Tutorial 12

Data Structures 1, 23. 5. 2025 `https://iuuk.mff.cuni.cz/~chmel/2425/ds1en/`

Geometric data structures

Exercise 1 (Worst case for a k-d tree)
Consider a 2-d tree, i.e., a binary tree where we split the plane alternately along the x and y axes. Show that a two-dimensional range query (counting the number of points in a rectangle) can take up to $\Omega(\sqrt{n})$ time. Specifically, find a set of points stored in the tree and a query that will take this long while not finding any points.
Bonus: try to generalize this to arbitrary k-d trees with time complexity $\Omega(n^{1-\frac{1}{k}})$.

Exercise 2 (Finding Minimum and Deletion in a k-d Tree)
We have a k-d tree. Design the operation of finding the minimum element in a given dimension. (That is, a function FindMin(d) that finds the element whose d-th coordinate is minimal.)
Next, design the implementation of the Delete operation. For simplicity, you may assume that no two points share any coordinate.
(In both cases, do not worry about time complexity.)

Exercise 3 (Nearest Neighbor and k-NN)
We have a k-d tree. Design how to implement the operation of finding the nearest neighbor, and the operation of finding the m nearest neighbors.
Hint: for m-NN, a heap might be useful.

Parallel data structures

Exercise 4 (BankAccountFactory)
We have this pseudocode in “Java” for a class that simulates a bank account. Assume that Lock is a class that corresponds to a standard lock, that *cannot be locked multiple times.* In such case, you would get stuck on waiting for yourself to unlock the lock. Show that in that case, you can create a deadlock.

```
class BankAccount {
 private int balance = 0;
 private Lock lk = new Lock();

 int getBalance() {
  lk.acquire();
  int ans = balance;
  lk.release();
  return ans;
 }

 void setBalance(int x) {
  lk.acquire();
  balance = x;
  lk.release();
 }

 void withdraw(int amount) {
  lk.acquire();
  int b = getBalance();
  if(amount > b) {
   lk.release();
   throw new WithdrawTooLargeException();
  }
  setBalance(b - amount);
```

```
   lk.release();
   }
 }
```

Your colleague fixed this: show that the function `newWithdraw` that replaces `withdraw` cannot deadlock.

```
 void newWithdraw(int amount) {
  lk.acquire();
  lk.release();
  int b = getBalance();
  lk.acquire();
  if(amount > b) {
   lk.release();
   throw new WithdrawTooLargeException();
  }
  lk.release();
  setBalance(b - amount);
  lk.acquire();
  lk.release();
 }
```

Finally, show that even if this new version cannot deadlock, it is not correct: it might happen that some withdrawal can get "lost".

Bonuses

Exercise 5 (Can we reach?)
Construct a probabilistic algorithm[1] for deciding reachability (i.e., answering the question "is there a path from u to v?"), which fails with probability at most 1/4 and uses only logarithmic space.
Technical note on logarithmic space: when we have an algorithm running in sublinear space, we assume the input is on a "read-only" tape, where we can access it but not modify it (but we can freely copy from it as needed).[2]
You may find the following claim useful: in a connected undirected graph with n vertices, if we start from vertex u and at each step choose uniformly at random one of the neighbors to move to next, the expected number of steps to reach another vertex v is at most $2n^3$.
It might also help to recall Markov's inequality: Let X be a non-negative random variable. Then for all $\varepsilon > 0$, we have $P[X \geq \varepsilon] \leq \frac{\mathbb{E}[X]}{\varepsilon}$. Equivalently, for any $d \geq 1$, $P[X \geq d \cdot \mathbb{E}[X]] \leq \frac{1}{d}$.

Exercise 6 (Skiing)
You want to start skiing, but you don't own skis. At the resort, you have the option to rent skis for a day for 1 dollar, or to buy them for C dollars. ($C \in \mathbb{N}$)
On the other hand, you have a premonition that your nemesis will show up one early morning and break your legs, thus ending your skiing career. The problem is, you don't know which day that will happen. So you only have one choice: each morning you find out whether something happened to you, and then decide whether to rent or buy the skis (if you haven't already bought them — once you do, you can ski freely). However, you'd like to use your money as efficiently as possible.
Find a deterministic online algorithm that will always pay at most twice as much as the optimum (which knows in advance on which day d the sabotage will occur). Can you also show that your online algorithm is optimal in terms of competitiveness (i.e., that no deterministic online algorithm can always pay less than $2 - \frac{1}{C}$)?

[1]You've probably already encountered probabilistic algorithms, but briefly: this is an algorithm that can make random decisions at each step from among several options

[2]Technically, this class is defined on Turing machines with a read-only input tape and a separate work tape. Only the space used on the work tape counts toward space complexity.