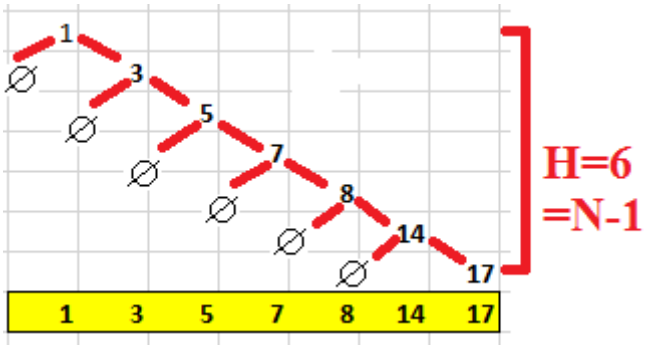
Read/Update

Read (find) – we can consider BST as pre-calculated binary search middle elements. Just imagine you run binary search algorithm for all possible values and record all middle values in a tree. Search process remains the same, that means complexity should be the same – O(log(n)). Build a binary search example (H = 2 = *ceil*(*log2*(n)) - 1):



Binary search algorithms will always converge in not more than T(log(n)), because it's search tree is **balanced by construction –** we always divide interval into equal slices. (*Refresh balanced tree theory here*). As a result – search of any non-present element (the longest search) will always take **equal time**.

For general BST these assumptions are false, because the tree **depends on the order of element insertion**. Just remember **degenerate** example. Let's add elements from our array one-by one (1, 3, 5, 7, 8, 14, 17), not in order of binary search. Resulting tree can be still used for search, but…



Now we can see, that “balanced” property is very important to preserve tree in balanced state.

Create

Create (add) – in common case complexity will be **O(find) + O(1)**. In case we would like to preserve tree balanced, it can take longer, but never longer than O(1: for rotations).

Delete

Delete has **3** options:

1) We delete leaf element – O(find) + O(1: to set null).
2) We delete element **– O(find) + O(1: to relink subtree to parent)**.

   *Discuss whether this complexity estimation is always true. Obviously **not**, because it strictly depends on internal tree representation. What will happen when we will try to move subtree one level up in array-based tree implementation? – moving array elements up will take O(n) operations and will keep tree in inconsistent mode for this time.*

3) Deletion with 2 leaves: O(find) + O(find: for max element) + O(1: to swap).

   Is this case better or worse for array-based implementation? Why? – (better, because search is fast, and swap is O(1))

You can make the following conclusion that according:

1) BST data structure is similar to binary search algorithm (according to *find()* method).
2) BST can make all **CRUD** operations in O(log(n)) ***iff*** tree is balanced. Keeping tree **balanced** – is a task for extended DS like AVL- and RB-trees.
3) Using array-based implementation is nice when you have immutable tree, but deletions and rotations will be painful, that is why better not to use it.

Practice

1) Implement linked binary search tree. Interface includes at least **find**(), **add**(), **delete**(), **size**() methods.
2) Write a method that calculated tree height. Can you do it for O(n)?
3) Create
   a. One BST and build it from your own binary search implementation (put mid-element values in a tree).
   b. Second BST and build it from ordered array.
4) Calculate both tree’s height.

AVL tree

Theoretical part

AVL trees are strictly balanced trees, that means it **always** stay in the state |h(left)-h(right)| <= 1. This leads us to **O(log n)** for read. There are also approaches balanced on probability, but they can promise only T(log n) for average case.

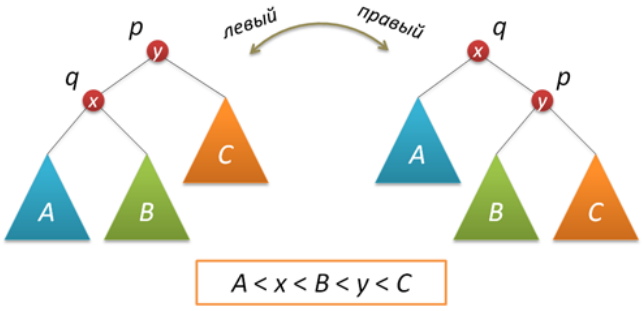
Search

AVL tree algorithms is a combination of plain binary tree and balancing algorithms. This means ***Search (find)*** operation will remain the same.

Restructuring/ Rotations

Restructuring approaches for AVL trees are also called **rotations**. To perform rotation on AVL tree, there should be special conditions – tree should be in unbalanced state. This condition depends on **balance factor = h(left) – h(right)**. It should be {-1, 0, 1} for each node. If it is {-2, 2} for any node – tree starting from this node (and up to root) should be restructured to reduce this difference. To have this value you can either store height of subtree or balance factor itself in each node.

The simplest way to balance subtree is to do **simple rotation**.



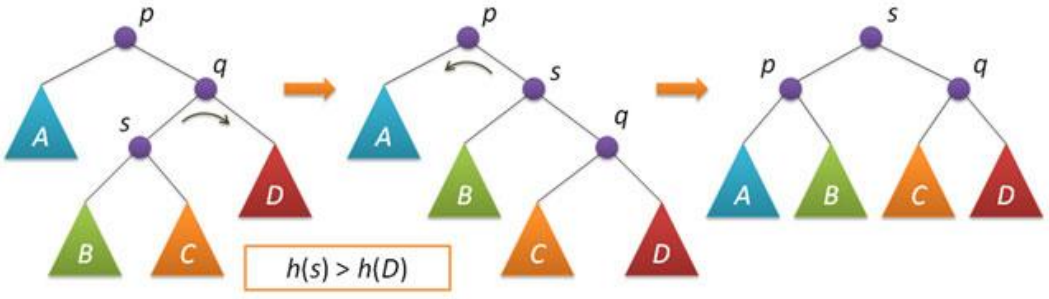
But this will work only
(for right): ***h(q) = h(C) + 2*** *&&* ***h(B) <= h(A).***
(for left): ***h(p) = h(A) + 2*** *&&* ***h(B) <= h(C).***

In other words “middle” subtree (B) should not be “higher” that side subtree (A for right rotation) of the same node before rotation. Else, it will again bring subtree in unbalanced state.

What should we do in other case?

Let’s do more complex things – **big rotation**. If “**middle**” subtree is “**higher**” than side subtree, we can firstly fix it. For left rotation it will be as follows:

Rotate (**q**) subtree in *opposite direction* to make **left branch of it subtree shorter (or equal) than right branch**. Then the condition for simple rotation will become true.



Insert

Now, to insert the element to the tree, you have to

1) Follow BST rules
2) Rebalance all path from inserted element to the root node.

```
public static <T> Node<T> insert(Node<T> root, Node<T> newNode) {
    if (root == null) return newNode;
    if (newNode.key < root.key)
            root.left = insert(root.left, newNode);
    else
            root.right = insert(root.right, newNode);
    return root.balance();
}
```

Delete

To delete the element, you have to:

1) Follow BST rules
2) Rebalance starting from “replacing node” (that was inserted instead of deleted node) to the root node.

```
Node<T> remove(int k) {
    if (k < key) left = left.remove(k);
    else if (k > key) right = remove(k);
    else {  // if (k == p->key)
            Node<T> q = left;
            Node<T> r = right;
            if (r == null) return q;
            Node<T> min = r.findMin();
            min.right = r.removeMin();
            min.left = q;
            return min.balance();
    }
    return balance();
}
```

Practical part

Having BST implemented:

1) Implement rotation methods in your Linked Tree classes. Carefully add balancing operation to insert() and delete() implementations. Test your tree on with the following actions:
    a. **insertion sequence**: *March, May, November, August, April, January, December, July, February, June, October, September*.
    b. **deletion sequence**: April, August, December.
    c. **find sequence**: January-December (all months present except deleted).
2) Measure algorithm complexity properties.
    a. Insert 1000000 (1M) values from 0 to 999999 into the tree.
    b. For each insertion, measure **operation time** and **tree height**.
    c. Delete elements from 0 to 999999.
    d. For each deletion, measure **operation time** and **tree height**.
    e. Build 2 graphs that show operation time and tree height depending on number of elements in a tree. Do they fit O(log n) estimation?

Red-black tree

Theoretical part

Properties

Red-black trees are balanced trees, but balance property is different – path from root to any leaf should include the **same number of black nodes**. This property itself means nothing, but according to other rules longest path will never be more than twice longer, than the shortest. These rules are:

- The root is black.
- All leaves (NIL) are black. Usually we just consider null pointers as black leaves.
- If a node is red, then both children are black. (in other words, never 2 red nodes in a row!)

Insertion

Start insertion as you do in BST. If new node is first in a tree (root) – just make it black (to preserve the rules). Otherwise – make it red.

If **parent is black**: everything is fine (none of rules are violated). We are done.

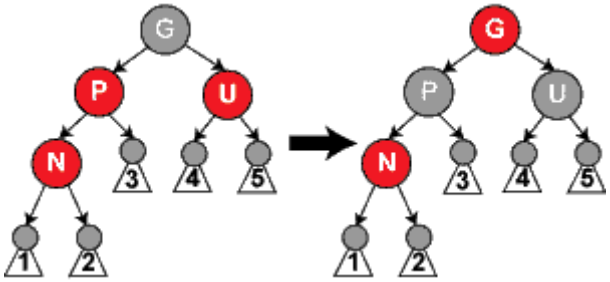
If **parent is red**, we've got the problem. "**Double red problem**".

Double red problem

1) **Parent and uncle are red**: color them in black and make grandfather red (flip color between levels is preserving "black count" rule). This can lead to:

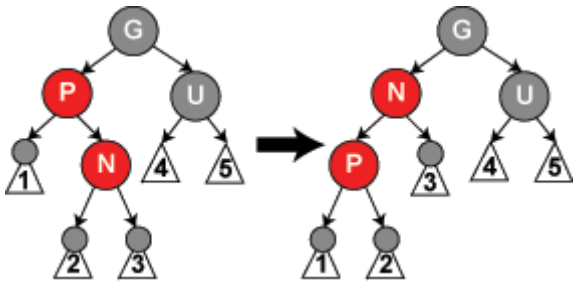
- grandfather is root - make him black.

- great- grandfather is red - restart insertion from grandfather's point (assume we inserted grandfather's node, so we again have "double red problem").



2a) **Parent is red, uncle is black** (+inserted is right, parent is left): rotate left (N becomes parent for right parent)

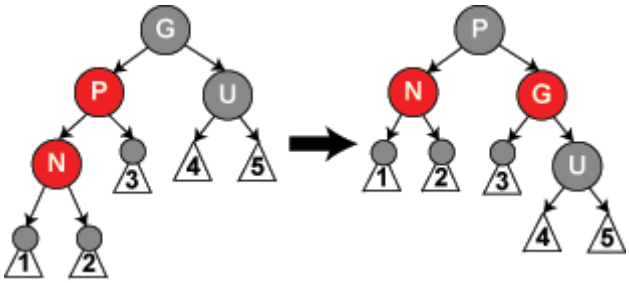
2b) **Parent is red, uncle is black** (+inserted is left, parent is right): rotate right (N becomes parent for left parent)



After (2) tree is still in violated state. So, go on to (3) for **ex-parent** (P node).

3a) **parent is red, uncle is black** (+inserted is left, parent is left): rotate grandfather to the right, make him red, parent - black.

3b) **parent is red, uncle is black** (+inserted is right, parent is right): rotate grandfather to the left, make him red, parent - black.



Deletion

When we delete from BST, then we replace deleted node with successor/predecessor that has maximum one child (always). That's why we can consider deletion as deletion of single-child (not more than single child) element always (is node has no children – we can call "child" any of NULL references).

Deleting red

Deleted is red - replace deleted with a child-subtree (all properties preserve).

Deleting black

Easy case

Deleted is black, child is red - replace deleted with a child and make this child black.

Hard case

Deleted and child nodes both black (e.g. when deleted is the last node with 2 nulls). Removal of any black node unbalances the tree.

1) Firstly, replace deleted node with his child.

2) Brother (sibling) of deleted is now brother of new node.

And now let's go:

1) **Deleted was a root node;** his child is a new root. We are done.

2) **Deleted had a red brother**. Flip parent and brother colors and then rotate left (right) over parent to make brother the grandfather of son. Then go to 4, 5, or 6.

3) **Parent, brother, brother's children are black**. Make brother red. Start from (1) for a parent (he is black)!

For next conditions consider child node of deleted becomes left child after deletion.

4) **Parent is red, sibling and his children are black**. Flip their colors, we are done.

5) **Brother is black, his children are different and right is black** (but we want him to be red!). Rotate right over brother to bring red on top. Then flip his and brother's colors. goto (6).

6) **Brother is black, his right child is red.** Rotate left over parent. Flip parent's and sibling's colors. Make ex-sibling's right child black.

For 4-5-6 consider symmetry.

Practice

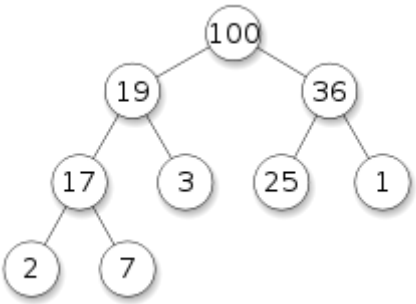
1) Implement Red-Black node class.
2) Implement “uncle” and “grandparent” methods.
3) Implement insertion operations for RB-tree.
4) Measure algorithm complexity properties.
    a. Insert 1000000 (1M) values from 0 to 999999 into the tree.
    b. For each insertion, measure **operation time** and **tree height**.
    c. Build 2 graphs that show operation time and tree height depending on number of elements in a tree. Do they fit O(log n) estimation?
    d. Compare to AVL tree measurement on the same graph.

Heap and heap sort

Heap properties

Heap is a balanced binary tree, and it is **different from BST.** Heap properties are:

- **Heap is a complete binary tree**. That means all levels are full (except, maybe the last), and last level nodes are filled *from left to right*. Property obviously leads to the idea that if you store heap as an **array-based k-ary (2-ary) tree**, data alignment will be compact (no *null*s inside the array). We need from 2^{h-1} up to 2^h-1 element array to store a tree of height **h**. The worst and the best cases for space consumption are different only by multiplier.
- Each parent node is **greater or equal than both children** (max-heap) (less or equal – min-heap). Yes, in heaps we are definitely allowed to store equal keys.



Idea of heap ADT is to preserve the smallest (biggest) element on the top to allow very fast access and deletion of this value. When you put elements to DS, and **pick the topmost element without any parameters** (using some rule), this approach is called **pool** (stacks and queues are pools). There is a data structure (container) that expects exactly this behavior for implementation – **priority queue**. This is a kind of queue, but you provide not only element, but **also a priority number** (key), that can help an element to move faster through the queue. Will it work as a normal queue if all the elements has the same priority (yes)?

Heap operations

Everything was ok with insertion until this moment…

Upheap

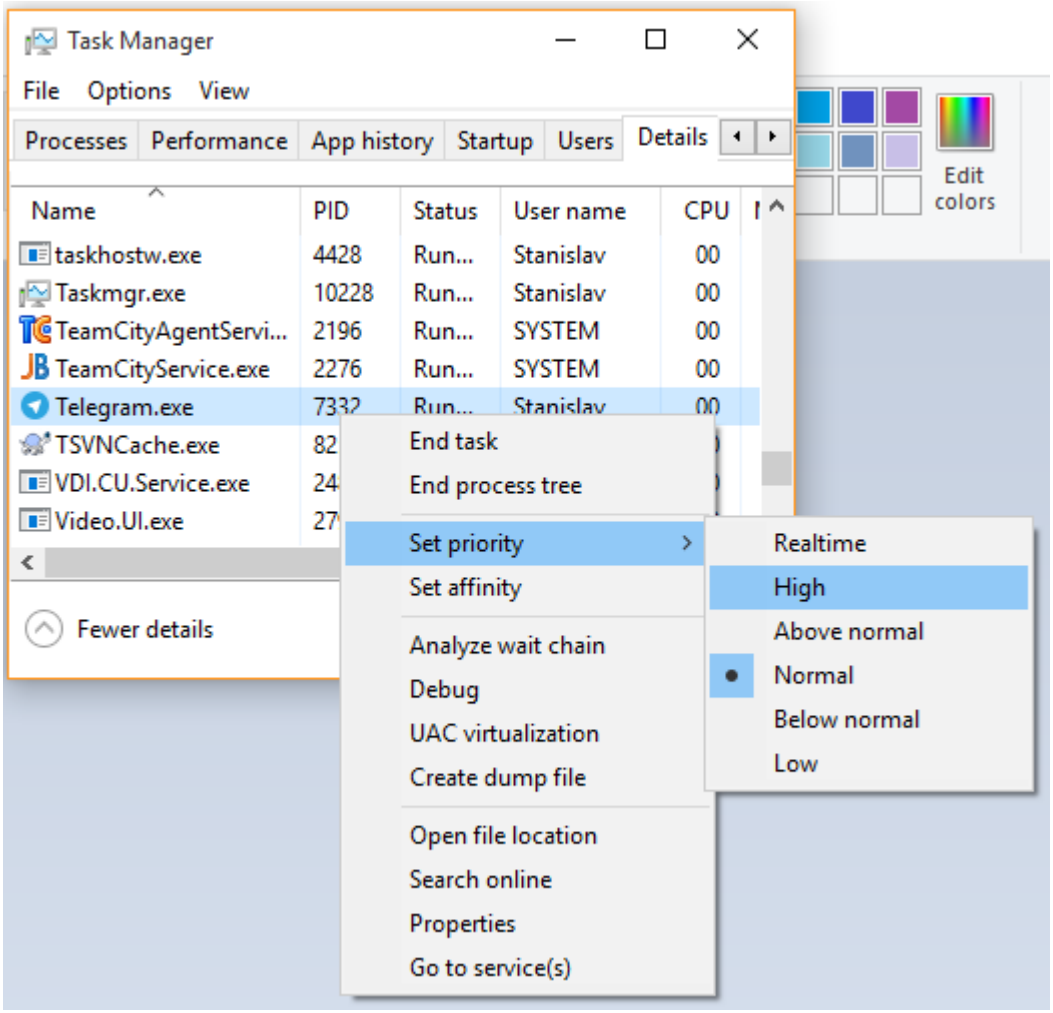
When you insert a value into an array-based heap, you just add new value as a last element (according to fill property of complete tree).

What will happen if after insertion **parent value is less than inserted**? Let's become a bubble! Proceed changing values with parents. Do it either with tail recursion, or a loop.

```
while (node.parent != null && node.value > node.parent.value) {
      swapvalues(node, parent);
      node = node.parent;
}
```

Heap height is O(log n). This can make us confident, that loop will end in O(log n) time.

What are other upheap applications? Consider you use heap for scheduler.



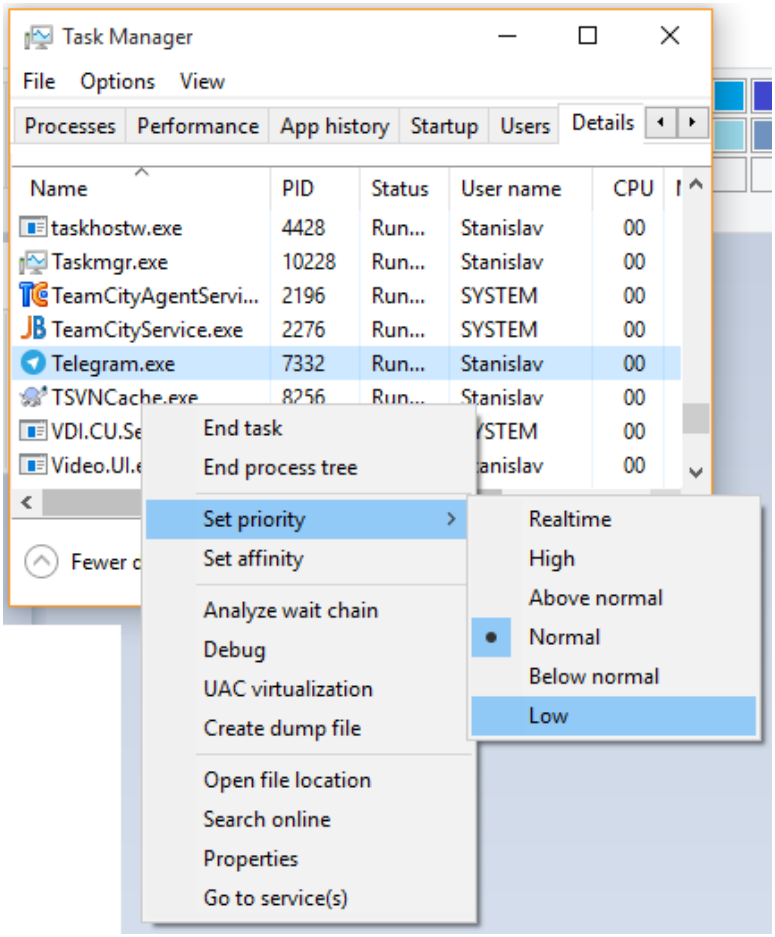
This operation requires to reconsider place for some node in a tree. How can we do it? Upheap!

Downheap

Now let's remember about the main idea of the data structure. It was designed to preserve the smallest (biggest) element on the top to allow very fast access and **deletion** of this value. As you delete the topmost (or first element in terms of array-based heap), you will now have 2 orphan subtrees. How can we fix it? Let's put **the last** (most probably one of the smallest) element to the top. But we know this deadbeat doesn't worth with place. So, let him **sink to the bottom:**

1) Select the biggest child.
2) Swap.
3) Repeat.

What are other downheap applications?



Now we know, how to:

1) Search for the greatest element (peek(), dequeue())
2) Insert element (add last + upheap)
3) Delete element (replace with last and downheap)
4) Update priority (either upheap or downheap)

Building a heap

What else can we do with the heap? We can build a heap from arbitrary data in O(n log n) time. How? There are few approaches all O(n log n). The easiest way is just start inserting elements from the first to the last. But there's a way to do it faster.

1) Take elements starting from the last one by one (N)
2) Proceed downheap (1 .. log N). This will move bigger elements closer to the beginning of array.
3) Repeat until the first element (Sum of N/2 * log(N) + N/4 * (log(n) -1) + …) -> n log n

Heap sort

We know how to build a heap from arbitrary array in O(n log n) time and O(1) time. Let's use this to build a sorted array. That's easy.

1) Build a heap from arbitrary array (O (n log n))
2) Loop N times
    a. Swap last element and first. Now last element in array is the biggest. O(1)
    b. Reduce heap size by one. We now do not consider last element as a part of array. O(1)
    c. Downheap. (O(log n))
    d. Repeat.

Estimate complexity.

Practice

1) Implement array based heap as a **Map** data structure.
    a. Downheap
    b. Upheap
    c. Heap building
2) Implement Priority Queue using your Heap.
3) Implement heap sort.

Quick sort and merge sort

Merge sort

Divide and conquer approach: we split a task that we don't know how to solve into a smaller subtasks that we know how to solve. These tasks are usually expressed in recursive algorithms. E.g. binary search.

Idea of merge sort is pretty close to binary search:

1) Split an array into 2 equal parts (by size)
2) **Sort them separately**
3) Merge these parts O(n)

As you can see, inside the sort algorithms we use sort. Can we use recursion to solve this problem?

1) We know base case: array of 0/1 elements is already sorted.
2) Each time we reduce array size twice. So, one day (in **log n** time) it will become 0 or 1.

Now the only problem now is to implement merge of 2 ordered arrays.

```
void Merge(
       int *A,
       int *L, int lCount,
       int *R, int rCount) {      |M(L+R = m) = 6 + m + 1 + 3 * m * 3 = 10m + 7
       int i,j,k;                                                    |3
       i = 0; j = 0; k =0;                                           |3
       while(i < leftCount && j < rightCount) {                      |m + 1
              if(L[i] < R[j]) A[k++] = L[i++];                       |
              else A[k++] = R[j++];                                  | max(3*m)
       }
       while(i < leftCount) A[k++] = L[i++];                         |max(3*m)
       while(j < rightCount) A[k++] = R[j++];                        |max(3*m)
}
```

```
void MergeSort(int *A,int n) {          S(n) = 2 * S(n/2) + M(n/2) + k*n + l
       int mid, i, *L, *R;                                  |4
       if(n < 2) return;                                    |1
       mid = n/2;                                           |1
       L = (int*)malloc(mid*sizeof(int));                   |n/p
       R = (int*)malloc((n - mid)*sizeof(int));             |n/p
       for(i = 0; i < mid; i++)                             |n/2*2 + 1
              L[i] = A[i];                                  |n/2
       for(i = mid;i<n;i++)                                 |n/2*2 + 1
              R[i-mid] = A[i];                              |n/2
       MergeSort(L,mid);                                    |S(n/2)
       MergeSort(R,n-mid);                                  |S(n/2)
       Merge(A,L,mid,R,n-mid);                              |M(n/2 + n/2) = M(n)
    free(L);                                                |1
    free(R);                                                |1
}
```

```
S(n) = 2*S(n/2) + [M(n) + n* (2/p+3)*n + 10] = 2 * S(n/2) + t*n + q
S(1) = 4
S(n) =  2*S(n/2) + t*n + q

        =  4*S(n/4) + 2*t*(n/2) + 2*q + t*n + q
        =  4*S(n/4) + 2*t*n + 3*q

        =  8*S(n/8) + 4*t*n/4 + 4*q + 2*t*n + 3*q
        =  8*S(n/8) + 3*t*n + 7*q
        =     ....
        =  L*S(n/L) + log(L)*t*n + (L-1)*q          | n == L
        =  n*S(1) + log(n)*t*n + (n-1)*q
        = (4+q)*n + t*n*log(n) - q = O(n log n)
```

Quick sort

This is another sorting based on recursive approach (divide and conquer). Select random middle. Compare all elements with the middle and place smaller in left part, bigger [and equal] in right. Splitting of elements takes Θ(n). Then repeat this for left and right parts recursively.

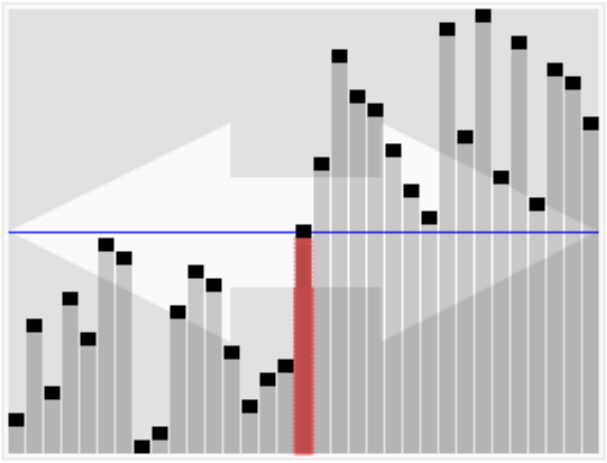
Best case: each recursion splits array into equal parts. So, it will happen $\log_2 n$ times (depth of recursion, same as binary search). So we have $\Theta(n) * \Omega(\log_2 n) = \Omega(n) * \Omega(\log_2 n)$ = **$\Omega(n^* \log_2 n)$**

Avg case: only probabilistic approach. **T(n* log n)**

Worst case: each time we select biggest (smallest) value element as middle. This leads to situation, that next level or recursion will deal with array of **n-1** size. Having n + (n-1) + … = n*(n+1)/2. This means $O(n^2)$

- 1) Split array into two parts where right parts dominates on left.
- ∃(*pivot* from i..j) ∀(*m* from i..j)

- if (*pivot* < m) then A[*pivot*] < A[m]
- if (*pivot* >= m) then A[*pivot*] >= A[m]
- 2) Sort both left and right parts recursively



```
QSort (array, low, high) {
      mid = split(array, low, high);
      QSort(array, low, mid-1);
      QSort(array, mid+1, high);
}
```

Graph representation

Graphs are special data structures that come not from algorithms (as hash tables and trees do – they solve the problem of efficiency for the same task), but from real-world/mathematical problems. Graphs are very good at describing problems, *modelling real-world cases*, where something (or someone) can "travel" between base points in unpredicted way. E.g. tourist is moving from town to town, electron is traveling over the circuit, TCP-frame can travel though internet, your work activity is travelling from one task to another, etc. Having this as a model, we can answer common questions about these journeys:

- *What is a **shortest path** for [tourist/TCP-frame/worker/…] to get from A to B in terms of [time/resistance/hops/…]? (shortest path)*
- *How can […] **visit all the points** in the fastest way? (travelling salesman problem)*
- *What is the **minimal number of** [roads/links/…] can remain to preserve connections between base points? (spanning tree)*
- *How many **different** […] can you make, that **none of the neighbors will have the same**? (coloring)*
- *How can I arrange my visits to be sure that some points will be accessed always after other points? (concerts, scheduling … - topological sort)*

As you can see, these questions can be asked in multiple practical cases, and answers for them are **existing graph-based algorithms**.

To solve using graphs, you should be able to do 4 things:

1) Understand graphs.
2) Formalize you problem domain as a graph (special type of graph: oriented, not oriented, weighted, complete, connected, …).
3) Implement graphs.
4) Know that there are graph-based algorithms, which can solve your problem.

Edge list structure

You graph data structure contains 2 lists: one is a list of vertices, another – list of edges. Why do we need both:

Graph can be not connected, but we still need a tool to visit all nodes and edges in O(n) time (e.g. you update road tax, or recalculate flight prices, …). Otherwise we will not be able to reach graph parts.

Adjacency list structure

We can add neighborhood lists to make `adjacent()` calls faster (scan not all the collection, but only corresponding edges).

Adjacency matrix structure

Adjacency matrix is the best one in terms of access speed – You can find nodes and edges in O(1) time, but they case redundant memory consumption for spare graph cases. Consider AMS as *edge list structure* + array containing connections for faster access:

```
private ArrayList<Edge> nodeConnections = new ArrayList<>();

private Edge getEdge(Vertex from, Vertex to) {
        return nodeConnections.get(vertices.size() * from.listPosition + to.listPosition);
}
```

For lightweight implementation, you can keep only weight inside the cell (not edge). Also, think how you will delete and add nodes to DS containing such a matrix.

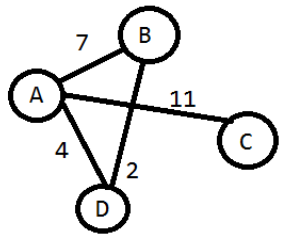
Practice

4) Implement adjacency list structure with methods:
    a. Add vertex
    b. Remove vertex
    c. Add edge
    d. Remove edge
    e. Get adjacent vertices
5) Write a code that can fill your graph from the following file:

```
A B C D E F G H Kolya Vasya
A B 5 D Kolya 1 G Kolya 5 Kolya Vasya 12 F B 7
```

*where first line contains node names separated by spaces, and second line contains triplets of **origin**, **destination** and **weight** for edges.*

E.g.




```
A B C D

A B 7 A D 4 B D 2 A C 11
```

6) Write the code that will serialize graph in the same way.

Graph traversal: DFS, BFS

Traversal: DFS (Depth first search)

DFS (even it is called SEARCH algorithms) is mostly a **traversal** algorithms with a beautiful recursive implementation that can lead to stack overflow for dense graphs (that's why better use iterative

implementation). For search it is used in **trees**. Usually algorithm is used as a part of greater tasks: in topological search, Dinic's algorithms for maximum flow calculation (for transport networks), etc. Naturally, this approach is used to **find way out from a maze** without any heuristics.

Consider depth first search as **preorder tree traversal of spanning tree**. We are interested in visiting all **nodes**, but edges are not important for us. You visit the node (root node of spanning tree), and then repeat operation for the first adjacent node (child in terms of spanning tree). To avoid duplicate visiting we use either recursion or stack (as in lecture).

```
dfs(node)
    visit(node);                // this can mean: add to the tree, e.g.
    for (Node n: node.adjacent())
         if (!n.visited) dfs(n);
```

Traversal: BFS (Breadth first search)

BFS is an algorithm of (a) graph traversal (b) finding a path. For **unweighted** graph it always produces the **shortest way**. BFS is also a way to build a spanning tree on the graph. It looks like building a tree level-by-level: we put one vertex on the top level (level 0, root), visit it. All adjacent vertices become children, visit them. Then we repeat search from each node of the first level. To preserve visiting order we can use queue, as show on the lecture.

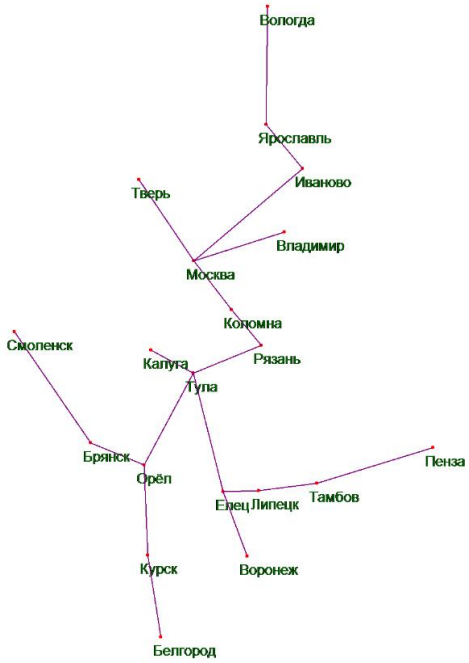
```
void BFS() {
  Queue<> q = new Queue<Node>();
  root.marked = true;
  q.push(root);
  while (!q.empty()) {
      int v = q.dequeue();
      visit(v);
      for (Node n: v.adjacent()) {
          if (!n.marked) {
            n.marked = true;
            q.push(n);
          }
      }
  }
}
```

Graph: minimal spanning tree

Minimum spanning tree for weighted tree

This task is very important for logistics – we can calculate minimum fuel consumption for logistics, minimum cable length to cover the area with network, etc. As we already know, spanning tree for unweighted tree is always a minimum tree. But that's unfortunately false for weighted graphs. How shall we build this tree? That's also very easy – let's iteratively select the best known candidate edge for our tree.

1) Start from any proposed root (you can always “hang” undirected tree on any vertex and call is “root”) and mark it as **visited**.
2) Put *adjacent to newly visited node* **edges** into a waiting list as “candidates”.
3) Pick up the shortest edge from the list and add it to the tree (we know both vertices, so we can find where to attach this edge).
4) Mark second node of the edge **as visited (“create an office/warehouse/router there”)**.
5) Clean up the graph: remove all candidate edges, that connect 2 already visited nodes.
6) Repeat 2-5 until candidate list is empty.



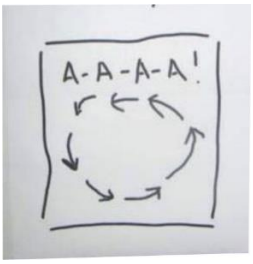
Practice

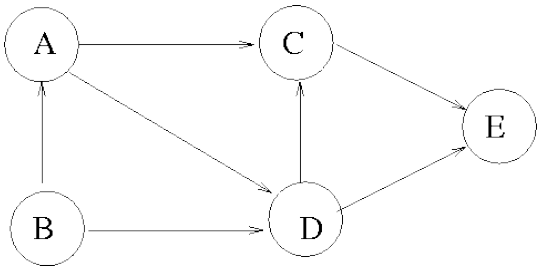
1) Read the graph from the proposed file (Appendix A).
2) Build spanning tree for this graph starting from «Москва» using DFS, BFS and minimum spanning tree algorithms. Calculate edge weight sum for all these cases. Save resulting graphs to a file. Visualize the graph and spanning trees using city coordinates.

Topological sort

Topological sort is important for scheduling work, usually of multiple agents, to achieve agreed and consistent timetable. **Result** of TS is a **list** where graph vertex order is preserved. The idea is that if you know that some action A is essential for another action B to execute (B depends on A directly or indirectly), than in a sorted list of actions action A will stand to the left from (“before”) B. It is very easy to achieve this if your graph satisfies the only constraint: it is **acyclic (has no cycles)**.

Left graph cannot be sorted.





If the graph **acyclic**, than if you start going from any point by **any path**, you will once reach the “end of the world” (nowhere to go). That means, there’s always **at least 1 vertex that has no successors**. Second idea about the acyclic graphs is if you remove one vertex from that graph, **it will remain acyclic**. These two ideas about acyclic graphs can converted to very easy algorithm for topological sort:

1) Find **one** lonely vertex (there can be few of them) at the end of the world. No one from the remaining graph depends on this vertex (so they all can seat to the left in the resulting sorted list). Attach this vertex to **the left of the list**.
2) Remove this vertex and all the adjacent edges from the graph.
3) Repeat 1-2 until graph is empty OR we find a cycle (no lonely edges).

Practice

3) You should explain Euler's formula to your small brother

$$e^{ix} = \cos x + i \sin x$$

He knows nothing about the math that mean you should cover all these themes:

- Natural numbers
- Integer numbers
- Rational numbers
- Real numbers
- Complex numbers
- Limits
- Exponentiation (bringing to a power)
- "e" constant
- Trigonometrical functions

Arrange these themes as a graph. Create a file with proposed graph. Create learning track for your brother using topological sort.

Read the graph from the file proposed at previous tutorial. Find the shortest path from «Вологда» to «Курск». Write distance and city sequence to a file.

Graph: shortest path algorithms

Shortest path: Dijkstra algorithm

Finds all shortest paths from starting vertex to all other in $O(v^2)$ time. Cannot work for negative weights. Algorithms at each iteration "visits" new vertex, calculates intermediate path length and refines path length for already visited. Stops when all vertices are visited. Starting with 0 for start node and $+\infty$ for other.

Algorithm step:

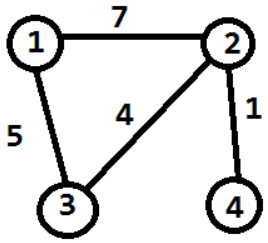
- Find **unvisited** vertex V_i with minimal calculated path length.
- Consider all simple paths, where V_i is the pre-last vertex – get all adjacent nodes to V_j (let's call them V_j) and calculate new path length for each as $PL(V_j) = PL(V_i) + |(V_i, V_j)|$. If this values is smaller that already assigned to a node – replace with smaller. If you want to restore the path – replace also "origin value" for V_i.

Floyd-Warshall algorithm

This is an algorithm to find shortest paths from all $\mathbf{V_i}$ to all $\mathbf{V_j}$. It will work for any weighted graph, but without negative cycles (that make whole problem unsolvable). This is a very simple algorithm.

Start from the adjacency matrix, filled with edge weights and ∞ if there is no edge.

DIST	**1**	**2**	**3**	**4**
1	0	7	5	∞
2	7	0	4	1
3	5	4	0	∞
4	∞	1	∞	0



Consider **each vertex V_k** as the possible transit vertex **for each path from V_i to V_j**. If going through Vk is a shortcut – update distance value.

```
for k = 1 ..|V| // transit nodes
    for i = 1 .. |V| // start nodes
        for j = 1 .. |V| // end nodes
            DIST[i][j] = min (DIST[i][k] + DIST[k][j], DIST[i][j]) // “KERNEL”
```

What is the complexity of this algorithm? ☺. This is obvious that it is not very useful to search for only weights. Let us change the kernel to find paths themselves. Create additional adjacency matrix, where values will be destination node indices, -1 otherwise.

NEXT	**1**	**2**	**3**	**4**
1	0	2	3	-1
2	1	0	3	4
3	1	2	0	-1
4	-1	2	-1	0

```
    ...
    if (DIST[i][k] + DIST[k][j] > DIST[i][j]) {
        DIST[i][j] = DIST[i][k] + DIST[k][j]
        NEXT[i][j] = NEXT[i][k]
    }
```

This approach will allow us to “unwind” the path with this procedure:

```
int curr = i;
while (NEXT[curr][j] != j) { PRINT(curr); curr = NEXT[curr][j]; }
```

or

```
while (NEXT[curr][j] > 0) { PRINT(curr); curr = NEXT[curr][j]; }
```

How can we speed up this $O(n^3)$ algorithm? Obviously, it is highly parallelizable while k is fixed, so we can run 2 internal loops independently in parallel. Other approach is to use SSE (streaming SIMD extention): e.g. for basic edition, we can use `PMAXUB mm1, mm2/m64` instead of kernel (this is a parallels “max” operation).

Or if we are just interested in connectivity components, we can use

```
W[i][j] = W[i][j] or (W[i][k] and W[k][j])
```

instead of kernel. Than we can store data and perform operations bitwise! In this case we can achieve up to $O(n^3/(SIMD\_count*core\_count))$

Practice

4) You should explain Euler’s formula to your small brother

   $$e^{ix} = \cos x + i \sin x$$

   He knows nothing about the math that mean you should cover all these themes:
   - Natural numbers
   - Integer numbers
   - Rational numbers
   - Real numbers
   - Complex numbers
   - Limits
   - Exponentiation (bringing to a power)
   - “e” constant
   - Trigonometrical functions

   Arrange these themes as a graph. Create a file with proposed graph. Create learning track for your brother using topological sort.
5) Read the graph from the file proposed at previous tutorial. Find the shortest path from «Вологда» to «Курск». Write distance and city sequence to a file.

Max flow problem

Flow network – <u>oriented</u> graph with 2 values: weights (**c**apacity) and **f**low.

- c(u,v) > 0
- f(u,v) <= c(u,v)
- f(u,v) = -f(v,u)
- sum(f(u,v))|$_v$=0 (for any *u != s, t*)

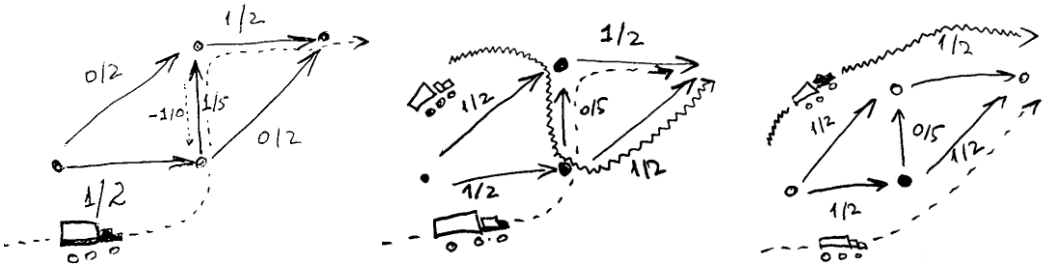
There are 2 special vertices**: source (**s**)** and **sink (**t**).** There's also a tem **cut**.

cut = incoming flow – outgoing flow

Residual network – if transport is already going, what is our «remaining» capacity

Flow and capacity:

- You have an edge (A->B), c = 5
  - **f(a, b) = 0**, f(b, a) = 0
  - **c(a, b) = 5**, c(b, a) = 0
- You send some traffic (**3**) from A to B
  - f'(a, b) = f(a, b) + 3 | *if f < c*
  - f'(b, a) = f(b, a) – 3 | *if f < c -- "castling" potential*
  - **c'(a, b) = c(a, b) – f(a, b)**
    - **c'(a, b) = 5 – 3 = 2**
    - **c'(b, a) = 0 – (-3) = 3**



Edmonds-Karp algorithm for max flow

- Initial **residual network** c(u,v) **== adjacency matrix** (no edge === 0)
- Initial **flow** f(u, v) == 0
- LOOP
  - *Find any valid path **from s to t** on c(u,v) (c(u,v)>0, even if it is a backward edge)*
  - **<u>if none – break;</u>**
  - Find a **bottleneck**
  - For each edge in path "send a N trucks":
    - f'(u,v) = f(u,v) – bottleneck
    - f'(v,u) = f(v,u) + bottleneck
    - c = c – f
- return **inCut(t)**

Appendix A

distances.txt

```
Белгород Брянск Владимир Вологда Воронеж Елец Иваново Калуга Коломна Курск
Липецк Москва Орёл Пенза Рязань Смоленск Тамбов Тверь Тула Ярославль

Тула Калуга 110 Тула Рязань 180 Тула Орёл 190 Тула Елец 210 Калуга Тула 110
Рязань Тула 180 Рязань Коломна 82 Коломна Рязань 82 Коломна Москва 110
Москва Коломна 110 Москва Иваново 110 Москва Тверь 170 Москва Владимир 190
Иваново Москва 110 Иваново Ярославль 120 Ярославль Иваново 120 Ярославль
Вологда 220 Тверь Москва 170 Орёл Тула 190 Орёл Брянск 128 Орёл Курск 160
Брянск Орёл 128 Брянск Смоленск 250 Курск Орёл 160 Курск Белгород 140
Белгород Курск 140 Владимир Москва 190 Елец Тула 210 Елец Липецк 82 Елец
Воронеж 130 Липецк Елец 82 Липецк Тамбов 130 Воронеж Елец 130 Тамбов Липецк
130 Тамбов Пенза 290 Вологда Ярославль 220 Смоленск Брянск 250 Пенза Тамбов
290
```

coordinates.txt

```
Москва     55° 45'    37° 37'
Воронеж    51° 43'    39° 16'
Ярославль  57° 37'    39° 51'
Липецк     52° 37'    39° 37'
Пенза 53° 12'    45° 0'
Рязань     54° 36'    39° 42'
Тула  54° 13'    37° 36'
Брянск     53° 16'    34° 25'
Иваново    57° 1'     40° 59'
Тверь  56° 52'   35° 55'
Курск  51° 44'   36° 11'
Смоленск    54° 47'   32° 3'
Белгород    50° 37'   36° 35'
Орёл   52° 58'   36° 4'
Калуга      54° 32'   36° 17'
Владимир    56° 9'    40° 25'
Тамбов      52° 43'   41° 25'
Вологда     59° 14'   39° 54'
Коломна     55° 5'    38° 47'
Елец   52° 36'   38° 31'
```