# Implementation of algorithms and data structures

## 1. seminar

Jirka Fink

https://ktiml.mff.cuni.cz/~fink/

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague

Winter semestr 2025/26
Last change 2. října 2025

Licence: Creative Commons BY-NC-SA 4.0

# The course

## Goals

- Learn how to implement advanced algorithms and data structures without tedious debugging
- Learn how to write and use various tests
- Learn to structure code and write API

## Entrance expectations

- Knowledge of some programming language (e.g. C/C++, Python, Java, C#)
- Theoretical knowledge algorithms and data structures from bachelor study
- Experience in implementing basic algorithm (e.g. graph search)

# Communication

## Web

`https://ktiml.mff.cuni.cz/~fink/`

## E-mail

fink@ktiml.mff.cuni.cz

## Recodex

- Enroll into my group on Recodex
- Submit your fully working programs here

## Gitlab

- Create a private fork of the repository
  `https://gitlab.mff.cuni.cz/finkj1am/implementation`
- Give me (finkj1am) access (reporter)

# Passing conditions

## Implement

- 3 testing assignments
- Red-black tree
- Goldberg algorithm for network flow problem

Bonus: Blossom algorithm for maximum matching in general unweighted graphs

## Assignments

1. **Red-black trees**
   Martin Mareš, Tomáš Valla: Průvodce labyrintem algoritmů, CZ.NIC, 2017
   Robert Sedgewick: Left-leaning Red-Black Trees, doi:10.1.1.139.282
2. **Network flows (Goldberg algorithm)**
   Martin Mareš, Tomáš Valla: Průvodce labyrintem algoritmů, CZ.NIC, 2017
   Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein: Introduction to algorithms, The MIT Press, 2001
   `http://mitp-content-server.mit.edu:18180/books/content/sectbyfn?collid=books_pres_0&id=8030&fn=Chapter%2026.pdf`
3. **Maximum matching in general unweighted graphs (Blossom algorithm)**
   Cunningham, Cook, Pulleyblank, Schrijver: Combinatorial optimization, John Wiley & Sons, 1997

# Type of tests

- Unit testing
- Fuzzy testing
- Integration testing
- System testing
- Acceptance testing
- Installation testing
- Regression testing
- Continuous testing
- Destructive testing
- Software performance testing
- Security testing
- VCR testing
- Internationalization and localization
- ...

# Testing methods

## Black-box testing

- No knowledge of codes is used
- Test fulfilling the specification
- First write tests, code later

## White-box testing

- Requires knowledge of codes
- Covers every part of code
- Reversed engineering: Searching for dangerous inputs for our code
- Verifies the internal structures

## Testing reported bugs

For a reported bug, a test is created before fixing.

# Test-driven development

## Development cycle

1. Write tests
2. Run tests and check that all fails
3. Implement the program
4. Use tests for debugging
5. Refactorization and clean up
6. Write documentation

## Advantages

- Tests may contain bugs, we verify that tests fails as expected
- Writing unit tests verifies usability of interface

# Unit testing

## Motivation

Verify correctness of new features and preserving functionality after changes in codes.

## Description

- Software testing method by which individual units of source code are tested to determine whether they are fit for use
- Tests should be independent

## Advantages

- Unit tests can be run repeatedly
- Fast discovery of a bug when code is changed
- Example of usage of a library

## Limitations and disadvantages

- Unit tests only proves that a program contains a bug
- Unit tests cannot prove that a program is correct (halting problem)
- Unit tests are not supposed to verify integration of modules
- Unit tests uses API without verifying correctness of internal data

# Unit testing: Sum of two numbers

```
1  public class TestAdder {
2    @Test
3    public void testSumPositiveNumbersOneAndOne() {
4      Adder adder = new AdderImpl();
5      TEST(adder.add(1, 1) == 2);
6    }
7    @Test
8    public void testSumPositiveNumbersOneAndTwo() {
9      Adder adder = new AdderImpl();
10     TEST(adder.add(1, 2) == 3);
11   }
12   @Test
13   public void testSumPositiveNumbersTwoAndTwo() {
14     Adder adder = new AdderImpl();
15     TEST(adder.add(2, 2) == 4);
16   }
17   @Test
18   public void testSumZeroNeutral() {
19     Adder adder = new AdderImpl();
20     TEST(adder.add(0, 0) == 0);
21   }
22   ...
23 }
```

# Unit testing: Graph

```
1 public abstract class AbstractGraphTest {
2   MutableGraph<Integer> graph;
3   @Before
4   public void init() {
5     graph = createGraph();
6   }
7   @After
8   public void validateGraphState() {
9     validateGraph(graph);
10  }
11  @Test
12  public void nodes_oneNode() {
13    addNode(N1);
14    TESTThat(graph.nodes().containsExactly(N1);
15  }
16  @Test
17  public void nodes_noNodes() {
18    TESTThat(graph.nodes().isEmpty();
19  }
20  @Test
21  public void adjacentNodes_oneEdge() {
22    putEdge(N1, N2);
23    TESTThat(graph.adjacentNodes(N1)).containsExactly(N2);
24    TESTThat(graph.adjacentNodes(N2)).containsExactly(N1);
25  }
26 }
```

Source: Wikipedia: Unit testing
https://github.com/google/guava/blob/master/guava-tests/test/com/google/common/graph/AbstractGraphTest.java

# How create unit tests?

## Initial questions about unit tests

- How should a unit be tested?
  E.g. which library should be used to write tests?
- What should be tested?
  E.g. what should be the content of unit tests?

## Libraries for unit tests

- Python: unittest — Unit testing framework
- Julia: Unit testing (standard library)
- Java: JUnit 5
- C#: Unit test basics
- C/C++: Tens of libraries

Wikipedia: List of unit testing frameworks

# What should be tested?

## General hints

- Write test verifying that task is fulfilled
- Test boundary cases

## Test sorting function

```
1 // First, small simple tests
2 TEST(sort([5,7,9,3]) == [3,5,7,9])
3 TEST(sort(['d','a','z']) == ['a','d','z'])
4 TEST(sort(["one","two","three"]) == ["one","three","two"])
5
6 // Tricky cases, some cases may depend on documentation
7 TEST(sort([]) == [])
8 TEST(sort([1,2,1,2,1]) == [1,1,1,2,2])
9 TEST(sort([1,False,5,True]) == [False, 1, True, 5])
```

Creating larger and random tests will be discussed later.

# What should be tested?

## Examples of tests of binary search trees

1. Create a new empty tree and destroy it
2. Create a new empty tree, insert one element and destroy the tree
3. Insert more elements
4. Delete some elements
5. Delete all elements
6. Combine insertion and deletions
7. Check the counter of the number of elements
8. Find existing and non-existing elements
9. Insert existing and delete non-existing elements
10. Find and delete an element in an empty tree

# Testing consistency of stored data

## Disadvantages of unit tests

- In non-trivial situations, all cases cannot be tested
- When a unit test fails, it does not say where a bug is
- Unit tests does not verify the correctness of stored data
- E.g. insertion of an element may be incorrect, but a bug may occur when it is deleted

## What can be tested in stored data?

- All conditions given in a definition of a data structure
- Invariants
- Properties implied by proofs of correctness of an algorithm
- Values of variables which can be computed from other data
  e.g. number of elements in a tree
- Values of all variables have expected values
  e.g. range of integers, enumerators

# Example: Doubly linked list

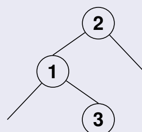
```
1 void list_test(list *l) {
2   if(!l->first) { // List is empty
3     TEST(!l->last);
4     return;
5   }
6   TEST(l->last);
7   TEST(!l->first->prev);
8   TEST(!l->last->next);
9   for(node *n = l->first; n; n = n->next) {
10    if(n != l->first)
11      TEST(n->prev && n->prev->next == n);
12    if(n != l->last)
13      TEST(n->next && n->next->prev == n);
14  }
15 }
```

# Example: binary search tree

```
1 void tree_test(tree *tree) { node_test(tree->root); }
2
3 void node_test(node *node) {
4   if(!node)
5     return;
6
7   TEST(!node->left || node->left->parent == node);
8   TEST(!node->right || node->right->parent == node);
9
10  TEST(!node->left || node->left->key <= node->key);
11  TEST(!node->right || node->right->key >= node->key);
12
13  node_test(node->left);
14  node_test(node->right);
15 }
```

## Question

Does this test guarantee that a binary search tree satisfying it is correct?



# Testing order in a binary search tree

```
1 // Returns a pair of the minimal and the maximal key in the subtree
2 pair<int,int> order_test(node *node) {
3   int min = node->key, max = node->key, cmp;
4
5   if(node->left) {
6     min,cmp = order_test(node->left);
7     TEST(cmp <= node->key);
8   }
9
10  if(node->right) {
11    cmp,max = order_test(node->right);
12    TEST(node->key <= cmp);
13  }
14
15  return pair(min, max);
16 }
```

# Example: Priority queue in an array

```
1 void heap_test(queue *q) {
2   for(int i = 1; i < q->size; i++)
3     // Parent is stored on position floor((i-1)/2)
4     TEST(q->array[i].priority > q->array[(i-1)/2].priority);
5 }
```

# More examples

## Hash table with separate chains

- A linked list in every bucket is correct
- Compute hash of every element to test whether it is stores in the proper bucket
- The ratio of the number of element to buckets is within expected range

## Graphs

- Incidence lists contains expected values
  e.g. indices of vertices are within expected range or pointers give vertices inside the graph
- Every edge is a member of incidence lists of both end-vertices

# The first assignment

## AVL tree

Binary search tree where for every vertex, the difference between heights of the left and the right subtree is at most one.

## Write tests verifying consistency of AVL tree

- There is a template on gitlab
- Implement one function testing consistency of AVL tree
- Some unit tests are given, but there are more of them on recodex