Implementation of algorithms and data structures

6. 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 26. listopadu 2025

Why we should avoid using recursion?

Why we should avoid using recursion?

- Recursion may cause stack overflow
- Recursion may have significant overhead
- Recursive algorithm may have exponential complexity

Exponential complexity

```
1 function fibonacci(n) {
2    if n <= 1
3       return n
4    return fibonacci(n-1) + fibonacci(n-2)
5 }
```

If possible, use Dynamic programming and store previous results.

Stack overflow

Examples of stack overflow

- Binary search trees with linear depth; e.g. Splay trees
- Quicksort

Stack overflow

Examples of stack overflow

- Binary search trees with linear depth; e.g. Splay trees
- Quicksort

Exercise

Consider a binary tree with nodes containing

- Key
- Left, Right, Parent pointers to nodes
- Number of elements in the subtree

Without recursion, verify consistency of a given tree.

Quicksort with stack of size $O(\log n)$

Quicksort with stack of size $O(\log n)$

```
1  stack ← [ ]
2  i ← 1
3  j ← n
4  while true do
       # Sort subarray i...j
5      m ← split(i, j)                          # Splits into i...m and m+1...j
6      if m − i < j − m − 1 then
           # The first subarray is shorter
7          if m + 1 < j then
8              stack.push((m + 1, j))                # Push the longer subarray
9          j ← m                                 # Continue on the shorter one
10     else
           # The second subarray is shorter
11         if i < m then
12             stack.push((i, m))                    # Push the longer subarray
13         i ← m + 1                             # Continue on the shorter one
14     if i == j then
15         if stack is non-empty then
16             (i, j) ← stack.pop()
17         else
18             return
```

- The length of the subarray stack[l] is at most $\frac{n}{2^{l-1}}$ for every l
- If stack contains k subarrays, then $j - i \leq \frac{n}{2^k}$

Introsort: Quicksort with $O(n \log n)$ worst-case complexity

Introsort: Quicksort with $O(n \log n)$ worst-case complexity

Introsort

Whenever the recursion of Quicksort reaches the depth $2 \log n$, use Heapsort instead.

Introsort: Quicksort with $O(n \log n)$ worst-case complexity

Introsort

Whenever the recursion of Quicksort reaches the depth $2 \log n$, use Heapsort instead.

Worst-case time complexity

- Quicksort excluding Heapsorts needs $O(n \log n)$
- Quicksort splits the input into subarrays of length $n_1, \ldots, n_k$
- Time complexity of all Heapsorts is $\sum_{i=1}^{k} n_i \log n_i \leq \log n \sum_{i=1}^{k} n_i = O(n \log n)$

Time complexity of Introsort is $O(n \log n)$

Tail recursion optimization

Some compilers can eliminate recursion if a function is tail recursive

```
1 function find_in_binary_search_tree(node, key) {
2    if node == NULL or node->key == key
3       return node
4    child = key < node->key ? node->left : node->right
5    return find_in_binary_search_tree(child, key)
6 }
```

Tail recursion optimization

Some compilers can eliminate recursion if a function is tail recursive

```
function find_in_binary_search_tree(node, key) {
   if node == NULL or node->key == key
      return node
   child = key < node->key ? node->left : node->right
   return find_in_binary_search_tree(child, key)
}
```

However, some recursion may be impossible to eliminate

```
function factorial(n) {
   if n <= 2
      return n
   return n * factorial(n-1)
}
```