Poul Klausen

JAVA 19

More algorithms and data structures

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 19: MORE ALGORITHMS AND DATA STRUCTURES

SOFTWARE DEVELOPMENT

Java 19: More algorithms and data structures: Software Development
1st edition

ISBN 978-87-403-2619-2
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

The current book is the nineteenth in this series of books on software development in Java, and it is a continuation of the previous book. The book provides examples of algorithm paradigms including classic algorithms. In addition, several other data structures are described, including the implementation of a red-black binary tree. General trees are also treated in the form of a B-tree and a B+ tree and they are implemented as Java classes. A large part of the book is deals with graphs, including how a graph can be implemented as a class in Java. In addition, classic graph algorithms are described and implemented such as Dijkstra's algorithm and Prim's algorithm. The book ends with a chapter that introduces the traveling salesman problem, primarily as an example of an algorithm, where one does not know any solution that can be used in practice.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in Java. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples and exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code to the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with one or two larger sample programs, which focus primarily is on process and an explanation of how the program is written. On the other hand appears the code only to a limited extent – if at all – and the reader should instead study the finished program code perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

The programming language is, as mentioned Java, and besides the books use the following products:

- NetBeans as IDE for application development
- MySQL to the extent there is a need for a database server (from the book Java 6 onwards)
- GlassFish as a web server and application server (from the book Java 11 onwards)
- Android Studio, that is a tool to development of apps to mobil phones

It is products that are free of charge and free to install, and there is even talk about products, where the installation is progressing all by itself and without major efforts and challenges. In addition, there are on the web detailed installation instructions for all the three products. The products are available on Windows and Linux, and it therefore plays no special role if you use Linux or Windows.

All sample programs are developed and tested on machines running Linux. In fact, it plays no major role, as both Java and other products work in exactly the same way whether the platform is one or the other. Some places will be in the books where you could see that the platform is Linux, and this applies primarily commands that concerning the file system. Otherwise it has no meaning to the reader that the programs are developed on a Linux machine, and they can immediately also run under Windows unless a program refers to the file system where it may be necessary to change the name of a file.

Finally a little about what the books are not. It is not “how to write” or for that matter reference manuals in Java, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

This book is an immediate continuation of the previous book on algorithms and data structures and contains primarily topics that were not available in the previous book. The book consists of two parts, in which the first part deals with algorithm paradigms and expands what has been said about algorithms in the previous book, while the other part deals with several data structures, and especially graphs. The topics are as follows:

1. *Greedy algorithms*, and as examples of greedy algorithms, the *load problem* and *Huffman* encoding are used.
2. *Divide and conquer*, and as examples of such algorithms, the *closest-points problem* and *The Tower of Hanoi* are used.
3. *Dynamic programming*, and as examples of problem solving after this template, the *coin problem* and the *0/1 Knapsack* problem are used.
4. *Backtracking*, and as examples, *The Eight Queens Problem* is used and the determination of all permutations for a positive integer n.
5. The class *DisjointSet*, which represents an equivalence relation. The class is used to implement an algorithm that finds its way through a maze.
6. *Randomization algorithms*, which are algorithms, where the next step in the algorithm is selected by a random generator. As an example, an algorithm that creates a *Sudoku* is used.
7. Implementing a binary tree as a *red-black* tree. Binary trees have been treated in the previous book, but the algorithm has been postponed to this book for reasons of space.
8. General trees, including general search trees. As an example, a B tree and a B+ tree are implemented as Java classes.
9. As the last graphs are treated, including an example of how to implement a graph as a data structure. In connection with graphs typical graph algorithms are treated:
   - *Dijkstra's algorithm*
   - *Bellman-Ford*
   - *Topological sort*
   - *Prim's algorithm*
   - An overall introduction to *The Traveling Salesman Problem*.

Code to many of the examples fills a lot and is only shown to a lesser degree in the book. Refer to the completed examples, which can be downloaded from the book's website, as well as the code for the final example showing an activity planning program.

2 GREEDY ALGORITHMS

There is a family of algorithms known as *greedy algorithms*, which try to construct an optimal solution to a problem through a number of steps. In each step, based on given criteria, a decision is chosen that deemed to be the best choice towards a final solution to the problem.

A greedy algorithm is performed in phases or steps, which performs in each phase, what seems to be the best, but without interfering with future consequences. This means that you always find a local best solution and thus use a "take what you can get now" strategy and hence the name of the algorithms. When the algorithm stops, one hopes that the local best solution is close to the optimal solution. If so, the algorithm is correct, but otherwise it is a sub-optimal solution.

Immediately, it is not obvious where such algorithms can be used, but typically it's algorithms that are easy to implement and are also effective, and in situations where one does not necessarily need an optimal solution, a greedy algorithm can be quite reasonable.

A little abstract, you can formulate the solution of a greedy algorithm as follows:

- *C*: a collection of candidates for a solution
- *S*: the collection of selected candidates
- *solution()* which checks whether a collection of candidates is a solution
- *feasible()* which checks whether a collection of candidates can lead to a solution
- *select()* which returns the best candidate for a solution

One of the classic examples of a greedy algorithm is the so-called repayment problem (or coin problem), where you have to exchange an amount in as few coins (money units) as possible. One can think of the problem in relation to a vending machine where a person throws in more money than what the goods costs and the vending machine will then return the excess amount, but with as few coins as possible. To describe the solution of this problem by using a greedy algorithm, where *b* is the amount to be returned, it could be something like the following:

- *C* the holding of coins
- *S* the coins to be thrown back
- *solution()* the value of *S* is equal *b*
- *feasible()* if the machine contains the coin and it can be used
- *select()* the largest coin that can be used

For example, if you think of Danish coins: 1-crown, 2-crown, 5-crown, 10-crown and 20-crown and the amount *b* to be exchanged is always in whole crowns, so is the strategy:

```
for (c = 20, 10, 5, 2, 1)
{
 while (feasible(c))
 {
  select(c)
  if (solution()) break;
 }
}
if (solution()) return S else error
feasible(c) { return C contains c and c <= b }
select(c) { S.add(c); C.remove(c); }
solution() { S.value == b }
```

Note that assuming the machine has enough coins, the algorithm finds a solution (*b* is an entire number of crowns and can therefore always be exchanged in 1-crown coins). It is an algorithm that is greedy about the value of the coin. It first tries to take all the 20-crowns coins as it can, then all the 10-crown coins as it can and so on, and every time without taking into account if that is the best choice.

To implement the algorithm in Java (*CoinProblem* project) I have first written a model class for a coin:

```
class Coin implements Comparable<Coin>
{
 private int value;
```

and the only thing here is to note is that *Coin* objects are comparable so they are arranged after falling value of the coin. I have then written a class that assign a number of coins to a *Coin* object:

```
class Coins
{
 private Coin coin;
 private int count = 1;

 …

 public void add(int count)
 {
  this.count += count;
 }

 …
```

```
 public String toString()
 {
  return String.format("[%3d, %3d]", coin.getValue(), count);
 }
}
```

There is not much to explain, but the class must represent a *Coin* of a given value and is used to keep track of how many coins with that value the machine should give back. Next, there is the class *Greedy* that implements the above algorithm:

```
class Greedy implements Iterable<Coins>
{
 private Map<Coin, Integer> holdings = new TreeMap();

 public void add(Coin coin, int count)
 {
  if (holdings.containsKey(coin)) count += holdings.get(coin);
  holdings.put(coin, count);
 }
```

```
public List<Coins> exChange(int amount) throws Exception
{
 List<Coins> coins = new ArrayList();
 for (Coin c : holdings.keySet())
  while (feasible(c, amount))
  {
   select(coins, c);
   if (solution(coins, amount -= c.getValue())) return coins;
  }
 for (Coins c : coins) add(c.getCoin(), c.getCount());
 throw new Exception("The amount could not be exchanged");
}

public Iterator<Coins> iterator()
{
 …
}

private boolean solution(List<Coins> coins, int amount)
{
 return coins.size() > 0 && amount == 0;
}

private boolean feasible(Coin coin, int amount)
{
 return holdings.get(coin) > 0 && coin.getValue() <= amount;
}

private void select(List<Coins> coins, Coin coin)
{
 int count = holdings.get(coin) - 1;
 holdings.put(coin, count);
 for (Coins c : coins)
  if (c.getCoin().equals(coin))
  {
   c.add(1);
   return;
  }
 coins.add(new Coins(coin));
}
}
```

The class's holdings of coins is stored in a *Map<Coin, Integer>*. It is a *TreeMap* that ensures that the coins are sorted decreasing according to the value of the coins. The class has a method *add()* used to fill coins on the machine. The most important thing is the method *exChange()* which exchange an amount. The method starts to create a list for the solution, and it is empty from the start, which corresponds to the algorithm. The method *solution()*

says that there is a solution if the list is not empty and the amount is 0 and thus the entire amount is exchanged and the method is iterating as long as no solution has been found or until all coins are tried without finding a solution. The method *feasible()* tests where a coin can be used, which is the case if the value of the coin is less than the amount and the machine has coins of that kind. The method *select()* selects the coin by reducing the machine holding and adding the coin to the result. In particular, note the last loop in the method *exChange()*, used to cancel the selected coins if the machine has to give up to exchange the amount.

Finally, there is the program or test method that attempts to exchange an amount of 20:

```
public class CoinProblem
{
 private static final Random rand = new Random();
 private static Greedy greedy = new Greedy();

 public static void main(String[] args)
 {
  greedy.add(new Coin(1), 50);
  greedy.add(new Coin(2), 40);
  greedy.add(new Coin(5), 30);
  greedy.add(new Coin(10), 20);
  greedy.add(new Coin(20), 10);
  for (int i = 0; i < 20; ++i) exChange(rand.nextInt(90) + 10);
 }

 private static void exChange(int amount)
 {
  try
  {
   System.out.printf("%d should be exchanged\n", amount);
   print(greedy.exChange(amount));
  }
  catch (Exception ex)
  {
   System.out.println(ex);
  }
 }

 private static void print(List<Coins> list)
 {
  …
 }
}
```

In practice, to write a greedy algorithm you must argue both for correctness and for an optimal solution. Above I have argued for correctness. As to whether the algorithm is optimal, it starts to assume that the 20s are available, and then it can be assumed that the amount is less than 20. If the amount is exchanged, there may be no more than 1 10-crown, 1 5-crown, 2 2-crowns and 1 1-crown. The 19 options for exchanging this amount with the minimum number of coins are

```
10    5     2     2
10    5     2     1
10    5     2
10    5     1
10    5
10    2     2
10    2     1
10    2
10    1
10
5     2     2
5     2     1
5     2
5     1
5
2     2
2     1
2
1
```

that exactly are the results for the algorithm, thus concluding that the algorithm is optimal. However, the coins could have been chosen so that the algorithm does not provide an optimal solution. Suppose, for example, that the coins instead were 1-crown, 3-crown and 4-crown, and you would like to change the amount 6. The algorithm would then give 1 4-crown and 2 1-crown, but a better solution would be 2 3-crown. The conclusion is that you should be careful about greedy algorithms as they are intuitive and apparently the obvious algorithm, but the result is not necessarily the right one.

In the introduction to this chapter, I mentioned that greedy algorithms are typically effective, as is the case with the above algorithm. If you have the amount N, N / 20 operations are needed to find all 20-crown and, at worst, 20 operations of finding the last coins so the number of operations is therefore something in the direction of N / 20 + 20 and thus a complexity that is linear. One can, however, do it better. If you in the class *Greedy* change the method *feasible()* to

```
private int feasible(Coin coin, int amount)
{
 return Math.min(holdings.get(coin), amount / coin.getValue());
}
```

and *select()* to

```
private void select(List<Coins> coins, Coin coin, int n)
{
 int count = holdings.get(coin) - n;
 holdings.put(coin, count);
 for (Coins c : coins)
  if (c.getCoin().equals(coin))
  {
   c.add(n);
   return;
  }
 coins.add(new Coins(coin, n));
}
```

can the algorithm be written as follows:

```
public List<Coins> exChange(int amount) throws Exception
{
 List<Coins> coins = new ArrayList();
 for (Coin c : holdings.keySet())
 {
  int n = feasible(c, amount);
```

```
  if (n > 0)
  {
   select(coins, c, n);
   if (solution(coins, amount -= n * c.getValue())) return coins;
  }
 }
 for (Coins c : coins) add(c.getCoin(), c.getCount());
 throw new Exception("The amount could not be exchanged");
}
```

Basically, it is the same algorithm, but now it performs in principle independent of the amount and has only 5 operations and thus has constant time complexity.

2.1 THE LOAD PROBLEM

I want to show another example of a greedy algorithm, which is also a classic example:

*A cargo space – such as a ship – must be filled with containers, all of which are of the same size but have different weights. Suppose there are n containers and that i*th *the container has the weight* w_i*. The problem is to load the cargo space with as many containers as possible, but on the condition that the maximum weight can not exceed c.*

Let x_i be a binary variable (a variable that kan have the values 0 and 1). If the variable x_i is 1, it means that, that the *i*th container should be loaded and if it is 0, the *i*th container should not be loaded. The problem can then be expressed by determining the values of *n* binary variables that meet the requirement

$$\sum_{i=1}^{n} w_i x_i \leq c$$

and which optimizes the function

$$\sum_{i=1}^{n} x_i$$

Any solution $(x_1, x_2, ..., x_n)$ that meets the condition (the inequality above) is called a possible solution, and a possible solution that optimizes the optimization function is an optimal solution.

If this problem is to be solved using a greedy algorithm, the method is

1. Set all variables x_i to 0
2. Set the solution to empty
3. If all containers are selected, you are done
4. Select the container among those that are not already loaded with the least weight
5. If the weight of that container plus weight of the solution is greater than the capacity, you have finished
6. Add the selected container to the solution (set x_i to 1).
7. Go to step 3

This means that you every time load the container with the least weight if there is room for it. Firstly, you should consider whether the above provides a solution, but it is apparent from 3 and 5: You are done when there are no more containers or if the weight exceeds the capacity. Secondly, one should consider whether it is an optimal solution, but replacing one of the containers in the solution with another will increase the weight of the solution, thus not making room for more. In each step, the most promising candidate is chosen and added to the solution without regard to other circumstances. Therefore, the method is greedy.

The above problem is often called *the load problem*. Note that the complexity of the above algorithm is basically $O(n^2)$ where n is the number of containers. If you start by sorting the containers by weight, you must always choose the first container that is left and for example by choosing quick sort, the complexity can then be changed to $O(n\log(n))$.

EXERCISE 1: THE LOAD PROBLEM

Write a program that can solve the load problem, where there are 12 containers and the capacity is 600:

```
public class LoadProblem
{
 private static int[] weight =
  { 100, 60, 200, 120, 50, 30, 90, 80, 150, 20, 40, 80 };
 private static int cost = 600;
```

The result of the program could be as follows:

```
  100   60   200   120   50   30   90   80   150   20   40   80

    x    x               x    x    x    x         x    x    x
Total: 550
```

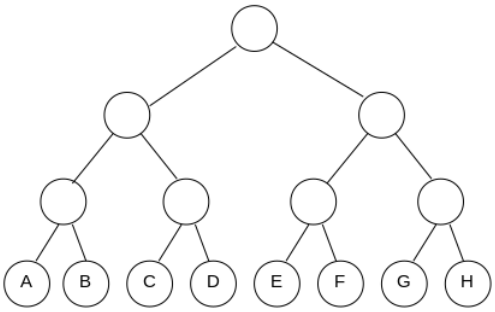
2.2 HUFFMAN CODES

As another example of a greedy algorithm, you can look at *Huffman encoding* of text that can be used to compress text files, and previously it was a widely used method, for example, if you were to transfer text over a communication line. If you look at text in the ASCII format, you have 256 different characters, each of which are encoded as 8 bits, and a file thus fills the number of bytes as there are characters in the file including various control characters. This means that very often used characters like e and t use 8 bits, which also applies to a characters that are not used so often. One can therefore consider an encoding where characters that are widely used are encoded with few bits while characters that are rarely used are encoded with more bits. That's exactly the idea of a Huffman encoding.

Suppose, for example, that you only need the characters A, B, C, D, E, F, G and H, and thus 8 characters. There is therefore a need for 8 character codes, and the relevant alphabet could then be encoded as:

A	000
B	001
C	010
D	011
E	100
F	101
G	110
H	111

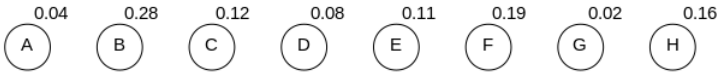
and all 8 characters would then be encoded as 3 bits. This encoding can be illustrated with the following binary tree, if you start the root and every time you go to the left, add a 0 and each time you go to the right add 1:



Suppose you also have knowledge of how often the characters in question occurs, for example

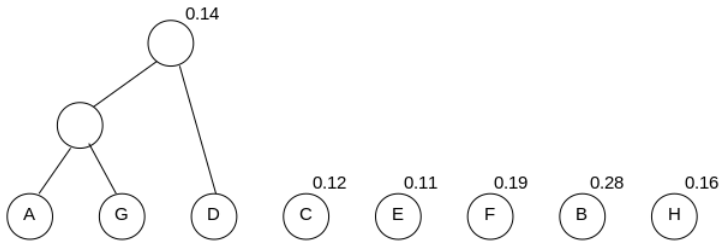
A	000	0.04
B	001	0.28
C	010	0.12
D	011	0.08
E	100	0.11
F	101	0.19
G	110	0.02
H	111	0.16

where the last column indicates the frequencies of the characters. Then you can construct a binary tree by starting with 8 sub-trees, where the numbers indicates the weight of the tree:

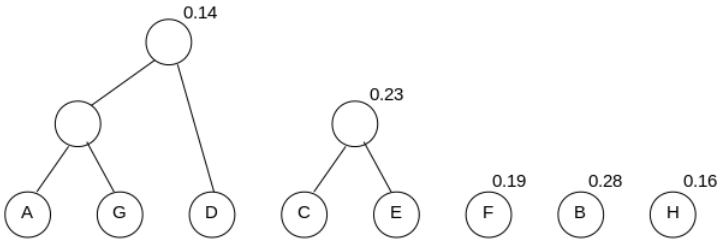


You now choose the two trees with least weight (that is A and G) and combine them into a new sub-tree that gets the weight 0.06:

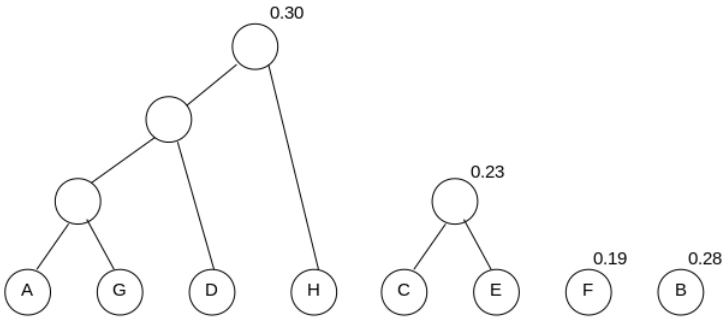
Now, repeat the process, and the two trees with the least weight are now the new sub-tree as well as D:



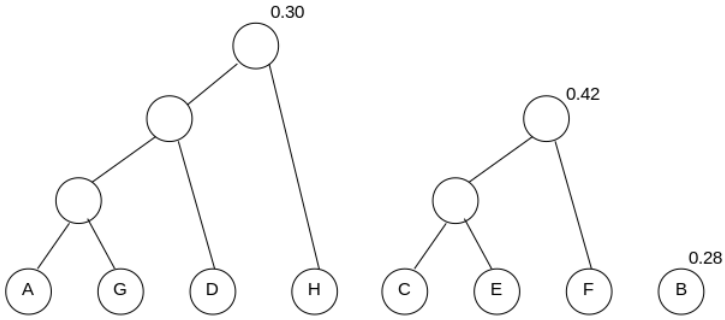
Now, the two sub-trees of minimum weight are C and E should be combined into a new sub-tree with a weight of 0.23:



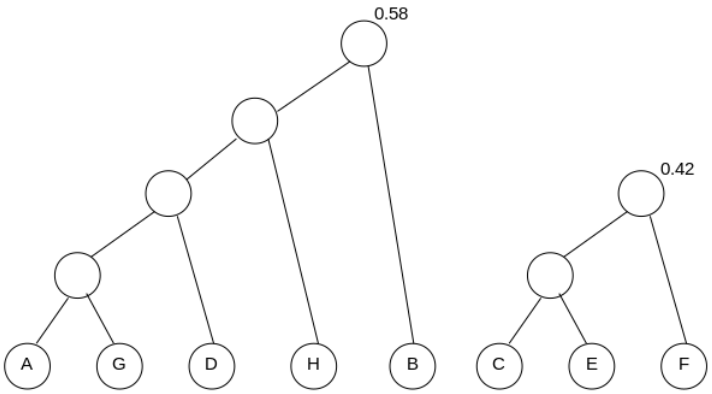
Next, H must be combined with the left of the above sub-trees, resulting in a sub-tree with weight 0.30



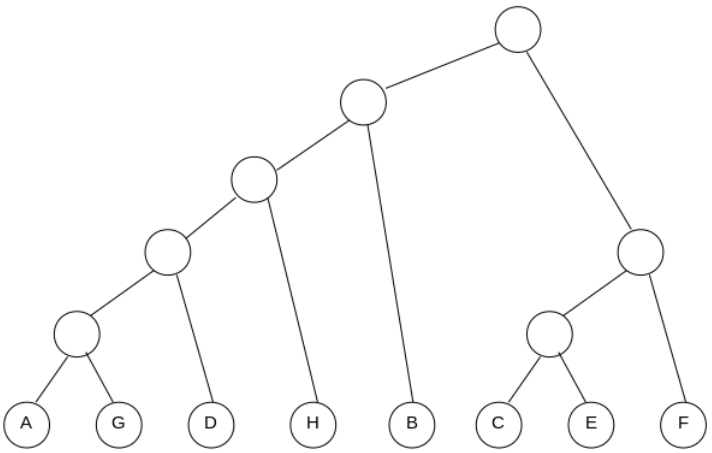
The next step is to combine the sub-tree with weight 0.23 and F:



Next to the last step, combine the two sub-trees with 0.30 and 0.28 weights:



Back there is only the combination of the last two sub-trees:



You can now, in the same way as above, determine an encoding by starting for each character in the root and adding a 0 each time you go left and 1 each time you go to the right:

A	000	0.04	00000
B	001	0.28	01
C	010	0.12	100
D	011	0.08	0001
E	100	0.11	101
F	101	0.19	11
G	110	0.02	00001
H	111	0.16	001

It is this encoding, which is called a Huffman coding. For example, if you have a 100-character file (of the possible 8 characters) and that they are distributed according to the frequencies in the above table, after the coding in column 2, the file will fill 300 bits, and after the encoding in column 4 the file will fill:

```
4*5 + 28*2 + 12*3 + 8*4 + 11*3 + 19*2 + 2*5 + 16*3 = 273
```

and thus a saving of 9%. It does not sound so much, but the savings could be greater if the difference in frequencies was greater than in the above example.

The above encoding is represented as a tree, sometimes called a code tree. It is characterized by the fact that it is only leaf nodes that represent characters, and it has the consequence that, from a bit pattern, it can be unambiguously determined which character the pattern represents (if it represents a character). The important thing is that no character's bit pattern can be prefix in another character. Suppose that you have the bit pattern 01101101001110010000101110001. You now starts in the root and follow the tree until you reach a leaf, and you definitely have a character. Then you start the root again and follow the tree until you have decided the next character, and continue until you have decided all the characters:

```
01|101|101|001|11|001|00001|01|11|0001|
  B   E   E   H  F  H     G  B  F   D
```

Considering how the above tree was constructed, the starting point was, that each leaf node (and thus to each character) was attached to a weight. Each leaf node can be appropriately interpreted as a tree with a single node and a weight. The algorithm was now to find the two trees with the smallest weights and let these two trees be combined in a new binary tree whose weight is the sum of the weights in the two sub-trees. This process is repeated until you only have one tree. Choosing the two trees with the smallest weights each time exactly corresponds to a greedy algorithm that is greedy to the least weight.

The program *Huffman* implements a Huffman compression of a file. The program is executed as a command where there are two options:

```
java -jar Huffman.jar -c file1 file2
```

that compresses *file1* and saves the result in *file2*.

```
java -jar Huffman.jar -d file1 file2
```

that decompresses *file1* and saves the result in *file2*.

The project consists of three classes in addition to the actual program. One is called *Bitmap* and represents a simple bitmap and hence the code for a character as a variable length bit pattern. I do not want to show this class here. First, it fills a part, and most of the class relates to regular bit-manipulation. The class HuffmanTree represents the code tree:

```
public class HuffmanTree implements Comparable<HuffmanTree>
{
 private Node root;
 private int weight;

 public HuffmanTree(byte value, int weight)
 {
  root = new Leaf(value);
  this.weight = weight;
 }

 public HuffmanTree(HuffmanTree t1, HuffmanTree t2)
 {
  root = new Node(t1.root, t2.root);
  weight = t1.weight + t2.weight;
 }
```

```
public void create(Map<Byte, Bitmap> byteMap, Map<Bitmap, Byte> codeMap)
{
 Stack<Integer> stack = new Stack();
 create(root, byteMap, codeMap, stack);
}

private void create(Node root, Map<Byte, Bitmap> byteMap,
 Map<Bitmap, Byte> codeMap, Stack<Integer> stack)
{
 if (root instanceof Leaf)
 {
  byte value = ((Leaf) root).value;
  Bitmap bmp = new Bitmap();
  for (int i = 0; i < stack.size(); ++i) bmp.set(i, stack.get(i));
  byteMap.put(value, bmp);
  codeMap.put(bmp, value);
 }
 else
 {
  stack.push(0);
  create(root.left, byteMap, codeMap, stack);
  stack.pop();
  stack.push(1);
  create(root.right, byteMap, codeMap, stack);
  stack.pop();
 }
}

@Override
public int compareTo(HuffmanTree t)
{
 return weight < t.weight ? -1 : weight > t.weight ? 1 : 0;
}

private class Node
{
 public Node left;
 public Node right;

 public Node(Node left, Node right)
 {
  this.left = left;
  this.right = right;
 }
}

private class Leaf extends Node
{
 public byte value;
```

```
  public Leaf(byte value)
  {
   super(null, null);
   this.value = value;
  }
 }
}
```

First, note that the class implements *Comparable<HuffmanTree>* that compares two trees according to their weight. It is used to construct the completed code tree with a greedy algorithm that is greedy by weight. The class has two variables, which in addition to the weight is the root of the tree, whose type is *Node*, which is an inner class that consists of nothing but two references to sub-trees. There is also another inner class *Leaf*, which is derived from *Node*, and which represents a leaf node and thus a character. *Leaf* expands *Node* with a variable to the character. The class has two constructors, where the first creates a tree consisting solely of a leaf node, while the other from two trees creates a new tree with the two trees as sub-trees.

However, the most important thing is the method *create()*, which creates the encoding from two maps. The first map assigns to each key (each character of the type *Byte*) in the form of a bitmap the character's binary code. The second map assigns for each key (each binary

code according to the encoding) the character that the particular *Bitmap* represents. These maps are initialized by a private method *create()* by traversing the code tree (actually by a pre-order traversing).

Then there is the class *CodeTable*, which builds the code tree and hence a *HuffmanTree*:

```
public class CodeTable
{
 private int[] code = new int[256];
 private Map<Byte, Bitmap> byteMap = new HashMap();
 private Map<Bitmap, Byte> codeMap = new HashMap();

 public CodeTable(int[] codes)
 {
  code = codes;
  createCodes();
 }

 public CodeTable(String filename)
 {
  try (BufferedInputStream stream =
   new BufferedInputStream(new FileInputStream(filename)))
  {
   byte[] buffer = new byte[4096];
   for (int count = stream.read(buffer); count > 0; count = stream.read(buffer))
   {
    for (int i = 0; i < count; ++i) ++code[buffer[i] & 0x000000ff];
   }
  }
  catch (Exception ex)
  {
   System.out.println(ex);
  }
  createCodes();
 }

 public int[] getCodes()
 {
  return code;
 }

 public Bitmap getBitmap(byte b)
 {
  return byteMap.get(b);
 }
```

```
 public Byte getByte(Bitmap bmp)
 {
  return codeMap.get(bmp);
 }

 private void createCodes()
 {
  PriorityQueue<HuffmanTree> queue = new PriorityQueue();
  for (int i = 0; i < code.length; ++i)
   queue.add(new HuffmanTree((byte)i, code[i]));
  while (queue.size() > 1)
  {
   HuffmanTree t1 = queue.poll();
   HuffmanTree t2 = queue.poll();
   queue.add(new HuffmanTree(t1, t2));
  }
  HuffmanTree tree = queue.poll();
  tree.create(byteMap, codeMap);
 }
}
```

The class starts by defining an array *code* with space for 256 values. The reason is that the program must be able to encode 256 different characters that correspond to the number of values that a byte can represent. The result is that the program can "compress" any file regardless of the content. It is elaborated below. In addition, the class creates the two maps mentioned above.

The class has two constructors, and I want to start with the last one that has a file-name as a parameter. It is the name of the file to be compressed, and the constructor starts to determine the frequency of each byte in the file. There are 256 options, and for each of the file's bytes, the corresponding position in the array *code* is counted by 1. After the frequencies are specified, the method *createCodes()* is called. It starts by placing each of the 256 bytes as a HuffmanTree (and thus a leaf node) in a priority queue, where the priority is the frequency. The next *while* loop combines trees. as all times, take the first two trees in the queue, combine them and paste the results into the priority queue. Finally, they are only one tree, and with this code tree, the two maps are initialized with codes.

After this constructor has been executed, for each byte in a file, you can return the corresponding *Bitmap*, which is its binary encoding, and subsequently save the result to a compressed file, but it must be possible to go the other way and decompress the content. Therefore, the frequencies (the array *code*) are stored along with the file's content, and this is where the other constructor is used to restore the code tree.

Finally, there is the actual program (the class *Huffman*), but I do not want to show the class here, as most of it has to do with reading and writing files, but basically the program has two methods *compress()* and *deCompress()*. In short, the methods works as follows.

compress() starts by creating a *CodeTable* object for the current file. Next, it reads the file and creates a large bitmap consisting of all the file's bytes converted to binary code corresponding to the current Huffman encoding. Next, the 256 codes, the bitmap length, and bitmap itself are saved in the output file.

deCompress() begins by reading the code table and creates from that a *CodeTable* object, which is the same Huffman encoding as the original file was converted with. After that, the rest of the file is read and used to create a large bitmap with the file's content. Using the encoding this bitmap is parsed to the files original bytes and is written to the output file.

As mentioned above, the program can compress any file, and the question is how good the program is. First, note that when compressing a file, the output file starts with 256 codes for the encoding frequencies and an integer for the bitmap length and thus a total of 1028 bytes. One can thus conclude that the program can not compress small files, and even the opposite, as small files actually fills more than the original file (a file with one *int* will fill 1032 bytes instead of 4). If the file is large, the 1028 bytes do not mean much and are not decisive if you get a win. What matters is the frequency of individual bytes. If all 256 bytes can occur and if they occur for the most part at high frequency, then the program will have no benefit. For example, if you test the program on an image (compresses and decompresses) you will see that the program works (the compressed file decompresses to the correct result), but the compressed file fills approximate the same (if not more) than the original file. If, on the other hand, the original file contains plain text, you will need to see a significant gain (maybe 30% or more), and the fewer characters that appear, the more the program will compress.

3 DIVIDE AND CONQUER

Another family of algorithms is *called divide and conquer*, and it is algorithms that are recursive, but not all recursive algorithms are divide and conquer algorithms. A divide and conquer algorithm consists of two parts:

1. *Divide*: The problem is divided into two or more disjunctive but simpler problems, which are resolved recursively if the basic case is not reached.
2. *Conquer* : The solution to the original problem is found by combining the solutions of the sub-problems.

Classic examples of divide and conquer problems are merge sort and quick sort, but also to find the *k*th smallest element in a collection and thus the median is solved with a divide and conquer algorithm.

Therefore, in order to use a divide and conquer algorithm, a problem must be divided into smaller sub-problems, each of which is solved recursively according to the same template. Here it is important to note that the division should result in sub-problems that are simpler than the original problem. It is also important to note that the division should result

in at least two sub-problems, and for example, a recursive binary search is not a divide and conquer algorithm. Finally, it is important to note that the sub-problems must be disjunctive. For example, I have previously looked at a recursive method that returns the *n*th Fibonacci number:

```
public static long fibo(int n)
{
 if (n < 2) return n;
 return fibo(n - 1) + fibo(n - 2);
}
```

and that the algorithm is highly ineffective for just slightly larger values of *n*. Although the algorithm divides the problem into two recursive sub-problems, it is not a divide and conquer algorithm as the sub-problems are not disjunctive.

3.1 CLOSEST-POINTS PROBLEM

As an example of a divide and conquer algorithm, I will look at the so-called *closest-point problem*. Given n points in a coordinate system

$$P = \{(x_{i1}, x_{i2}) \mid i = 0,1,\ldots,n-1\}$$

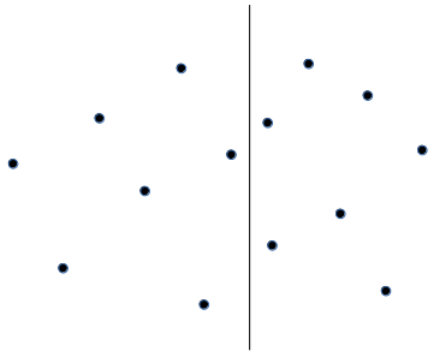
the problem consists in determining the shortest distance between two points where the distance between (x_1, x_2) og (y_1, y_2) is determined as

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

Note that it is allowed that two or more points are the same and the distance between two identical points is 0. The problem can be illustrated by the following figure, where the two points associated with a line are the two points closest to each other:

If there are n points, then there are $n(n - 1) / 2$ distances, and a simple solution would then be to determine all distances between pairs of points and then choose the least one. Such an algorithm is simple (and simple to implement), but it is clear that it has square time complexity. It is therefore obvious to seek a better algorithm.

Suppose you have an array containing the points and suppose the array is sorted by the points x-coordinates. You can then immediately split the points into two halves

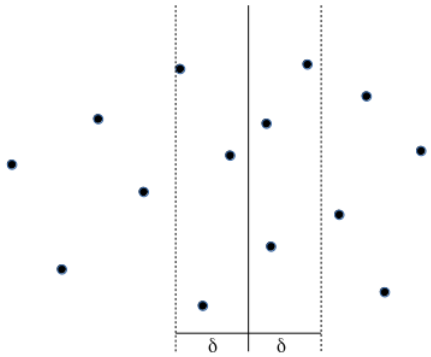


There are now three options:

1. The two points closest to each other are in the left half
2. The two points closest to each other are in the right half
3. The two points closest to each other are in each half and thus on each side of the line

One can then determine the solution by choosing the best (measured by the distance of the points) of the three solutions for the three sub-problems. Here the first two sub-problems are simple to solve as they are solved by repeating the above breakdown on each of the two sub-problems and thus by recursion. The problem is therefore to solve the third problem, and if you do it with linear time complexity, it is similar to what I have shown for merge sort, that the algorithm can be implemented with a time complexity that is $O\big(Nlog(N)\big)$. The solution thus consists in dividing the problem into two recursive sub-problems and a third, that is to be solved iterative. Thus, it is a divide and conquer algorithm.

To analyze the middle problem let δ_L and δ_R denote the solution (the minimum distance between points) of respectively the left and the right sub-problem and set $\delta = min\{\delta_L, \delta_R\}$. Let δ_C denote the solution of the third sub-problem. Now there is only reason to be interested of δ_C if it is les than δ and thus for points falling within a strip about the dividing line:

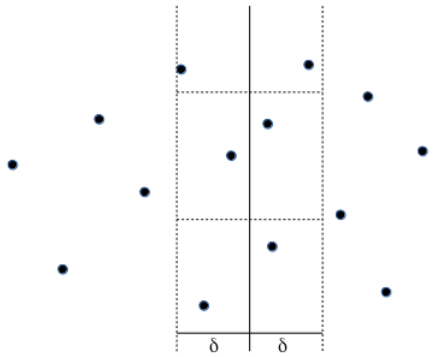


which reduces the number of points that needs to be processed. This can be done in the following way where *a* and *b* are start and end indexes for points within the strip:

```
for i = a to b
 for j = i + 1 to b
  if (dist(p[i], p[j]) < delta) delta = dist(p[i], p[j])
```

This algorithm has still square complexity, but it is enough to compare points *p[i]* with the points within the strip where the differences between the y values are smaller than δ (other points will not provide a better solution). Assuming, at the same time, that the points

within the current strip are sorted by the y-values and if you consider a particular point *p[i]*, then because of the choice of δ (which is less than or equal to the shortest distance between points in both the left and the right part) there may not be more than 7 points within a δ square:



and the algorithm can therefore be improved as shown below, where the inner loop now due to the maximum 7 points can be perceived as constant:

```
for i = a to b
 for j = i + 1 to b
  if (p[j].y > p[i].y) break
  else if (dist(p[i], p[j]) < delta) delta = dist(p[i], p[j])
```

As a result, the complexity of the third sub-problem is now linear, and the finished algorithm has the complexity $O(Nlog(N))$. However, there are two things that are not entirely obvious. First, the array of points can not be sorted by both x-values and y-values respectively. This problem can be solved by having two copies of the array each sorting after the two criterion. Another thing is that from the start it is assumed that the array of points is sorted (after the x-values), but since an array can be sorted by, for example, merge sort, which has $O(Nlog(N))$ complexity, you can start by sorting the array without changing the algorithm's time complexity as $O(Nlog(N))$.

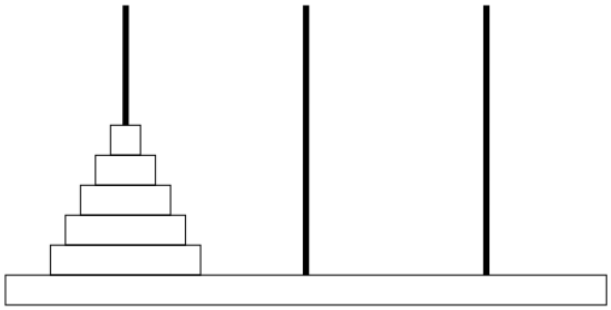
PROBLEM 1: IMPLEMENT THE CLOCEST-POINTS PROBLEM

In this problem you should try to implement the algorithm that solves the closest points problem as described above. Once you have implemented the algorithm, of course, you should also test it and try to write a test method that supports the algorithm's time complexity as $O(Nlog(N))$.

3.2 TOWER OF HANOI

Another classic problem is *The Tower of Hanoi* problem, and the example is included as a good example of using divide and conquer algorithms. The problem is usually described as follows:

At the end of the nineteenth century, a game called *The Tower of Hanoi* was introduced as a news in Europe. The popularity of the game was due to an anecdote, saying that priests in *The Temple of Brahma* worked to solve a problem and that the earth's demise would occur when the problem was solved. The game that the priests worked on consisted of a brass plate with three diamond "sticks", on which were placed 64 gold discs of different diameters. Initially, all discs are placed on the left stick with the largest at the bottom, then the second largest, etc., that is as a tower. The game now must move the tower over to the right stick, but only one disk at a time can be moved, and such that a disc never rests on a smaller diameter disc.



First of all, it is not entirely clear that the game can be solved, and secondly, as many priests have observed, you have to move many discs – and with 64 discs, really many.

In this section, I want to write a program that can simulate this game where the user can move the discs by dragging a disc with the mouse, as well as an option for the program to move the discs automatic and thus a demo feature. I do not want to show the code, which fills a part, but far most concerns on the user interface, and it's actually only the event handler for the demo feature, which is a divide and conquer problem.

If you look at the above figure (with *n* discs), you can describe the procedure as:

1. move the tower with the top *n-1* discs over to the middle stick
2. move the lower disc onto the right stick
3. move the tower (with *n-1* discs) on the middle stick on the right stick

This is obviously not a solution, but it means that the original problem is divided into two simpler problems and a trivial problem – it is easier to move a tower with *n-1* discs than a tower with *n* discs. The solution is then a step in the right direction, and as a tower always can be moved to a larger disk, it is clear that you can solve the problem by a recursive algorithm.

If you have a tower with 2 discs, the algorithm is:

1. move the top disc to pin 2
2. move the bottom disc onto pin 3
3. move the disc on the middle pin to pin 3

That is, the tower can be moved by 3 moves or $2^2 - 1$. Suppose you can move a tower with *n-1* discs using $2^{n-1} - 1$ moves. If you now have a tower with *n* discs, you can move the tower according to the above procedure by

$$2^{n-1} - 1 + 1 + 2^{n-1} - 1 = 2^n - 1$$

moves, and the above algorithm is then a $O(2^N)$ algorithm. Think on

$$2^{64} = 18446744073709551616$$

so the earth will not demise with the first because of the Tower of Hanoi problem. Written in Java, the algorithm can be implemented as follows:

```
private void moveTower(int a, int b, int c, int n)
{
 if (n > 0)
 {
  moveTower(a, c, b, n - 1);
  pause();
  moveDesk(a, b);
  moveTower(c, b, a, n - 1);
 }
}
```

4 DYNAMIC PROGRAMMING

As illustrated several times in the previous, many problems can be described using a recursive algorithm, perhaps because the problem is recursive by nature. Implementing recursive algorithms is easy, but as mentioned, it can lead to algorithms that are highly ineffective because the number of recursive calls explodes. The classic example of such an algorithm is a recursive method to determining the Fibonacci numbers, and here the solution instead was to write an iterative method. It is an example of dynamic programming. The idea is to state with a simple case that can be solved. You now save the solutions and proceed, but so that the next solution is determined by the solutions already decided. The iterative algorithm to determine the Fibonacci numbers starts by determining the first two (which is 0 and 1). Then you continue, but so that you always decide the next number as the sum of the two previous ones. That is, starting with the simple case and gradually moving forward. It is opposite recursion, where starting with the complex case and breaking up in minor problems until one reaches a simple case.

In conjunction with greedy algorithms, I saw on the coin problem, where you has to exchange an amount with as few coins as possible, and in that context, I also showed that the coins

may have such values that a greedy algorithm does not provide an optimal solution. However, the problem can be solved optimally with an algorithm that uses dynamic programming.

If you have to exchange an amount *amount*, the strategy is to determine the minimum number of coins that will be used to exchange all amounts

1, 2, 3, …, amount

You start by deciding how many coins to use to exchange 1, which is 1. Then you iterates all amounts from 1 through *amount*. For each value *a*, you iterates over all coins *c* (denoted as *c1*, …, *cn*), all the coins whose value is smaller or equal to *a*, and you determines

m = min { exhange(a – c1), …, exchange(a – cn) }

Then the minimum number of coins to be used to exchange *a* are equals *m + 1*. A little more formally, the method can be described as:

```
amount = the amount to be exchanged
values[1..amount] coins to exchange
loop a from 1 to amount
{
 min = a
 loop c for all coins
 {
  if (c.value() <= a
  {
   min = coins to exchange a – c.value() + 1
  }
 }
 values[a] = min
}
return values[amount]
```

You should note that it is precisely an algorithm based on dynamic programming by determining the result for all values less than or equal to the desired amount, and constantly applying solutions to minor problems that have already been solved.

To implement the algorithm, I have started with a copy of the program *CoinProblem*, which I have called *CoinProblem1*. The two classes *Coin* and *Coins* are unchanged, and the same goes for the test program, except that instead of the class *Greedy*, a class *Dynamic* is used:

```
class Dynamic implements Iterable<Coins>
{
 private Map<Coin, Integer> holdings = new TreeMap();
```

```
public void add(Coin coin, int count)
{
 if (holdings.containsKey(coin)) count += holdings.get(coin);
 holdings.put(coin, count);
}

public List<Coins> exChange(int amount) throws Exception
{
 List<Coins>[] arr = createArray(amount + 1);
 arr[0] = new ArrayList();
 for (int b = 1; b <= amount; ++b)
 {
  int min = b;
  Coin val = new Coin(1);
  int n = 0;
  for (Coin coin : holdings.keySet())
  {
   if (coin.getValue() > b) continue;
   int j = b - coin.getValue();
   if (arr[j].size() + 1 <= min)
   {
    min = arr[j].size() + 1;
    val = coin;
    n = j;
   }
  }
  arr[b] = new ArrayList();
  Coins coins = new Coins(val, 1);
  arr[b].add(coins);
  for (Coins c : arr[n])
   if (c.getCoin().equals(val)) coins.add(c.getCount()); else arr[b].add(c);
 }
 for (Coins c : arr[amount])
  if (holdings.get(c.getCoin()) < c.getCount())
   throw new Exception(amount + " could not be exchanged");
  for (Coins c : arr[amount])
   holdings.put(c.getCoin(), holdings.get(c.getCoin()) + c.getCount());
  return arr[amount];
}

public Iterator<Coins> iterator()
{
 return new CoinIterator();
}

class CoinIterator implements Iterator<Coins>
{
 private Iterator<Coin> keys = holdings.keySet().iterator();
```

```
  public boolean hasNext()
  {
   return keys.hasNext();
  }

  public Coins next()
  {
   Coin coin = keys.next();
   return new Coins(coin, holdings.get(coin));
  }
 }

 private <T> T[] createArray(int length, T… arr)
 {
  return Arrays.copyOf(arr, length);
 }
}
```

and here it is only the method *exChange()* that has been changed. The method creates an array of *List<Coins>* objects, where the object at position *n* must contain the *Coin* object to be used to exchange the amount *n*. At position 0 is added the empty list as it does not require coins to exchange the amount 0. As next step iterates over all amounts *b* from 1

through *amount*. Each iteration starts by seting *min* to b and *val* to the coin with the value 1. This corresponds to the use of b *val*-coins to exchange b. n is the index used to keep track of which coins are already is selected. For each amount there is an inner loop that iterates over all coins. If a coin has a value greater than b, it can of course not be used and you can proceed with the next coin. Otherwise, you test, if b minus the value of the coin can be exchanged with fewer coins than *min*, and if so, set *min* to that value and *val* to the current coin. After you have iterated over all coins, the array is updated with the result, you have found.

Finally, the program must test if there are sufficient coins and if not raise an exception. Otherwise, the holding of coins is counted down and the method returns the result.

If you consider the method, you can say that it consists of an outer loop that iterates N times (if the amount is N) and an inner loop that iterates K times if there are K coins. It is followed by two loops each with K iterations. The algorithm therefore performs in round numbers:

$$NK + 2K = (N + 2)K \sim NK$$

operations. Now K usually will be small in relation to N and of a fixed size (for Danish coins 5), and you can therefore perceive K as a constant. Doing so, the algorithm has linear complexity.

4.1 THE PRIMES

The usual prime algorithm, which tests where an integer, is a prime, is about checking if the number has a divisor. This can be done by trying to divide with all numbers less than the number to be tested for prime number. Since 2 is the only equal prime, you will therefore typically optimize the algorithm by dividing only with the odd numbers and if you notice that it is enough to divide with all numbers less than or equal to the square root of the number to be tested, the algorithm can be written as follows:

```
public static boolean isPrime(long n)
{
 if (n == 2 || n == 3 || n == 5 || n == 7) return true;
 if (n < 11 || n % 2 == 0) return false;
 for (long k = 3, m = (long)Math.sqrt(n) + 1; k <= m; k += 2)
  if (n % k == 0) return false;
 return true;
}
```

Note, however, that the number of iterations is $\sqrt{n}$ and therefore the time complexity of the algorithm is $O(\sqrt{N})$, which is a quite good time complexity, but for large values, many divisions still need to be performed and the algorithm becomes slow.

If you look at the algorithm, you can notice that it is enough to divide with all primes less than or equal to the square root of n. All else being equal, it will give fewer divisions and thus a more efficient algorithm. If *primes* is a list of all primes less than or equal to the square root of n, the prime algorithm can be written as follows:

```
public static boolean isPrime(List<Long> primes, long n)
{
 long m = (long)Math.sqrt(n) + 1;
 for (Long t : primes)
 {
  if (t > m) break;
  else if (n % t == 0) return false;
 }
 return true;
}
```

The problem is just that you do not know the prime numbers to divide with (you do not know *primes*). Assume that the task is to write a method that can return the *n*th prime. Using the first of the above methods, you can implement a solution as follows:

```
public static long prime1(int n)
{
 if (n == 0) return 2;
 long t = 1;
 for (int i = 0; i < n; )
 {
  for (t += 2; !isPrime(t); t += 2);
  ++i;
 }
 return t;
}
```

The method consists in determining the first *n* primes (where the prime number with index 0 is set to 2) and if you store these prime numbers in a list, you can use the other prime method and thus dynamic programming:

```
public static long prime2(int n)
{
 List<Long> list = new ArrayList(n / 10);
 if (n == 0) return 2;
 long t = 1;
```

```
 while (list.size() <= n)
 {
  for (t += 2; !isPrime(list, t); t += 2);
  list.add(t);
 }
 return t;
}
```

The question is then whether there is any gain, and it is actually very limited, but there is a factor 4 different with the advantage to the last method. The last method is probably based on fewer divisions, but conversely, one must maintain the list, which also takes time. Below is a test program:

```
public static void main(String[] args)
{
 StopWatch sw = new StopWatch();
 int N = 10000000;
 sw.start();
 long t1 = prime1(N);
 sw.stop();
 System.out.println(sw.getMilliSeconds());
 sw.start();
```

```
 long t2 = prime2(N);
 sw.stop();
 System.out.println(sw.getMilliSeconds());
 System.out.println(t1);
 System.out.println(t2);
}
```

and if you run the program you could get the result:

```
395033
118524
179424691
179424691
```

The project is called *ThePrimes* and you should note that the project uses the class *Primes* in the class library *palib* from the book Java 18. This project is also copied to the project files for this book as the current book will expand the library with new classes and also modify existing classes.

4.2 THE 0/1 KNAPSACK PROBLEM

The load problem discussed during greedy algorithms is a special case of a more general problem called the *0/1 Knapsack problem*. It's a very classic problem within the algorithm theory, and the following is a short introduction to the problem and an example of how to implement an algorithm to solve the problem using dynamic programming. The problem can be formulated as follows:

Maximaze the function

$$f(x_1, x_2,..., x_n) = \sum_{i=1}^{n} p_i x_i$$

where $x_1, x_2,...., x_n$ *are binary variables among the solutions to the inequality*

$$\sum_{i=1}^{n} w_i x_i \leq c$$

As a simple example of a 0/1 Knapsack problem where n = 3, c = 7 and

w = { 3, 4, 3 }
p = { 5, 2, 3 }

The problem is then to maximize the function

$$f(x, y, z) = 5x + 2y + 3z$$

when x, y and z is binary variables and

$$3x + 4y + 3z \leq 7$$

In this case, the problem is easy to solve because each variable can only have 2 values (0 and 1). There are only 8 possible solutions and you can list all combinations in a table:

x	y	z	3x + 4y + 3z	5x + 2y + 3z
0	0	0	0	0
0	0	1	3	3
0	1	0	4	2
0	1	1	7	5
1	0	0	3	5
1	0	1	6	8
1	1	0	7	7
1	1	1	10	10

Here you can see that the lower option can not be used because it does not match the inequality, and for the remaining 7 options the last column shows that the optimal solution is (1, 0, 1).

In this case, the problem is solved with a *brute force algorithm*, which simply means that you systematically test all options and in that way determines the best result. It is of course an algorithm that you can always use, but the problem is time complexity. If you have a 0/1 Knapsack problem with n variables, then the result will be an algorithm where time complexity is $O(2^n)$ which is hopeless. There is therefore reason to be interested in a better algorithm.

There are several strategies to solve a problem and you can, for example, apply a greedy algorithm that is greedy for p-values (profit) and constantly select the element with the largest p-value. You can be greedy for w-values (weight) and always choose the element with the smallest w-value or you may be greedy after the quotient p / w. In the load problem as formulated in Chapter 2, all p values were equal to 1, and the algorithm was greedy

for weight, but as illustrated by the coin problem, a greedy algorithm does not necessarily provide the optimal solution, and instead you can consider dynamic programming.

Dynamic programming can be used to solve a problem when the solution can be described as the result of a sequence of decisions. Think of determining the *n*th Fibonacci number, which occurs by determining the first, the next, the third, and so on. This strategy can be used to solve the 0/1 Knapsack problem.

Suppose that $(y_1, y_2, ..., y_n)$ is an optimal solution for a 0/1 Knapsack problem. Removing the first variable from the problem causes another 0/1 Knapsack problem:

Maximize

$$\sum_{i=2}^{n} p_i x_i$$

among the solutions to

$$\sum_{i=2}^{n} w_i x_i \leq d$$

where $d = c - w_1 y_1$

Then it is clear (by an ordinary control) that $(y_2,..., y_n)$ is a solution to the reduced problem. Suppose it is not an optimal solution. Then there is another solution $(z_2,..., z_n)$ (a better solution) such that

$$\sum_{i=2}^{n} p_i y_i < \sum_{i=2}^{n} p_i z_i$$

When

$$\sum_{i=2}^{n} w_i z_i \leq d = c - w_1 y_1 \Rightarrow w_1 y_1 + \sum_{i=2}^{n} w_i z_i \leq c$$

follows that $(y_1, z_2,..., z_n)$ is a solution to the original problem and thus (because $(y_1, y_2,..., y_n)$ is an optimal solution) that

$$\sum_{i=1}^{n} p_i y_i \geq p_1 y_1 + \sum_{i=2}^{n} p_i z_i \Rightarrow \sum_{i=2}^{n} p_i y_i \geq \sum_{i=2}^{n} p_i z_i$$

which is a contradiction. That is, a 0/1 Knapsack problem satisfies if you remove a variable, you can remove the same variable in an optimal solution, and you get an optimal solution for the reduced Knapsack problem (since there is no magic by removing the first variable instead of another).

Given a 0/1 Knapsack problem, you can consider the following sub-problem $P_k(d)$:

Maximize

$$\sum_{i=k+1}^{n} p_i x_i$$

among the solutions to

$$\sum_{i=k+1}^{n} w_i x_i \leq d$$

and hence the problem where the first variable has been removed. Here is $P_0(c)$ the original problem. Let $f_k(d)$ denote the value of an optimal solution to the sub-problem, where $f_0(c)$ is the value of an optimal solution to the original problem. Suppose all weights and c are positive numbers. Then follows immediately that

- $f_n(d) = 0$ for all $d > 0$ since the optimization function in this case is constant equal to 0 (the sum is empty)
- $f_k(0) = 0$ for all k since the problem's constraints can only be met if all binary variables are 0
- $f_k(d) = -\infty$ for all $d < 0$ and all k which simply means that the solution is empty

Let $(y_k,...,y_n)$ be an optimal solution to $P_{k-1}(d)$ with the value $f_{k-1}(d)$. That is

$$f_{k-1}(d) = \sum_{i=k}^{n} p_i y_i$$

According to the above are $(y_{k+1},....,y_n)$ an optimal solution to $P_k(d - w_k y_k)$, and then

$$f_k(d - w_k y_k) = \sum_{i=k+1}^{n} p_i y_i$$

That is

$$f_{k-1}(d) = \sum_{i=k}^{n} p_i y_i = \sum_{i=k+1}^{n} p_i y_i + p_k y_k = f_k(d - w_k y_k) + p_k y_k$$
$$= \max\{f_k(d), f_k(d - w_k) + p_k\}$$

where the last identity follows from y_k being either 0 or 1. That is, that one can decide f_{k-1} from f_k. This is where the dynamic comes in, because f_n is trivial you can successively decide f_k and as the last f_0. The formula is that

1. $f_n(d) = 0$ for all $d > 0$
2. $f_k(0) = 0$ for all k
3. $f_k(d) = -\infty$ for all $d < 0$
4. $f_{k-1}(d) = max\{f_k(d), f_k(d - w_k) + p_k\}\ k = n, n-1, \ldots, 0$

Consider as example a 0/1 Knapsack problem where n = 5, c = 10 and

w = { 3, 4, 2, 1, 5 }
p = { 5, 10, 2, 4, 8 }

Note that the problem is:

Maximize:

$$5x_1 + 10x_2 + 2x_3 + 4x_4 + 8x_5$$

when

$$3x_1 + 4x_2 + 2x_3 + x_4 + 5x_5 \leq 10$$

It is easy to manually calculates the example (where you have the possible values for d and where the table is filled using the formula above):

p	w		1	2	3	4	5	6	7	8	9	10
5	3	f_5	0	0	0	0	0	0	0	0	0	0
10	4	f_4	0	0	5	5	5	5	5	5	5	5
2	2	f_3	0	0	5	10	10	15	15	15	15	15
4	1	f_2	0	2	5	10	10	12	15	15	17	17
8	5	f_1	4	4	6	10	14	14	16	19	19	21
		f_0	4	4	6	10	14	14	16	19	19	22

which results in the maximum to be 22. Then, to determine the sequence that gives the maximum solution, start with the maximum capacity 10 and compare $f_0(10)$ and $f_1(10)$. Since they are different, x_5 has contributed to the capacity, so $x_5 = 1$, and the capacity is

reduced with $10 - w_5 = 5$. Then $f_1(7)$ compared to $f_2(7)$, and as they are different $x_4 = 1$, and then reducing capacity with $5 - w_4 = 4$. Next time are $f_2(4)$ and $f_3(4)$ compared, and they are both 10 and then x_3 has not contributed to the capacity so that $x_3 = 0$. The next comparison is $f_3(4)$ with $f_4(4)$ there are different and $x_2 = 1$ and the capacity is reduced to $4 - w_2 = 0$. Finally, is $f_4(0)$ compare to $f_5(0)$ which is 0 and thus is $x_1 = 0$. The overall result is as follows:

Maximum is 22 for the following sequence:

$$x_1 = 0$$
$$x_2 = 1$$
$$x_3 = 0$$
$$x_4 = 1$$
$$x_5 = 1$$

Note that $10x_2 + 4x_4 + 8x_5 = 22$.

Also note that if the problem has *N* variables and the capacity is *C* then the algorithm can be implemented with a time complexity that is $O(CN)$.

EXERCISE 2: THE KNAPSACKPROGRAM

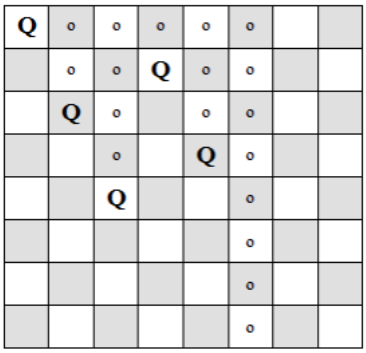
Write a program that implements the 0/1 Knapsack problem corresponding to the above description. The hardest thing is actually to make sure that you index correctly in the arrays used.

5 BACKTRACKING

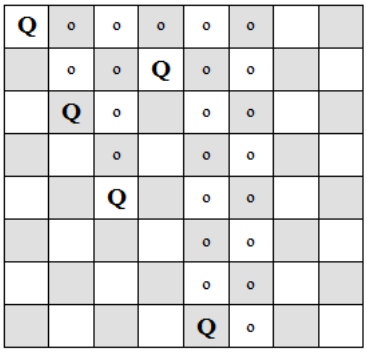
Often the algorithm to solve a problem is obvious, but in other contexts it may be difficult to find an algorithm, and in particular it may be difficult to choose the correct algorithm. In this chapter, I will look at a family of algorithms, known as *backtracking*, and it is best characterized as a systematic way to search for a solution to a problem by reviewing all possible solutions. As an example, I will look at the so-called *Eight Queens Problem*:

A chessboard consists of 64 fields organized in a square with 8 rows and 8 columns. In a chess game, the queen can be moved horizontally in a row, vertically in a column and diagonally, thus striking another piece on one of the places to which the queen can be moved. The problem now is to place 8 queens on a chessboard so none of them can strike each other.

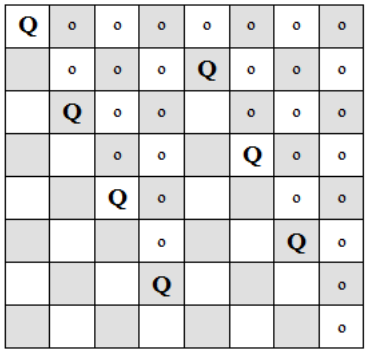
One solution to the problem could be to let a computer systematically review all of the options until a solution is found. A chessboard has 64 fields and placing 8 queens on a chessboard consists in choosing 8 out of 64 fields and the number of ways to do it is the binomial coefficient , which is the number 4426165368 so that even on a fast computer it is not a particularly passable road. However, the problem can be reduced by observing that each row can contain no more than one queen, and the same applies to each column. It significantly reduce the problem. For each of these options, check that none of the queens can hit each other diagonally. However, the solution method can be further improved by proceeding more systematically. You start by placing a queen in the first column of the top row. Next, place a queen in the second column by starting at the top of the column and continuing until you find a place where it can not strike the queen in the first column. Now continue with the third column where you start again at the top and go down until you find a place where it can not strike one of the first two queens. You then continue as long as possible. When placing a queen in the sixth column, it goes wrong. Whichever row you try, it will be able to strike one of the five queens that are located.



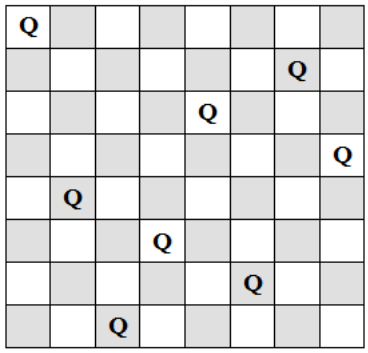
You now skips the last queen that is located (the queen in the fifth column). You are backtracking and then try one of the places that have not been tried. It's going well at the last place. You then tries again with the sixth column, but it is still not good.



You must therefore skip the last queen, but as there are no more places in this column (the fifth column), you must backtrack once again and skip the queen in the fourth column and try with a new location. One tries again, and this time it is possible to place queens in the first seven columns, but in the eighth it goes wrong and you have to backtrack.



You must continue until you find a solution. Here it is important that the problem has a solution as otherwise you will continue indefinitely. The idea is to systematically try to find a solution by going on until you have either found the solution or until you can find that you have ended up in a dead end. If that's the case, you'll get back and try again. Note that the problem has many solutions and that the method simply provides a single solution.



If you are to write a program that can solve this problem, you must be able to represent the game, and you must keep track of which locations have been tried. Regarding the first, the game can simply be represented as a 2-dimensional array of *boolean* elements, which indicates whether a place contains a queen or not. To keep track of which locations have been tried, you can use on two counters for rows and columns. The following class shows a solution of the eight queen problem:

```
package eightqueens;

public class Queens
{
 private boolean[][] board = null;

 public Queens(int size)
 {
  board = new boolean[size][size];
```

```
 }

 public int getSize()
 {
  return board.length;
 }

 public boolean isQueen(int row, int col)
 {
  return board[row][col];
 }

 public void setQueen(int row, int col, boolean value)
 {
  board[row][col] = value;
 }

 public boolean nextQueen(int col)
 {
  if (col == getSize()) return true;
  for (int row = 0; row < getSize(); ++row)
   if (ok(row, col))
   {
    board[row][col] = true;
    if (!nextQueen(col + 1)) board[row][col] = false; else return true;
   }
  return false;
 }

 public boolean ok(int row, int col)
 {
  for (int j = 0; j < col; ++j) if (board[row][j]) return false;
  for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; --i, --j)
   if (board[i][j]) return false;
  for (int i = row + 1, j = col - 1; i < getSize() && j >= 0; ++i, --j)
   if (board[i][j]) return false;
  return true;
 }
}
```

Note that the class' constructor has a parameter that indicates the size of a chess game. In relation to a real chess game (which always has a size of 8 × 8), of course, it does not matter much, but the eight queens problem is actually general and applies to sizes other than 8, so this parameter for the constructor. The method *ok()* tests whether you can put a queen in the place *(row, col)*, while it is the method *nextQueen()* that uses backtracking. If *col* is equal to the square size, the problem is solved and the method returns *true*. If not

you run over all rows in the column *col*, and for each row you try to set a queen. If you can do that, put a queen

```
board[row][col] = true;
```

and continue (recursively) with the next column, and if it is not possible, you must backtrack:

```
board[row][col] = false;
```

Also note that if you start setting the first queen differently (not in row 1), you will find another solution, and the problem actually has many solutions.

The following program used the class *Queens* to solve the eight queens problem, partly with 8 queens and partly with 15 queens:

```
package eightqueens;

public class EightQueens
{
 public static void main(String[] args)
 {
```

```
  test1();
  test2();
 }

 private static void test1()
 {
  Queens q = new Queens(8);
  if (q.nextQueen(0)) show(q);
  else System.out.println("The problem could be solved…");
 }

 private static void test2()
 {
  Queens q = new Queens(15);
  if (q.nextQueen(0)) show(q);
  else System.out.println("The problem could be solved…");
 }

 private static void show(Queens q)
 {
  for (int i = 0; i < q.getSize(); ++i)
  {
   for (int j = 0; j < q.getSize(); ++j)
    if (q.isQueen(i, j)) System.out.print("Q "); else System.out.print("x ");
   System.out.println();
  }
 }
}
```

Backtracking is a solution technique that can be used in many contexts where you try to find a solution to a problem with a partial solution and where you try to get back and forth when you realize that you are building a solution which can not lead to a result. Backtracking is significantly better than just trying with all possibilities (a brute force algorithm), but it should be noted that backtracking typically does not have a good time complexity.

EXERCISE 3: EIGHTQUEENS, ALL SOLUTIONS

Create a copy of the program *EightQueens*. You must add another test method that solves the eight queens problem (with 8 queens) when the method must determine all solutions to the problem (there are 92).

5.1 PERMUTATIONS

As another example of a problem that can be solved with a backtracking algorithm, I will look at a problem that consists in determining all permutations of *n*, where *n* is a positive integer. By a permutation of a positive number *n*, one understand the *n* positive integers that are less than or equal to *n* in one order or another. For example there are 6 permutations of 3:

```
1  2  3
1  3  2
2  1  3
2  3  1
3  1  2
3  2  1
```

Note that in general there is *n!* permutations of *n*. A strategy for determining all permutations of *n* could be:

1. Repeat for all *k* from 1 to *n* and an empty permutation
2. Add *k* to the permutation
3. If it results in a legal partial solution repeat step 2–4 recursively for all j from 1 to n
4. Else bactrack

that is to say, you try successively with all numbers until you find one that can be used. Of course, you should be careful that during the placement of a number you do not try the same number several times. It can be formulated in the following algorithm:

```
package permutations;

import java.util.*;

public class Permutation
{
 private int[] arr = null;

 private Permutation(int n)
 {
  arr = new int[n];
  for (int i = 1; i <= n; ++i) arr[i - 1] = i;
 }

 private Permutation(Permutation p)
 {
  arr = new int[p.getLength()];
  for (int i = 0; i < p.getLength(); ++i) arr[i] = p.arr[i];
 }
```

```
public int getLength()
{
 return arr.length;
}

public int getValue(int i)
{
 return arr[i];
}

@Override
public String toString()
{
 StringBuilder text = new StringBuilder(getLength());
 for (int i = 0; i < arr.length; ++i)
 {
  text.append(arr[i]);
  if (i < arr.length - 1) text.append(' ');
 }
 return text.toString();
}
```

```
 public static List<Permutation> permutations(int n)
 {
  List<Permutation> list = new ArrayList();
  permute(list, new Permutation(n), 0);
  return list;
 }

 private static void permute(List<Permutation> list, Permutation p, int i)
 {
  if (i == p.getLength() - 1) list.add(new Permutation(p));
  else for (int j = i; j < p.getLength(); ++j)
  {
   swap(p.arr, i, j);
   permute(list, p, i + 1);
   swap(p.arr, i, j);
  }
 }

 private static void swap(int[] arr, int i, int j)
 {
  if (i != j)
  {
   int t = arr[i];
   arr[i] = arr[j];
   arr[j] = t;
  }
 }
}
```

The class *Permutation* represents a permutation as an array. The class has two private constructors, and therefore can only be instantiated by the class's own methods, which occurs in the methods *permutations()* and *permute()*. In addition to the constructors, the class has properties for the permutation size and its values. The most important method is *permutations()*, which returns a list of all permutations of the number n. The method creates the list, but the permutations are created and added to the list in the recursive method *permute()*, which is an algorithm that works by backtracking. The method has 3 parameters, where the first is the list of permutations, the next is the permutation that is worked on, while the last one is how far the method is reached.

The algorithm is not so easy to understand, but the method is as follows. The method *permute()* is called with the trivial permutation of the form $(1,2,3,\ldots,n)$, and the method must then create permutations by systematically switching the numbers. From start is i equal to 0 that is equivalent to that none of the numbers are selected for the permutation. Each time the method is called, a loop is passed from i to the end (the numbers selected are not changed) and you select a new number for the permutation among the numbers that are

not used. Then the method calls itself recursively, but with i 1 larger, since a number has now been chosen. Recursion stops when all numbers are selected and the permutation is added to the list (a copy is created). When the permutation is added to the list, the current *permute()* method terminates, which means that the last swap in p is canceled, which is equivalent to backtracking.

5.2 A DISJOINTSET

I will mention a simple data structure, usually referred to as a *DisjointSet*. It does not have anything to do with backtracking and when the data structure is presented in this place, it's because I need it in the next example.

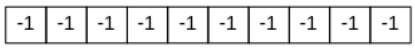
In mathematics, an equivalence relation plays an important role as a relation that divides a set into disjoint subsets. If the relation is called ~ an equivalence relation on a set S is a relation that satisfies:

$$\forall a \in S: a \sim a$$
$$\forall a, b \in S: a \sim b \Rightarrow b \sim a$$
$$\forall a, b, c \in S: a \sim b \wedge b \sim c \Rightarrow a \sim c$$

All elements, that with this relation are related to the element a, are called the equivalence class determined by a and are sometimes referred to as $[a]$ and from the above definition follows that two elements in the same equivalence class determine the same equivalence class. That means

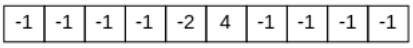
$$[a] = [b] \Leftrightarrow a \sim b$$

A set with an equivalence relation is sometimes also called for a class division of the set. I will describe a class that can be used to create a class division of the numbers 0, 1, …, n-1. The starting point is an array of n elements, where all from the start have the value -1, and with 10 elements it can be illustrated as follows:

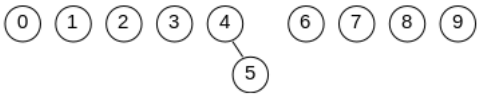


It must be interpreted in such a way that all numbers from 0 to 9 each constitute their own equivalence class. One can also think of the relationship as 10 trees, where each tree alone consists of a root:

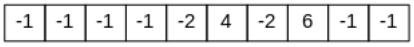
The class must have a method *union()*, to combine two equivalence classes, and if you want to combine the equivalency class for 4 and 5, you can do this as follows:



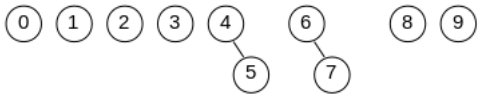
The value of 4 is -2 and must be interpreted as the number is negative as 4 is root in (determines) an equivalency class with the depth 2. That the value in position 5 is 4 it means that 5 is in the same equivalency class as the element in position 4, so that 4 and 5 are now are in the same equivalency class. To illustrate it with a tree is the result



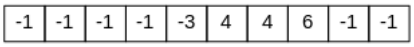
If you then combine the equivalency classes for 6 and 7, you get



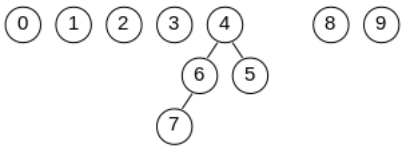
and written as a tree:



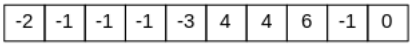
That is, the 10 elements are now divided into 8 equivalence classes. As a next step, I want to execute a union of the two equivalence classes containing 5 and 6. Here's 5 not root in an equivalency class, and the method union works by combining two equivalence classes using the class's root. It is therefore necessary with a method *find()* that can find the root of the equivalency class that contains a particular element. For 5, the method *find()* finds the element 4 while it for 6 finds the element itself. You have to combine the two equivalence classes, where the root is 4 and 6:

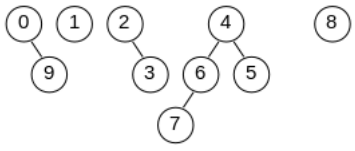


Here you should note that the element 7 refers to 6, which refers to 4, which is the root of the new equivalence class, while the element 5 also refers to 4. Thus, the methods must be implemented so that the path from one element to its root becomes so short as possible, and it just corresponds to the perception of the elements organized in a tree:

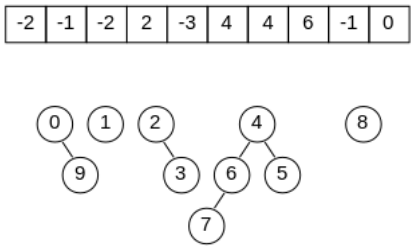


As the next operation, I want to combine the sub-tree for 0 and 9, which is simple as both elements are roots in a sub-tree:

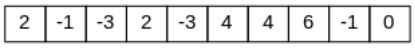




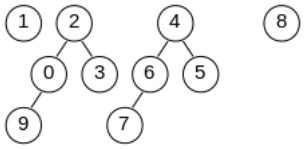
and if you combine 2 and 3:



If you then combine the equivalence classes determined by 0 and 2, you get



0 and 2 are each root in an equivalency class, but the method *union()* must ensure that the path from an element in an equivalency class to the root is as short as possible:



Then the 10 elements are divided into 4 equivalency classes. If you try with a *union()* of 0 and 3, nothing should happens, because the two elements are already in the same equivalency class.

The task is to write a class *DisjointSet* corresponding to the above, and it is actually very simple, but since the class has some practical interest, I will add it to my class library *palib*. The class must, in addition to a constructor who creates the set, only have two methods *find()* and *union()*:

```
public class DisjointSet
{
 private int[] S;

 public DisjointSet(int n)
 {
  S = new int[n];
  for (int i = 0; i < S.length; ++i) S[i] = -1;
 }
```

```
 public void union(int a, int b)
 {
  if (S[a] >= 0) return;
  if (S[b] >= 0) return;
  if (a == b) return;
  if (S[b] < S[a]) S[a] = b;
  else
  {
   if (S[a] == S[b]) --S[a];
   S[b] = a;
  }
 }

 public int find(int t)
 {
  if (S[t] < 0) return t;
  return S[t] = find(S[t]);
 }
}
```

The method *find()* returns the root of an element's equivalency class (the value with a negative value). The method is written recursively, and it modifies the path of elements towards the root in order to make the road as short as possible. The method *union()* is called with two

elements, and it is assumed that these elements are each root in an equivalency class. If this is not the case, or are both elements the same, the method is interrupted. If the value of the last element (which is negative) is less than the value of *a*, then the element *a* should refer to *b* (the equivalency class where *b* is root) and otherwise the value of *b* must refer to *a*, but possibly the value of *a* is counted down by 1.

Below is a test program that corresponds to the above explanation:

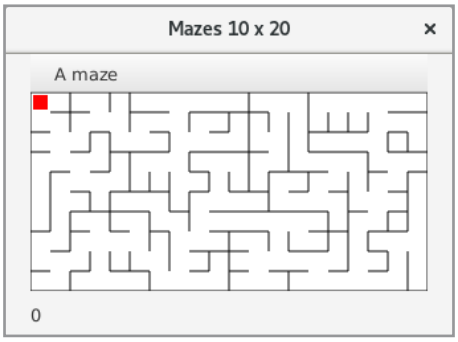
```
public class DisjointTest
{
 public static void main(String[] args)
 {
  DisjointSet ds = new DisjointSet(10);
  ds.print();
  ds.union(ds.find(4), ds.find(5));
  ds.print();
  ds.union(ds.find(6), ds.find(7));
  ds.print();
  ds.union(ds.find(5), ds.find(6));
  ds.print();
  ds.union(ds.find(0), ds.find(9));
  ds.print();
  ds.union(ds.find(2), ds.find(3));
  ds.print();
  ds.union(ds.find(2), ds.find(0));
  ds.print();
  ds.union(ds.find(0), ds.find(3));
  ds.print();
 }
}
```

The program assumes that the class *DisjointSet* is expanded by a method *print()* that prints the content of the array, and then the test method gives the following result:

```
[0:-1] [1:-1] [2:-1] [3:-1] [4:-1] [5:-1] [6:-1] [7:-1] [8:-1] [9:-1]
[0:-1] [1:-1] [2:-1] [3:-1] [4:-2] [5:4] [6:-1] [7:-1] [8:-1] [9:-1]
[0:-1] [1:-1] [2:-1] [3:-1] [4:-2] [5:4] [6:-2] [7:6] [8:-1] [9:-1]
[0:-1] [1:-1] [2:-1] [3:-1] [4:-3] [5:4] [6:4] [7:6] [8:-1] [9:-1]
[0:-2] [1:-1] [2:-1] [3:-1] [4:-3] [5:4] [6:4] [7:6] [8:-1] [9:0]
[0:-2] [1:-1] [2:-2] [3:2] [4:-3] [5:4] [6:4] [7:6] [8:-1] [9:0]
[0:2] [1:-1] [2:-3] [3:2] [4:-3] [5:4] [6:4] [7:6] [8:-1] [9:0]
[0:2] [1:-1] [2:-3] [3:2] [4:-3] [5:4] [6:4] [7:6] [8:-1] [9:0]
```

5.3 A MAZE

I will finish this chapter on backtracking with a program that shows a maze:



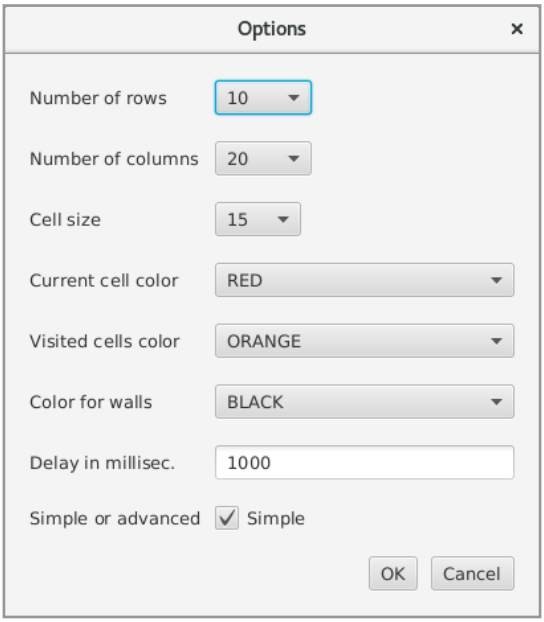
The user basically should be able to do two things:

1. You must be able to move the red field from the top left corner to the bottom right corner by clicking the mouse, but of course you can not move between two cells separated by a wall.
2. The user should be able to start a demo where the program itself moves the field.

The program fills a part, but most of it has to do with the user interface, and in reality there are only two challenges:

1. How to create a maze with a certain number of rows and columns
2. How to write an algorithm that automatically moves the field through the maze

Before I look at the code, I would like to mention that there is a menu item called *Options*, where the user can make some settings:



- You can choose the size of the maze in terms of number of rows and columns.
- You can choose the size of the individual cells – how much to fill on the screen.
- You can choose the color of the current cell and thus the color of the red square.
- You can choose the color used to draw walls between the individual cells.
- You can specify how much time (measured in milliseconds) that should be between the computer's moves of the square in connection with the demo function.
- Finally, two difficulty levels can be selected (although there are not big difference), but the meaning is explained below.

I do not want to show the code for the above dialog but basically maintains the dialog the following model:

```
public class Model
{
 public static final int MINROWS = 5;
 public static final int MAXROWS = 100;
 public static final int MINCOLS = 5;
 public static final int MAXCOLS = 200;
 public static final int MINSIZE = 5;
 public static final int MAXSIZE = 50;
 public static final Map<String, Color> colors = new TreeMap();
```

```
private int rows = 10;
private int cols = 20;
private int count = 0;
private int size = 15;
private String currentPos = "RED";
private String visitedPos = "ORANGE";
private String walls = "BLACK";
private int delay = 1000;
private boolean simple = true;
```

The first constants indicates the minimum and maximum values for the dialog box's ComboBox's, while colors are a *Map<String, Color>*, which indicates the colors that the application can use identified with a name.

The labyrinth's cells are defined by the following class:

```
public class Cell extends Canvas
{
 private Model model;
 private ObjectProperty<EventHandler<ActionEvent>> onActionProperty =
  new SimpleObjectProperty();
 private IntegerProperty row = new SimpleIntegerProperty();
 private IntegerProperty col = new SimpleIntegerProperty();
 private BooleanProperty left = new SimpleBooleanProperty(true);
 private BooleanProperty top = new SimpleBooleanProperty(true);
 private BooleanProperty right = new SimpleBooleanProperty(true);
 private BooleanProperty bottom = new SimpleBooleanProperty(true);
 private IntegerProperty state = new SimpleIntegerProperty(0);
```

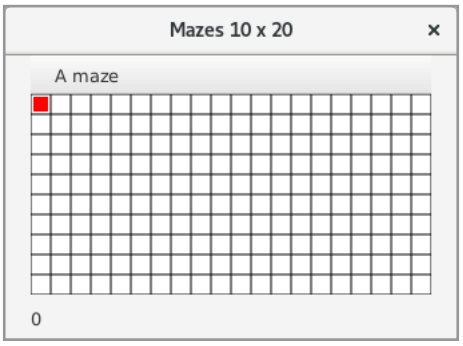
The class is, in principle, simple, but the code fills a part because of the many properties. The first two indicates the row and column for the cell in question. The next four indicate whether there is a wall, which there is from start, and it is then part of the algorithm that creates the maze to remove some of these walls. Finally, there is the last property *state* which means the following:

- 0, the cell is unvisited
- 1, the cell is visited
- 2, the cell is the current and contains the red square

Note that the class is a *Canvas*, and it is thus the class that is responsible for drawing the individual cells that occurs in a method *draw()*. Finally, the class can raise an *ActionEvent*, which happens if you click on a cell, and in the current case, it means that the main window will be notified that a cell has been clicked and which cell is clicked.

Back there is the class *Mazes* which fills a part, but the most important is the implementation of the above two algorithms.

To create a new maze, start creating the cells and placing them in a *GridPane*. From start all walls exist and the user interface only shows a number of squares:



and the task is now random to remove walls until the maze is made. Now you can not just remove walls by random, because the maze must be detachable, so there's a way through it. This is where you can use the class *DisjointSet*. If the maze as above has 10 rows and 20 columns, there are 200 cells, and if you arrange these cells in an array and using a *DisjointSet* there is a path through the maze when the first and the latter belong to the same equivalency class. The algorithm should therefore works as follows:

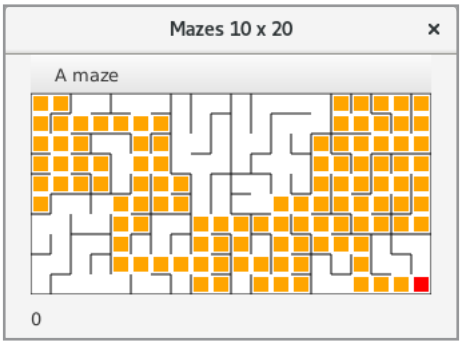
```
select a random cell
select random a neighbour cell
if (these cells do not belong to the same equivalency class)
{
 break down walls between the two cells
}
repeat the above until the first and last cell belong to the same
equivalence class
```

In principle, the algorithm is quite simple and works fine, but it has a problem as it can provide groups of cells that are closed and to where you can not move the red square. This problem can be solved by changing the last statement in the algorithm to:

```
repeat the above until all cells belongs to the same equivalence class
```

The algorithm will then break down more walls, and the red square can be moved to all cells. That's what I called a difficulty, because it basically gives a maze, which is harder to find through, as there are more options to go in the wrong direction. The downside is that the algorithm has a poor time complexity, as you can observed by a large maze.

Then there is the last algorithm where it is the machine that has to find through the maze:



Here the program will try to find through the maze. The program uses a strategy where it tries to go to the right if possible and if it is a place where it has not already been. If it is not possible it then try it next to the same guidelines to go down, then left on finally up. If it can not reach any of the places, it must backtrack until it comes back to a cell from which it can go on. In principle, it is simple (but not particularly effective) and the only challenge is to keep track of which cell to go to in backtracking. It can be solved simple with a stack.

6 OTHER ALGORITHMS

Above I have mentioned 4 different algorithm templates that are well known, and if you want to write an algorithm and the solution is not obvious, you can consider whether one of the above paradigms can be used. If so, it can provide guidelines for the problem's solution as well as an indication of the complexity of the solution. Now, however, not all algorithms can be characterized by these descriptions, and in this chapter I will give a few examples of algorithms that can not.

6.1 BRUTE FORCE ALGORITHMS

I have previously mentioned *brute force algorithms*, which are simply solving a problem by trying all the possibilities. In principle, there is nothing wrong with these kinds of algorithms, and you often can not do anything else. As an example, below is shown an algorithm that determines the divisors in a non negative integer:

```
public class BruteForce
{
 private static final Random rand = new Random();
 private static final long N = 100000;

 public static void main(String[] args)
 {
  for (int i = 0; i < 10; ++i)
  {
   List<Long> list = divisors(Math.abs(rand.nextLong()) % N);
   if (list.size() < 100)
   {
    for (Long t : list) System.out.print(t + " ");
    System.out.println();
   }
   else System.out.println("Number of divisors; " + list.size());
  }
 }

 private static List<Long> divisors(long t)
 {
  List<Long> list = new ArrayList();
  for (long d = 1; d <= t; ++d) if (t % d == 0) list.add(d);
  return list;
 }
}
```

Is the number called *N* the algorithm simply try to divide with all natural numbers that are less than or equal to *N*. Typically, brute force algorithms are simple (as is the case here too), but the problem is the time complexity and above it is *O(N)* and there is, of course, nothing wrong with that, but if *N* is large, the algorithm becomes slow.

As mentioned, it is not always possible to do much more than apply a brute force algorithm (think of linear search), but if you have a brute force algorithm to solve a problem, there is reason to consider whether there is a better algorithm.

6.2 RANDOMIZED ALGORITHMS

As another kind of algorithms, I would like to mention algorithms whose complexity depends on randomly chosen values. These are algorithms where it can be difficult to specify the time complexity and where one can only specify the complexity with one or other probability, but it can also be algorithms that determine the correct result with a given probability. Perhaps it may be difficult to imagine the use of algorithms where one can not be sure of the result, but if an algorithm is very complex and takes a very long time, a result that is likely to be correct, can be better than no results at all.

In the previous book, I have treated Quick sort as an $O\big(Nlog(N)\big)$ sorting algorithm, at least in average, but at worst, the complexity can be square, which occurs if the array is already sorted or almost sorted. The complexity depends on how lucky you are with the choice of pivot point and there are several techniques for solving this problem. When I examined quick sort I argued for choosing the median as a pivot point, but another option is to choose the pivot point random, which also provides a good solution.

As another example of how random values can be included in algorithms, I will once again look at the prime algorithm where the classic algorithm is:

```
public static boolean isPrime(long n)
{
 if (n == 2 || n == 3 || n == 5 || n == 7) return true;
 if (n < 11 || n % 2 == 0) return false;
 for (long t = 3, m = (long)Math.sqrt(n) + 1; t <= m; t += 2)
  if (n % t == 0) return false;
 return true;
}
```

In fact, it is nothing but an improved brute force algorithm, where it has been found that it is enough to divide with all odd numbers and where it is enough to try with all numbers

that are less than or equal to the square root of n. The algorithm's time complexity is thus $O(\sqrt{N})$.

Within the number theory, Fermat's Lesser Theorem is says:

If p *is a prime goes for all* $n = 1,2,3,\ldots,p-1$ *that* $a^{p-1} \equiv 1\, mod(p)$

This result can be used to determine if a number is a prime, for if a^{n-1} is not equivalent to 1 modulo n, then is n certainly a composite number, but if $a^{n-1} \equiv 1\, mod(n)$ is n possibly a prime number. Indeed, there are actually natural numbers such that $a^{n-1} \equiv 1\, mod(n)$ and so n is not a prime. As a step towards an algorithm that tests whether n is a prime you can start with

```
select a random number a, 1 < a < n-1
if (a^(n-1) % n == 1) n is probably a prime
else n is not a prime
```

This is of course not a safe algorithm, but it can be improved by using the following theorem, which also originates from the number theory:

If p is a prime the equation $x^2 \equiv 1\, mod(p)$ *only has the solutions* 1 and $p-1$ *within the set* $\{1,2,\ldots,p-1\}$.

To implement a corresponding algorithm, you can start by noticing the following method that raises *x* to the power of *n*:

```
public static long pow(long x, long n)
{
 if(n == 0) return 1;
 if(n == 1) return x;
 if(n % 2 == 0) return pow(x * x, n / 2);
 else return pow(x * x, n / 2) * x;
}
```

If you use that you can calculate modulus *n* in each iteration, and use that *n* from the beginning is uneven and add an additional parameter *k*, which from the beginning is *n-1* and then use the last of the above statements, test for prime can be written in the following manner:

```
private static long fermat(long a, long k, long n)
{
 if (k == 0) return 1;
 long x = fermat(a, k / 2, n);
 if( x == 0 ) return 0;
 long y = (x * x) % n;
 if (y == 1 && x != 1 && x != n - 1) return 0;
 if (k % 2 != 0) y = (a * y) % n;
 return y;
}
```

You can now write a method that is parallel to *isPrime()* and tests if an integer is a prime number:

```
public static boolean isPrime(long n, int t)
{
 if (n == 2 || n == 3 || n == 5 || n == 7) return true;
 if (n < 11 || n % 2 == 0) return false;
 for(int i = 0; i < t; ++i)
  if (fermat(random(2, n - 2), n - 1, n) != 1) return false;
 return true;
}
```

```
public static long random(long a, long b)
{
 return Math.abs(rand.nextLong()) % (b - a + 1) + a;
}
```

where *random()* is a method that returns a random integer from the closed interval [a; b]. You can not be sure of the result, among other things because the value of the parameter *a* in the method *fermat()* is selected randomly, but here you should note that the method *fermat()* is executed in a for loop and each iteration uses a new (random) value for *a*. That is, if *fermat()* returns that it may be a prime number, try several times (at least if *t* is greater than 1). That is, the higher values of *t*, the greater is the chance that the method returns the correct result, but it costs on performance.

Below is a test method (similar to the corresponding program from chapter 3):

```
public class PrimeTester
{
 private static final int N = 10000000;
 private static final int T = 5;

 public static void main(String[] args)
 {
  System.out.println(test(N).size() + " errors");
  StopWatch sw = new StopWatch();
  sw.start();
  long t1 = prime1(N);
  sw.stop();
  System.out.println(sw.getMilliSeconds());
  sw.start();
  long t2 = prime2(N);
  sw.stop();
  System.out.println(sw.getMilliSeconds());
  System.out.println(t1);
  System.out.println(t2);
 }

 public static List<Long> test(long n)
 {
  List<Long> list = new ArrayList();
  for (long t = 0; t <= n; ++t)
   if (Primes.isPrime(t) != Primes.isPrime(t, T)) list.add(t);
  return list;
 }
```

```
 public static long prime1(int n)
 {
  if (n == 0) return 2;
  long t = 1;
  for (int i = 0; i < n; )
  {
   for (t += 2; !Primes.isPrime(t); t += 2);
   ++i;
  }
  return t;
 }

 public static long prime2(int n)
 {
  if (n == 0) return 2;
  long t = 1;
  for (int i = 0; i < n; )
  {
   for (t += 2; !Primes.isPrime(t, T); t += 2);
   ++i;
  }
  return t;
 }
}
```

The method *test()* tests which of the numbers from 0 to N is prime (where N is 10000000), and each time the usual prime algorithm (which gives the correct result) and the above, is used and the method determines which results are incorrect. The method *prime1()* determines the first *N* primes and returns the last and hence the *N*th prime, but using the usual prime algorithm. The method *prime2()* performs the same, but instead uses the new prime algorithm. *main()* performs the three test methods and prints the results, but for the last two methods, how long the operations has taken, measured in milliseconds. Here you should note the constant *T*, which indicates how many times the new prime algorithm should try and thus the likelihood that the result is correct. Below is an example of a run of the program, where T = 1:

```
186 errors
414421
46504
179424691
179411257
```

and you can see that out of 10 million tests for prime numbers, the program in 186 cases has given a wrong result. That is why the last two prime numbers are different. On the other hand, there is a difference in the time factor by approximate a factor 10. As another run, the result is shown below, where T = 5:

```
0 errors
412047
66565
179424691
179424691
```

Now there is no mistake, but in return it costs on time.

6.3 SUDOKU

Sudoku is a popular puzzle game where you have to fill a square with 81 numbers, which must have values 1 to 9. An example could be as shown below, where a Sudoku has been defined, 36 of the fields are filled and thus the person who must solve the Sudoku has to fill the 45 other places when the following rules must be complied:

1. Each row must contain the numbers 1–9
2. Each column should contain the numbers 1–9
3. Each of the 9 3×3 squares must contain the numbers 1–9

and the task is solved when all the places are filled. Obviously, the severity depends on how many places are pre-filled (by the one who has defined the Sudoku, which below is the computer), but it also depends on how the numbers are located. It is said that a Sudoku can be solved if it has a unique solution, and as far as I know, it is not possible to define a Sudoku in which less than 17 places are pre-filled.

4				9	6			8
	5	9		2	4			6
	6		3				9	4
		2					6	
6	8					4	5	1
	7						8	
8	1	5	4			6	2	7
7						8		
2				6	8		1	5

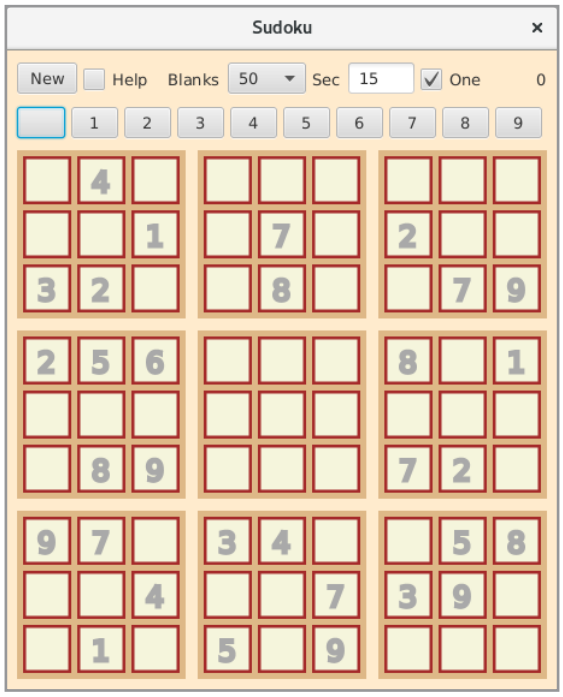
4	2	7	1	9	6	5	3	8
3	5	9	8	2	4	1	7	6
1	6	8	3	5	7	2	9	4
9	4	2	5	8	1	7	6	3
6	8	3	9	7	2	4	5	1
5	7	1	6	4	3	9	8	2
8	1	5	4	3	9	6	2	7
7	3	6	2	1	5	8	4	9
2	9	4	7	6	8	3	1	5

In this section I will show a program that can play Sudoku and where it is the user's task to solve the Sudoku. It is a program that has many versions, so it's easy to download a solution that you can compare to. There are generally two challenges:

1. The user interface
2. Generate a Sudoku

and in relation to the current book, it is of course the last one that is the most interesting.

As for the user interface, the application opens a window, as shown below. At the top, there is a kind of toolbar that has a start button, so you can create a new Sudoku. There is a check box, and if you put a check mark here, the program will check if the value in a cell is conflicting with the rules (the same number twice in the same row and so on) and, if so, you can not fill a cell with the number. The combo boxes are used to indicate how many blank cells there should be, and thus the severity, and finally there is a label that indicates how many times you have changed the value of a cell. The toolbar has two more components, which are an entry field and a check box, respectively. The meaning of these components is explained below.



To fill in a Sudoku you can either use the mouse or use the keyboard. If you click in a field (which is not filled in by the machine), the field is selected and you can either clear it or insert a number by clicking one of the buttons above the Sudoku. Alternatively, you can use the arrow keys and the numeric keypad. If you use the arrow keys and no cell is selected, the first cell is highlighted in the top row – the first cell that is not filled in by the computer and moving the selected cell using the arrow keys, skip the cells that the computer has filled. If you want to clear a filled cell, press the Delete key and you want to unselect a cell, press End.

I do not want to show the code of the user interface, which contains nothing new and by the way is not particularly complex, but I would rather mention the other challenge of making a new Sudoku. It requires a strategy and thus an algorithm, and I have used the following procedure:

1. create a empty Sudoku
2. fill the first column with the 9 numbers in random order
3. fill the next two cells in the top row randomly so that they do not violate the Sudoku rules for the first block
4. fill the rest of the top row with the remaining numbers in random order
5. fill the last 64 cells such that the results a legal Sudoku
6. blanks randomly the wanted number of cells

Here, step 2, 3 and 4, together with step 6, will ensure randomness so that you do not get the same Sudoku every time and the problem is so reduced to fill the remaining 64 cells, however, such that the cells values are determined randomly, but of course such that the rules for a Sudoku are constantly met. Step 5 is thus a complex problem, in particular because the Sudoku must be determined so that it has a unique solution. For the same reasons, step 6 is also complex because deletion of a field can lead to a Sudoku with multiple solutions. The solution of step 5 can be illustrated as follows:

```
1. while (there are cells that can be filled in a unique way) fill
   a cell
2. if (the Sudoku is filled) return true
3. find the cell where there are fewest choices for a number
4. for each of these cells
   {
   fill the cell
   repeat the algorithm
```

The solution of step 6 for clearing fields can be described as

```
1. select a random cell that is not blank
2. if (setting this cell blank leads to a solved Sudoku) blank
   this cell
3. repeat step 1 and step until the desired number of cells are
   blank
```

The next thing is to write an algorithm in Java according to these guidelines, which is not quite simple. The starting point is the following class, which represents a Sudoku:

```
public class Puzzle
{
 private int[][] values = new int[9][9];
```

If a value is 0, it is interpreted as the corresponding cell is blank. If the value is negative, it is interpreted as a value filled by the computer while a positive value is interpreted as the value set by the user. The class is generally simple and has only simple methods.

The next class is called *Solver*, and the class is used to fill a Sudoku. It's a relatively complex class, and the following explains the most important things, but you should look into the code for the details:

```
public class Solver
{
 // The method is recursive and in order to prevent the recurrence from running
 // indefinitely, an upper limit is defined for how many methods solve() must
```

```
// try to fill the sudoku
private static final int MaxTries = 5000;
// Refers to a solution if there is a solution found with the method solve()
private Puzzle solution = null;
// indicates the available options to fill in a cell
private boolean[] list = new boolean[10];
// number of solutions found
private int solutions = 0;
// used to count the number of recursions
private int count = 0;

…

// Method that tries to fill a Sudoku. If it goes well, the property solution
// will refer to the filled Sudoku. Otherwise it's zero.
// The first parameter puzzle is a Soduko which is partially filled.
// The second parameter unique indicates whether the Sudoku should have a
// unique solution. If required, the method may take a long time.
// The method is recursive and in order to
avoid too many recursions, the method
// is interrupted if the number of recursions exceeds the constant MaxTries.
public boolean solve(Puzzle puzzle, boolean unique)
{
```

```
 // work is done on a copy of the current Sudoku so you can try again
 // if it fails to fill it
 Puzzle copy = new Puzzle(puzzle);
 ++count;

 // fill cells as long as there are cells that can be filled in a unique way
 single(copy);
 // used to return three values from the method find() as Java only
 // has value parameters
 Triple triple = new Triple();
 // if there is a solution (the table is filled out) the method returns
 if (isSolved(copy))
 {
  ++solutions;
  // if the Sudoku should be unique the method give up
  if (unique && solutions > 1) return false;
  // solution is the finished Sudoku
  solution = copy;
  return true;
 }
 // otherwise the cell that can be filled in fewest ways is determined
 // if it is not possible to find a cell the method gives up
 else if (!find(copy, triple)) return false;
 // triple contains the index for the row and
 column for the cell and the number
 // of ways that the cell can be filled
 // the values that can be used to fill the cell is in the instance variable
 // list
 boolean ok = false;
 boolean solved = false;

 // tries to fill the cell with one of the possible digits indicated in list,
 // after which the method calls itself recursively
 for (int i = 1; !ok && i <= triple.cnt; ++i)
 {
  int k = select();
  list[k] = false;
  copy.setValue(triple.row, triple.col, k);
  if (count < MaxTries) solved = (solve(copy, unique)); else solved = false;
  if (!unique) ok = solved;
 }
 return solved;
}

…

// Returns true if t can be placed in the cell (r, c).
public static boolean ok(Puzzle puzzle, int r, int c, int t)
{
 …
}
```

```
 // References a cell and an index between 0 and 9
 class Triple
 {
  int row;
  int col;
  int cnt;
 }
}
```

With this class in place, I can write a class *Generator* that implements a method that creates and returns a Sudoku with a certain number of blanks. Again, I would only like to outline the solution and refer to the completed code regarding the details:

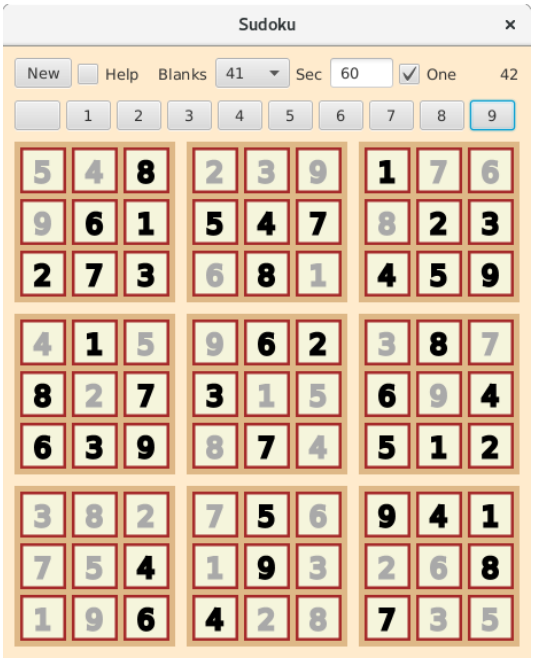
```
public class Generator
{
 public static final Random rand = new Random();
 // The largest number of trys to blank a cell before giving up and
 // selecting another cell
 private static final int MaxTries = 5000;

 // Creates a new sudoku with a certain number of blank and within a given
 // number of seconds. If the method can not form a Sudoku with the desired
 // number of blank cells within this period, it gives up and returns
 // the best Sudoku found, that is the Sukodu with as many shiny as possible.
 // The parameters are the number of blanks, how many seconds before the
 // method gives up and where the Sudoku must have a unique solution.
 public Puzzle create(int blanks, int seconds, boolean unique)
 {
  Puzzle puzzle = tryCreate(blanks, unique);
  int t1 = getTime();
  // if you do not get the desired result, repeat the process until you
  // exceed the maximum allowed number of seconds.
  while (puzzle.getBlanks() < blanks)
  {
   Puzzle temp = tryCreate(blanks, unique);
   if (temp.getBlanks() > puzzle.getBlanks()) puzzle = new Puzzle(temp);
   int t2 = getTime();
   if (t2 - t1 > seconds) break;
  }
  return puzzle;
 }

 // Creates a new sudoku with a certain number of blanks.
 // The algorithm works as follows:
 // The first column is filled in with the 9 numbers in random order.
 // The next two numbers in the top row are filled randomly so that they
 // do not violate the Sudoku rules for the first block.
```

```
// The rest of the top row is filled in with the remaining 6 numbers
// in random order.
// The rest of the table is filled using the class Solver.
// The method randomly blanks the number of cells you want.
// If it is not possible to blanks the desired number of cells, the
// operation is interrupted and you get a Sudoku with the blank cells that
// that has been blanked.
private Puzzle tryCreate(int blanks, boolean unique)
{
 Puzzle puzzle = new Puzzle();
 int row = 0;
 int col = 0;
 // set1 is the 9 values
 List<Integer> set1 = new ArrayList();
 for (int i = 1; i <= 9; ++i) set1.add(i);
 List<Integer> set2 = new ArrayList();
 int t = set1.get(rand.nextInt(9));
 puzzle.setValue(row, col, t);
 set1.remove(new Integer(t));
 // fill the first column
 for (row = 1; row < 9; ++row)
 {
  t = set1.get(rand.nextInt(set1.size()));
  set2.add(t);
```

```
   set1.remove(new Integer(t));
   puzzle.setValue(row, col, t);
  }
  // fill the next two cells in the first row
  for (row = 0, col = 1; col < 3; ++col)
  {
   t = set2.get(rand.nextInt(set2.size()));
   while (t == puzzle.getValue(1, 0) || t == puzzle.getValue(2, 0))
    t = set2.get(rand.nextInt(set2.size()));
   set2.remove(new Integer(t));
   puzzle.setValue(row, col, t);
  }
  // fill the last 6 cells in the top row
  for (col = 3; col < 9; col++)
  {
   t = set2.get(rand.nextInt(set2.size()));
   set2.remove(new Integer(t));
   puzzle.setValue(row, col, t);
  }
  // use the class Solver to fill the rest of the Sudoku
  Puzzle solution = null;
  do
  {
   Solver solver = new Solver();
   solver.solve(new Puzzle(puzzle), unique);
   solution = solver.getSolution();
  }
  while (solution == null || solution.isBlank());
  // blanks the desired number of cells
  puzzle = blank(solution, blanks, unique);
  return puzzle;
}

// Blanks cells corresponding to the desired number blanks.
// Each time a cell has been set (possibly two cells), the method tries
// to solve the Sudoku with a Solver object. If blanking cels does not
// lead to a solution, the method give it up the try another cell.
// If you try with a cell more than MaxTries times, give up.
public Puzzle blank(Puzzle puzzle, int blanks, boolean unique)
{
 Puzzle temp = new Puzzle(puzzle);
 Counter counter = new Counter();
 int tries = 0;
 do
 {
  Puzzle copy = new Puzzle(temp);
  blank(temp, counter);
  Solver solver = new Solver();
  if (!solver.solve(new Puzzle(temp), unique))
  {
```

```
    temp = new Puzzle(copy);
    tries++;
   }
  }
  while (counter.count < blanks && tries < MaxTries);
  puzzle = temp;
  puzzle.setReadonly();
  return puzzle;
 }

 // Blank one or two cells that are symmetrical in the same row,
 // in the same column or diagonally.
 // The first cell is selected randomly and blanked while the last cell
 // is blanked if it is not already.
 private void blank(Puzzle puzzle, Counter counter)
 {
  …
 }

 …
}
```

Then I can explain the meaning of the last two components of the toolbar in the user interface, where you can enter the maximum number of seconds that the program may use to display cells and partly if the solution should be unique. If a Sudoku has a unique solution (what it should in principle have) it's more difficult to solve it, but the program should be used for general enjoyment, it may not mean that much and therefore these two settings.



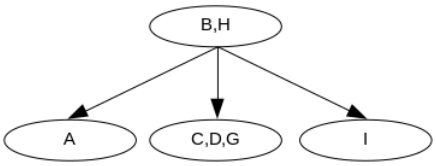
7 MORE ABOUT TREES

In the previous book I have treated binary trees, and especially binary search trees. As an example of a balanced binary search tree, I have shown an AVL tree. In this chapter I will look a bit more on trees, where I will partly show the implementation of another balanced binary search tree, called a *red-black tree*, and I will also give an introduction to general search trees.

7.1 2-3-4 TRESS

Before looking at the implementation of a red-black tree, I will look at a tree, which is called a 2-3-4 tree, which is a tree of degree 4 and hence a tree where a node can have no more than 4 children. I do not want to implement the tree, but it can be used to explain another method of balancing binary trees.

A node may have one, two or three values:

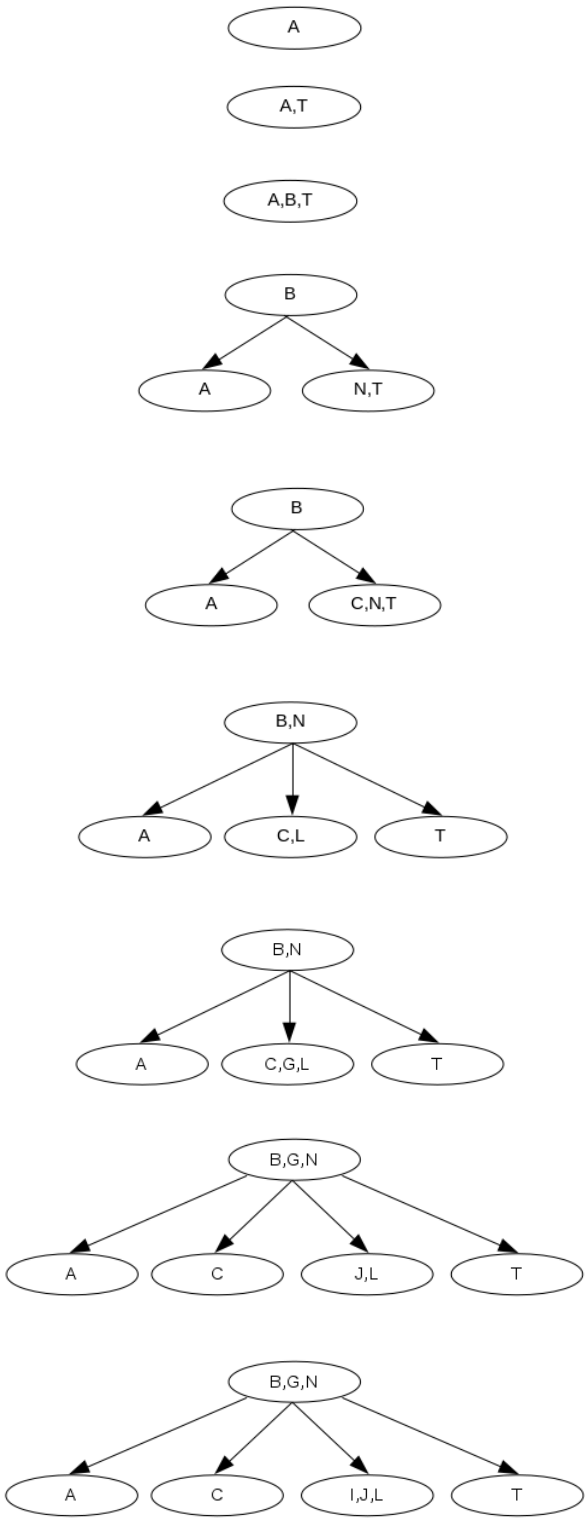


It is a tree with a root, and (assuming that the same element must not occur twice) are the rules:

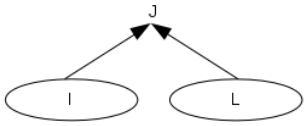
1. A node has elements that are arranged in ascending order.
2. A 2-node (one-value node) has two sub-trees, and the value is greater than all elements in the left sub-tree and less than all values in the right sub-tree.
3. A 3-node (a two-value node) has three sub-trees, where the first element is larger than all elements in the left and less than all values in the middle sub-tree and the second element is greater than all the values in the middle sub-tree and less than all values in the right sub-tree.
4. A 4-node (a three-value node) has four sub-trees (as numbered from the left, respectively, the first, second, third and fourth sub-trees), where the first element is greater than all values in the first sub-tree and less than all values in the second sub-tree, the middle element is greater than all values in the second sub-tree and less than all values in the third sub-tree and the last element is greater than all values in the third sub-tree and less than all values in the fourth sub-tree.

If you want to insert an element, you search from the root and find the node where the element is to be inserted. If it is a 2-node or a 3-node, the element can be inserted directly. On the other hand, if it is a 4-node, the node is split into two 2-nodes, where the middle value is inserted in the parent. The new element can then be inserted into one of the two 2-nodes that results from the split. When the middle element is moved to the parent, the result can of course be that parent becomes full, and if so, the split must be repeated at this level.

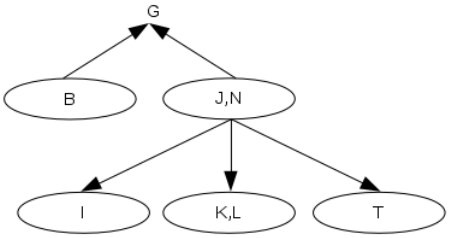
As an example, I have shown below what happens if you construct a 2-3-4 tree consisting of the elements A, T, B, N, C, L, G, J, I and inserted in this order:



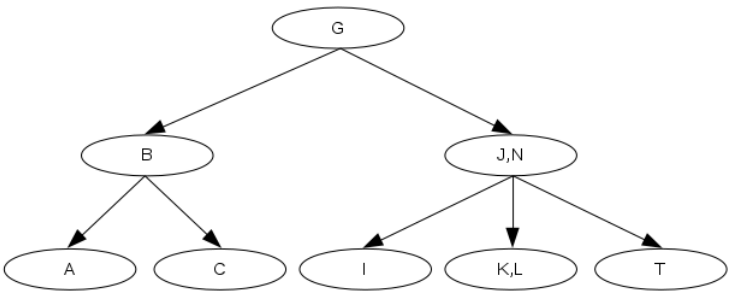
Suppose that you then have to insert K. The insertion must take place in a 4-node, and it is therefore necessary to split again:



where then J bubbles up in the tree. Since J must also be inserted in a 4-node, it is necessary to split again:



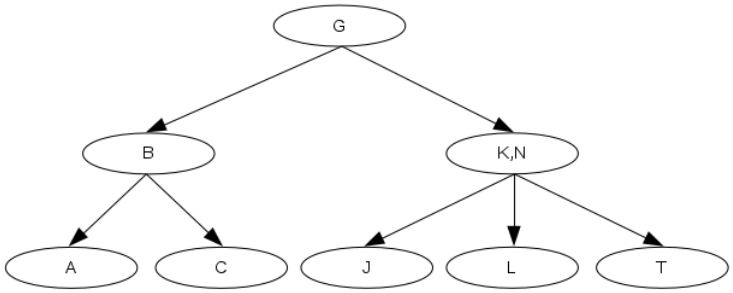
The result is that G becomes a new root, as shown below. Interestingly, the result is a balancedtree. In this case it is a tree with 10 elements and the height 2.



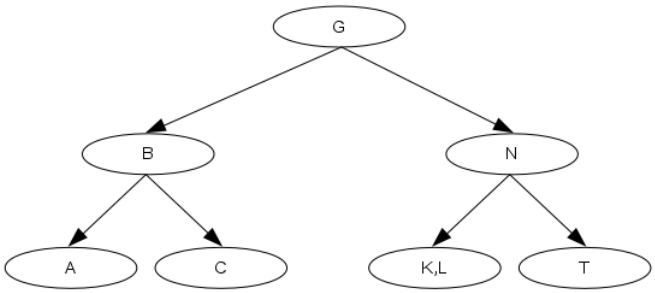
Deleting an element is more difficult (there are several cases), and it is based on rotations and fusions of nodes (it merges three 2-nodes together to a single 4 node). Deleting an element can be described as follows:

- Find the element to be deleted.
- If the element is not a leaf node, you mark the position and continue searching until you comes to a leaf that contains the element's successor (it may be either the largest element that is smaller than the element to be deleted, or it may be the smallest element larger than the element to be deleted). The easiest way is to adjust the tree top down, as described below, such that the leaf node you reach is not a 2-node. If that is the case, the element in the node marked (the element to be deleted) can immediately be replaced by the successor in the leaf node.
- If the element to be deleted is in a leaf node and it is not a 2-node, the element can be deleted immediately.
- If the element to be deleted is in a leaf node and it is a 2-node, the tree must be adjusted as described below.
- Perform the following adjustments of the tree if on the road from the root to the above leaf node you meet a 2-node (but not for the root, which is specially treated):
    1. If parent is a 2-node and the sibling nodes is also 2-node, all three nodes are combinedto a new 4 node. Then the parent's current node will never be a 2-node.
    2. If there is a sibling on one of the sides that is not a 2-node, a rotation with this sibling is performed. As a result, elements from the sibling are moved to parent, while an element from parent moves to the current node, which is then a 3-node.
    3. If there is no sibling with more than one value, a fusion is performed with the parent and a sibling. The result is a new 4-node.

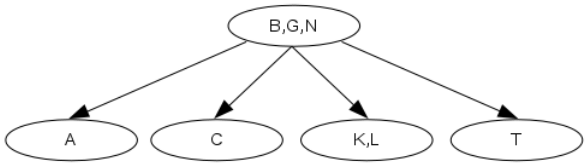
The procedure can best be illustrated by a few examples. Suppose that in the above tree you want to delete I. On the way to I, you will not meet 2-nodes (except for the root), so no adjustments will be made to the tree. You are in a 2-node and it has a sibling that is not a 2-node. Therefore, a rotation must be performed where K moves up one level while J moves down to the same node as I, which can then be deleted. The result is as shown below:



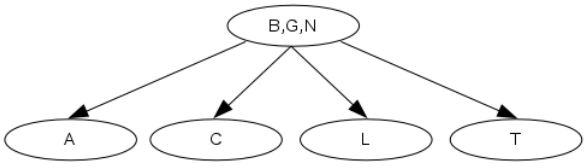
Suppose then you want to delete J. On the way down to J there are no 2-nodes, so there is no adjustment of the tree. J is in a 2-node, and its sibling is also a 2-node, and therefore there must be a fusion of the two nodes containing J and L and the element K from the parent node. The result is a 4 node from which J can be deleted:



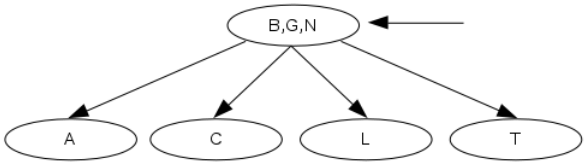
Finally, suppose you also want to delete K. When searching for K, you come to N, there is a 2-node, and since its sibling is also a 2-node, there must be a fusion:



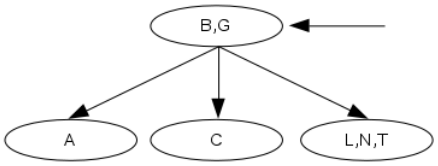
Then the search can continue and you get to the node containing K and the element can immediately be deleted, since it is not a 2-node:



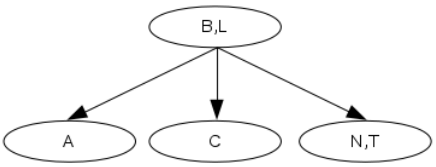
Assume as the last example that you want to delete G. You mark the node containing G and continue searching to the successor L:



L's siblings are 2-node so there must be a fusion with one of them, for example T:



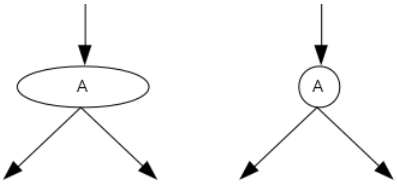
After that, L must replace G in the selected node:



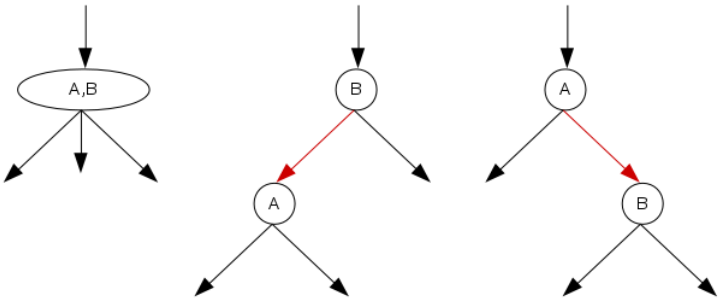
7.2 RED-BLACK TRESS

As mentioned above, you do not directly implement a 2-3-4 tree, among other things because there are several kinds of nodes, and both insertion and deletion results in many special cases. However, you can easily convert a 2-3-4 tree to a binary tree, which is called a *red-black tree*, and the result becomes a reasonably balanced tree. The idea is that all links between two nodes are assigned a color that is either red or black. It is said that a node is black if its parent's link is black while a node is red if its parent link is red. A 2-3-4 tree can be transformed into a binary tree as follows:

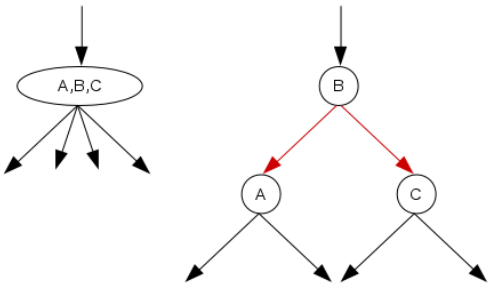
A 2-node is transformed into a usual binary node with two black children (which may be empty):



A 3-node is mapped to two binary nodes, one being red (there are two options):



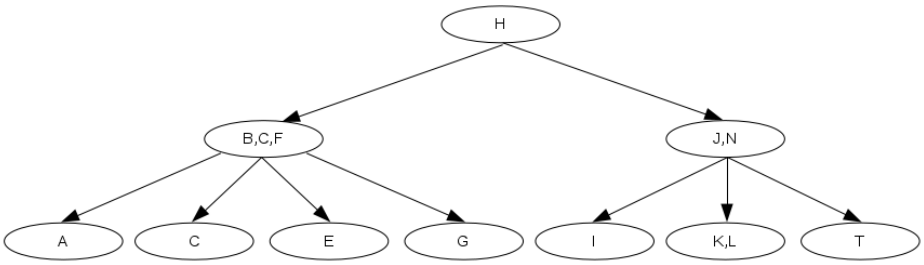
Finally, a 4-node is mapped to three binary nodes, where both children are red:



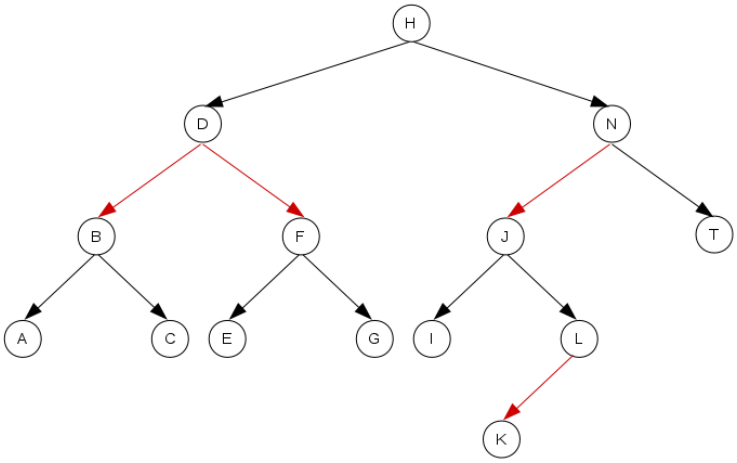
The result of this transformation is as follows, and is then the definition of a *red-black tree*:

1. There can not be two red nodes next to each other (in the above figures, links whose successor are not shown are all black).
2. As the black depth of a node, you understand the number of black links from the root to the appropriate node. In a red-black tree, all leaf nodes have the same black depth. This is because all red links correspond to 3-nodes and 4-nodes and therefore do not contribute to the height of the corresponding 2-3-4 tree.
3. There is no link to the root, and it is assumed that it means that the root is black.

Consider as example the 2-3-4 tree shown below:



If you convert it into a red-black tree, the result is:



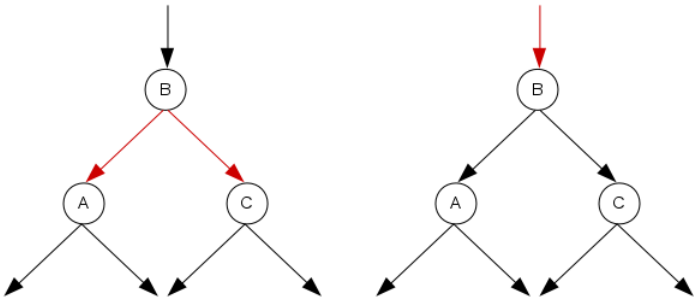
Here you should note that there are now not two red nodes that are successors, and that all leaf nodes have the same black depth that is the height of the corresponding 2-3-4 tree.

A red-black tree is not balanced, but if you counting only black links it is. A 3-node and a 4-node may insert an additional depth in the tree, and therefore the height of a red-black tree is no more than double the corresponding 2-3-4 tree and thus the tree is reasonably balanced.

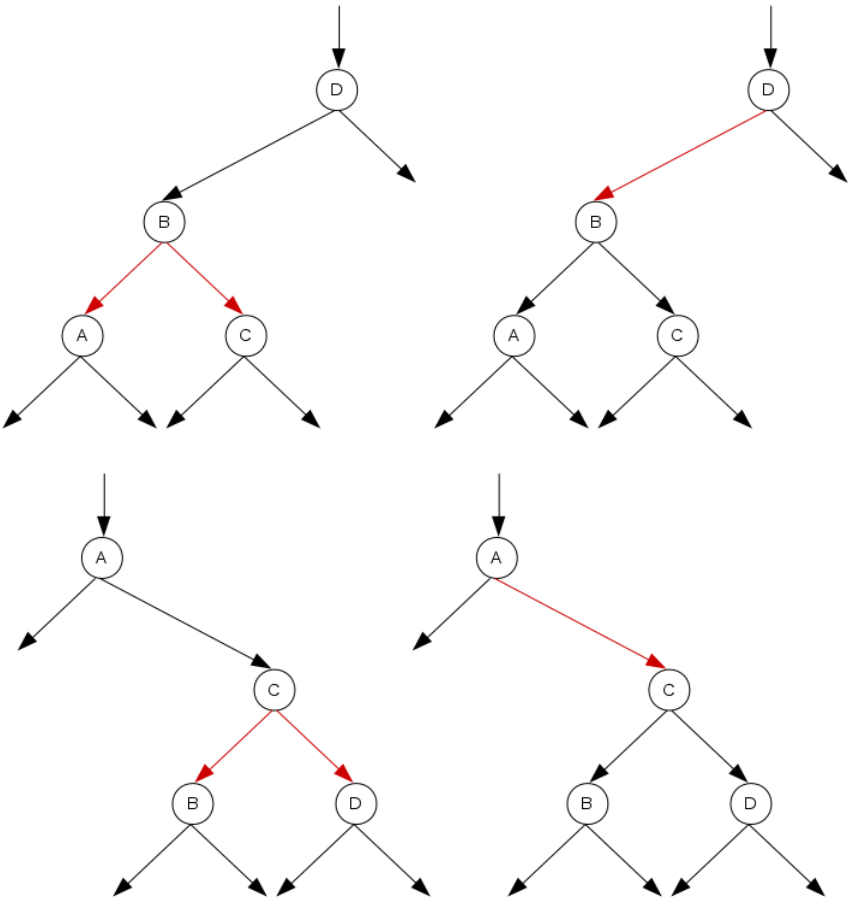
A red-black tree is especially a binary search tree and therefore, all that has been said in the previous book about binary search trees, including balanced search trees, are still validly understood that it is possible to implement insertion and deletion so that the requirements for a red-black tree is maintained. It is possible, but it's not simple.

I will start with insertion. Above I described insertion into a 2-3-4 tree as a search down in the tree to find where the new element is to be inserted and followed by a return to split any 4-nodes that are filled up. That is, there was both a top-down pass and a bottom-up pass (similar to the AVL tree). It is possible to perform this insertion in a single top-down pass by ensuring that the new element is not inserted into a full 4-node. This can be done by searching down the tree and split all the 4 nodes encountered in two 2-nodes. In a

red-black tree, it is an extremely simple process, as a 4-node is known to have two red children, and the split consists of a simple color change:

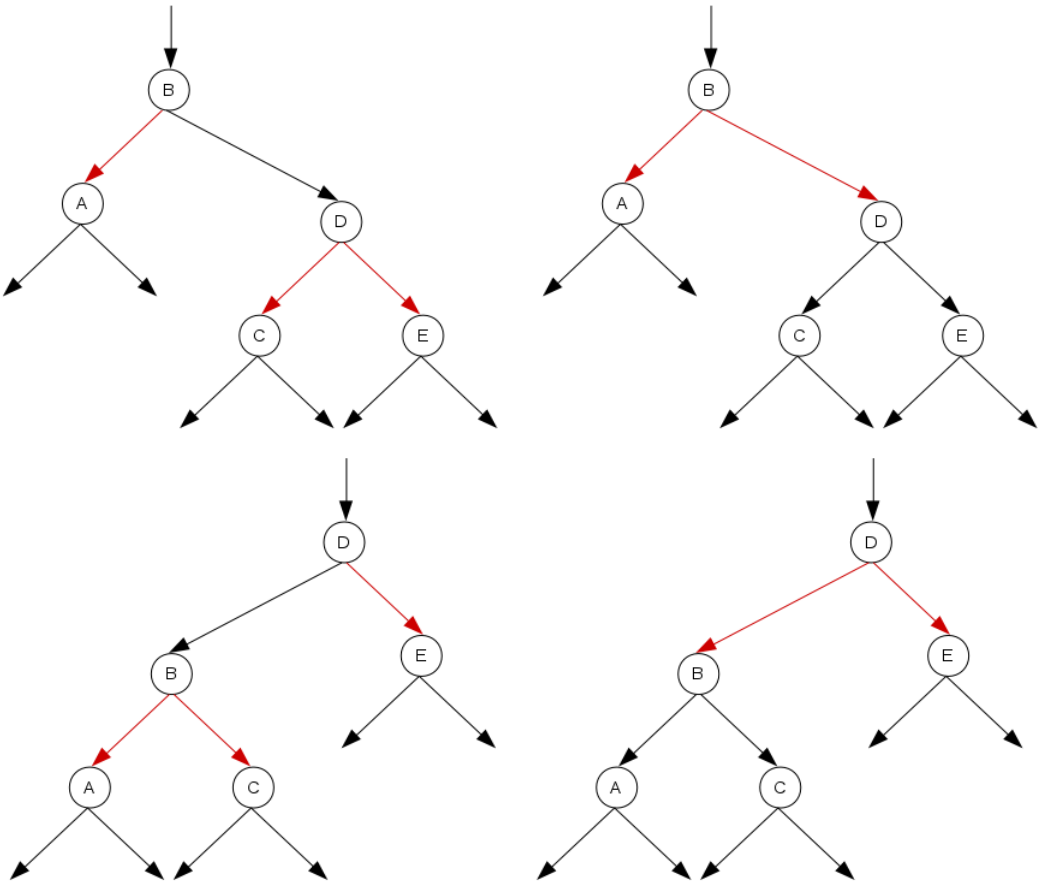


This preserves the black depth of all leaf nodes. Since B is now red, there is a problem if B's parent is also red, as you have two subsequent red nodes. This problem is solved with rotations. If a 4-node is succeeding to a 2-node, there are two options:

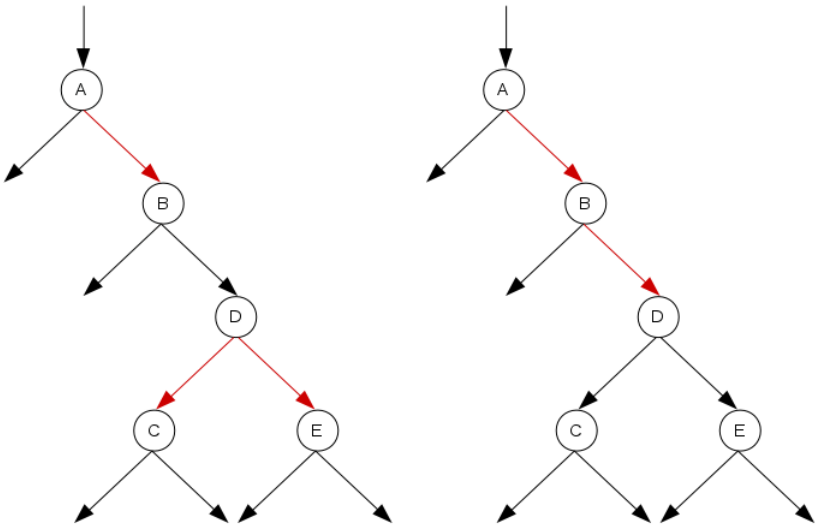


None of these situations cause problems. If a 4-node instead follows a 3-node, there are three options, as the 4 node may be a left child, a right child or a middle child. As shown above, the conversion of a 3-node to a red-black tree can lead to two orientations, so there

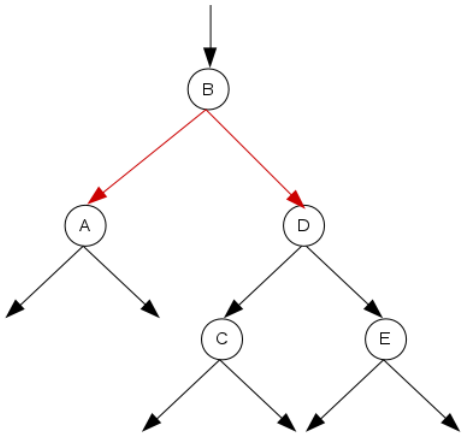
are actually 6 cases. Below are two situations where a 4-node is a right child and a left child for a 3-node but a 3-node with two different orientations:



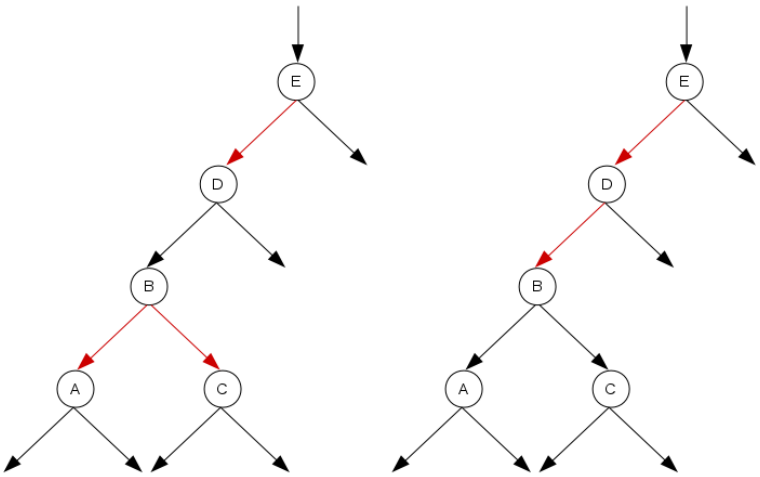
In both cases, the color change does not cause a problem (no two red nodes occurs as successors). If, on the other hand, a 4-node still sits as a right and a left child, but opposite:



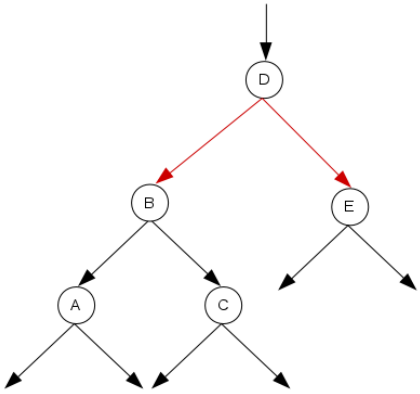
then the color change causes a problem as there will be two subsequent red nodes. The problem is solved with a left rotation about B:



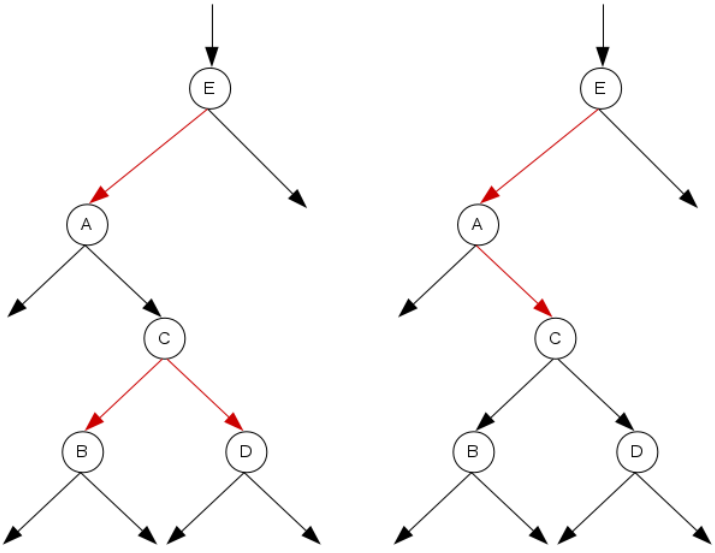
Here too there is a parallel situation where the orientation is the opposite:



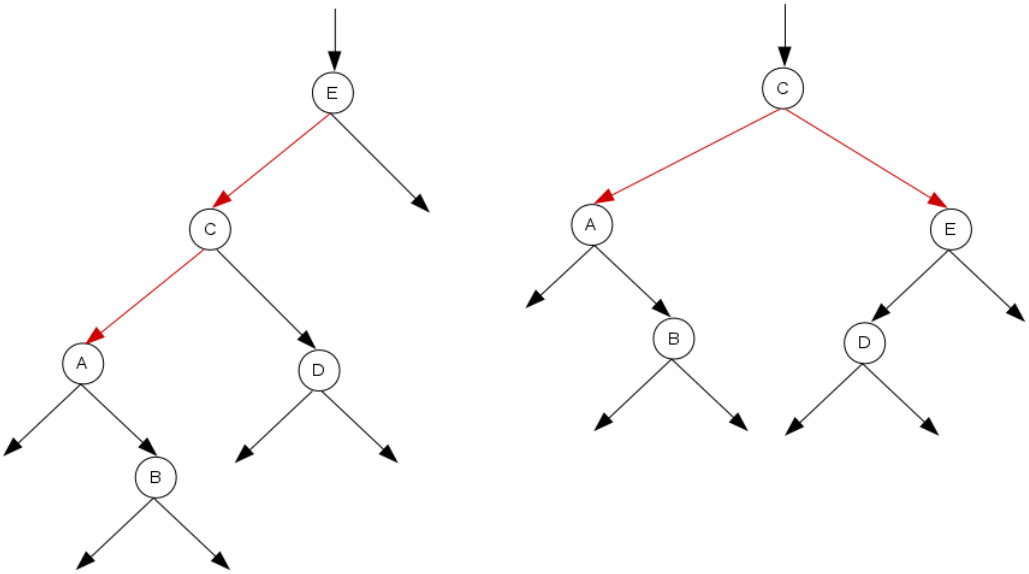
and this time the problem is solved by of a right rotation around E:



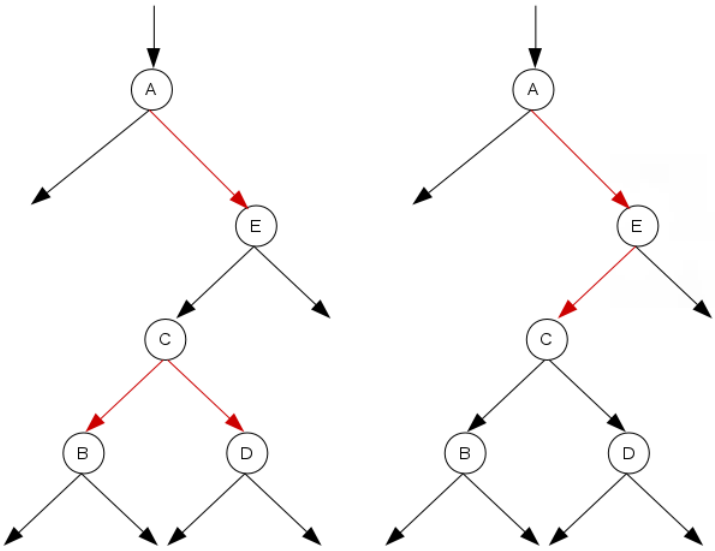
A 4-node can also be the middle child to a 3-node, and if you make a color change here, the result is two subsequent red nodes:



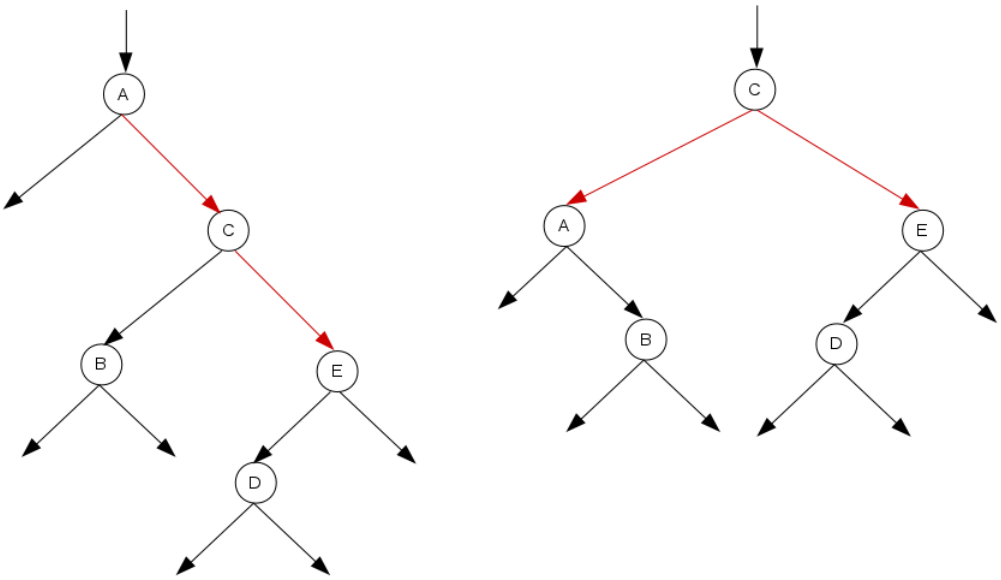
This time, the balance is restored by first performing a left rotation in A and then a right rotation in E:



Here too, there is another case where the 3-node has the opposite orientation:



This time, the balance is restored with a right rotation in E and then a left rotation in A:



Immediately there is another case where a 4 node could be a child to another 4 node, but it is inherently impossible because such a 4-node parent would already be split.

The result of all these operations is that when you find the leaf where the new element is to be inserted, it may happen in a 4-node that should be split, but on the way to the leaf node all 4 nodes are removed, and parent will have room for an additional element. This means that the insertion can be done with a single top-down pass and in such a way that the tree retains its black depth and is still "nicely" balanced.

A red-black tree can, in the same way as an AVL tree, be implemented as a class that inherits *BinSearchTree<T extends Comparable<T >>* (see the previous book on binary trees). To implement the tree, it is necessary to add an additional attribute to a node that can control the color:

```
class RBNode<T extends Comparable<T>> extends DoubleNode<T>
{
 private TreeColor color = TreeColor.BLACK;
```

Here, *TreeColor* is a simple enumeration that defines the two colors. After that, insertion of an element can be implemented as described above, but it requires some code lines, and it can be difficult to keep track of what happens. You are encouraged to study the code, and the easiest is always thinking of a 2-3-4 tree.

Back there is removing a node, and it is implemented as described under 2-3-4 tree. That is, searching down the tree for the element to be deleted. If you get to a 2-node, you do a fusion and a new 4-node will be created, which in a red-black tree will consist of a color change. This can lead to problems with two subsequent red nodes, and the balance must then be restored with rotations. There are several options, and I will not illustrate the many cases with examples here, but you may want to examine the completed code that is both long and complex.

The finished class in the library *palib* and is called *RBTree*.

EXERCISE 4: TEST RED-BLACK TREE

You must write a program that is almost identical to the program *AVLProgram* from the book Java 18 (exercise 16), except that an *AVLTree* must be replaced by an *RBTree* everywhere.

7.3 GENERAL TREES

In the rest of this chapter, I will look at the implementation of general trees, which were briefly mentioned in the previous book. In fact, it is not easy to write an abstract data type that represents a general tree, at least not so that it is effective, and can be used for something, but vice versa there are many examples where data is organized in a general tree (and just think of a file system for a disk or, for example, a *TreeView* control in *JavaFX*), so there are good reasons to be interested in writing a class that represents a general tree.

In this section, I would like only to interfere with a tree where there are no requirements besides that the are elements organized in a hierarchy, where each element has a unique parent (except for the root that does not have any parent). The tree can be defined as follows:

```
public interface Tree<T> extends Iterable<TreeNode<T>>
{
 public TreeNode<T> getRoot();
 public TreeNode<T> add(TreeNode<T> parent, T element);
 public TreeNode<T> find(T element);
 public IList<TreeNode<T>> getPath(TreeNode<T> node);
 public boolean remove(TreeNode<T> node);
 public int getSize();
}
```

and here you should note that there are few methods that are primarily limited to inserting elements, deleting elements and using *find()* asking if an element exists in the tree. You should also note that the methods directly refer to the internal nodes and thus to the implementation of the tree, which is one of the problems of implementing a general tree. Data encapsulation is not as desired. Finally, you should note that the interface says that a tree should implement the iterator pattern.

The tree is constructed as *TreeNode<T>* nodes, where T is the data type for the elements that the tree has to organize. The class is as follows:

```
public class TreeNode<T>
{
 private T element;
 private TreeNode<T> parent;
 private IList<TreeNode<T>> children = new ArrayList();
 private int level;

 public TreeNode(T element, TreeNode<T> parent)
 {
  this.parent = parent;
  this.element = element;
```

```
 level = parent == null ? 0 : parent.level + 1;
}

public T getElement()
{
 return element;
}

public int getLevel()
{
 return level;
}

public int getDegree()
{
 return children.getSize();
}

public TreeNode<T> getParent()
{
 return parent;
}
```

```
 public IList<TreeNode<T>> getChildren()
 {
  return children;
 }

 public TreeNode<T> add(T element)
 {
  TreeNode<T> node = new TreeNode(element, this);
  children.add(node);
  return node;
 }

 public boolean remove()
 {
  if (parent == null) return false;
  return parent.children.remove(this);
 }

 @Override
 public int hashCode()
 {
  return element.hashCode();
 }

 @Override
 public boolean equals(Object obj)
 {
  return element.equals(obj);
 }
}
```

The class has more references this time, where a node can refer to other nodes. Since the degree of a node can be arbitrary, a node has a list of references to child nodes and thus to the sub-trees. You should note that this list is always created, which is not necessary as all leaf nodes do not need the list. One could therefore achieve better efficiency by just creating the list for composite nodes, and when I always creates the lists, it is only to make the code a bit simpler. Also note that a node has a parent pointer, thus referring to its parent. The aim of these pointers is to make the implementation of some of the methods more effective. Finally, note that a node has a level, so you know at what level it is part of a tree. Note that the number of elements in the children list is the degree of the node.

The class's methods are somewhat trivial, but you must note the method *add()*, which creates a new node and puts it as a child to the current node. You should also note the method *remove()*, which deletes the current element. This is done by deleting the current element in the parent node's child list. Note that this means deleting the sub-tree that has the current element as root. The method *remove()* starts by testing if the node's parent is zero because it should not be possible to delete the root.

The tree itself can then be implemented as follows:

```
public class GeneralTree<T> implements Tree<T>
{
 private TreeNode<T> root;
 private int size = 1;

 public GeneralTree(T element)
 {
  root = new TreeNode<T>(element, null);
 }

 public TreeNode<T> getRoot()
 {
  return root;
 }

 public TreeNode<T> add(TreeNode<T> parent, T element)
 {
  ++size;
  return parent.add(element);
 }

 public TreeNode<T> find(T element)
 {
  for (TreeNode<T> node : this) if (node.getElement().equals(element))
   return node;
  return null;
 }

 public List<TreeNode<T>> getPath(TreeNode<T> node)
 {
  List<TreeNode<T>> path = new ArrayList();
  for ( ; node != null; node = node.getParent()) path.add(node);
  return path;
 }

 public boolean remove(TreeNode<T> node)
 {
  if (node.remove())
  {
   --size;
   return true;
  }
  return false;
 }
```

```
public int getSize()
{
 return size;
}

public Iterator<TreeNode<T>> iterator()
{
 return new TreeIterator();
}

private class TreeIterator implements Iterator<TreeNode<T>>
{
 private Queue<TreeNode<T>> queue = new LinkedQueue();

 public TreeIterator()
 {
  queue.enqueue(root);
 }

 @Override
 public boolean hasNext()
 {
  return !queue.isEmpty();
 }
```

```
 @Override
 public TreeNode<T> next()
 {
  try
  {
   TreeNode<T> node = queue.dequeue();
   for (TreeNode<T> child : node.getChildren()) queue.enqueue(child);
   return node;
  }
  catch(Exception ex)
  {
   return null;
  }
 }
}
```

Here the implementation of the methods is simple, and the only exception is the iterator implemented as an inner class. The first is to decide in what order the individual elements are to be visited, and I have chosen that it will take place in level order. The iterator can therefore be implemented using a queue in much the same way as in the previous book I implemented level order traversing a binary tree.

If you consider the methods *add()* and *remove()*, they are actually extremely effective because they are implemented with constant time complexity, but it is only apparent because they require that you know the node in which the new element is to be inserted or the node to be deleted. Also note that when implementing remove with constant time complexity, it is because a node has a parent pointer. To find a node (which must deleted or be parent to a new node), use the method *find()*, and this is the problem, because it has a linear time complexity.

Finally, note the method *getPath()*, which returns the path from the root to a node. When you can implement this method with reasonable efficiency, it is again due to the parent reference.

The class has not implemented a method to return the height or a method to the degree of the tree. It is not so easy to do it in a good way as they typically will have linear complexity. One could try to maintain two instance variables for height and degree, respectively, which should be maintained in conjunction with *add()* and *remove()*. It is simple in connection with *add()*, but is not so easy in connection with *remove()*. With the current implementation you can therefore determine the height and the degree by traversing the tree.

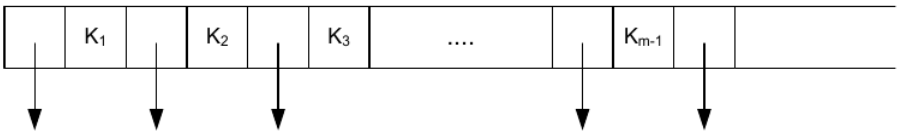
EXERCICE 5: TEST A GENERAL TREE

You must write a program that you can call *TreeTester* when the program is used to test the above class *GeneralTree*. The program can be outlined as follows:

```
public class TreeTester
{
 public static void main(String[] args)
 {
  // a general tree to integers
  ITree<Integer> tree = new GeneralTree(0);
  // add the values 1, 2, 3, …, 9 to the tree such that 1 is parent to 2,
  // 2 parent to 3 and so on

  // add 100 random numbers less than 100000 to the tre, when a number t
  // must be a child to the node whose value is the cross sum of t

  // print the tree
 }

 // Method which prints the tree with one line for each level. In the line
 // the elements should be separated with a space, and an element
 // should be printed as two values:
 // level:value
 // The method must also prints the size, the height and the degree of the tre.
 private static void print(ITree<Integer> tree)
 {
 }

 // Returns the cross sum of the integer t.
 private static int digetSum(int t)
 {
 }
}
```

7.4 A SEARCH TREE

In the previous book and above under red-black trees, I have looked at binary search trees, which are trees, where you can find an element with the logarithmic time complexity. You can also consider general search trees, which is a tree of the order *n* and which consists of nodes, where each node consists of max *n-1* search values and max *n* pointers to sub-trees:



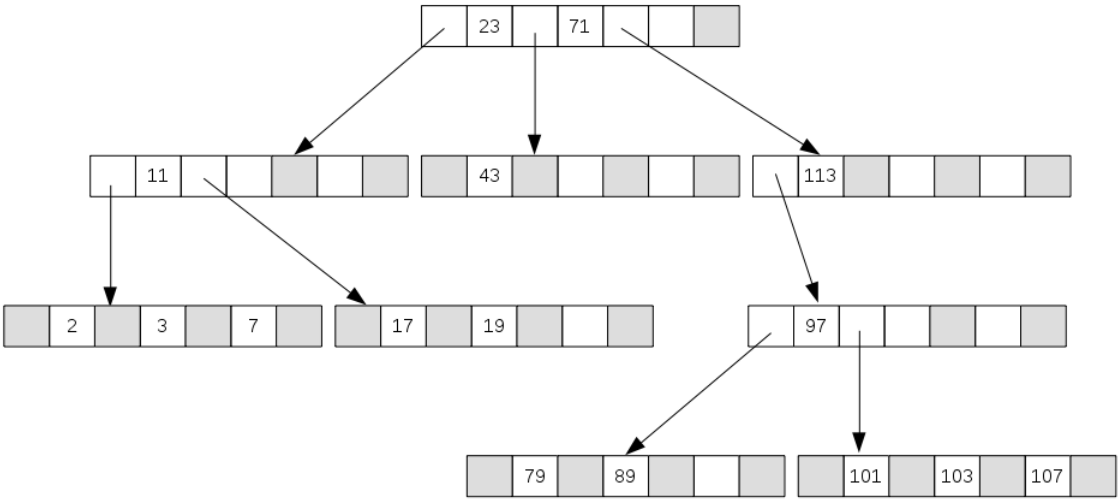
where:

1. $m \leq n$
2. K_i are ordered search values so $K_1 < K_2 <\ldots< K_{m-1}$
3. each pointer is an either zero or points to a node and thus a sub search tree

If a pointer P points to a node, the following requirements must be met:

1. if P is the first pointer all values X in the sub-tree, that P points on must meet $X < K_1$
2. if P is the last pointer all values X in the sub-tree, that P points on must meet $X > K_{m-1}$
3. if P is the *i*th pointer where $1 < i < m$ must all values X in the sub-tree, that P points on meet $K_{i-1} < X < K_i$

Note the last blank area in the above figure, which will illustrate that a node is not necessarily filled, that means that $m < n$. A 2-3-4 tree is thus a search tree of the order 4. As an example, the figure below illustrates a search tree, where the gray boxes illustrate zero pointers:



If you want to search for a particular element in such a tree, you search recursively through the tree until you either find the element or find that the element is not there. Assume that, for example, you will search for the number 89 in the above tree:

1. by searching in the root you can find that if the number is in the tree it must be in the right sub-tree
2. by searching in the root of this sub-tree you can deduce that if 89 is found in the tree it must be in the left sub-tree
3. by searching in the root of this sub-tree you can once again find that the number is in the left sub-tree
4. and by searching in the root of this sub-tree (which is a leaf node) the number is found

It was therefore necessary to visit 4 nodes. Assume, as another example, that you want to find the number 47:

1. when searching in the root you can find that if the number is in the tree it must be in the sub-tree as the pointer between 23 and 71 points to
2. by searching in the root of this sub-tree you will find that the number must be in the sub-tree to the right of 43, but it is empty, and thus 47 are not found

The tree thus has in many ways the same characteristics as a binary search tree.

At worst, you have go down to the lower level to determine if an element exists or does not exist (just in the same as in a binary search tree). Suppose the tree has t elements, that all nodes are filled out and that the tree is "nicely" balanced. If you start searching in the root, it corresponds to selecting a sub-tree of t/n elements. Repeating the search here, you

select next time a sub-tree with t/n^2 elements. After repeating the operation k times, you have a sub-tree with t/n^k elements. In the worst case, you have finished the search when

$$\frac{t}{n^k} = 1 \Leftrightarrow t = n^k \Leftrightarrow log_n(t) = k$$

That is in a "nicely" balanced search tree of the order n is the depth $log_n(t)$ where t is the number of elements, and you can therefore at worst find an element by searching in $log_n(t)$ nodes. Note that you can use binary search in the individual nodes. Suppose that you have a search tree of the order 100 and there are 10000000 elements. In the worst case, you have to search in

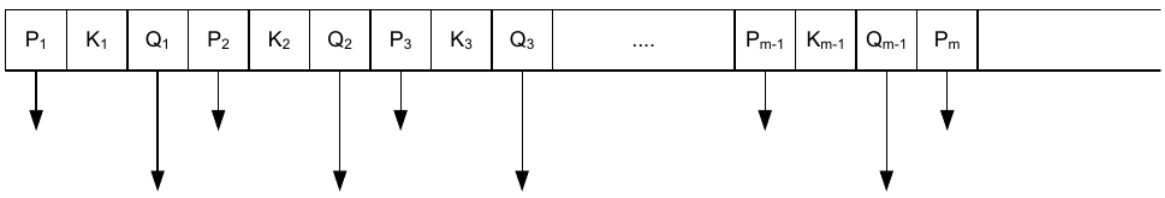
$$log_{100}(10000000) = \frac{log_{10}(10^7)}{log_{10}(100)} = \frac{7}{2} \sim 4$$

nodes. The example tells us that a balanced search tree has a very small depth, and that is the exact goal, as search trees are usually used for data stored on disk. One then loads one disk block at a time, and as reading a disk block takes a long time compared to corresponding memory operations, it is important to have few disk operations – although afterwards you have to search the individual disk blocks, but it happens in memory.

It is possible to implement a search tree, as described above, but it does not have the great interest, as its efficiency is completely dependent on where the tree is balanced, and since it should also be such that the individual nodes are not nearly empty. I would therefore like to describe a variant that accommodates these two conditions. It is thus a data structure that implements a generally tree that is reasonably balanced and ensures a good search complexity, but as mentioned, it is a data structure that primarily has interest for objects that are permanently stored on disk as the time to search the individual nodes can be significant.

7.5 B TREES

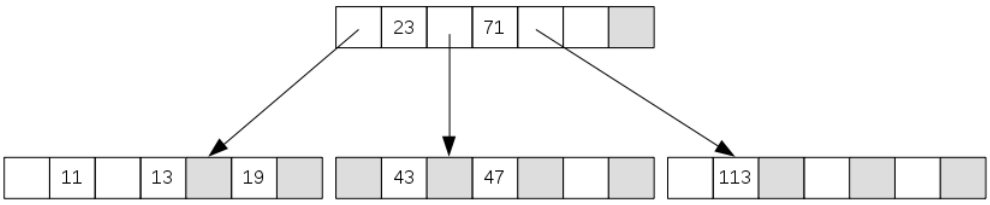
A *B tree* of the order n is a search tree of the order n, but where the data elements instead are a pair of the form [*K, Q*], where *K* is the key and *Q* is a pointer to the place where the data of the key *K* is stored. *Q* could for example be (and in practice will be) the address of a disk block. Similarly, a node in a B tree can be illustrated as follows:



where the *P* pointers are as in the ordinary search tree, while the *Q* pointers point to data blocks. In addition, the tree must meet:

1. All nodes except the root and leaf nodes have at least *[n/2] P* pointers (that is at least *[n/2]* sub-trees).
2. The root has at least two *P* pointers unless the root is the only node in the tree.
3. All leaf nodes are on the same level, and a leaf node is characterized by that all *P* pointer being zero.

For example, a B tree of the order 4 is shown below (note that I have not shown the *Q* pointers):



These requirements ensures that the tree is reasonably balanced and the challenges are to implement insertion and deletion of elements in such a way that the above requirements are always respected.

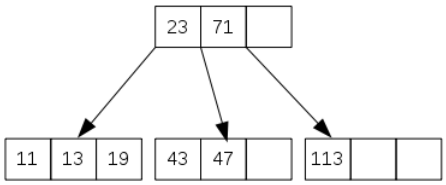
Inserting an element in B tree can be described informally as follows:

A B tree starts with a single root node, and elements are insert here as long as space is available. When the root is filled up and you add the next element, the root is split into two node at level 1. The middle element will stay in the root while the remaining elements are split between the two new nodes. If an element is inserted into a filled node that is not the root, it is split into two nodes at the same level and the middle element is moved to the parent node. If it then is filled, the process is repeated here, and it continues and eventually ends with the creation of a new root.

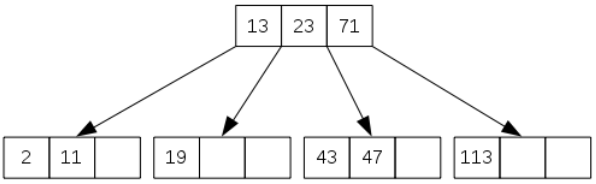
Below is the result of inserting the following elements into the above tree:

2, 3, 5, 7, 41, 53, 29, 31, 37, 63, 59

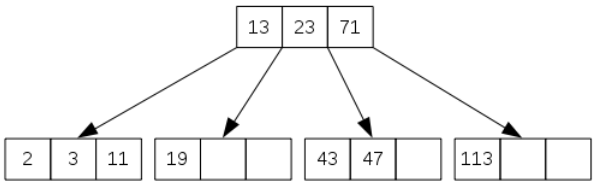
when the elements are inserted in the order shown. The above tree has the order 4, and when you must split a node, there are 4 elements where the middle must be moved to the parent node, while the others are to be distributed on two child nodes. For technical reasons, I do not want to show room for the pointers in the individual nodes, and the starting point is thus the following tree:



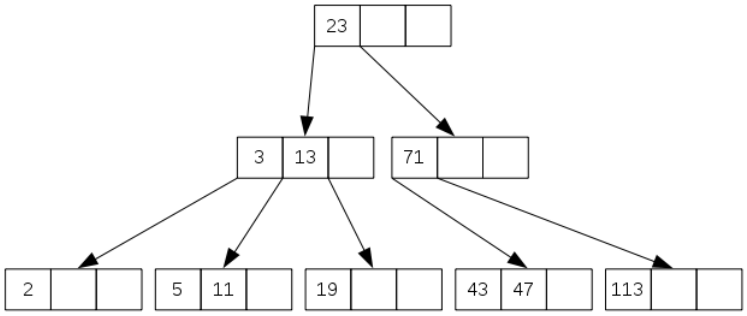
Below is what happens to the tree as the 11 numbers are inserted. Here after 2 has been inserted:



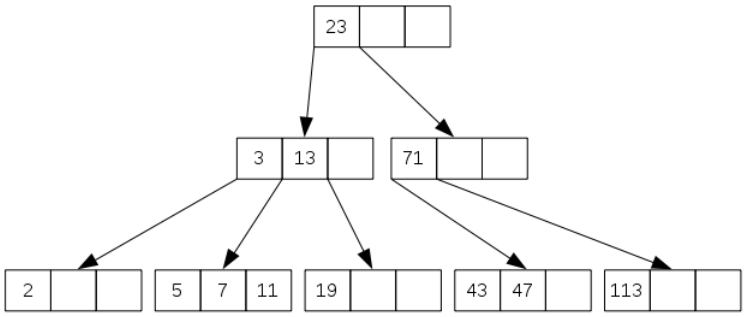
After 3 is inserted:



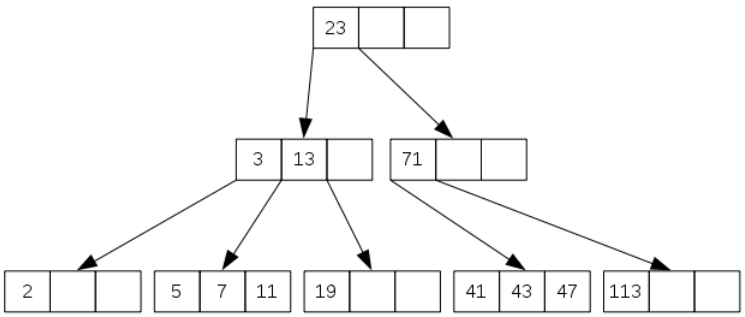
After 5 is inserted:



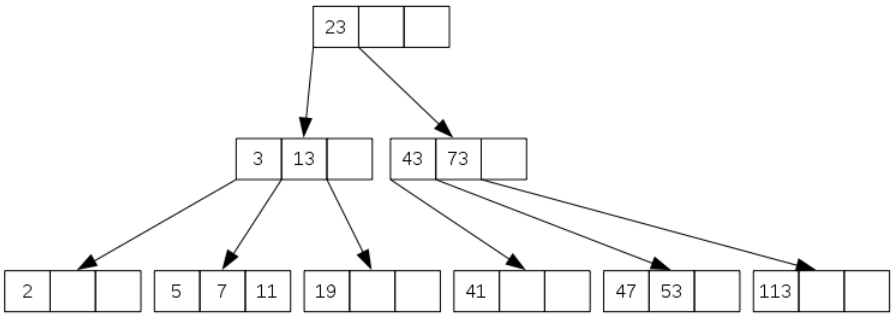
After 7 is inserted:



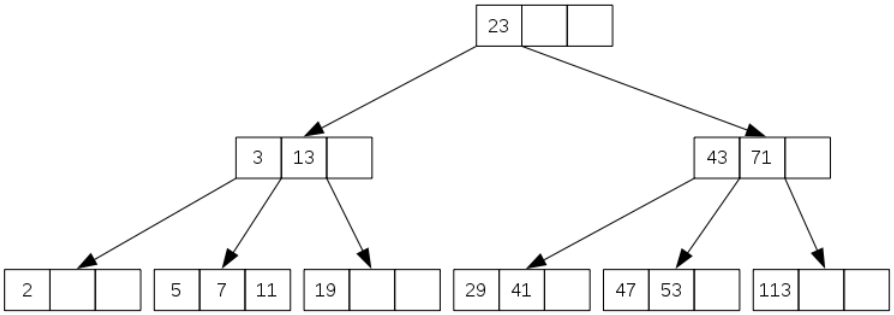
After 42 is inserted:



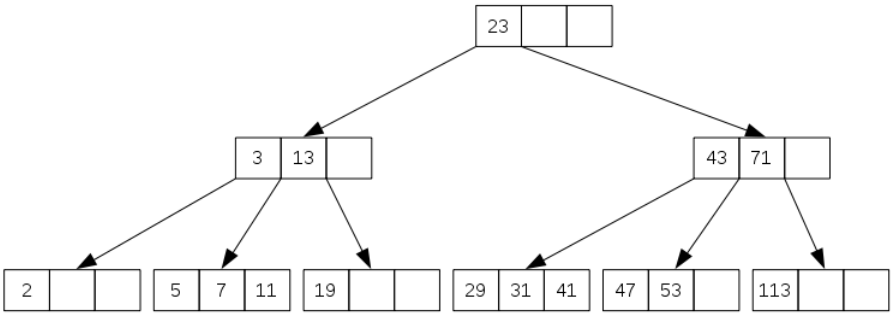
After 53 is inserted:



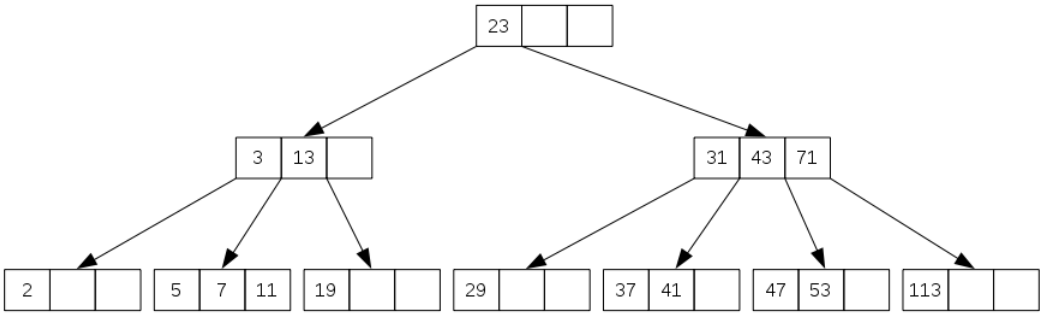
After 29 is inserted:



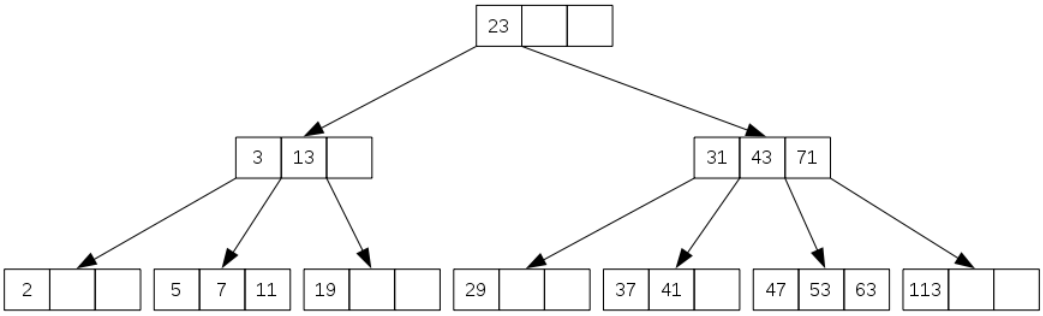
After 31 is inserted:



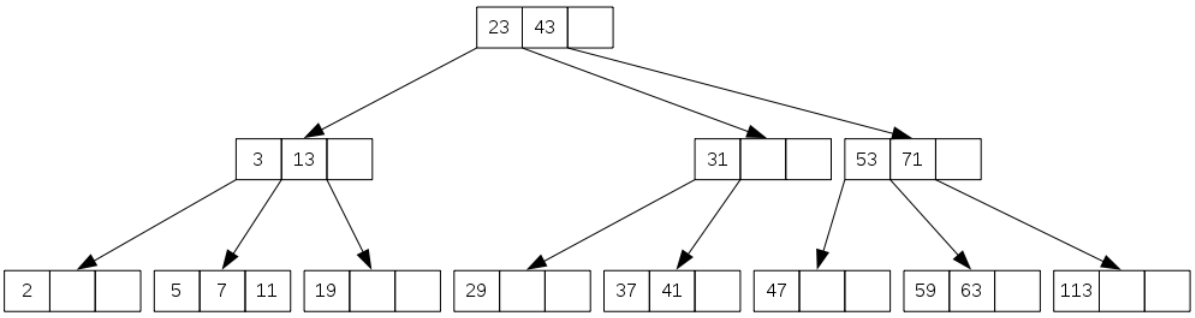
After 37 is inserted:



After 63 is inserted:



After 59 is inserted:



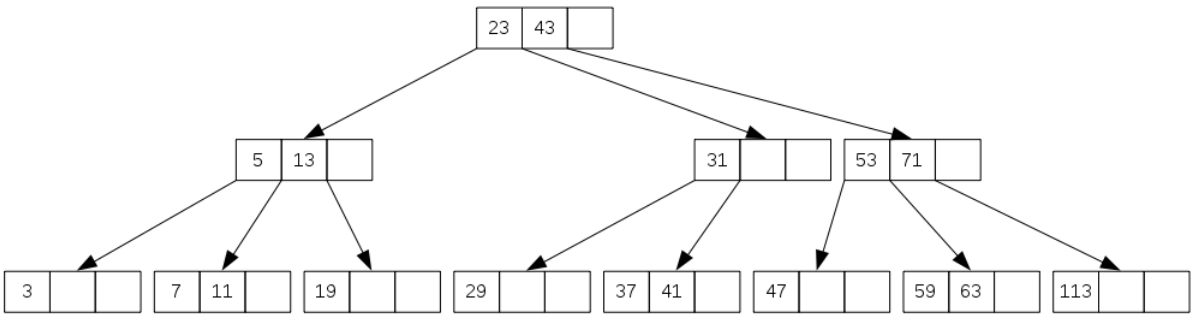
In this case, the number of elements in a full node is odd. If the number is even, it is necessary to choose a strategy for which element to move up one level – there are two options. It does not matter if you choose one or the other.

When you delete in a B tree, a node may become less than half full, and then a node must be combined with a neighbor. That is you must merge two nodes and it is an operation that can propagate up through the tree and can result in a tree with a smaller depth. In fact, it is not easy to implement deletion. The strategy is roughly as follows.

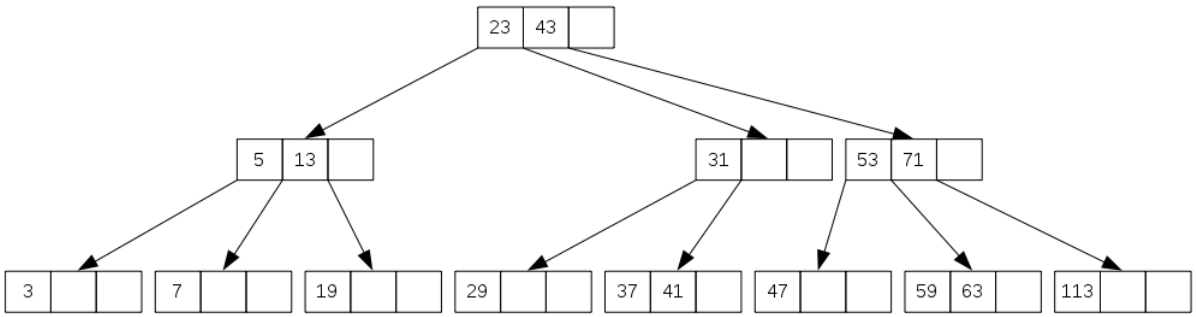
You start by finding the node that contains the element to be deleted. Next, as in the 2-3-4 tree, the node that contains the element's successor, which is a leaf node, should be found. The element to be deleted is then replaced with its successor, and the successor is deleted in the leaf node. After that it may happen that the leaf node is no longer half full, and the result is no longer a B tree, and you must now adjust the tree. If the node has a direct sibling, which is more than half full, the problem can be solved with a rotation, moving an element from this sibling to the parent, while moving an element from parent to the leaf where an element was deleted. If this is not the case, the leaf node where an element is deleted is merged with a neighbor sibling and an element from parent.

Below are some examples where elements are deleted in the above tree, but since a node can only have 4 sub-trees, a node is only half full when it becomes empty so rotations and merging will only be necessary when a leaf is empty.

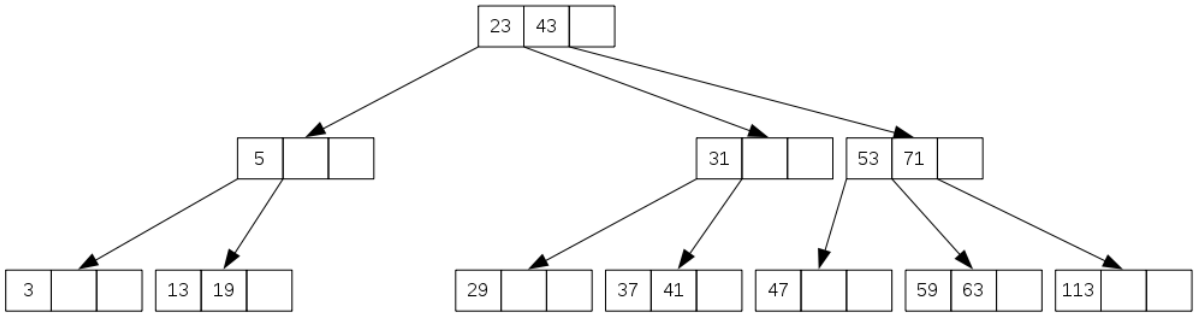
After 2 is deleted. A left rotation has been performed:



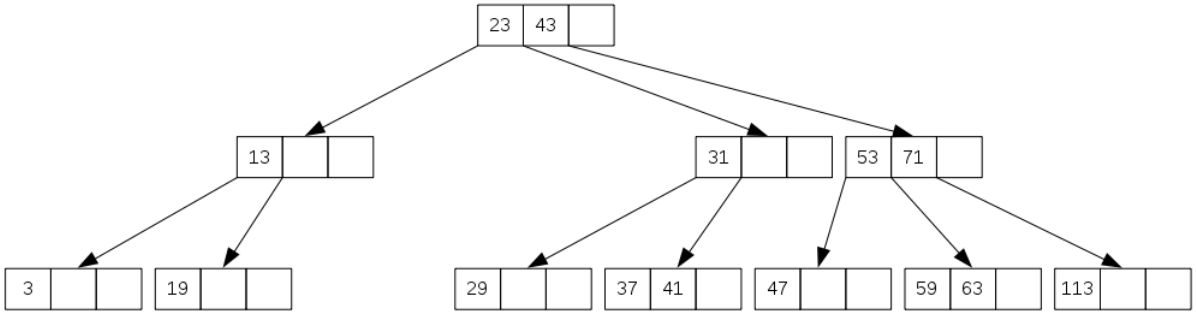
After 11 is deleted. The node containing 11 will not be empty, so no further operations need to be done:



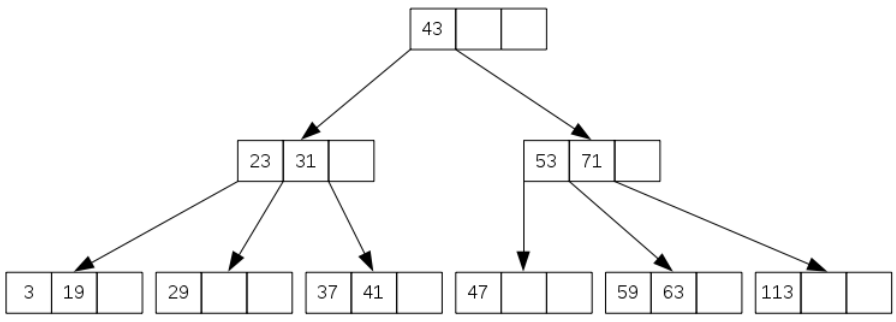
After 7 is deleted. The result is an empty node and a merge with the right sibling and an element from parent is performed:



After 5 is deleted. There is only a replacement with the successor, that immediately can be deleted without further adjustments of the tree:



After 13 is deleted. 13 is replaced with 19, resulting in an empty node. A merge with the left sibling and element 19 from parent is performed. Then parent is empty and a merge with the right sibling and the element 23 from parent is performed:



A B tree is implemented slightly in the same way as a binary tree, and the biggest difference is that a node is defined differently, as it should accommodate multiple elements and be able to refer to multiple sub-trees. The class *BTree* implements a B tree with typical operations, but it is more for the example than it is for utility. The work consists primarily of implementing *insert()* and *remove()* to ensure that the tree is still a B tree after the operations has been

completed. The class organizes memory objects in a B tree, meaning that with a large order (many elements in a node), the tree gets a low depth and thus the search path to a leaf node becomes short, but it is offset by the fact that you must search in the individual nodes. In this case, the nodes are searched linearly, and as a node is ordered, you could possibly improve the tree by instead searching binary. Especially with a tree of a large order one should consider to implements binary search in the individual nodes.

For the reasons mentioned above, a B tree has not the great interest in organizing memory objects, and the goal of the B tree is also to organize keys and references to disk blocks in memory, thus enabling efficient search in a file and only load the disk blocks when they are required.

EXERCISE 6: TEST BTREE

You must write a program that you can call *BTreeTester*, which will test the class *BTree*. The program must have a method *test1()* that creates a *BTree* of the order 6 for *Integer* objects. You must then print the tree's order, the number of elements and the height of the tree. The method must then insert the numbers from 10 to 99 in the tree, but in random order. Next, the method must print the same information as above, the smallest and the largest element, the elements of the tree on one line using an iterator, and finally the tree itself using the class's *toString()* method.

You must then write a method *test2()*, which creates a *BTree* for *Character* objects. Next, the method must insert the 26 letters A–Z in the tree when they must be inserted in random order. The method must then perform a loop, which for each iteration prints the same information as *test1()* and then deleting a random element in the tree. This iteration must be repeated until the tree is empty.

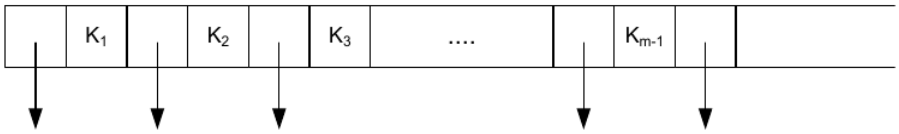
The class *RBTreeTester* (exercise 5) has a method *test3()*. Copy this method to the current class *BTreeTester*. The method compares the performance of three trees (a *RBTree<Integer>*, a *BinSearchTree<Integer>* and a *java.util.TreeSet<Integer>*). Expand the method with another fourth tree, but this time of the type *BTree<Integer>* to perform the same operations on the tree. Try experimenting a little with the tree order (the tree faneout) and whether it is important for the performance.

7.6 B+ TREES

A B tree is characterized by a big faneout – or that's what it's all about: Big faneout and low depth = few disc operations. For all leaf nodes, it means that they contain many pointers for sub-trees, which are always *null*, and to avoid that, you usually implement a variant of the B tree, which is called a B+ tree. A B+ tree is a B tree with two kinds of nodes:

1. inner nodes
2. leaf nodes

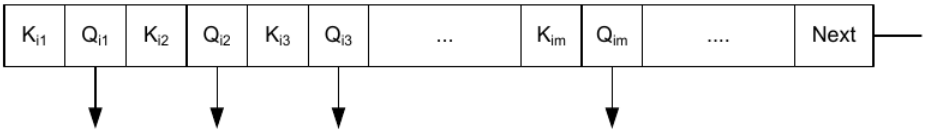
An inner node has the same format as a node in a general search tree:



but there are no data pointers, but only search values and pointers to the sub-trees. In general, an inner node must meet the same criteria as for a B tree, but regarding sub-trees the following requirements are stated:

1. if *P* is the first pointer, all values *X* in the sub-tree that *P* point to must fulfill $X < K_1$
2. if *P* is the last pointer, all values *X* in the sub-tree that *P* point to must fulfill $X \geq K_{m-1}$
3. if P is the *i*th pointer where $1 < i < m$, all values *X* in the sub tree that *P* point to must fulfill $K_{i-1} \leq X < K_i$. That is that a search value *K* is the largest search value that occurs in the sub-tree that the previous pointers point to.

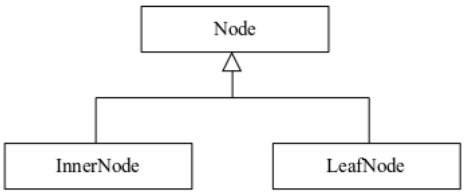
A leaf node has the form



where the *Q* pointers are data pointers for the object (data block) that has the key *K*. That is, a leaf node consists of pairs *[K, Q]*, which are arranged according to the key field.

Next there is a pointer to the next leaf node, thus giving a simple possibility of traversing all the tree's elements sorted according to the key.

The class *BTree2* implements a B+ tree, but its implementation is different than the class *BTree*, as this time two types of nodes are defined as inner classes:



Node is an abstract base class that contains the keys:

```
private abstract class Node
{
 IList<K> keys;
```

while the two concrete nodes are defined as follows:

```
private class InnerNode extends Node
{
 IList<Node> children;

private class LeafNode extends Node
{
 IList<V> values;
 LeafNode next = null;
```

The class itself is defined as:

```
public class BTree2<K extends Comparable<K>, V> implements Iterable<V>
{
 private Node root = null;    // the root of the tree
 private int count = 0;       // the number of elements in the tree
 private int order = 0;       // the order of the tree
```

The code consists primarily of programming the two node classes, but to facilitate the work, the class *Tools* is expanded with a new binary search method:

```
public static <T extends Comparable<T>> int binSearch(IList<T> list,
T elem)
{
 int a = 0, b = list.getSize() - 1;
 while (a <= b)
 {
  int m = (a + b) / 2;
  if (list.get(m).equals(elem)) return m;
  if (list.get(m).compareTo(elem) < 0) a = m + 1; else b = m - 1;
 }
 return -(a + 1);
}
```

The method searches in a list instead of an array, but the important thing is the return value if the element is not found, which is the index that the element should have been but with negative sign. Java's Collections classes implement binary search in the same way. The class *BTree2* uses binary search in the individual nodes, and the above implementation makes it easier to search down the tree.

Also the class *ArrayList* (and the interface *IList*) has been changed so that the class is expanded with a new method that deletes all elements within an interval:

```
public void clear(int a, int b)
{
 if (b > a)
 {
  for (int i = a, j = b; j < count; ++i, ++j) elements[i] = elements[j];
  count -= b - a;
 }
}
```

Java's *ArrayList* has a similar method and the purpose of implementing the method is performance.

EXERCISE 7: TEST BTREE2

Write an application that you can call *BTree2Tester*, which can be used to test a *BTree2* tree:

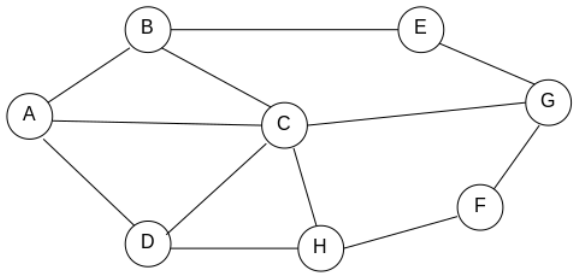
```
public class BTree2Tester
{
 private static java.util.Random rand = new java.util.Random();
```

```
public static void main(String[] args)
{
}

private static void test1()
{
 // Create a IList<Integer> that contains all
 even numbers in a interval [A; B].
 // Create a BTree2<Integer, String> with fanout 6.
 // Insert an element for each key in the
 above list, when the keys must be
 // inserted in random order, and the value should be a random String.
 // Print the tree.
 // Search 10 keys in the tree, and print for
 each key where it is found or not.
 // Get a sub range of values – the method
 getValues() – and print the values.
}

private static void test2()
{
 // Create two lists list1 and list2 of the
 type IList<Integer>, and initializes
 // the lists with the keys from 100 to 200.
 // Create a BTree2<Integer, String> with a fanout that is 4.
 // Add for each key in list1 an element to
 the tree, when the elements must be
 // added in random order of the keys.
 // Print the tree.
 // Remove elements random, until the tree contains 20 elements.
 // Print the tree.
}

private static void test3()
{
 // Create two lists list1 and list2 of the type IList<Integer>, and initializes
 // the lists with the keys from 100 to 1000.
 // Create a BTree2<Integer, String> with a fanout that is 10.
 // Add for each key in list1 an element to
 the tree, when the elements must be
 // added in random order of the keys.
 // Print the tree.
 // Remove elements random, until the tree is empty, and print the keys,
 // as they are removed, on a line.
 // Print the tree.
}
```

```
private static void test4()
{
 // Creates a BTree2<Integer, String> with a fanout on 256.
 // Insert N elements in the tree, which has random keys les than 2N.
 // Print
 // the number of elements in the tree
 // the number of milliseconds used to insert the elements
 // the height of the tree
 // Search the tre for M random key values.
 // Print
 // the number of milliseconds used for the M searches
 // the number of matches
}

private static void print1(BTree2<Integer, String> tree)
{
 // Print method for test1() and test3() that prints
 // the tre, that is the value of toString()
 // the number of elements
 // the height
}

private static void print2(BTree2<Integer, String> tree)
{
 // Print method for test2() that prints
 // the tre, that is the value of toString()
 // the number of elements
 // the height
 // the values in the tree by using the iterator
}

private static String createString(int a, int b)
{
 // Returns a sting with random characters between A and Z.
 // The length of the string should have a random value between a and b.
}
```

8 GRAPHS

A graph is a collection of objects, such that there between two objects can be a relationship. For example, a graph can be illustrated as follows:



where there are 8 objects, as indicated by a letter. The lines illustrates relationships, and there is, for example, a relationship between C and G. In a graph, all objects play the same role, and thus there are no objects that are root or start objects. A little with inspiration in the above figure, you call the objects for vertices, while the relationships are called edges,

and you define a graph as a data structure *G* consisting of two sets *V* (for vertices) and *E* (for edges):

G = (V, E)

The graph shown in the above figure is thus defined by the following sets

V = {A, B, C, D, E, F, G, H}

E = {{A, B}, {A, C}, {A, D}, {C, D}, {C, G}, {C, H}, {D, H}, {H, F}, {F, G}, {B,C}, {B, E}, {E, G}}

In conjunction with a graph, you also define the concept of a *road* or *path* between two vertices as a sequence of edges that lead from one vertex to another. In the graph above are

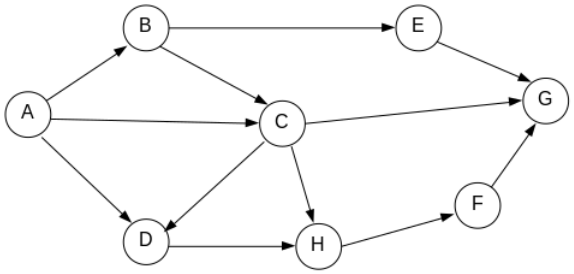
(A, C, G)

(A, B, C, H, F, G)

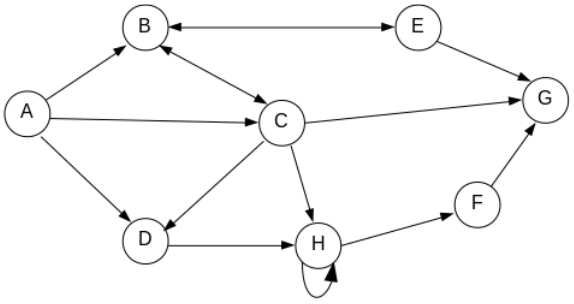
both paths from *A* to *G*. A bit more formally, a path in a graph *G* is a sequence of vertices $(V_1, V_2, \ldots, V_n)$ with the property that for all i = 1, 2, …, n-1, there is an edge (V_i, V_{i+1}). The same vertex may appear several times, but if all vertices are different, one talks about a simple path.

A graph like the above is called a non-oriented graph, in which an edge is interpreted as a double edge corresponding to the relationship is going both ways. That is there is both an edge from *A* to *B* and an edge from *B* to *A*. You can also have an orientated graph where the edges are not directional:

E = { (A,B), (A,C), (A ,D), (C ,D), (C,G), (C,H), (D,H), (H,F), (F,G), (B,C), (B,E), (E,G)}



Although if a graph is oriented, there must be edges in both directions – and if so, you usually illustrates it with a double arrow, and there must also be an edge from a vertex to itself:



In an non-oriented graph you can always go in both directions, and you can therefore also characterize a non-oriented graph as an oriented graph where each edge is a double arrow – there is an edge in both directions.

In an oriented graph one calls a path between two vertices for a cycle if the two vertices are the same, that is, if the path starts in a vertex and ends in the same vertex. With the above example is

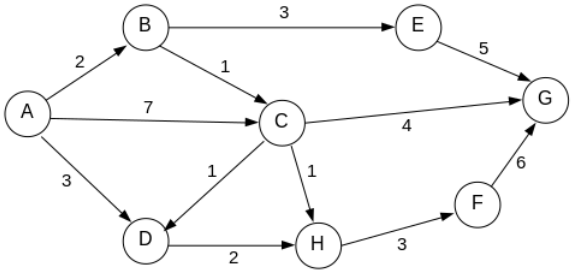
(C, B, C)

a cycle. An oriented graph without cycles is called an oriented acyclic graph.

Since a vertex can have an edge to itself, for all vertices there can be an edge to any other vertex (including the vertex itself). If the graph has n vertices, it may then highest have n^2 edges.

A graph that has many edges is called a *dense* graph, while a graph with few edges is called a *sparse* (thin) graph. There is no exact measure of when a graph is dense or sparse, and it is solely a question that a graph may be more or less sparse. It, on the other hand, has an impact on how to implement a graph.

The above graphs are all non-weighted graphs, but we can also have weighted graphs, which means that for each edge we have a weight in the form of a number:



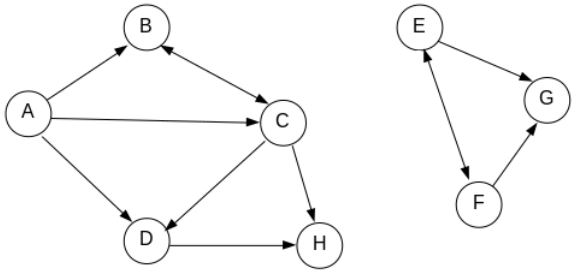
In practice, weights could represent distances or costs.

A graph is a data structure, and as explained above, the graph can be defined as two sets *G = {V, E}* of respectively vertices and edges. Of course, operations are also associated with the data structure, and for a given implementation there must be operations that build the graph and remove edges and vertices. In addition, there are the following operations:

- Traverses the graph where you start at a vertex and from there go to all the other vertices that are possible to reach by following the graph's edges.
- The shortest way, that is to find the shortest path from a vertex to all other vertices. What shortest means is interpreted differently depending on whether the graph is weighted or not.

- The least spanning tree that deals with (if possible) removing so many edges that the result is a tree.
- A topological sorting, which arrange the vertices of the graph in a certain order.

As a last remark regarding graphs, the above drawings illustrate coherent graphs, but it is not a requirement, and the figures below also illustrates a graph:



However, in most cases, you will be interested in coherent graphs (as a non-coherent graph just is a family of coherent graphs), and I will, in the following – unless otherwise stated – assume that a graph is coherent.

A graph is also called a network, and many applications relate to, for example, computer networks or networks of geographical locations.

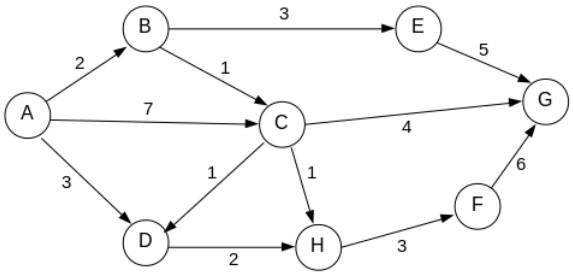
Remark

Graph theory is actually a large area with many complex algorithms, and the following is by no means exhaustive and is only an introduction to the subject. The graph theory is extensive and is based on stringent mathematical profs both for the correctness of the algorithms and their time complexities. The following approach is practical oriented and the primarily aims are to implement a graph as a data structure in Java, and the different algorithms are illustrated without precise profs, and the same applies to their effectiveness.

8.1 IMPLEMENTING GRAPHS

In general, it is not easy to implement a graph as a data structure, especially because it is difficult to implement the graph's operations so that they are general and with a reasonable time complexity. The implementation of the classic graph operations will often depend on the internal representation of a graph and thus becomes closely linked to the data presentation.

One way is to implement a graph is using a 2-dimensional array. Consider again the graph:



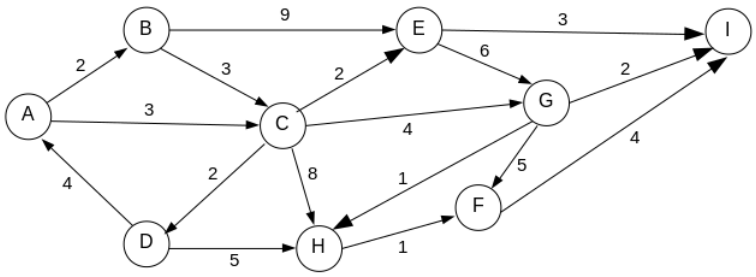
If you use a 2-dimensional array, which indexes are the vertices of the graph and you in elements in the array, writes the weights of the edges, you can represent the graph as follows:

	A	B	C	D	E	F	G	H
A		2	7	3				
B			1		3			
C				1			4	1
D								2
E							5	
F							6	
G								
H						3		

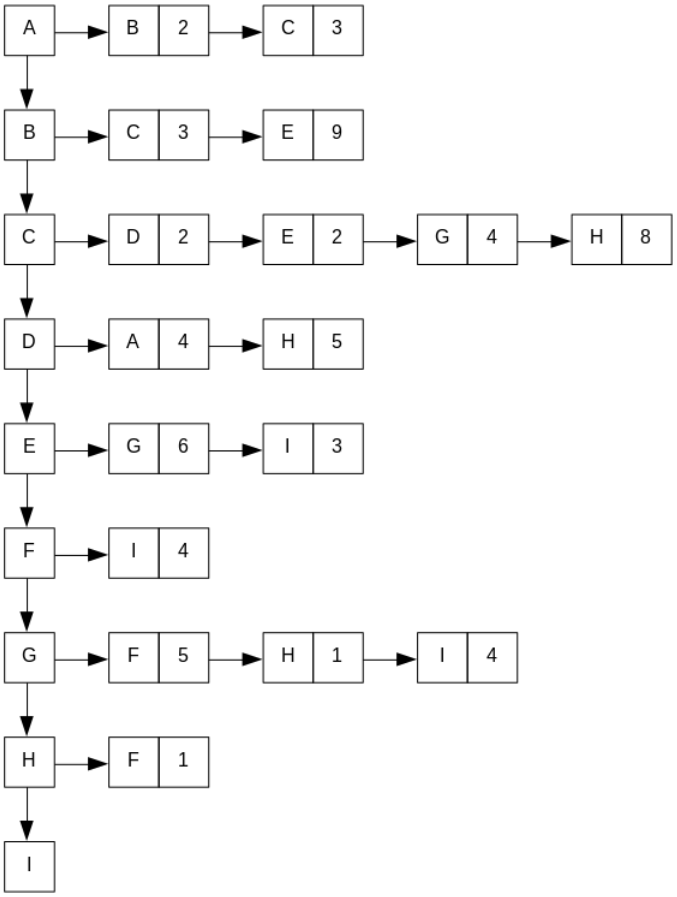
The empty cells indicates that there is no edge between the two vertices. For example, there is an edge from *A* to *B* with weight 2, but there is no edge from *B* to *A*.

Implementing a graph in this way is somehow natural, but the way has some major disadvantages. First, the way is best if the graph is dense (has many edges), as the matrix else will get many empty elements and thus a large amount of unused space, but the biggest problem is that it is difficult to implement graph algorithms effectively as many of the algorithms result in searches, and as another problem, it is difficult to expand the graph with new vertices, and even worse to delete vertices. For these reasons and because most graphs are sparse in practice, usually another implementation is used, where the graph is implemented as a list where each node consists of the vertex as well as a list of edges from that vertex.

For example, I will look at the following graph:



which is a weighted graph with 9 vertices and 17 edges. It is therefore a relatively thin graph with few edges (17 out of 81 possible). An implementation of the graph can be illustrated as follows:



where you vertical to the left has a collection with vertices, and each vertex has a list of direct successors, that is, a list of the edges that originate from that vertex. For each element in this list, the weight of the edge is also attached.

For example, as a collection for the vertices, you can use a hash table. This ensures that it is easy to add new vertices, but most importantly, you can find a vertex with constant time complexity. With regard to a collection for edges in each vertex, one can also use a hash table, but in most practical applications, the number of edges will be few, so you often select a list or perhaps a linked list instead. The search in the list for edges will probably be linear, but if the graph is sparse (and the lists are short), the linear time is outweighed by the overhead, by using a more complex collection like a hash set.

With these considerations in place, I will address writing a class that can implement a graph. The first problem is to define which operations the class should have, and thus what you can do with a graph. As a step along the way, I have defined the following interface:

```
package dk.data.torus.collections;

import dk.data.torus.PaException;
import java.util.Iterator;

public interface IGraph<T> extends Iterable<T>
{
 /**
   * @return The type of the graph.
   */
 public GraphType getType();

 /**
   * @return The number of vertices in the graph
   */
 public int getSize();

 /**
   * @return The number of edges in the graph
   */
 public int getCount();

 /**
  * Return the weight of the edge [from, to] or
    NaN if the edge has no weight.
  * @param from Start vertex
  * @param to End vertex
  * @return The weight of the edge [from, to]
  * @throws PaException If the edge does not exists
  */
 public double getWeight(T from, T to) throws PaException;

 /**
  * Change the weight of the edge [from, to] or assign a new weight if the
  * edge has no weight.
  * @param weight The new weight
  * @param from Start vertex
  * @param to End vertex
  * @throws PaException If the edge does not exists
  */
 public void setWeight(double weight, T from, T to) throws PaException;

 /**
  * Add a new vertex to the graph.
  * @param elem The data element for the new vertex
  * @throws PaException If the graph already has
    a vertex with data element elem
  */
 public void add(T elem) throws PaException;
```

```
/**
 * Add a new edge [from, to] to tre graph. If the graph already has an
 * edge [from, to] the operation is ignored. If the graph does not have a
 * vertex to, a vertex is added to the graph.
 * @param from Start vertex
 * @param to End vertex
 * @return True if a new to vertex is added to the graph
 * @throws PaException If the graph does not have e vertex for from
 */
public boolean add(T from, T to) throws PaException;

/**
 * Add a new edge [from, to] to tre graph. If the graph already has an
 * edge [from, to] the operation just update the weight. If the graph does
 * not have a vertex to, a vertex is added to the graph.
 * @param from Start vertex
 * @param to End vertex
 * @param weight The weight for the edge.
 * @return True if a new to vertex is added to the graph
 * @throws PaException If the graph does not have e vertex for from
 */
public boolean add(T from, T to, double weight) throws PaException;
```

```
/**
 * Check if the graph contains a vertex with the value elem.
 * @param elem The element to be searched
 * @return True if the graph contains a vertex with the value elem
 */
public boolean contains(T elem);

/**
 * Check if the graph contains an edge [from, to]
 * @param from The start vertex
 * @param to Then end vertex
 * @return True if the graph contains an edge [from, to]
 */
public boolean contains(T from, T to);

/**
 * Remove all edges and vertices in the graph.
 */
public void clear();

/**
 * Remove the vertex with the value elem.
 * @param elem The element to be searched
 * @return True if the vertex is found and removed
 */
public boolean remove(T elem);

/**
 * Remove the edge [rom, to].
 * @param from The start vertex
 * @param to The end vertex
 * @return True if the edge is found and removed
 */
public boolean remove(T from, T to);

/**
 * Return a list with all successors to the vertex with value elem.
 * @param elem Value of the vertex to be searched
 * @return A list with all successor vertices
 */
public IList<Vertex<T>> getSuccessors(T elem);

/**
 * Return a list with all predecessors to the vertex with value elem.
 * @param elem Value of the vertex to be searched
 * @return A list with all predecessor vertices
 */
public IList<Vertex<T>> getPredecessors(T elem);
```

```
 /**
  * Returns an iterator that iterates all vertices in the graph.
  * @return An iterator that iterates all vertices in the graph
  */
 public Iterator<Vertex<T>> getVertices();
}
```

where a single helper type is defined:

```
package dk.data.torus.collections;

public enum GraphType { Directed, Undirected }
```

which indicates whether a graph is oriented or not. Finally, the following types are used, which represent the vertices and edges of the graph:

```
public class Vertex<T>
{
 private final T element;                  // the data element
 private final IHash<Edge<T>> edges;       // set of edges
 private double distance = Double.POSITIVE_INFINITY;
 private Vertex<T> prev = null;
 private boolean used = false;
 private int count = 0;

class Edge<T>
{
 private Vertex<T> to;
 private double weight = Double.NaN;
```

Here you should note that a *Vertex* object exactly matches an element in the above figure with a collection to the vertices, and that an *Edge* object exactly matches an element in a collection of edges. That is, the two classes *Vertex* and *Edge* defines the types that correspond to the above illustration of a graph. In relation to the above figure, the class *Vertex* has a further 4 properties. In relation to the implementation of a graph, these 4 properties have no use, but they are used as auxiliary variables of the individual graph algorithms. Basically, these variables should not be attributes to the class *Vertex*, and instead you could apply a specialization or wrapper class, but the solution allowing the variables to be attributes to the *Vertex* class simplify and streamline the implementation of the graph algorithms, so this choice.

The class *Graph<T>* implements the interface *IGraph<T>* and thus represents a graph. You should note that the class alone supports building the graph with vertices and edges, but no graph operations – which I have not mentioned yet. There is some code, and you should primarily note the following.

First, there is the data presentation where I have used a map for vertices and with the generic type *T* as a key. It requires that the type *T* should override both *equals()* and *hashCode()* with value semantics. Then note that a property has been defined for the graph type (oriented or non-oriented) and that the type is initialized in the constructor, which by default sets the type to be oriented. Assuming that a vertex has few edges (whatever it may be), for most operations, one must expect a time complexity that with approximation is constant. However, if the graph is dense (so that from every vertex there is an edge to all (many) other vertices), several of the methods will have linear complexity. It is difficult to estimate the complexities as they depend on the density of the graph, but since graphs in practice have more edges than vertices, and since these edges often are distributed evenly across the vertices, this complexity is often seen as *log (N)* where *N* is the number of vertices.

One of the most important methods is the method *add()* that adds an edge. It starts by testing if there is already an edge between the two vertices, and if so, the operation is ignored (for a non-weighted graph) or the weight is updated. If the edge is not already there, find the corresponding vertex by "pick the corner up" into the map and if it is not there, it will be created. That is, the method *add()* will add new vertices if a vertex is not already in the graph. With the two vertices you can then add an edge to the one. If the graph is not oriented, then another edge will be added automatically. It's a simple (and

also a little expensive) solution that you can discuss and there are other and better ways, but I perceive non-oriented graphs as the exception and have therefore allowed to get rid of it by just adding yet an edge.

Also note the implementation of the iterator, which consists simply in returning the map's iterator. This means that you can not assume anything about the order of iteration over the vertices. The implementation of the methods *remove()* is not quite simple, and it is hard to remove a vertex. Should you remove a vertex, you must also delete all edges that this vertex is part of, and here it is especially the edges that have the vertex as a successor, that causing problems as you can not find them without searching all the edges, that is, run over all vertices and run over all the edges of each vertex.

Then there is the method *getSuccessors()* that will return all successors for a particular vertex. It's a simple method, as a vertex contains a list of its successors. On the other hand, the opposite method *getPredecessors()*, which will return all predecessors to a vertex is not quite as simple. Here you must iterate over all vertices and for every vertex check if it has an edge to the appropriate vertex. If you again assume that the graph is sparse, you'll expect that *getSuccessors()* has constant complexity (you can only examine the edges of the vertex, which can be set to constant on time), while *getPredecessors()* has linear complexity, since you have to run over all vertices. Instead, speaking of a dense graph, the two complexities will be *O(N)* and $O(N^2)$.

Finally, note the method *getVertices()*, which returns an iterator that iterates over all vertices and where the iterator returns *Vertex* objects. It is important for the implementations of the graph algorithms.

EXERCISE 8: BUILD A GRAPH

Write a program that you can call *GraphTester1*. Add a method

```
private static IGraph<Character> build1()
{
```

that creates and returns the graph illustrated above. That is, a weighted oriented graph with 9 vertices and 17 edges.

The program must also have the following methods:

```
private static void test(IGraph<Character> graph)
{
 // print the graph (the method print())
 // remove the edge from G to H
 // remove the edge from F to I
 // print the graph
 // remove the vertex C
 // print the graph
}

private static void print(IGraph<Character> graph)
{
 // print number of vertices
 // print number of edges
 // print the graph (the value og toString())
 // print the vertices on a line using the graph iterator
}
```

Perform the test method from *main()*. Also write a method:

```
private static IGraph<Character> build2()
{
```

which creates the same graph, but so it should be a non-weighted and non-oriented graph. Also, perform the test method on this graph.

PROBLEM 2: A DENSE GRAPH

In this task you must write a graph class, which you can call *ArrayGraph*. Start with a new project, which you can call *GraphTester2* and add a class *ArrayGraph<T>*. The class should implement the interface *IGraph<T>*, but the class must now be implemented using a 2-dimensional array as described in the introduction to this chapter. Instead of directly creating a 2-dimensional array, you can make the work easier by defining the data structure as follows:

```
public class ArrayGraph<T> implements IGraph<T>
{
 private IMap<T, Integer> map = new HashMap();
 private IList<IList<Double>> weight = new ArrayList();
 private GraphType type = GraphType.Directed;
```

where the first variable is a map that maps the vertices to indices in the array, while the next represents the 2-dimensional array. Note that it may be difficult to implement some of the methods in *IGraph<T>* with a good time complexity.

Once you have written the data structure, test it in the same way as in *GraphTester1*. Finally, expand with another test method:

```
private static void test2()
{
 int N = 1000;
 // Create a IGraph<Integer> called graph1 of the type ArrayGraph.
 // Create a IGraph<Integer> called graph2 of the type Graph.
 // Initialize both graphs with vertices from 1 to N both inclisive, but in
 // the same random order.
 // Try to add 20N random edges to both graphs, where the weight is a random
 // value between 0 and 1 – remark that the result
    could be less than 20N edges.
 // Print both graphs with the below print() method.
 // Remove N/10 random vertices from the graph.
 // Print both graphs with the below print() method.
```

```
 // Insert the above vertices in the graph again, and try to add 2N random
 // edges to both graphs.
 // Print both graphs with the below print() method.
}

private static void print(IGraph<Integer> graph1, IGraph<Integer> graph2)
{

 // For both graphs calculate the sum of all edges and measure how long
 // time it takes.
 // For both graphs print
 // the number of vertices
 // the number of edges
 // the sum of the weights
 // the time i micro seconds to calculate the sum
}
```

If all things seems to work properly, try experimenting with the value of *N*.

8.2 GRAPHTOOLS

If you look at the graph class, you can not really do much with the class, in addition to building a graph with vertices and edges, and you can iterate over the vertices. As a data structure or collection, the graph is not interesting by itself, but there are many algorithms that relate to graphs, and in combination with graphs, it often occurs in practice (roads between cities, routers in a computer network, airplanes between airports, etc.) that makes graph algorithms interesting. In the following I will give some examples of some classic algorithms, but there are many others. These algorithms are all implemented as static methods in the class *GraphTools*. As a parameter, all methods in the class *GraphTools* have a *Graph<T>* object as the graph that the methods will work on. The methods will know the internal structure of the class *Graph* through the method *getVertices()* and the two classes *Vertex<T>* and *Edge<T>* and are closely linked to this class.

As an alternative, instead, you could have implemented all graph algorithms as instance methods in the class *Graph* or perhaps even better in specialized classes. When I choose to do it as static methods in a special tool class, it is better to focus on the algorithms, but also because despite the above-mentioned strong coupling, yet some form of decoupling between the implementation of the graph and the implementation of the algorithms, but the price is an overhead in the form of copying the vertices.

8.3 TRAVERSING A GRAPH

As the above graph is implemented, it immediately supports traversing the graph and returns an iterator that can be used to iterate over all vertices. The problem with this iterator, however, is that it does not guarantee anything in which order you reach the individual vertices. Typically, you are interested in starting in a certain vertex and then moving forward along the edges of the graph according to one or another rule. Two ways are called respectively

1. dept first traversing
2. breadth first traversing

For both ways, you begin from a start vertex.

Depth first

At depth first, you will then go to the start vertex, and them followed by a successor to that vertex (if there is one that you have not already visited). This continues until you can not reach anymore vertices (you have reached as deep in the graph as you can). Then backtrack until you reach a vertex where you can go to a successor where you have not already been. Finally, you can not go any further either because you have visited all vertices or because there is a vertex for which there are no paths from the start vertex. Looking at the above graph, the result of a depth first traverse with start in A could be:

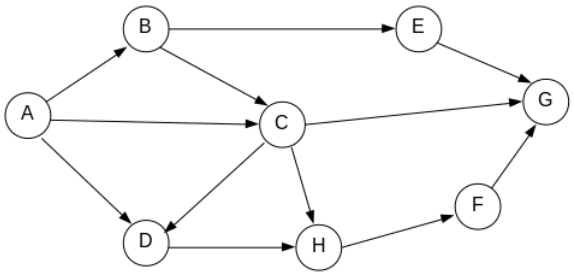
```
A B E I G F C D H
```

Note that there are other sequences, because the algorithm does not tell you what to do if there are more options, that is if you are in a vertex, and there are more successors that you have not visited.

The procedure can be described a little more precisely as follows:

```
Set C = the start vertex
Print C and mark it as handled
Push C on a stack
Repeat as long as the stack is not empty
 C = Peek the stack
 E = a none marked successor to C
 If E is null
  Pop the stack
 Eles
  Print E and mark it as handled
  Push E on the stack
```

where *Print* has to be interpreted as the operation that handles the vertex you have reached. Consider, for example, the following simple graph with 8 corners:



Below is an example of a depth first traversal and what happens to the stack when the traversing begins in A. When I have to choose a successor to a node, I have chosen each time the node that is not used and has a minimum index (value). Note that the selection of the smallest index is not part of the algorithm, and if the choice was different then the sequence will be another.

```
                  G
               F  F  F
            H  H  H  H  H
         D  D  D  D  D  D  D
      C  C  C  C  C  C  C  C  C     E
   B  B  B  B  B  B  B  B  B  B  B  B  B
A  A  A  A  A  A  A  A  A  A  A  A  A  A  A
```

To implement the algorithm in Java, two things are required: A stack and a way to mark the vertices that are processed. For the sake of the last, there is in the file for the class *GraphTools* defined a class that extends a vertex with multiple fields:

```
class Node<T> extends Vertex<T>
{
 private double distance = 0;
 private Node<T> prev = null;
 private boolean used = false;
 private int scratch = 0;

 public Node(T element, IList<Edge<T>> edges)
 {
  super(element, edges);
 }
```

In this case, only the field *used* is used, and is used to mark a vertex as visited. The other fields are used by other graph algorithms, which I will come to later. The class *GraphTools* has a method that encapsulates a graph's vertices in *Node* objects:

```
private static <T> IMap<T, Node<T>> getNodes(Graph<T> graph)
{
 IMap<T, Node<T>> nodes = new HashMap<T, Node<T>>();
 for (Iterator<Vertex<T>> itr = graph.getVertices(); itr.hasNext(); )
 {
  Vertex<T> v = itr.next();
  nodes.insert(v.getElement(), new Node(v.getElement(), v.getEdges()));
 }
 return nodes;
}
```

Here you should note that the method uses the iterator defined by *getVertices()* in the class *Graph*. It is a method with package visibility and can therefore be used in the class *GraphTools* to grab the vertices. The method is simple, but it is used in all of the following graph algorithms – and of course with a small performance loss, as a map is created.

All graph algorithms use a start vertex, and the class *GraphTools* therefore also implements a simple auxiliary method that finds a starting vertex. The method returns *null* if the vertex does not exist:

```
private static <T> Node<T> findNode(IMap<T, Node<T>> nodes, T element)
{
 return nodes.get(element);
}
```

Finally, there is a last helper method:

```
private static <T> Node<T> getSuccessor(IMap<T, Node<T>> nodes, Node<T> node)
{
 for (Edge<T> e : node.getEdges())
 {
  Node<T> to = nodes.get(e.getTo().getElement());
  if (!to.isUsed()) return to;
 }
 return null;
}
```

which returns a successor to a node if there is a successor that is not marked. If this is not the case, the method returns *null*. With all that available, the algorithm for depth first traverse can be written as follows:

```
public static <T> Iterator<T> dfs(IGraph<T> graph, T element)
{
 return new DfsIterator((Graph<T>)graph, element);
}
```

where

```
static class DfsIterator<T> implements Iterator<T>
{
 private IMap<T, Node<T>> nodes;
 private IStack<Node<T>> stack = new ArrayStack();

 public DfsIterator(Graph<T> graph, T element)
 {
  nodes = getNodes(graph);
  Node<T> node = findNode(nodes, element);
  if (node != null)
  {
   node.setUsed(true);
   stack.push(node);
  }
 }
```

```
@Override
public boolean hasNext()
{
 return !stack.isEmpty();
}

@Override
public T next()
{
 T element = null;
 try
 {
  Node<T> node = stack.peek();
  element = node.getElement();
  node.setUsed(true);
  do
  {
   Node<T> next = getSuccessor(nodes, node);
   if (next == null)
   {
    stack.pop();
    if (!stack.isEmpty()) node = stack.peek();
   }
```

```
     else
     {
      stack.push(next);
      break;
     }
    }
    while (!stack.isEmpty());
   }
   catch (Exception ex)
   {
    return null;
   }
   return element;
  }
}
```

is a static inner class. Compared to what has been said before, there is not much to explain. The constructor in the iterator class starts by creating a new map with corners, but this time expanded to nodes with additional information for the graph algorithms. Next there is the start vertex – the vertex that has the value *element*. As the next point, a stack is created and the first node is placed on the stack. Then the iterator points to the first vertex and the loop is running as long as there is something on the stack. For each repeat, try to find a successor to the node located on the top of the stack. If it is not possible, you pops the stack, but otherwise if a successor is found it will be marked and placed on the stack. Then the iterator points to this node. When the stack is empty, the iterator has pointed all the vertices that can be reached from the starting vertex.

Breadth first

Breadth first traverse looks like depth first. You start with a certain vertex, and then you handle all successors to this vertex. You then take these successors and for each one handle the successors (if they are not already handled). This continues until all vertices are handled – possibly except vertices that can not be reached from the starting vertex. As with depth first, the challenge is to keep track of which vertices you have already dealt with and which vertices to continue with when you finish a vertices' successors. At depth first, the task was solved using a stack, but in relation to width breadth first, the problem is solved instead with a queue:

```
Set C = start vertex
Print C and mark it as handled
Enqueue C in the queue
repeat as long the queue is not empty
```

```
C1 = Dequeuq the queue
C2 = a none marked successor to C1
repeat as long C2 not is null
 Print C2 and mark it as handled
 Enqueue C2 in the queue
 C2 = a none marked successor to C1
```

With the same simple graph as at depth first, the result of a breadth first traversing, in which I again have chosen the least index when I chose a successor, is as shown below. I have also shown what happens to the queue. The column on the left shows the element that was last taken out of the queue and thus the element whose successors are to be handled next.

```
A  B  C  E  D  G  H  I  F

    A
A   B
    B   C
B   C   E
C   E   D
    E   D   G
    E   D   G   H
E   D   G   H   I
D   G   H   I
G   H   I   F
H   I   F
I   F
F
```

Implementing the algorithm in Java is much as for depth first and the method uses the same helper methods.

In principle, the implementation of the two travers algorithms is the same and they both implements the iterator pattern. The question is how much it costs to move the iterator to the next node, and here is the work in the helper method *getSuccessor()*, and its complexity is determined by the density of the graph. If the graph is thin, the complexity in practice will be (almost) constant, but if the graph is dense, the complexity is linear.

When traversing a graph, the weights are not used, and the above methods can therefore be used for both weighted and non-weighted graphs. Both methods are based on finding the successors of a node, and they assume that the graph is oriented. If you compare with the way my graph is implemented, a graph is always oriented, as a non-oriented graph always adds an extra edge. Therefore, the travers methods work for both oriented and non-oriented graphs.

EXERCISE 9: TRAVERSE A GRAPH

Write a program that can test the depth first and breadth first traversing of a graph. You can call the project for *GraphTester3*. The project must have the following method:

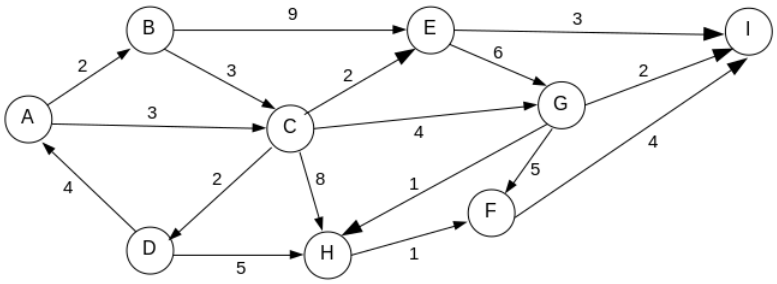
```
private static void print(IGraph<Character> graph)
{
 // print the number of vertices
 // print then number of edges
 // print the vertices on a line using the default traversing of the graph
 // print the vertices on a line using a depth first traversing of the graph
 // print the vertices on a line using a breadth first traversing of the graph
}
```

Build the same graph as in the program *GraphTester1* (with 9 corners and 17 edges). Print the graph using the above method, where A should be the start vertex.

Also build another graph, where the vertices are represented by letters, but where this time there should be the vertices A, B, …, Z but added to the graph in random order. Next, add random edges until the graph has 200 edges. Print the graph in the same way as above, but note that there may be vertices to which there is no path.

8.4 SHORTEST PATH

It's a whole family of algorithms, all of which aim are to find a path from one vertex to another vertex, but such that it is the shortest path. Usually, the algorithms are implemented so they from a starting vertex they finds the shortest path to all other vertices in the graph, where there is a path. In order for the algorithms to make sense, one has to measure something and it is here that the weights get into the picture. By the shortest path from one vertex to another vertex is meant a path so the sum of the weights of the edges that are part of the path is minimal. For example, if considering the graph below, is the shortest path from A to I is (A, C, G, I), which has the length (or cost) 9. It is easy to figure out in this case, but with a more complex graph it is nevertheless easy. I've described the problem as finding the shortest path, but of course you can also look at the problem finding the longest path, and the difference will be primarily to reverse the inequalities. In the following, I will only focus on the shortest paths.



MinPath

Shortest path algorithms are naturally defined for weighted graphs. However, I want to start somewhere else and look at a non-weighted graph where the shortest path should measure the number of edges, and the task is to find a path from a starting vertex to any other vertex, but so that the path have as few edges as possible. The algorithm can be described as follows:

```
Set C = the start vertex
Set the distance to C to 0
Enqueue C in a queue
repeat as long the queue is not empty
 C = Dequeue the queue
 repeat for all edges E from C
  if there not already is found a shortest path to E
   Set distance to E to distance to C + 1
   Note the predecessor to E as C
   Enqueue E in the queue
```

The idea is that if you first found the shortest path (like the number of edges) to a vertex, you can not do better by going another way. If you consider the above graph (and ignore all weights) and if you start in A, it is clear that the shortest paths to the direct followers B and C are 1 and you can not do it better. If you continue with B (by taking it from the queue), you have already found a path to its successor C, and of course it is not better to count one edge more. Conversely, no path has been found to E, and the shortest path becomes the path to B plus 1 and thus 2. To continue, next time, C will be taken from the queue.

A method that implements this algorithm, I have called *minPath()*. It must return an iterator, which iterates over all vertices, and returns for each vertex the shortest path to the starting vertex. For that I have defined the following class

```
public class Path<T>
{
 private IList<T> nodes;
 public double cost;
```

which represents a path from one node to another, as well as the accompanying cost or weight by following the path. The class *GraphTools* is expanded with two new helper methods that can create a path to *node*:

```
private static <T> Path<T> getPath(Node<T> node)
{
 IList<T> list = new ArrayList();
 addNodes(node, list);
 return new Path<T>(list, node.getDistance());
}

private static <T> void addNodes(Node<T> node, IList<T> nodes)
{
 nodes.add(node.getElement());
 if (node.getPrev() != null) addNodes(node.getPrev(), nodes);
}
```

They are used to construct a path (a *Path* object) to a *node*. These methods assume that an algorithm has previously been executed, which finds a shortest path stored in the *Node* object's *distance* property, and that the *Node* object's *prev* variable refers to the node's predecessor. That is, knowing the distance to the node, you can go back to the start vertex via *prev* and thus get the path.

With these remarks, the algorithm can be implemented as the following iterator:

```
static class PathIterator<T> implements Iterator<Path<T>>
{
 private IMap<T, Node<T>> nodes;
 private Iterator<T> keys;

 public PathIterator(IGraph<T> graph, T element)
 {
  nodes = getNodes((Graph)graph);
  Node<T> node = findNode(nodes, element);
  IQueue<Node<T>> queue = new Queue();
  queue.enqueue(node);
  node.setDistance(0);
  while (!queue.isEmpty())
  {
   Node<T> next = queue.dequeue();
   for (Edge<T> e : next.getEdges())
   {
    Node<T> to = nodes.get(e.getTo().getElement());
    if (Double.isInfinite(to.getDistance()))
    {
     to.setDistance(next.getDistance() + 1);
```

```
        to.setPrev(next);
        queue.enqueue(to);
       }
      }
     }
     keys = nodes.iterator();
    }

    public boolean hasNext()
    {
     return keys.hasNext();
    }

    public Path<T> next()
    {
     return getPath(nodes.get(keys.next()));
    }
  }
```

Basically, the algorithm works as the traversal algorithms, where a local map is created with *Node* objects, but otherwise the code corresponds exactly to the above algorithm. You should note how to test if a shortest path is found by comparing *distance* with *infinity*. The variable *distance* is initialized to this value in the class *Node*.

Note that the algorithm that finds the shortest paths in a non-weighted graph really is just a special case of the next shortest path algorithm, where the weights at all edges are 1.

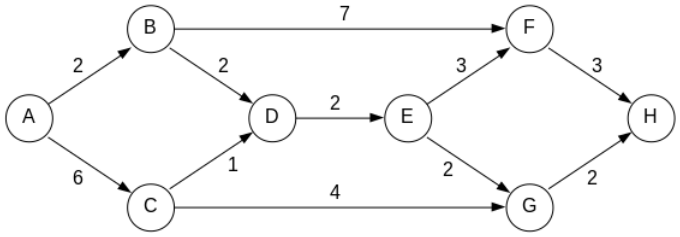
Dijkstra's algorithm

It is a well-known and classic algorithm to find the shortest paths in an oriented weighted graph. The algorithm assumes that all weights are not negative. The procedure is a bit the same as in *minPath()*, but you must keep in mind that the first path you determine is not necessarily the best:

1. Set the distance to all other vertices to infinity.
2. Select a starting vertex and set the distance to 0. Mark the vertex as completed, as the distance 0 must be a shortest distance.
3. Set the distance to all direct successors to the starting vertex to the weight.
4. Choose from the vertices that have not completed the vertex with the smallest distance.
5. Mark this vertex as completed.

6. Set the distance to all direct successors that are not completed to the sum of the distance to the vertex and the weight of the edge if it is a better distance than that already determined.
7. Repeat steps 4–6 until all vertices are completed.

Consider, for example, the following graph:



The easiest is to visualize the algorithm with a table where there is a row for each vertex. The first column indicates the vertices, the next is used to keep track of which paths have been completed, the third indicates the distance, while the last indicates the predecessor on the shortest path up to that vertex:

A		∞	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	
H		∞	

I will choose *A* as the starting vertex. The distance to A is 0 and there is no predecessor. I therefore mark *A* as completed and indicate the shortest paths to *A*'s immediate successors:

A	ok	0	
B		2	A
C		6	A
D		∞	
E		∞	
F		∞	
G		∞	
H		∞	

So far, I've only found the shortest path to A. I've also found paths to B and C, but it's not necessarily the shortest paths, but it's the shortest for the time being. I now choose a vertex, among those where I have not already found the shortest path (here is B–H), but the vertex which has the shortest path. It is *B* that has the distance 2. This is where it is

important that the distances from the start are set to infinity. There is no better way to *B*, because if so, this path would already be found in a previous pass. I now look at those of *B*'s successors, where no shortest path has already been found, that is to say, the successors who have not been completed (it's D and F) and investigate if I can get a cheaper way by going over B. In both cases I can (the routes found so far are infinite):

A	ok	0	
B	ok	2	A
C		6	A
D		4	
E		∞	
F		9	
G		∞	
H		∞	

Note that the new distances are determined as the distance to *B* plus the weight of the edge from *B* to the successor. Now the process repeats and I find the vertex that has the smallest distance and which is not marked as completed. It is *D*, which has the length of the path that is 4. *D* has a single successor *E*, and since it is not marked as completed, I investigate if I can get a better way of crossing *D*. I can (the length is infinity), and if I go through *D*, the distance can be calculated to 6:

A	ok	0	
B	ok	2	A
C		6	A
D	ok	4	B
E		6	D
F		9	B
G		∞	
H		∞	

Next time there are two vertices to choose from. Both *C* and *E* have the distance 6, which is the smallest distance. The algorithm does not tell me which vertex I should choose (it

does not matter) and I choose the first *C*. There are two successors, but the shortest to *D* is already determined so I just have to check if I can decide a better path to *G*, and I can:

A	ok	0	
B	ok	2	A
C	ok	6	A
D	ok	4	B
E		6	D
F		9	B
G		10	C
H		∞	

The next vertex is *E*, which has two successors (*F* and *G*), and none of them are marked as completed. The distance to *E* is 6, and the edge from *E* to *F* is 3, which gives a total distance of 9 to *F*. It's no better than the path already found, so I keep it, but note that it shows that the shortest path is not unique and that (not surprisingly) there can be several paths with the same distance. If you look at the path up to *G* via *E*, the distance is 8, and it is better than the distance already found. Therefore, the path to G must be updated as it is better to go over *E* than over *C* (which had the distance 10):

A	ok	0	
B	ok	2	A
C	ok	6	A
D	ok	4	B
E	ok	6	D
F		9	B
G		8	E
H		∞	

Next, according to the algorithm, choose *G*, which has the distance 8. *G* has only one successor named *H*, and it is not visited, and the table is updated as follows:

A	ok	0	
B	ok	2	A
C	ok	6	A
D	ok	4	B
E	ok	6	D
F		9	B
G	ok	8	E
H		10	G

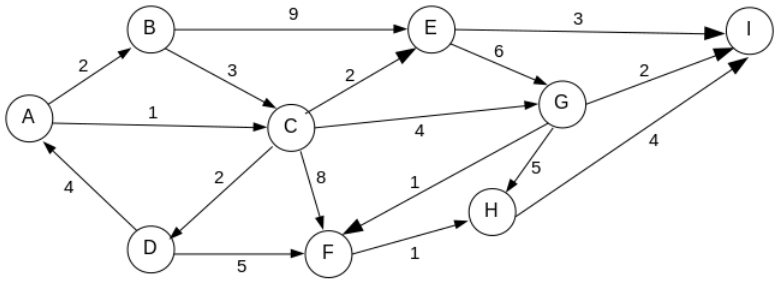
The next vertex is *F* and its only successor is *H*. The distance to *F* is 9, and the distance to *H* is 3 with a total distance of 12, and it is not better than the 10 that have already been found when going through *G*:

A	ok	0	
B	ok	2	A
C	ok	6	A
D	ok	4	B
E	ok	6	D
F	ok	9	B
G	ok	8	E
H		10	G

Then the task is solved, as *H* does not have successors that are not completed (in this case no one at all). The question is then what it's found. For example, if you take *G*, then the shortest path is from *A* is 8, and it goes via *E*, from which you come from *D*, and to *D* you come from *B* and from *A*. You can therefore summarize the result as follows:

To	Dist	Path
A	0	A
B	2	AB
C	6	AC
D	4	ABD
E	6	ABDE
F	9	ABF
G	8	ABDEG
H	10	ABDEGH

Consider as another example the graph:



The following tables show how to determine the shortest paths which starts in *A*:

A			
A	ok	0	
B		2	A
C		1	A
D		∞	
E		∞	
F		∞	
G		∞	
H		∞	
I		∞	

C			
A	ok	0	
B		2	A
C	ok	1	A
D		3	C
E		3	C
F		9	C
G		5	C
H		∞	
I		∞	

B			
A	ok	0	
B	ok	2	A
C	ok	1	A
D		3	C
E		3	C
F		9	C
G		5	C
H		∞	
I		∞	

D			
A	ok	0	
B	ok	2	A
C	ok	1	A
D	ok	3	C
E		3	C
F		8	D
G		5	C
H		∞	
I		∞	

E			
A	ok	0	
B	ok	2	A
C	ok	1	A
D	ok	3	C
E	ok	3	C
F		8	D
G		5	C
H		∞	
I		6	E

G			
A	ok	0	
B	ok	2	A
C	ok	1	A
D	ok	3	C
E	ok	3	C
F		6	G
G	ok	5	C
H		10	G
I		6	E

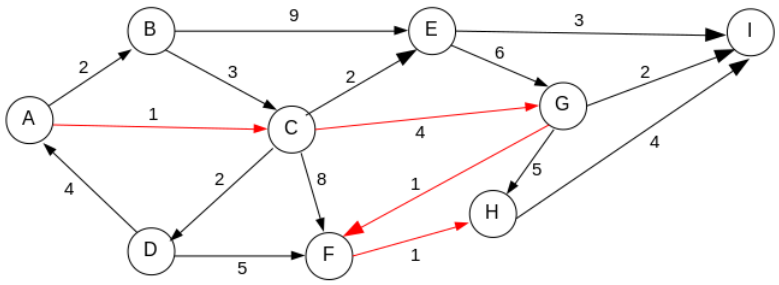
F			
A	ok	0	
B	ok	2	A
C	ok	1	A
D	ok	3	C
E	ok	3	C
F	ok	6	G
G	ok	5	C
H		7	F
I		6	E

I			
A	ok	0	
B	ok	2	A
C	ok	1	A
D	ok	3	C
E	ok	3	C
F	ok	6	G
G	ok	5	C
H		7	F
I	ok	6	E

A	ok	0	
B	ok	2	A
C	ok	1	A
D	ok	3	C
E	ok	3	C
F	ok	6	G
G	ok	5	C
H	ok	7	F
I	ok	6	E

Vertex	Distance	Path
A	0	A
B	2	AB
C	1	AC
D	3	ACD
E	3	ACE
F	6	ACGF
G	5	ACG
H	7	ACGFH
I	6	ACEI

and as an example, I have below shown the shortest path from A to H:



That was the algorithm, and the next thing is to get it written in Java. The idea in the algorithm is that you have decided the shortest way to the first vertices. The rest has a shortest path that is either infinite or not yet finally decided. You then take the vertex from the vertices whose shortest path is not yet determined, and which has the best so far smallest shortest path and calls it for the current vertex. It is now the next vertex whose shortest path is determined, and then you may need to adjust the distance to the current vertex's direct successors. The last is simple and the biggest challenge is to keep track of how far you have reached and which vertex should be the current vertex. For that a priority queue (or a heap) is used, where the vertices are inserted with a priority after the distance. Therefore, there is a need for a wrapper class that enclose a vertex and the distance, but such that the comparison is done only on the distance:

```
class Route<T> implements Comparable<Route<T>>
{
 private Node<T> dest;
 private double cost;
```

The algorithm can be implemented in the same way as shown above using an iterator, but this time it is more complex to implement the iterator. Basically, it's done with a priority queue and hence a heap, and the code is shown below, where the comments should explain how the code works:

```
static class DijkstraIterator<T> implements Iterator<Path<T>>
{
 private IMap<T, Node<T>> nodes;
 private Iterator<T> keys;

 public DijkstraIterator(IGraph<T> graph, T element) throws PaException
 {
  // encapsulates the graph's vertices with auxiliary fields for the
  // graph algorithm
  nodes = getNodes((Graph)graph);

  // set a reference to the start vertex
  Node<T> start = findNode(nodes, element);

  // the vertices for where a path is found are placed on a heap ordered
  // after the distance to the start vertex
  // it is equivalent to inserting them in a priority queue with the
  // distance as a priority to and associate the
  // distance for a vertex and the vertex are encapsulated in a wrapper
  // type called Route
  IHeap<Route<T>> heap = new Heap();

  // the start vertex is added to the heap with the distance 0, as there
  // is definitely a path from the start vertex to itself
  heap.add(new Route<T>(start, 0));
  start.setDistance(0);

  // the algorithm iterates over all vertices in the graph and so long there
  // are vertices left on the heap
  // if the heap is empty, it indicates that it is not a connected graph
  int count = 0;
  while (!heap.isEmpty() && count < nodes.getSize())
  {
   // the top element on the heap and thus the element that represents
   // the vertex with the smallest distance among the vertices whose
   // shortest path is not yet determined
   Route<T> route = heap.remove();
   Node<T> node1 = route.getDest();
```

```
// if the vertex refers to a vertex whose shortest path is determined,
// nothing else should happen, but the vertex should be ignored
// and you continue with the next element on the heap
if (node1.isUsed()) continue;

// otherwise the vertex will be marked as used to indicate that its
// shortest path is determined.
// it is now the current vertex
node1.setUsed(true);
++count;

// run over all the immediate successors of the current vertex
for (Edge<T> e : node1.getEdges())
{
 // node2 is a direct successor
 Node<T> node2 = nodes.get(e.getTo().getElement());
 double cost = e.getWeight();

 // If the weight is negative, the algorithm is interrupted with an error
 if (cost < 0) throw new PaException("Graph has negative edges");

 // if the edge from the current node to node2 leads to a better path
 // to node2, the path to node2 is updated and node2 is placed on the heap
```

```
      if (node2.getDistance() > node1.getDistance() + cost)
      {
       node2.setDistance(node1.getDistance() + cost);
       node2.setPrev(node1);
       heap.add(new Route<T>(node2, node2.getDistance()));
      }
     }
    }
    keys = nodes.iterator();
   }

   public boolean hasNext()
   {
    return keys.hasNext();
   }

   public Path<T> next()
   {
    return getPath(nodes.get(keys.next()));
   }
  }
```

You are encouraged to study the algorithm thoroughly and possibly try to follow the code through a test. Dijkstra's algorithm is a very well-known algorithm within graph theory. The algorithm is relatively effective and you can determine the time complexity as something towards the following:

$$O\big(Elog(V)\big)$$

where E is the number of edges, whereas V is the number of vertices.

The Bellman-Ford algoritmen

A requirement for Dijkstra's algorithm is that all weights are non-negative. Although it may not happen so often, there is nothing that says that a graph may not have negative weights. If so, one can still be interested in the shortest paths, but it is difficult to determine the shortest paths, and Dijkstra's algorithm can not be used immediately. The problem is that once you have found a shortest path to a vertex – the current vertex – a successor with a negative weight can lead to a shorter path to one of the vertices where you have already determined the shortest path. Another problem is that if a negative weight is included in a cycle, the cycle can lead to a shorter and shorter distance, and you can not decide a shortest path. It complicates the implementation of the algorithm and you are encouraged to study the code.

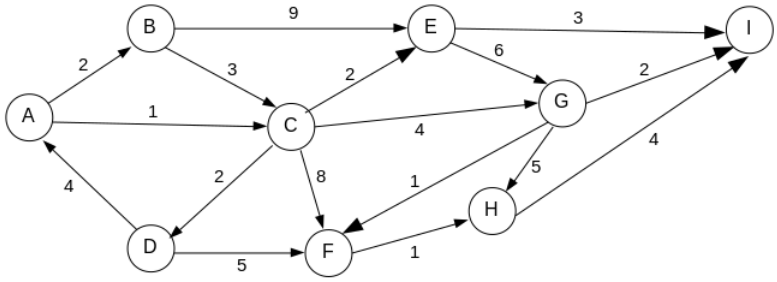
Not only is the implementation of *Bellman Ford* more difficult to understand, it also has a poorer time complexity than Dijkstra. Dijkstra basically consists of an iteration over the vertices, and what must happen for each vertex depends on the number of direct successors. If the graph is thin (few edges), it means that the complexity of Dijkstra with good approximation is as shown above. In Bellman Ford you must at worst compare all paths with all edges (every time you try with an edge, you must at worst check what it means for all paths) and the result has therefore the worst complexity that is *O(EV)* where E is the number of edges and V is the number of vertices.

EXERCISE 10: SHORTEST PATHS

In this exercise you should write a program that can test the three shortest path algorithms. Start with a project that you can call *GraphTester4*. Start by adding a method

```
private static IGraph<Character> build1()
```

which builds and returns the following graph:



You must then write a method *test1(IGraph<Character> graph)*, which for a graph determines and prints the shortest paths using the methods *minPath()*, *dijkstra()* and *bellmanFord()*. Test the method with a graph created with the method *build1()*.

Write a corresponding method *test2(IGraph<Character> graph)* that does the same, but only using *bellmanFord()*. Write a method *build2()* that creates the same graph as above, but with the difference that the edge from G to H has the weight -6. Test the method *test2()* using a graph created by *build2()*.

Write a method *build3(int n)*, which returns an *IGraph<Integer>*, which has *n* vertices with values from 1 to *n* when the vertices are to be added in random order. The method must also try to add *20n* random edges when the weight for each edge must be a random *double* between 0 and 1. Then try the following test method:

```
private static void test3()
{
 StopWatch sw = new StopWatch();
 for (int n = 1000; n <= 30000; n += 1000)
 {
  try
  {
   IGraph<Integer> graph = build3(n);
   sw.start();
   GraphTools.dijkstra(graph, 1);
   sw.stop();
   long t1 = sw.getNanoSeconds();
   double z1 = graph.getCount() * Math.log(graph.getSize());
   System.out.println(String.format("Dijkstra: %6d%10d%15d%15.6f",
    graph.getSize(), graph.getCount(), t1, t1 / z1));
   sw.start();
   GraphTools.bellmanFord(graph, 1);
   sw.stop();
   long t2 = sw.getNanoSeconds();
   double z2 = ((double)graph.getCount()) * graph.getSize();
   System.out.println(String.format("Bellman-Ford: %6d%10d%15d%15.6f",
    graph.getSize(), graph.getCount(), t2, t2 / z2));
  }
```

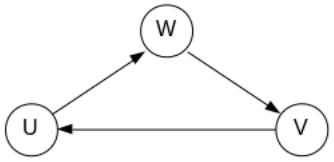
```
  catch (Exception ex)
  {
   System.out.println(ex);
  }
 }
}
```

which you can use to test whether the above claims regarding complexities seem reasonable.

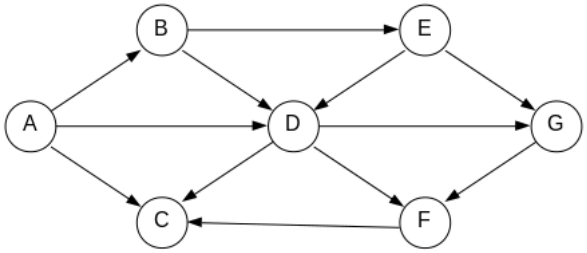
8.5 TOPOLOGICAL SORT

I will now look at an algorithm that performs a so-called topological sorting. It is an algorithm that relates to oriented graphs without cycles – also called an acyclic graphs. A topological sorting arranges the vertices in an oriented acyclic graph in such a way that if there is a path from U to V then V appears after U.

Note first that the requirement for an acyclic graph is necessary, for else if two vertices U and V are vertices of a cycle, there will be a path both from U to V and from V to U:

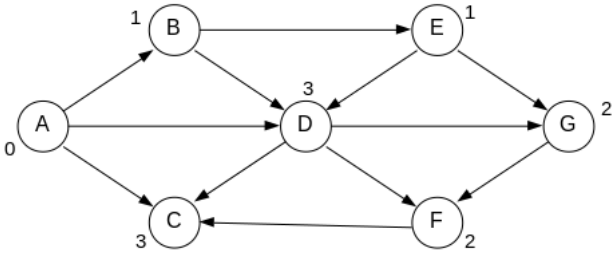


as [U, W, V] and [V, U]. Also note that a topological sorting makes sense for an non-weighted graph. Finally, note that a topological sorting is usually not unambiguous and there will usually be more solutions that make up a topological sorting. It is relatively simple to implement a topological sorting, and the algorithm is best explained as an example. Consider the following graph:

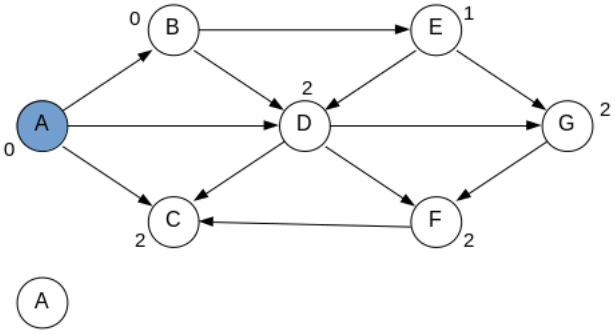


and it is easy to convince that it is an acyclic graph. At the fan-in for a vertex is meant the number of predecessors. A vertex with fan-in that is 0 has no predecessors. This applies

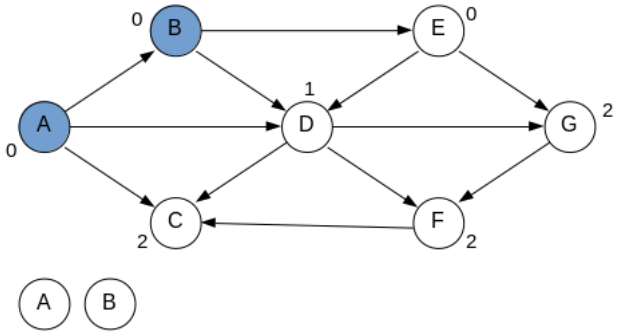
to the vertex A, and for example D has fan-in 3. The method is now to mark each vertex with its fan-in:



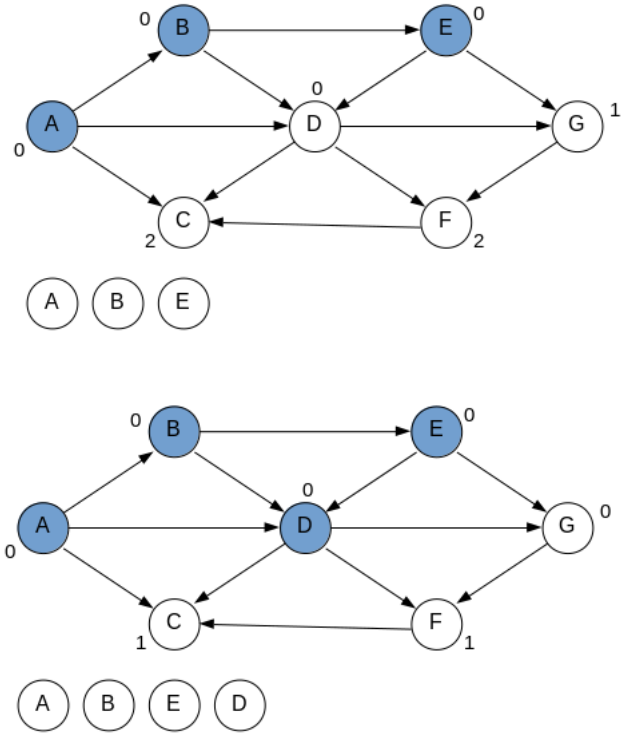
You now start with the vertex that has fan-in 0 (and if there are more you choose one). This will be the first vertex in the topological sorting (see below). At the same time, you reduce the fan-in for all direct successors to the selected vertex. In this case, there are B, C and D. At least one of these vertices will then have a fan-in, which is 0, because if not, the graph has a cycle. The algorithm now goes on in each iteration to select a vertex which has a fan-in that is 0 (and if there are more than one it does not matter which one you choose) and add it to the list of vertices for the topological sorting. In addition, the fan-in is reduced by 1 in all direct successors. Continue until all nodes (if possible) are added to the topologically sorted list. Below is the result of the first iteration:

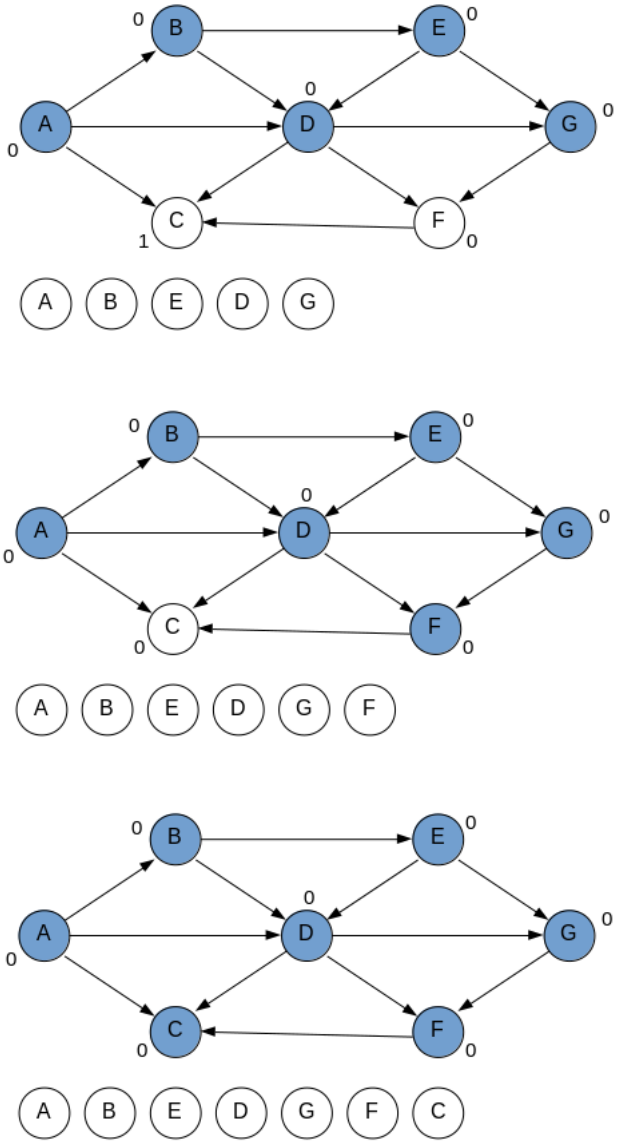


As next step, B should be added to the list:



and below the result of the other steps in the algorithm:





The algorithm can therefore also be described as follows: Iterates over all vertices, where you in each iteration selects a vertex with a fan-in that is 0. The processing consists of adding the vertex to the result and reducing the fan-in of all successors. You have to "look" on all edges, and the time complexity will therefore depend linearly on the number of edges.

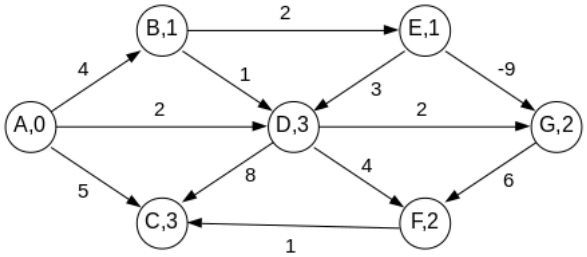
It is relatively easy to implement the algorithm. Each time you get to a vertex with fan-in 0, it is inserted in a queue and you continue until the queue is empty. In fact, I have implemented two versions of the algorithm:

```
public static <T> IList<T> topologicalSort(IGraph<T> graph) throws
PaException
{ … }
```

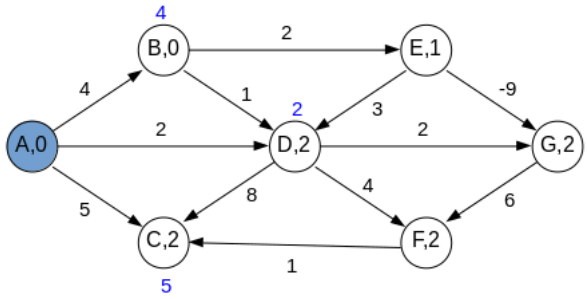
that determines a topological sort of the elements in a graph and returns the result as a list, as well

```
public static <T> boolean hasCycles(IGraph<T> graph) { … }
```

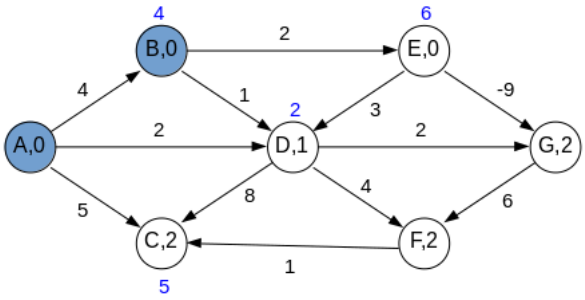
that tests if a graph has cycles. An application of topological sorting is the determination of the shortest paths. Suppose you have the following acyclic weighted graph, for which I have written fan-in in the individual vertices for technical reasons:



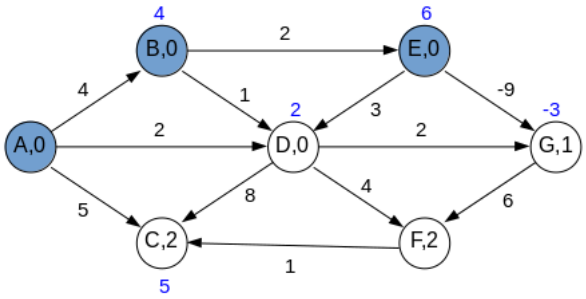
I want to start in A and determine the shortest paths to all other vertices. The idea is now to go through the vertices in a topological sorting and continuously update the distance to the direct successors. If you are at a vertex with fan-in 0, there are no successors who can give a better result (there are no predecessors that have not been visited) and you have therefore found the shortest path. When starting in A (which has fan-in 0), you determine the preliminary shortest path to the successors (marked as blue in the figure), while decreasing the fan-in for the successors (B, C and D):



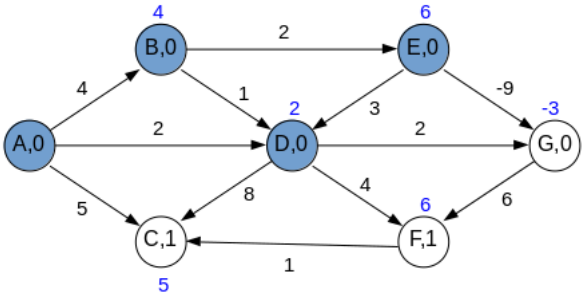
The next vertex to be treated is B. Since B does not have other predecessors, the shortest path to B is 4. Next, determine the shortest paths to the successors as the distance to B plus the weight of the edge. If it is better than the path that is already found (as may be infinite), the distance is updated and else nothing happens. In this case, the distance to E is set to 6, while distances to D are not changed (4 + 1 > 2). After the distances are updated, the fan-in in the successors is counted down by 1:



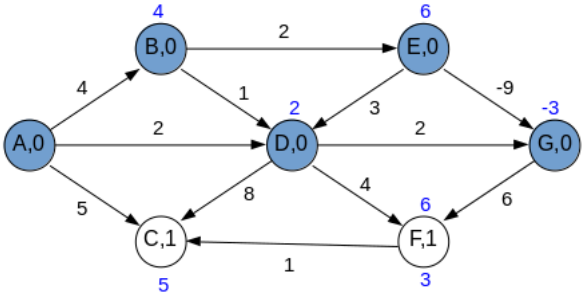
The next vertex is E. You have then found the shortest paths to A, B and E, and you are updating the distances to D and G. Note, in particular, that the distance to G becomes negative. This method of determining the shortest path works in the same way as Bellman Ford on negative weights. On the other hand, it is a requirement that the graph is acyclic.



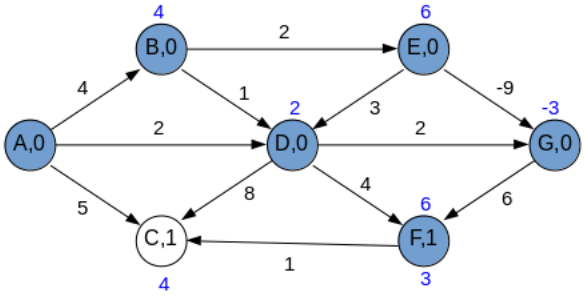
The next vertex is D:



It only leads to updating the distance to F. Next vertex is G:



That means the distance to F must be updated, then -3 + 6 < 6. Finally, the vertex F must be treated:

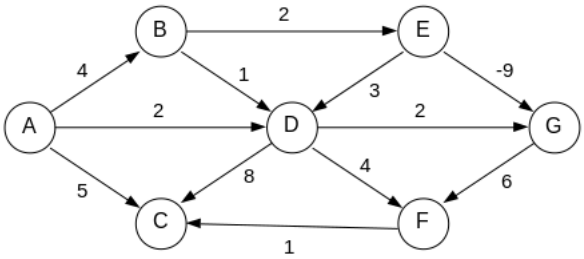


which causes the distance to C to be updated. Note that in the above figures I have shown only the distance to a vertex, but not the path. Also note that in the example I started with A, which have no predecessors. It does not have to be the case and the method works even if you start with a vertex that has predecessors.

The implementation of the algorithm is identical to the implementation of topological sorting, only you for each repetition must measure the distances to the successors.

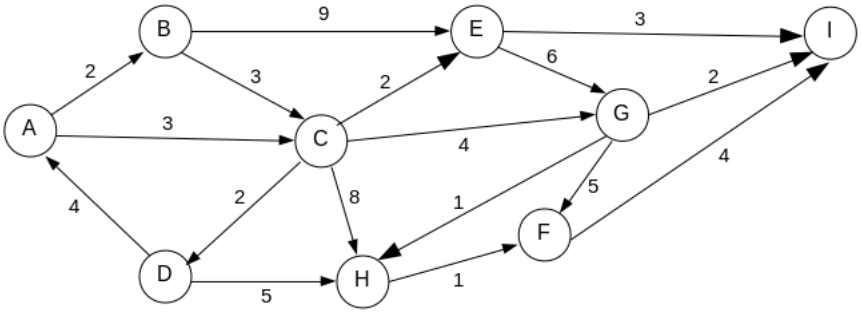
EXERCISE 11: GRAPHTESTER5

Create a project that you can call *GraphTester5*. The project must have a method that creates the following graph:



Add a test method that prints a topological sort of vertices in this graph.

Add a method that creates the following graph:

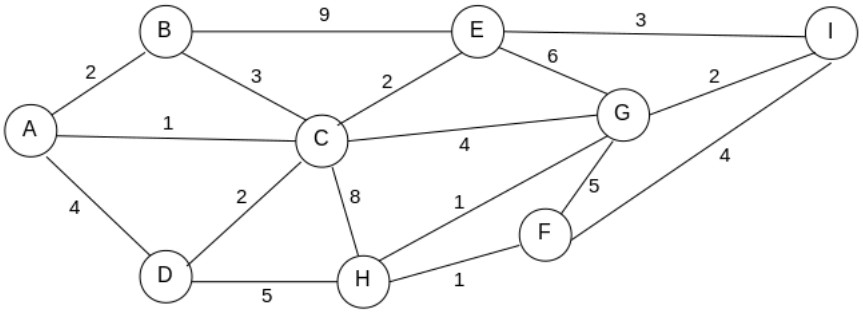


Add a test method that tests if this graph is cyclic (what it is). Try removing the edge from D to A and test again if the graph is cyclic.

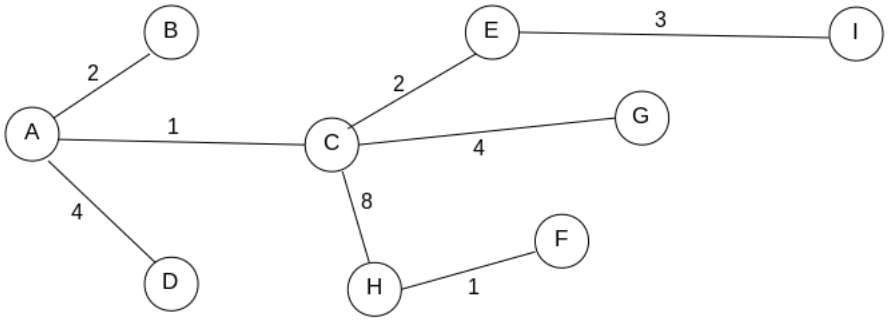
Finally, write a test method that determines the shortest paths with A as start in the first of the above graphs when using the method *GraphTools.acyclicPaths()*.

8.6 MINIMUM SPANNING TREE

A tree can be perceived as a graph with a starting element, because a tree consists of nodes (vertices) and each node has a unique parent (edge). Since each node has exactly one parent, it is an acyclic graph. Generally, a graph is not a tree, but given an non-oriented weighted graph, for example



you can remove edges so that it becomes a tree:

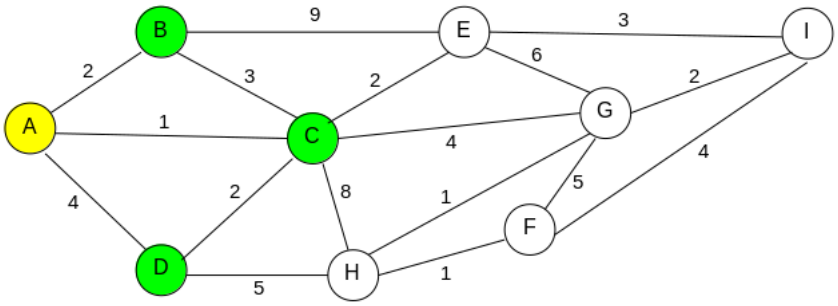


It is a tree, and such a tree is called for a spanning tree, as there is a path from the root to any other vertex – all vertices are included. In this case the graph is weighted and you can then try to determine the least spanning tree which is a spanning tree so the distance from the root to the vertices is minimal.

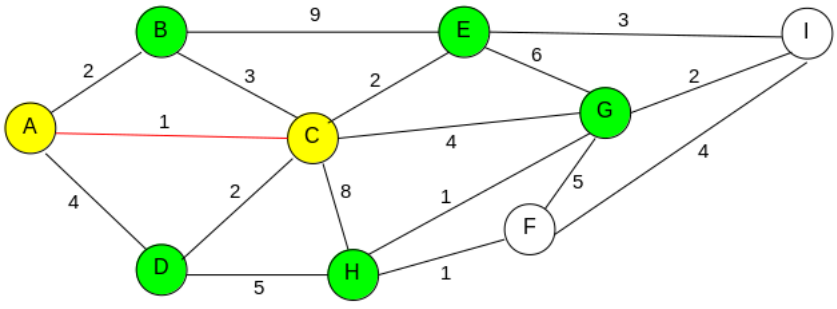
There are several algorithms to determine the least spanning tree for a non-oriented weighted graph, and I will show a very simple algorithm called *Prim*'s algorithm. The procedure is:

1. Begin by choosing a start vertex and add it to the least spanning tree.
2. Repeat as long as all vertices are not added to the tree: Find the smallest edge from the tree to a vertex that has not already been added to the tree and add it to the tree.

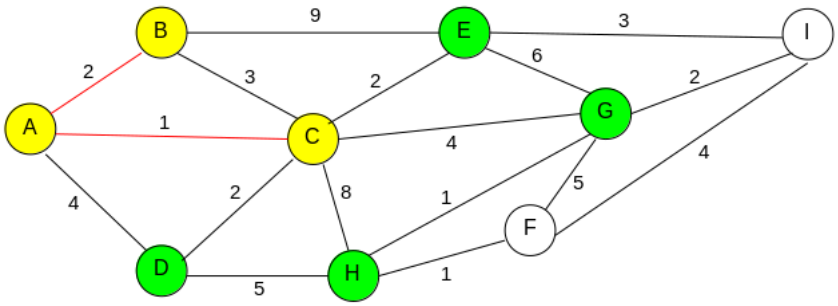
If you use the above tree as an example and starts in A, it is added to the tree as root, and at the same time you mark the successors as candidates:



Now you choose the shortest path to a candidate, that is the edge to C, and this vertex is now added to the tree, while at the same time selecting its successors as candidates:



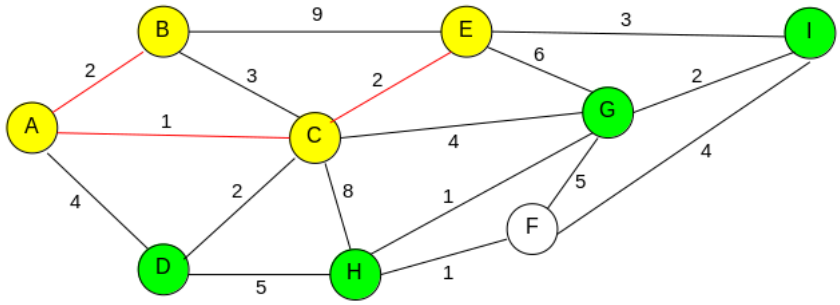
That is that the tree now consists of the edge between A and C. As the next step, you now take the candidate who has the least distance to the tree:



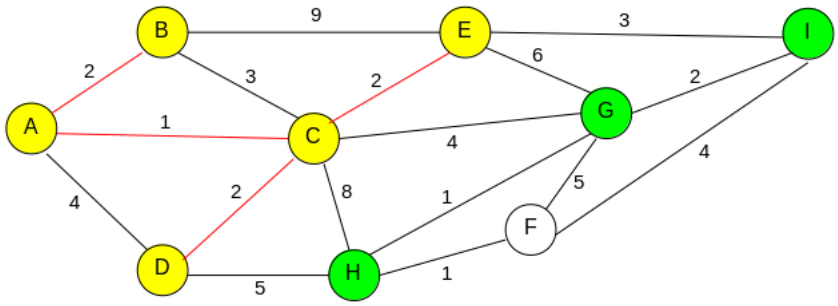
This time there were several options:

[A, B], [C, E], [C, D]

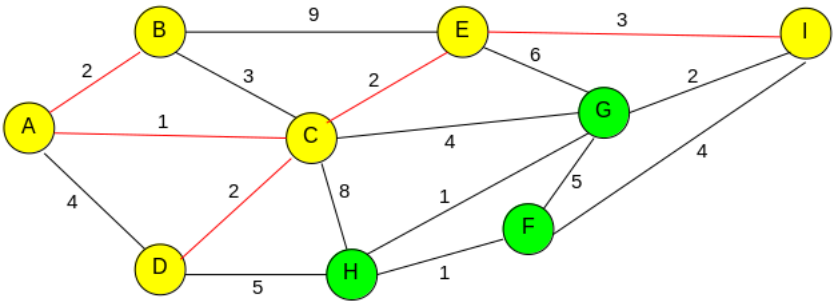
where the distance to the tree is all 2 and I have chosen the first one. By adding B to the tree, it now consists of three nodes and there is a single new edge as a candidate (the edge between B and C will not be candidate because C is already a node in the tree). To choose a new vertex for the tree, I will again choose the candidate with the least weight and there are two options this time. I choose E:



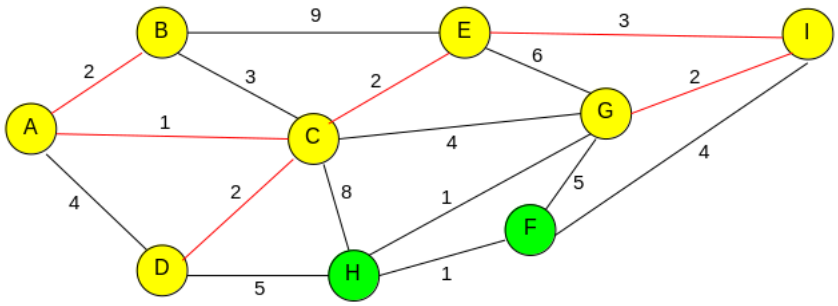
At the same time, the edge between E and I becomes a candidate. Next time there is only one option, namely the edge between C and D:



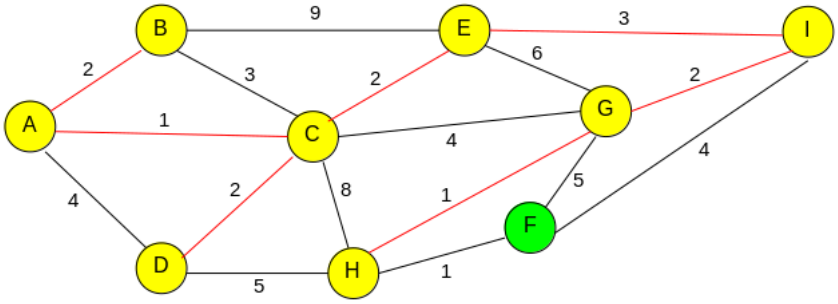
and the edge between D and H becomes candidate. The smallest candidate that now connects a vertex to the tree is the edge between E and I so that you insert it into the tree:



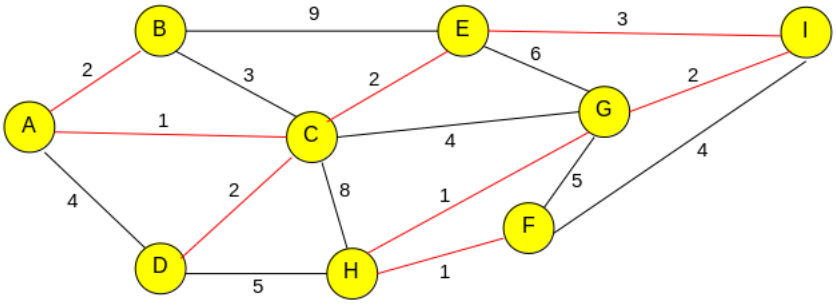
At the same time there are two new candidates, and here is the edge between I and G with the weight 2 the best:



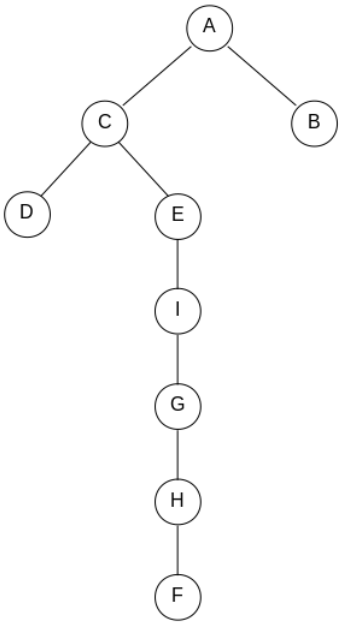
The next edge becomes the edge between G and H:



and finally there is the edge between H and F:



The result is a spanning tree as shown below:



The algorithm is relatively simple, but back, of course, there is also where the algorithm is solving the task, how good it is and how to implement it.

Regarding the first, there are several things to consider. First, one should consider that a connected non-oriented graph generally has a spanning tree. With it in place and if the graph is weighted, you must explain that the above procedure leads to a smallest spanning tree. The above illustrates an algorithm, but does not prove it, and although it is beyond the scope of this book, I can state that it is possible to formally prove Prim's algorithm.

The class *GraphTools* implements the algorithm (the method *prim()*), where the comments should explain the procedure. The method returns the spanning tree as a data structure of the form:

```
public class SpanningTree<T>
{
 private T element;
 private double cost;
 private IList<SpanningTree<T>> childs = new ArrayList();

 public SpanningTree(T element, double cost)
 {
  this.element = element;
  this.cost = cost;
 }
```

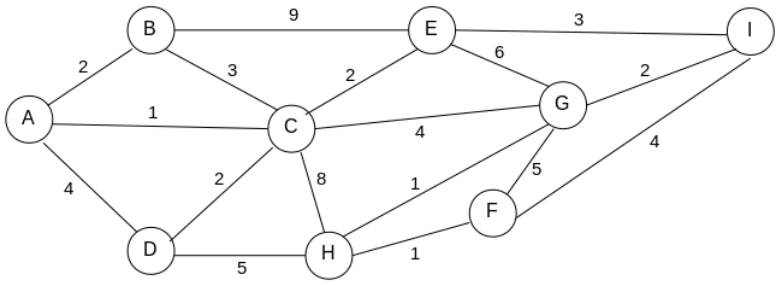
```
public T getElement()
{
 return element;
}

public double getCost()
{
 return cost;
}

/**
   * Returns all subtrees for the spanning tree in the order they are added to
   * the tree.
   * @return All subtrees for the this spanning tree
   */
public IList<SpanningTree<T>> getChilds()
{
 return childs;
}
```

```
 /**
   * Add a subtree to this tree.
   * @param child The subtree
   */
 public void add(SpanningTree<T> child)
 {
  childs.add(child);
 }
}
```

I do not want to show the code for the method *prim()*.

EXERCISE 12: PRIM'S ALGORITHMS

Write a program that, using Prim's algorithm, determines the least spanning tree for the graph:



That is, a method that builds the graph and a method that determines and prints the least spanning tree.

It is possible to show that the time complexity of the Prim's algorithm is

$$(V + E)log(V)$$

where V is the number of vertices and E is the number of edges. Add the following method that creates a graph with n vertices:

```
private static IGraph<Integer> build(int n)
{
 IGraph<Integer> graph = new Graph(GraphType.Undirected);
 try
 {
  IList<Integer> keys = new ArrayList();
  for (int i = 1; i <= n; ++i) keys.add(i);
  while (keys.getSize() > 0)
  {
```

```
    int i = rand.nextInt(keys.getSize());
    graph.add(keys.get(i));
    keys.removeAt(i);
   }
   for (int i = 0; i < 10 * n; ++i)
   {
    int a = rand.nextInt(n) + 1;
    int b = rand.nextInt(n) + 1;
    double w = rand.nextDouble();
    graph.add(a, b, w);
   }
  }
  catch (Exception ex)
  {
   System.out.println(ex);
  }
  return graph;
}
```

Use this method to validate the above formula for the time complexity by means of time measurements.

9 TRAVELING SALESMAN PROBLEM

This and the previous book have provided algorithms and data structures, and I will end with a chapter that presents a problem, the literature referred to as *The Traveling Salesman Problem* or short TSP. The problem is formulated as follows:

> *Given a number of cities and roads with road lengths between the cities, the task is to determine a route from a particular city and back again so that the route visits each city exactly once and so that the total route length is minimal.*

Similar to the previous chapter, the task can thus be formulated as finding the shortest path through an orientated weighted graph when the road must be a cycle that almost the start and the end vertex contains all vertices exactly once. You should note that it is an oriented graph and that usually between two vertices there will be an edge in both directions, but it does not have to be the case, nor is it necessary that the two roads in both directions have the same length.

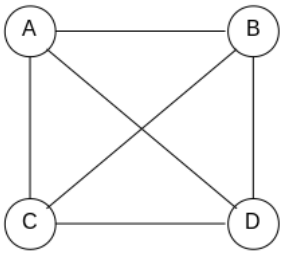
The problem, of course, has practical interest (even in many different situations). The problem has been formulated in the early '30s and has since been of great theoretical interest, but when I want to mention the problem here, it is partly as an example of a very complex problem, and partly as an example of a problem in which not a solution is known – at least not a general solution. However, there are algorithms that find local solutions for TSP, and in the following I will mention an example. It should be added that both the theory and the number of algorithms are extensive and the following does not go much further than a short presentation, but it may be a good practice to investigate what you can find online, including the many examples of algorithms.

9.1 MORE ON GRAPHS

For the sake of the following and TSP, I would like to add some additional definitions / remarks to graphs, and in the following G = (V, E) refers to a graph.

Two vertices are said to be *adjacent* if there is an edge between the vertices. At the degree of a vertex *v* you understand the number of vertices that are adjacent to *v*, and one often use the notation *deg(v)*. If the graph has *n* vertices, the degree of a vertex can therefore be max *n-1* (if the vertex does not have an edge to itself).

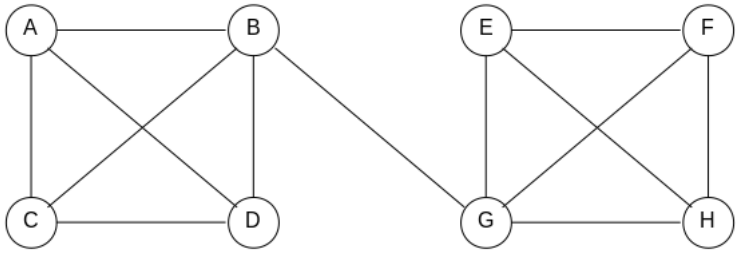
By a complete graph one understands a graph where two arbitrary different vertices are adjacent:



In the previous chapter, I have defined a path between two vertices as a sequence of edges that connect the two vertices. At that point it was allowed that the same vertex could occur several times, but in this chapter I would like a path to consist of different vertices apart from the first and the last vertex. In particular, a cycle has no repetitions beyond the start vertex, and sometimes one calls a cycle for a closed path.

A path that encompasses all vertices in the graph is called a Hamiltonian path, and in particular, you are talking about a Hamiltonian cycle.

In chapter 7 I have mentioned connected graphs (what all the examples shown above have been), but exactly one can say that a graph is connected if there is a path from A to B for each pair of vertices A and B. If a graph is not connected, it means that it consists of several sub-graphs, and here each sub-graph is called for a component. If you have a connected graph and if you have an edge where the result of removing the edge is a non-connected graph, the edge is called a bridge. For example, the edge {B, G} below is a bridge:



If you have a complete graph, it is clear that there is a Hamiltonian cycle, because there is an edge between all pair of vertices, but if the graph is non-complete you can not be sure about that. For example, if a graph has a bridge, there is not a Hamiltonian cycle, because if the start vertex is on the one side of the bridge, one can not cross to the other side without passing the bridge and the you can not get back again without visiting the same vertices.

It is clear that where a graph has a Hamiltonian cycle or not, is determined by how dense the graph is. There are several statements that indicates when a graph has a Hamiltonian

cycle and, for example, I would like to mention one. If a graph has n vertices, where n is greater than or equal to 3, and if

$$\forall v \in V: deg(v) \geq \frac{n}{2}$$

the graph has a Hamiltonian cycle. The statement is proved in the field of graph theory.

9.2 MORE ON COMPLEXITIES

If you consider a Hamiltonian cycle in a complete graph with n vertices and a start vertex, then you from the start vertex can go to *n-1* other vertices. From here there are *n-2* options to go to the next vertex and next time there are *n-3* options, and that means there are *(n-1)!* options to go all the way around. Since there are also opportunities for starting the vertex, the graph contains *n!* Hamiltonian cycles. Now it's not so wrong, because each cycle has an opposite cycle, where the difference is only if you go the one way or the another. One can conclude that for a given start vertex, the maximum number of Hamiltonian cycles is

$$\frac{\big((n-1)!\big)}{2}$$

which corresponds to a permutation of the vertices. One can therefore solve TSP with a brute force algorithm, that for a given graph and at a given start vertex, determine all Hamiltonian cycles and then determine the one with the smallest path length. The problem is simply that the factorial grows incredibly fast, and therefore a just slightly larger TSP problems require another algorithm. The idea is to divide a TSP problem into sub-problems and hope that they are of a simpler complexity class.

A complexity class covers a family of problems that have a complexity that resembles each other. If a problem can be solved with a time complexity expressed by means of a polynomial, it is said that the problem's complexity class is P, and that it is a P problem. Note that it applies to most of the problems I've shown in this and the previous book, such as linear algorithms, square algorithms, and more.

A decision problem is a problem whose solution is either a yes or no. A family of decision problems that can be solved by a deterministic algorithm is said to be in the complexity class NP. TSP is such a problem since the problem can be formulated as:

> *Given an non-directed graph and the distances between all vertices, does there exist a Hamiltonian cycle whose distance is less than c, for some given total distance c?*

This is a yes or no question, and thus is it is a NP problem.

Let P1 and P2 be two decision problems. If an algorithm can be reduced from a solution for P1 to a solution for P2 in polynomial time, it is said that P1 is polynomial reducible to P2. A problem is NP-complete if it can be proven that it is in the complexity class NP, and it can be shown that it is polynomial reducible to a problem in polynomial time that it is already known to be NP-complete.

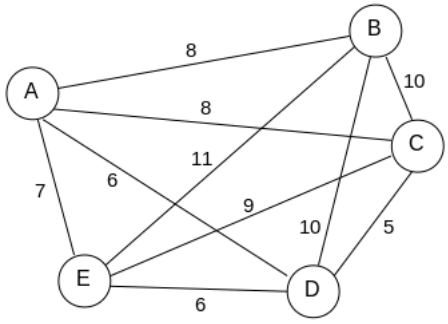
A problem is NP-hard if and only if it is at least as hard as an NP-complete problem. TSP is such a problem, since its solution includes to find all possible Hamiltonian cycles and then find the shortest one. Traveling Salesman Problem is thus classified as NP-Hard.

9.3 TSP ALGORITHMS

For the time being, one do not know any solutions of the general TSP, but there are several algorithms that solve local versions of TSP, and I will below show an example. It should be noted that most examples include algorithms that are used solely on complete graphs.

As an example, I will use the following simple graph with only 5 vertices. It is a complete graph, and although I above have described TSP as a problem regarding oriented graphs,

the problem also includes non-oriented graphs, as it just are graphs with an edge in both directions.



The following example is implemented in the project TSP. Before I look at the algorithm, I will implement a brute force algorithm, which is quite simple. The project contains the class *Permutation* from earlier, and with that class available, the algorithm can be written as follows:

```
private static <T extends Comparable<T>> Path<T> bruteForce(IGraph<T>
graph)
{
 List<T> keys = new ArrayList();
 for (T t : graph) keys.add(t);
 Collections.sort(keys);
 List<Permutation> edges = Permutation.permutations(graph.getSize() - 1);
 Path<T> path = new Path();
 path.setCost(Double.POSITIVE_INFINITY);
 try
 {
  for (Permutation p : edges)
  {
   Path<T> temp = new Path();
   int v = 0;
   temp.addVertex(keys.get(0));
   for (int i = 0; i < p.getLength(); ++i)
   {
    T to = keys.get(p.getValue(i));
    temp.addCost(graph.getWeight(keys.get(v), to));
    temp.addVertex(to);
    v = p.getValue(i);
   }
   temp.addCost(graph.getWeight(keys.get(v), keys.get(0)));
   temp.addVertex(keys.get(0));
   if (path.getCost() > temp.getCost()) path = temp;
  }
 }
```

```
 catch (Exception ex)
 {
  System.out.println(ex);
 }
 return path;
}
```

where the class *Path<T>* is the following:

```
public class Path<T> implements Iterable<T>
{
 private List<T> vertices = new ArrayList();
 private double cost = 0;
```

You should note that the algorithm to make it simple every time starts in the first vertex (after they are sorted), which one could of course change. The TSP project has a test method *test1()*, which uses the algorithm to solve TSP for the above graph. If you do that you get the result:

```
A B C D E A
36.0
```

where it in this case is easy to convinces yourself that it is correct. The project also has the following method:

```
private static IGraph<Character> build2(int n)
{
 IGraph<Character> graph = new Graph();
 List<Character> list = new ArrayList();
 char c = 'A';
 for (int i = 0; i < n; ++i, ++c) list.add(c);
 try
 {
  for (Character ch : list) graph.add(ch);
  for (Character c1 : list) for (Character c2 : list)
   if (!c1.equals(c2)) graph.add(c1, c2, rand.nextInt(90) + 10);
  Collections.shuffle(list);
  for (int i = 0; i < list.size(); ++i)
   if (list.get(i).equals('A'))
   {
    if (i > 0)
    {
     list.set(i, list.get(0));
     list.set(0, 'A');
    }
    break;
   }
```

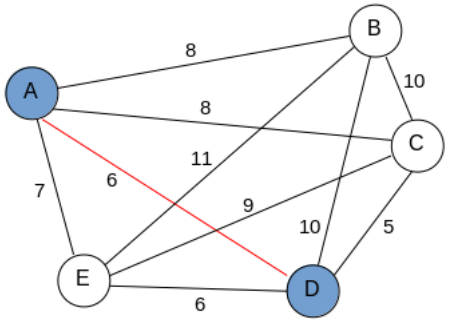
```
  for (int i = 1; i < list.size(); ++i)
    graph.add(list.get(i - 1), list.get(i), rand.nextInt(9) + 1);
  graph.add(list.get(list.size() - 1), list.get(0), rand.nextInt(9) + 1);
 }
 catch (Exception ex)
 {
  System.out.println(ex);
 }
 return graph;
}
```

which creates a graph with the vertices A, B, C, … where the parameter indicates the number of vertices. The method creates a complete graph and assigns random weights with values between 10 and 100. Then, the values of the vertices are shuffled in random order followed by A being placed in the first place. Finally, the weights on a random cycle are changed so they all have a random value between 1 and 10. The result is a Hamiltonian cycle through the graph with less weights that the algorithm should find. If you try out the method *test1()* with a graph created by this method, it seems apparently fine, but only as long that n is small. If n becomes greater than 10, the method becomes slow to finally stops completely.

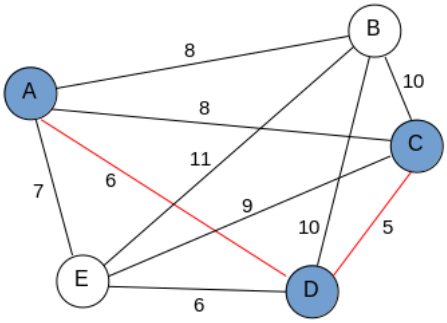
Note that the algorithm actually solves TSP, but its time complexity is just so slow that it can not be used for anything in practice. Note that efficiency can be improved by not using the class *Graph* and instead representing the graph using a 2-dimensional array. Similarly, it is a little expensive to use the class *Permutation*, as its complexity is also *O(N!)*, and here you could also seek for a better solution, but none of the parts change the complexity of the brute force algorithm, which is thus unusable in practice.

9.4 NEAREST-NEIGHBOR

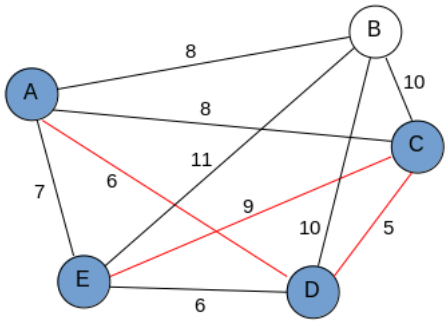
A very simple algorithm is starting in a vertex, and here you choose the way to the nearest successor and then reach another vertex. Here you repeat the process and choose the vertex to the successor with the least weight, and if there are several options, one must choose one of the options. It's an extremely simple algorithm, and the only thing to be aware of is that you should not go to a vertex where you have already been. If the graph is as above and if you start in A, there are 4 options and you must go to D where the weight is 6:



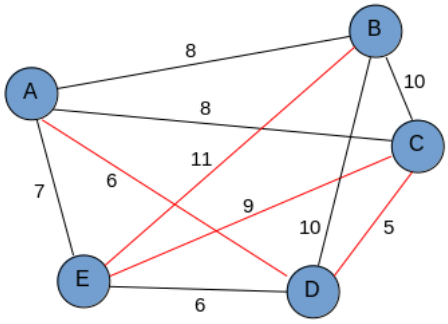
Then there are three options, and you have to go to C where the weight is 5:



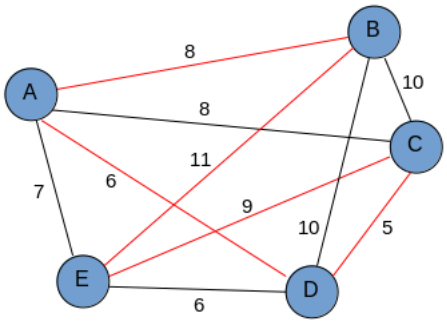
Next time there are only 2 options, because A and D are already added to the path, and since the least weight for B and E is 9, go to E:



Then there is only one option, namely to go to B:



and finally, the path closes by going back to A:



Thus, the algorithm finds returns

```
A D C E B A
39
```

It's a greedy algorithm. When you stand in a vertex, you choose the path that from this place seems to be the best solution without distinction as to whether another choice in the long term could be better. If you compare the result with the brute force algorithm, you

can see that next-neighbor does not in this case decide an optimal solution, but on the other hand, it is both a simple and effective algorithm. The algorithm can be implemented in Java as follows:

```
public static <T> Path<T> nearestNeighbor(IGraph<T> graph, T start)
 throws PaException
{
 ISet<T> used = new HashSet();
 Path<T> path = new Path();
 path.addVertex(start);
 used.insert(start);
 T current = start;
 while (path.getSize() < graph.getSize())
 {
  IList<Vertex<T>> list = graph.getSuccessors(current);
  double c = Double.POSITIVE_INFINITY;
  T t = null;
  for (Vertex<T> v : list)
  {
   if (v.getElement().equals(current)) continue;
   if (used.contains(v.getElement())) continue;
   if (graph.contains(current, v.getElement()))
   {
```

```
    double w = graph.getWeight(current, v.getElement());
    if (w < c)
    {
              t = v.getElement();
              c = w;
    }
   }
  }
  if (t == null) throw new PaException("No Hamiltonian cycle found");
  path.addVertex(t);
  path.addCost(c);
  used.insert(t);
  current = t;
 }
 path.addVertex(start);
 if (graph.contains(current, start))
  path.addCost(graph.getWeight(current, start));
 else throw new PaException("No Hamiltonian cycle found");
 return path;
}
```

This time you can specify the start vertex and if you apply the method to a graph created using the method *build2()*, the result could be the following, but this time with the value 26 for the parameter:

```
A I C K Y H J R Z F S L O U V Q M W D E T N X B G P A
125.0
```

You should note that this method finds a shortest path in a 26-node graph, and it does not take long, but also that the result is correct (otherwise the total length of the path would be at least 260). However, one can not be sure that the algorithm finds the right solution, as is also apparent from the first example – it is the price that it is a greedy algorithm.

As the algorithm is written, it also works for non-complete graphs. However, one can not be sure that it finds a solution, even if the graph has a Hamiltonian cycle. Consider the following method that constructs a graph with n corners, and so the degree of each corner is at least $n / 2$:

```
private static IGraph<Integer> build3(int n)
{
 IGraph<Integer> graph = new Graph();
 try
 {
  IList<Integer> keys = new dk.data.torus.collections.ArrayList();
  for (int i = 1; i <= n; ++i) keys.add(i);
```

```
   while (keys.getSize() > 0)
   {
    int i = rand.nextInt(keys.getSize());
    graph.add(keys.get(i));
    keys.removeAt(i);
   }
   while (!hasHamiltonian(graph))
   {
    int a = rand.nextInt(n) + 1;
    int b = rand.nextInt(n) + 1;
    double w = rand.nextDouble();
    graph.add(a, b, w);
   }
  }
  catch (Exception ex)
  {
   System.out.println(ex);
  }
  return graph;
 }

 private static <T> boolean hasHamiltonian(IGraph<T> graph)
 {
  int n = graph.getSize() / 2;
  for (T t : graph)
   if (graph.getPredecessors(t).getSize() + graph.getSuccessors(t).getSize() <= n)
    return false;
  return true;
 }
```

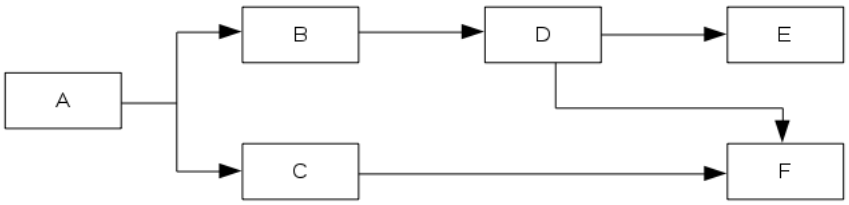
The method is not effective as it tries to add random edges until the degree of all vertices is sufficiently large, but after that the graph, according to the theory, should have a Hamiltonian cycle. The test method *test3()* tests *nearestNeigbor()*, but so that for each graph, all vertices are tried as start vertex until a solution is found – if one exists. Although a graph has a Hamiltonian cycle, it is not certain that *nearestNeigbor()* finds it.

If you finds that the method is slow, it is not due to *nearestNeigbor()* but instead *build3()*. Here you can try changing the loop to something like the following:

```
for (int i = 0, m = n * n / 4; i < m; ++i)
```

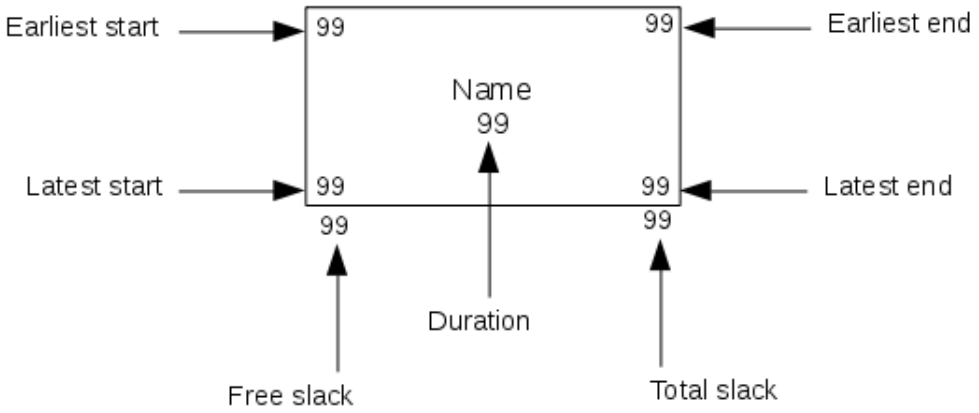
10 FINAL EXAMPLE: ACTIVITYPLANNER

To manage and follow up the time spent in the implementation of projects, time charts are often used as tools. The time chart consists of a series of activities linked with arrows to show the interdependence of the activities, for example:



The diagram is read as follows: Activity A must be completed before activities B and C can start. Activity D can not start until activity B is complete. B is the direct predecessor of

D and A is thus an indirect predecessor of D. There are several terms or values associated with an activity as shown in the following figure:



and I will use the following abbreviations:

- An activity: A
- Duration: D
- Latest, direct predecessor: P
- Earliest, immediate successor: S
- Earliest stat: Es
- Earliest end: Ee
- Latest start: Ls
- Latest end: Le
- Free slack: Fs
- Total slack: Ts

The project's start time is set to 0. It is applied as the earliest start time for all boxes representing the activities that can be started at the beginning of the project because they are without predecessors.

Then, for each of these activities, the calculated earliest end time is determined and added to the boxes. Earliest end time of an activity is found by adding the earliest start time and the duration ($Ee = D + Es$).

Earliest start time for other activities are determined by transferring the earliest end time of the most recent direct predecessor ($Es = Max\ (Ee, Ee, \ldots \mid all\ predecessors$). That is, all direct predecessors are expected to be completed before an activity can start. Then the earliest end time can be calculated as above.

The duration of the project is the largest earliest end time. The calculated project duration is considered as the most recent acceptable end time of the activities that constitute the completion of the project, that is all activities that have no successors.

The duration of the project is applied to all activities without successors as the last end time, after which the last start time for these activities can be calculated by subtracting the duration of the activity from the last end time (*Ls = Le-D*).

Most recent start and end times for the other activities can now be calculated by scanning all activities from right to left. The last end of an activity is determined by transferring the most recent start time of the earliest direct successor (*Le = Min (Ls, Ls, … | all successors*). The last start time of the other activities is found by subtracting the duration of the activity from the activity's latest end time.

Critical activities, critical roads and slacks

After the calculation of the most recent times, the diagram shows which activities can be temporarily postponed and what activities are fixed. The last category of activities that can be recognized by the fact that the earliest and most recent end times for each activity are the same, and they are also called critical activities as they constitute one or more critical paths through the network from start to finish. It is characteristic of a critical path that

1. it is measured by time the longest way in the network
2. if the duration of one or more of the activities on a critical path is extended, the duration of the project will be extended accordingly

The maximum displacement option for each activity is called the total slack. All activities on a critical path therefore have a total slack = 0.

The total slack is calculated on condition that all predecessors are completed as early as possible and all successors begin as soon as possible. The calculation of the total slack for an activity is done by subtracting the earliest end time from the latest end time (*Ts = Le – Ee*).

A delay in a critical network activity causes a corresponding extension of the project's duration, unless the duration of one or more of the subsequent critical activities is reduced correspondingly to their planned duration. Activities with a small total slack must therefore be monitored particularly carefully during the project time.

In the same way as it is possible to calculate the displacement option of an activity, assuming that all predecessors are completed as early as possible and all successors begin as soon as possible, it is also possible to calculate the displacement option of an activity, provided that all successors begin as early as possible. This displacement option is called the free slack. The calculation of the free slack for an activity is done by subtracting the activity's earliest end time from the earliest starting time of the earliest direct successor:

$Fs = Min(Es, Es, \ldots \mid all\ successors) - Ee.$

The free slack is used in the monitoring of the project. As you usually use the earliest start and end times when completing a project, it may be nice to know which displacement options the activities also have under these assumptions.

10.1 TASK DESCRIPTION

The task now is to write a program that can be used as a tool for calculating and printing information about a project and its activities. That is, for each activity

1. name
2. duration
3. earliest start
4. earliest end
5. latest start
6. latest end
7. total slack
8. free slack
9. the activity's direct successors

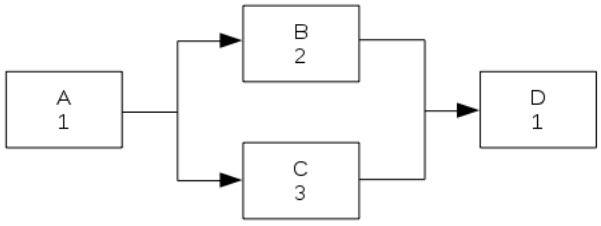
and for the whole project

1. the project's duration
2. the critical roads

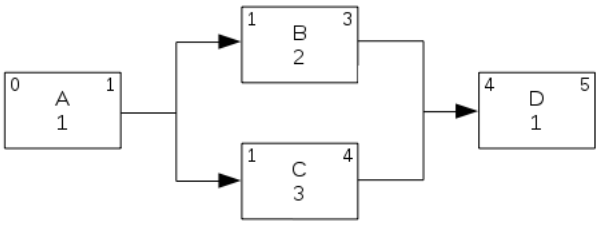
Calculation of times and time reserves can only be done if there is no cycle in the schedule. If that is the case, a message must be printed.

10.2 ANALYSIS

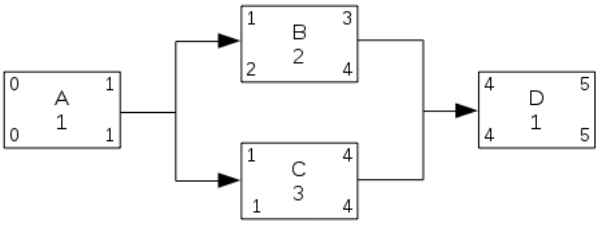
I will start with simple simple examples of activity charts and how to perform the above calculations. The first chart consists solely of four activities where there is a single start activity:



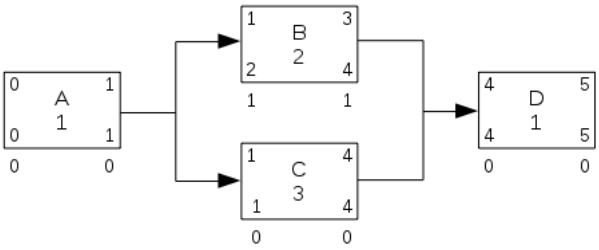
Below is the chart after I have determined the earliest start times and the earliest end times:



Below is the chart after I have determined the latest end times and the latest start times:



Finally, I have shown the chart below after I have calculated slack:



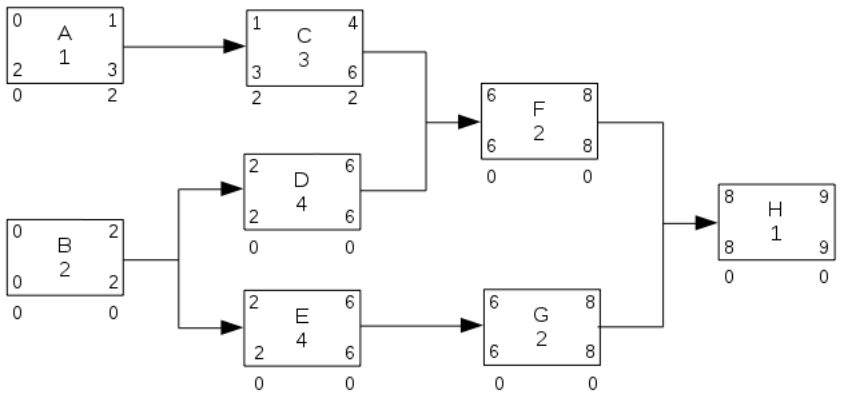
The result is thus:

Activity	Duration	Es	Ee	Ls	Le	Fs	Ts	Succes.
A	1	0	1	0	1	0	0	B C
B	2	1	3	2	4	1	1	D
C	3	1	4	1	4	0	0	D
D	1	4	5	4	5	0	0	

Duration of the entire project: 5

There is a single critical path: A C D

Below is another example after all times are filled out:



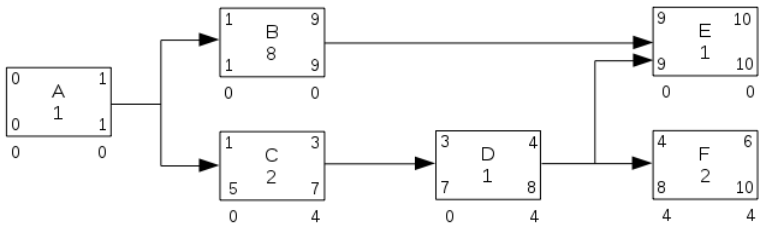
In this diagram there are two start activities that can be started in parallel. The result is as follows:

Activity	Duration	Es	Ee	Ls	Le	Fs	Ts	Succes.
A	1	0	1	2	3	0	2	C
B	2	0	2	0	2	0	0	D E
C	3	1	4	3	6	2	2	F
D	4	2	6	2	6	0	0	F
E	4	2	6	2	6	0	0	G
F	2	6	8	6	8	0	0	H
G	2	6	8	6	8	0	0	H
H	1	8	9	8	9	0	0	

Project Duration: 9

Critical paths: BEGH and BDFH

As a last example, I will show a diagram with two end activities:



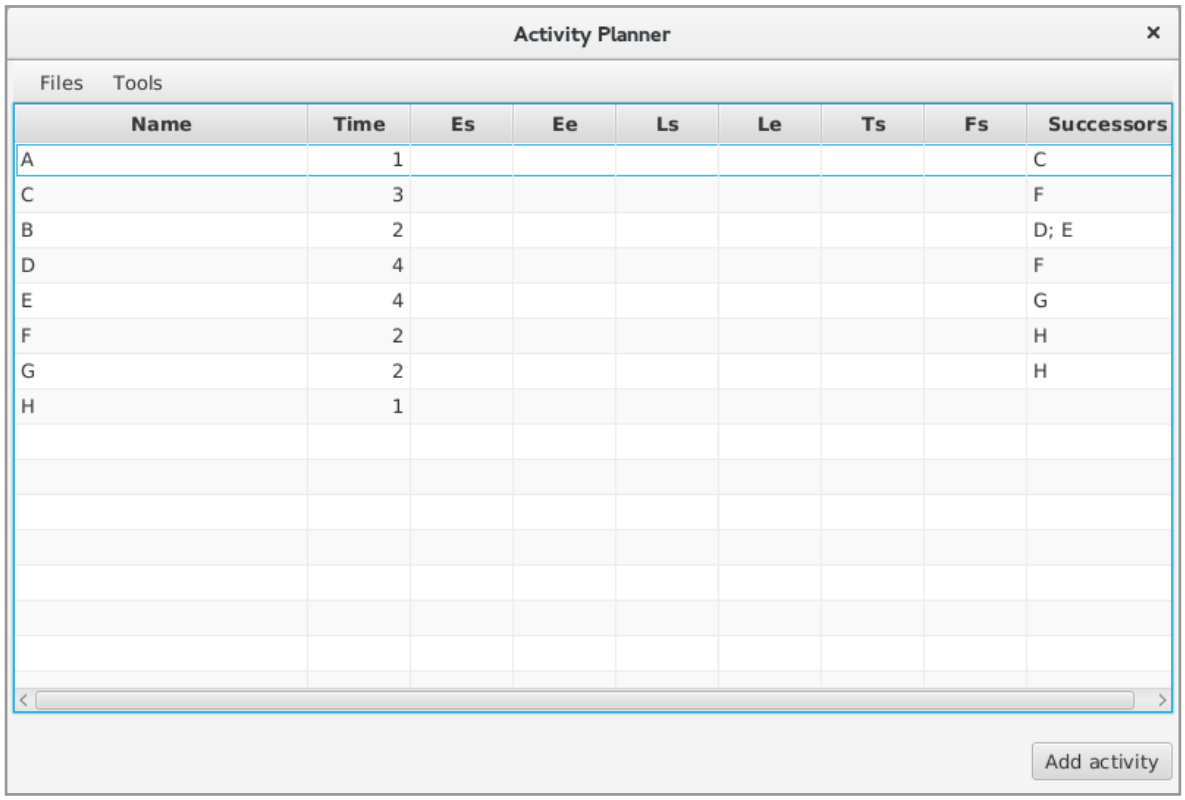
Activity	Duration	Es	Ee	Ls	Le	Fs	Ts	Succes.
A	1	0	1	0	1	0	0	B C
B	8	1	9	1	9	0	0	E
C	2	1	3	5	7	0	4	D
D	1	3	4	7	8	0	4	E F
E	1	9	10	9	10	0	0	
F	2	4	6	8	10	4	4	

Project duration: 10

Critical path: ABE

10.3 THE USER INTERFACE

The program's user interface consists of a simple *TableView*, and the result could be the following, which shows the second of the above examples:



The window has a menu that must have the following features:

Files

- New
- Open
- Save
- Save as

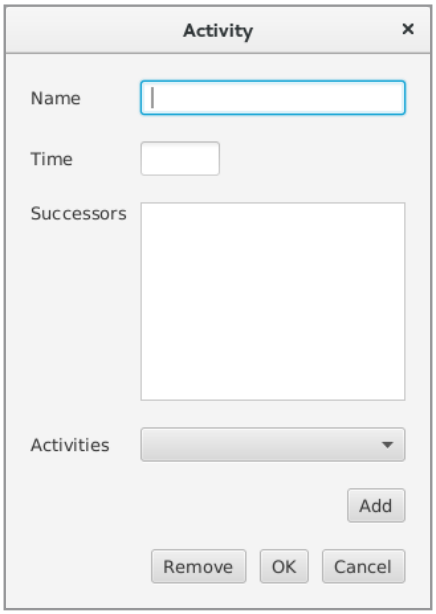
Tools

- Add activity
- Calculate
- Show result

where the meaning of the *Files* menu should be self explanatory. As for the *Tools* menu, the first menu item (and should then have the same function as the button) is used to create a

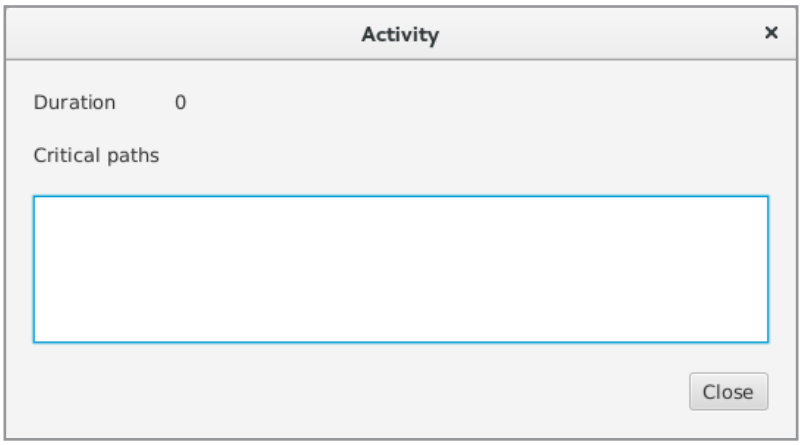
new activity, while the second function is used to fill the activity chart. The third function is used to show the total time and critical paths.

The button and corresponding menu item opens the following dialog box:



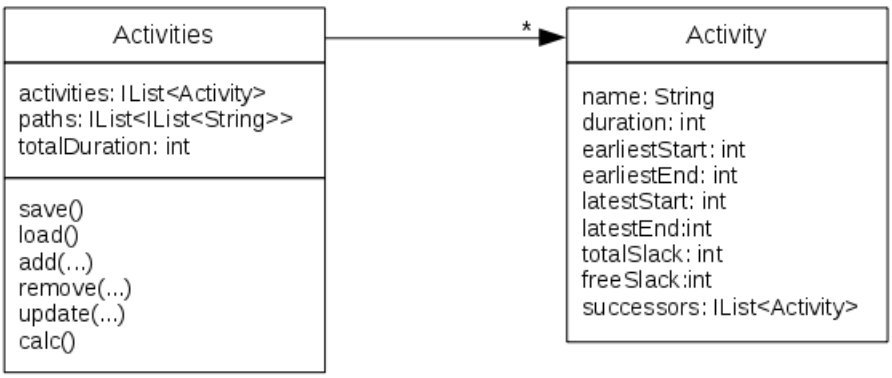
where you should be able to create an activity by entering name and time, and additionally you should be able to add successors. The same dialog box should be used to edit an existing activity. The name of the activity must be unique as it is perceived as a key.

Finally, the program must have the following dialog box, which shows the result after the chart has been calculated:



10.4 DESIGN

The program's model should basically consist of two classes, which represent an activity and a collection of activities respectively:



Here is the class *Activity*, a rather simple model class that has variables for the attributes associated with an activity. The Activities class is in principle a list of *Activity* objects. In addition, there is a list of critical paths and a variable that indicates the duration of the entire project. Otherwise, the class's methods will allow you to save and open activity charts. In addition, there are methods for maintaining the plan, and finally there is a method that calculates the times.

To save an activity plan, it is decided for each activity to save only the name of the activity, its duration and successors. All variables relating to calculations are not saved, and it is the same for the critical paths and the total duration of the project. The reason is to make it simple to save an activity plan, but then you need to recalculate the plan for each load. An activity plan is saved as a simple single text file with one line for each activity, and the above example is saved as:

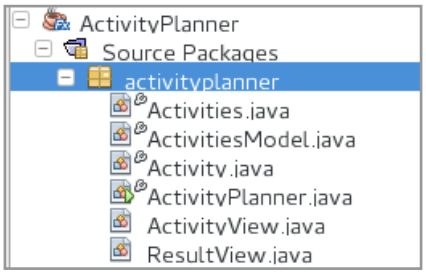
```
A;1;C
C;3;F
B;2;D;E
D;4;F
E;4;G
F;2;H
G;2;H
H;1
```

The method *calc()* must calculate the earliest and most recent times as described during the analysis. First, the activities must be sorted as in the shown diagrams. An activity plan can be perceived as an oriented graph, where each activity is a vertex, and a successor represents an edge. You can therefore arrange the activities by creating a corresponding graph and performing a topological sorting. Then the calculations corresponding to what is shown during the analysis can be done as

1. Calculate the earliest times at an iteration from start to finish
2. Calculate the latest times in an iteration in the opposite direction
3. Calculate the total slack and the free slack in an iteration from start to finish
4. Determine the critical paths starting with activities where the earliest start is 0 and the total slack is 0 and determine the critical paths using backtracking

10.5 PROGRAMMERING

The completed program consists of 6 classes:



I do not want to show the code here, but the most important classes are the implementation of the two model classes, and here you should especially study the class *Activities* and how the class *Graph* is used.

For the test, the book's files contain a directory that contains 6 files with activity plans.