# DATA STRUCTURES AND ALGORITHMS

## LECTURE 14

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2023 - 2024

# In Lecture 13...

- AVL trees

# Today

- Applications
- Examples
- Recap
- Exam

# Applications of different containers / data structures

- In seminar 5 we have already talked about one such application: evaluating an arithmetic expression, which is an example of using stacks and queues.
- In Lecture 8 we also mentioned some applications of stacks and queues (finding the exit in a maze, card game).
- Now we are going to talk about two other applications where different: bracket matching and the Huffman encoding.

# Bracket matching

- Given a sequence of round brackets (parentheses), (square) brackets and curly brackets, verify if the brackets are opened and closed correctly.
- For example:
  - The sequence ()([][][(())]) - is correct
  - The sequence [()()()()] - is correct
  - The sequence [()]) - is not correct (one extra closed round bracket at the end)
  - The sequence [(]) - is not correct (brackets closed in wrong order)
  - The sequence {[[]] () - is not correct (curly bracket is not closed)

# Bracket matching - Solution Idea

- Stacks are suitable for this problem, because the bracket that was opened last should be the first to be closed. This matches the LIFO property of the stack.
- The main idea of the solution:
  - Start parsing the sequence, element-by-element
  - If we encounter an open bracket, we push it to a stack
  - If we encounter a closed bracket, we pop the last open bracket from the stack and check if they match
  - If they don't match, the sequence is not correct
  - If they match, we continue
  - If the stack is empty when we finished parsing the sequence, it was correct

# Bracket matching - Extension

- How can we extend the previous idea so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
    - Open brackets that are never closed
    - Closed brackets that were not opened
    - Mismatch
- Keep count of the current position in the sequence, and push to the stack $< delimiter, position >$ pairs.

# Huffman coding

- The *Huffman coding* can be used to encode characters (from an alphabet) using variable length codes.
- In order to reduce the total number of bits needed to encode a message, characters that appear more frequently have shorter codes.
- Since we use variable length code for each character, *no code can be the prefix of any other code* (if we encode letter E with 01 and letter X with 010011, during decoding, when we find a 01, we will not know whether it is E or the beginning of X).

# Huffman coding

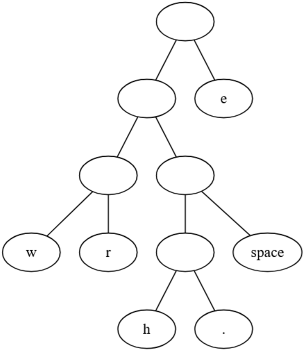
- When building the Huffman encoding, we need to have a message that we want to encode.
- First we have to compute the frequency of every character from the message, because we are going to define the codes based on the frequencies.
- Assume that we have the following message: WE WERE HERE.
- It contains the following characters and frequencies:

| w | e | r | h | space | . |
|---|---|---|---|---|---|
| 2 | 5 | 2 | 1 | 2 | 1 |

# Huffman coding

- For defining the Huffman code a binary tree is build in the following way:
  - Start with trees containing only a root node, one for every character. Each tree has a weight, which is frequency of the character.
  - Get the two trees with the least weight (if there is a tie, choose randomly), combine them into one tree which has as weight the sum of the two weights.
  - Repeat until we have only one tree.
- The implementation can simply be done with a Priority Queue which stores these partially built trees (and considers the weight as priority).

# Huffman coding



# Huffman coding

- Code for each character can be read from the tree in the following way: start from the root and go towards the corresponding leaf node. Every time we go left add the bit 0 to encoding and when we go right add bit 1.

- Code for the characters:
    - w - 000
    - r - 001
    - h - 0100
    - . - 0101
    - space - 011
    - e - 1

- We can see that the most frequent character (e), indeed has the shortest code, and the least frequent ones have the longest. Also, no code is the prefix of another.

- In order to encode a message, just replace each character with the corresponding code.

# Huffman coding

- Assume we have the following code and we want to decode it: 0110110001000100111001000000
- We do not know where the code of each character ends, but we can use the previously built tree to decode it.
- Start parsing the code and iterate through the tree in the following way:
  - Start from the root
  - If the current bit from the code is 0 go to the left child, otherwise go to the right child
  - If we are at a leaf node we have decoded a character and have to start over from the root
- The decoded message (quotation marks added to see the spaces at the beginning): ”  wewereerww”

# (Bad) examples

- During the semester we have talked about the most frequently used container ADTs and the data structures which are used in general to implement them.

- In the following, I am going to show you 3 real-life examples (and one artificial one), where knowing data structures and containers, can help you write simpler / more efficient code.

# Example 1

- Consider the following problem (simplified version of a problem given a few years ago for the Bachelor exam):

- *You have an ADT Student which has a name and a city. Write a function which takes as input a list of students and prints for each city all the students that are from that city. Each city should be printed only once and in any order.*

- How would you solve the problem? What container would you use?

# Example 2

- Consider the following algorithm (written in Python and mentioned in our first Lecture as well):

```
def testContainer(container, l):
'''
container is a container with integer numbers
l is a list with integer numbers
'''
    count = 0
    for elem in l:
        if elem in container:
            count += 1
    return count
```

- The above function counts how many elements from the list *l* can be found in the container. What is the complexity of *testContainer*?

# Example 3

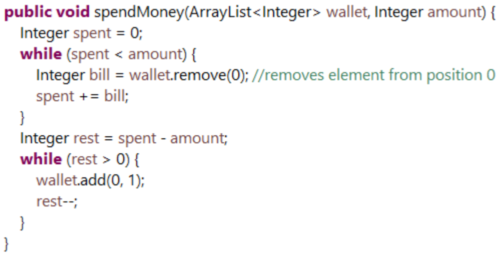
- Consider the following problem: *We want to model the content of a wallet, by using a list of integer numbers, in which every value denotes a bill. For example, a list with values [5, 1, 50, 1, 5] means that we have 62 RON in our wallet.*
  *Obviously, we are not allowed to have any numbers in our list, only numbers corresponding to actual bills (we cannot have a value of 8 in the list, because there is no 8 RON bill).*
  *We need to implement a functionality to pay a given amount of sum and to receive rest of necessary.*
  *There are many optimal algorithms for this, but we go for a very simple (and non-optimal): keep removing bills of the wallet until the sum of removed bills is greater than or equal to the sum you want to pay.*
  *If we need to receive a rest, we will receive it in 1 RON bills.*

# Example 3

- For example, if the wallet contains the values [5, 1, 50, 1, 5] and we need to pay 43 RON, we might remove the first 3 bills (a total of 56) and receive the 13 RON rest in 13 bills of 1.

# Example 3

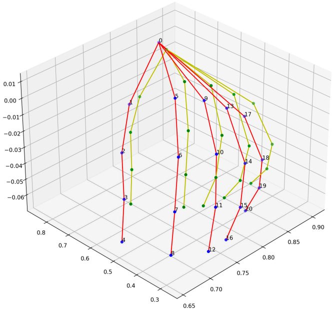
- For example, if the wallet contains the values [5, 1, 50, 1, 5] and we need to pay 43 RON, we might remove the first 3 bills (a total of 56) and receive the 13 RON rest in 13 bills of 1.
- This is a Java implementation provided by a student. ArrayList is an implementation of ADT List using a dynamic array as data structure. What is wrong with it?



```
public void spendMoney(ArrayList<Integer> wallet, Integer amount) {
    Integer spent = 0;
    while (spent < amount) {
        Integer bill = wallet.remove(0); //removes element from position 0
        spent += bill;
    }
    Integer rest = spent - amount;
    while (rest > 0) {
        wallet.add(0, 1);
        rest--;
    }
}
```

# Example 4

- Consider an application which processes a video recording of a hand gesture (for ex. thumbs up).
- The figure on the right is a visualization of a hand for which 20 key points are identified by a library (actually there are two hands, one with red and one with green, but only the red one has the numbers displayed).
- That library actually provides the coordinates (x, y, z) for all 20 points.



# Example 4

- The points are actually organized as a tree and the goal of the code below is to transform the coordinates of each key point (landmark in the code) to be relative to its parent.
- What is wrong with the code?

# Example 4

```
def transform_to_relative_joints(frame):
# Defining hand hierarchy starting from wrist, ending with each finger's tip
  parent_relationships = [
         (0, 1), (1, 2), (2, 3), (3, 4),  # Thumb
         (0, 5), (5, 6), (6, 7), (7, 8),  # Index finger
         (0, 9), (9, 10), (10, 11), (11, 12),   # Middle finger
         (0, 13), (13, 14), (14, 15), (15, 16),  # Ring finger
         (0, 17), (17, 18), (18, 19), (19, 20)]  # Little finger
  relative_joints = []
  #Processing each landmark of a frame
  for i, landmark in enumerate(frame):
    [x, y, z] = landmark
    parent_index = None
    # For a given landmark, search for its parent landmark/joint
    for parent_relationship in parent_relationships:
      if parent_relationship[1] == i:
        parent_index = parent_relationship[0]
        break

   if parent_index is not None:
   # When found, compute relative coordinates
       ...
       ...
  return relative_joints
```

# Recap

# Recap

- During the semester we have talked about the most important containers (ADT) and their main properties and operations
    - Bag, SortedBag, Set, SortedSet, Map, SortedMap, Multimap, SortedMultimap, List, SortedList, (Sparse)Matrix, Stack, Queue, Priority Queue and Deque, Binary Tree.
- We have also talked about the most important data structures that can be used to implement these containers
    - Dynamic array, Linked lists, Binary heap, Hash table (collision resolution with separate chaining, coalesced chaining, open addressing and linked hash table), Binary Search Tree, AVL Tree, Binary Tree.
- We have talked (briefly) about other data structures as well:
    - Skip list, Binomial heap, Cuckoo hashing, Perfect hashing.

# Exam organization

- Exam dates: 13, 17, 22, 26 June (for more information check the Teams post).
- Without the required number of attendances, you cannot participate at the exam (new column in attendance sheet).
- Exam will be closed-book and will last for 2 hours.

# Exam organization

- Problems will be divided into 3 parts:
    - **PART A:** 4 multiple choice questions (0.4 points each)
    - **PART B:** 4 drawing questions (0.85 points each)
    - **PART C:** 1 problem, where either the complexity of a given piece of code needs to be computed, or some simple implementation needs to be done. (1.5 p) + 1 more complex implementation problem (2.5 p)
- Solutions for each part need to be written on a different sheet of paper (so bring at least 3 sheets of paper) and your name and group needs to appear on all three of them.
- Justifications are important!