Implementation of algorithms and data structures

4. 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 22. října 2025

Testing intermediate results

- When a program returns an incorrect result, how do we find a bug?
- A unit test gives us input to reproduce an error but only limited information where the bug is.
- For data structures, we can test their internal structure after every operation.
- Can we verify results of intermediate steps of an algorithm?
- Can we also verify intermediate states of data structure during complex operation?

Quicksort

```
1  function quicksort(array a of length n) {
2     stack = [(1,n)]
3     while(stack is non-empty) {
4        i,j = stack.pop()
5        m = choose_pivot_and_split(a,i,j)
6        test_split(a,i,j)
7        if(i < m) stack.push((i,m))
8        if(m+1 < j) stack.push((m+1,j))
9        test_stack(a, stack)
10    }
11 }
```

Quicksort

```
1  function quicksort(array a of length n) {
2     stack = [(1,n)]
3     while(stack is non-empty) {
4        i,j = stack.pop()
5        m = choose_pivot_and_split(a,i,j)
6        test_split(a,i,j)
7        if(i < m) stack.push((i,m))
8        if(m+1 < j) stack.push((m+1,j))
9        test_stack(a, stack)
10    }
11 }
```

What should be tested after splitting

- $i \leq m \leq j$
- All elements between i and m are smaller than all elements between m+1 and j
- The array contains exactly given elements

Quicksort

```
1  function quicksort(array a of length n) {
2     stack = [(1,n)]
3     while(stack is non-empty) {
4        i,j = stack.pop()
5        m = choose_pivot_and_split(a,i,j)
6        test_split(a,i,j)
7        if(i < m) stack.push((i,m))
8        if(m+1 < j) stack.push((m+1,j))
9        test_stack(a, stack)
10    }
11 }
```

What should be tested after splitting

- $i \leq m \leq j$
- All elements between i and m are smaller than all elements between m+1 and j
- The array contains exactly given elements

How stack can be tested?

Quicksort

```
1  function quicksort(array a of length n) {
2     stack = [(1,n)]
3     while(stack is non-empty) {
4        i,j = stack.pop()
5        m = choose_pivot_and_split(a,i,j)
6        test_split(a,i,j)
7        if(i < m) stack.push((i,m))
8        if(m+1 < j) stack.push((m+1,j))
9        test_stack(a, stack)
10    }
11 }
```

What should be tested after splitting

- $i \leq m \leq j$
- All elements between i and m are smaller than all elements between m+1 and j
- The array contains exactly given elements

How stack can be tested?

- $stack[z][1] < stack[z][2] < stack[z+1][1]$ for every z in the stack (except last)
- When we replace every subarray $stack[z][1] \ldots stack[z][2]$ by its minimal and maximal elements, the resulting array must be sorted

Testing intermediate results

- Quicksort, Mergesort, Heapsort
- Knuth-Morris-Pratt, Aho-Corasick
- Shortest path
- Minimum spanning tree
- Maximum flow (Ford-Fulkerson, Dinic)
- Red-black trees

Using SAT/CSP/ILP solvers

- We have a search or optimization problem
  - e.g. coloring, matching, scheduling

Using SAT/CSP/ILP solvers

- We have a search or optimization problem
  - e.g. coloring, matching, scheduling
- Instead of writing a specialized program, we can use SAT/CSP/ILP solvers
  - Problem is described using variables and constraints

Using SAT/CSP/ILP solvers

- We have a search or optimization problem
  - e.g. coloring, matching, scheduling
- Instead of writing a specialized program, we can use SAT/CSP/ILP solvers
  - Problem is described using variables and constraints
- Boolean satisfiability
  - Variables are logical
  - Constraints are disjunctive clauses

Using SAT/CSP/ILP solvers

- We have a search or optimization problem
  - e.g. coloring, matching, scheduling
- Instead of writing a specialized program, we can use SAT/CSP/ILP solvers
  - Problem is described using variables and constraints
- Boolean satisfiability
  - Variables are logical
  - Constraints are disjunctive clauses
- Constraint satisfaction programming
  - Variables have finite domains
  - Constraints restrict assignments for subsets of variables

Using SAT/CSP/ILP solvers

- We have a search or optimization problem
  - e.g. coloring, matching, scheduling
- Instead of writing a specialized program, we can use SAT/CSP/ILP solvers
  - Problem is described using variables and constraints
- Boolean satisfiability
  - Variables are logical
  - Constraints are disjunctive clauses
- Constraint satisfaction programming
  - Variables have finite domains
  - Constraints restrict assignments for subsets of variables
- Integer linear programming
  - Variables are real numbers
  - Constraints are linear inequalities

Testing and debugging SAT/CSP/ILP solvers

- Possible problems
  - A solver finds a solution not satisfying our conditions
  - A solver claims that our model has no solution, although we have one
  - An ILP solver finds a suboptimal solution
- How to find the bug and write tests?