DATA STRUCTURES AND ALGORITHMS

LECTURE 14

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2024 - 2025

In Lecture 13...

- AVL trees
- Cuckoo hashing
- Perfect hashing

Today

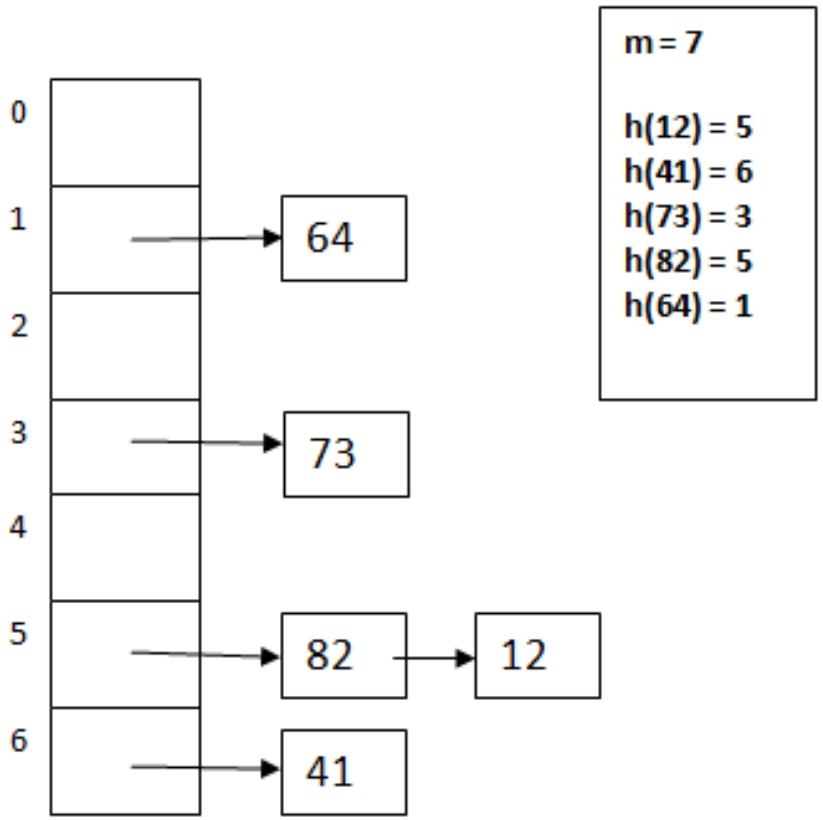
- Linked hash table
- Applications
- (Bad) Examples
- Recap
- Exam

Linked Hash Table

Linked Hash Table

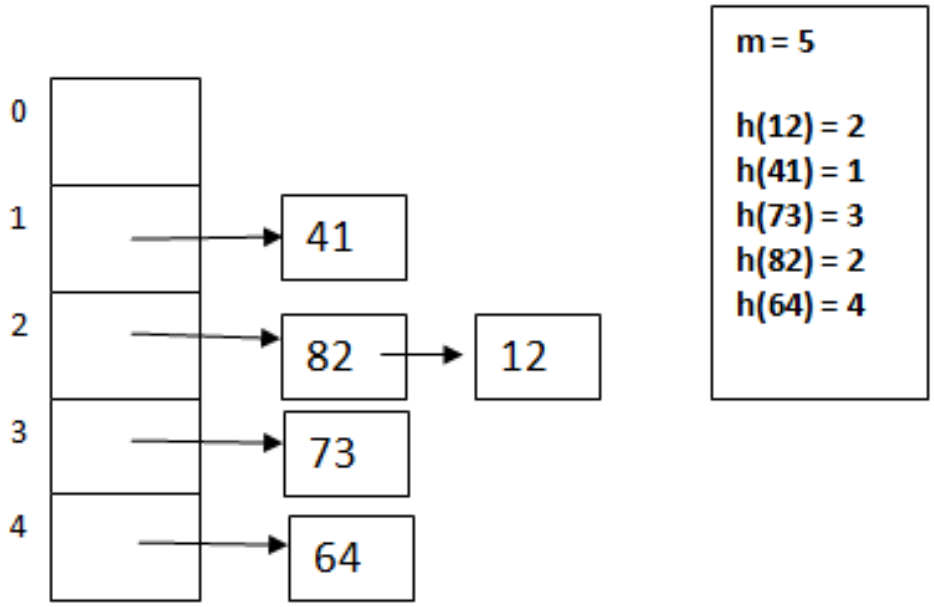
- Assume we build a hash table using separate chaining as a collision resolution method.
- We have discussed how an iterator can be defined for such a hash table.
- When iterating through the elements of a hash table, the order in which the elements are visited is *undefined*
- For example:
  - Assume an initially empty hash table (we do not know its implementation)
  - Insert one-by-one the following elements: 12, 41, 73, 82, 64
  - Use an iterator to display the content of the hash table
  - In what order will the elements be displayed?

Linked Hash Table



- Iteration order: 64, 73, 82, 12, 41

Linked Hash Table



- Iteration order: 41, 82, 12, 73, 64

Linked Hash Table

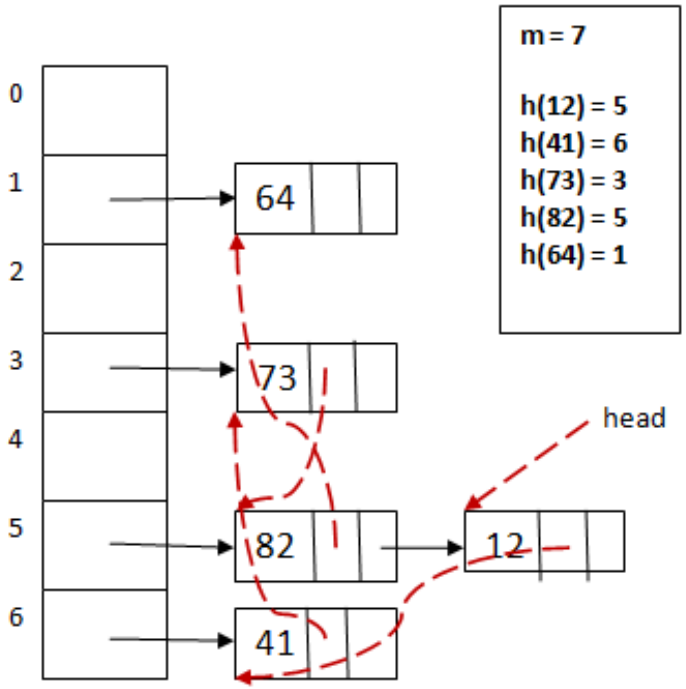
- A *linked hash table* is a data structure which has a *predictable* iteration order. This order is the order in which elements were inserted.
- So if we insert the elements 12, 41, 73, 82, 64 (in this order) in a linked hash table and iterate over the hash table, the iteration order is guaranteed to be: 12, 41, 73, 82, 64.
- How could we implement a linked hash table which provides this iteration order?

Linked Hash Table

- A linked hash table is a combination of a hash table and a linked list. Besides being stored in the hash table, each element is part of a linked list, in which the elements are added in the order in which they are inserted in the table.

- Since it is still a hash table, we want to have, on average, $\Theta(1)$ for insert, remove and search, these are done in the same way as before, the *extra* linked list is used only for iteration.

Linked Hash Table



- Red arrows show how the elements are linked in insertion order, starting from a *head* - the first element that was inserted, 12.

Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

- The only operation that cannot be efficiently implemented if we have a singly linked list is the *remove* operation. When we remove an element from a singly linked list we need the element before it, but finding this in our linked hash table takes $O(n)$ time.

Linked Hash Table - Implementation

- What structures do we need to implement a Linked Hash Table?

Node:
info: TKey
nextH: ↑ Node *//pointer to next node from the collision*
nextL: ↑ Node *//pointer to next node from the insertion-order list*
prevL: ↑ Node *//pointer to prev node from the insertion-order list*

LinkedHT:
m:Integer
T:(↑ Node)[]
h:TFunction
head: ↑ Node
tail: ↑ Node

Linked Hash Table - Insert

- How can we implement the *insert* operation?

```
subalgorithm insert(lht, k) is:
//pre: lht is a LinkedHT, k is a key
//post: k is added into lht
    allocate(newNode)
    [newNode].info ← k
    @set all pointers of newNode to NIL
    pos ← lht.h(k)
    //first insert newNode into the hash table
    if lht.T[pos] = NIL then
        lht.T[pos] ← newNode
    else
        [newNode].nextH ← lht.T[pos]
        lht.T[pos] ← newNode
    end-if
//continued on the next slide...
```

Linked Hash Table - Insert

```
    //now insert newNode to the end of the insertion-order list
    if lht.head = NIL then
        lht.head ← newNode
        lht.tail ← newNode
    else
        [newNode].prevL ← lht.tail
        [lht.tail].nextL ← newNode
        lht.tail ← newNode
    end-if
end-subalgorithm
```

Linked Hash Table - Remove

- How can we implement the *remove* operation?

```
subalgorithm remove(lht, k) is:
//pre: lht is a LinkedHT, k is a key
//post: k was removed from lht
    pos ← lht.h(k)
    current ← lht.T[pos]
    nodeToBeRemoved ← NIL
    //first search for k in the collision list and remove it if found
    if current ≠ NIL and [current].info = k then
        nodeToBeRemoved ← current
        lht.T[pos] ← [current].nextH
    else
        prevNode ← NIL
        while current ≠ NIL and [current].info ≠ k execute
            prevNode ← current
            current ← [current].nextH
        end-while
//continued on the next slide...
```

```
        if current ≠ NIL then
            nodeToBeRemoved ← current
            [prevNode].nextH ← [current].nextH
        else
            @k is not in lht
        end-if
    end-if
//if k was in lht then nodeToBeRemoved is the address of the node containing
//it and the node was already removed from the collision list - we need to
//remove it from the insertion-order list as well
    if nodeToBeRemoved ≠ NIL then
        if nodeToBeRemoved = lht.head then
            if nodeToBeRemoved = lht.tail then
                lht.head ← NIL
                lht.tail ← NIL
            else
                lht.head ← [lht.head].nextL
                [lht.head].prev ← NIL
            end-if
//continued on the next slide...
```

```
        else if nodeToBeRemoved = lht.tail then
            lht.tail ← [lht.tail].prev
            [lht.tail].next ← NIL
        else
            [[nodeToBeRemoved].next].prev ← [nodeToBeRemoved].prev
            [[nodeToBeRemoved].prev].next ← [nodeToBeRemoved].next
        end-if
        deallocate(nodeToBeRemoved)
    end-if
end-subalgorithm
```

Applications of different containers / data structures

- In a seminar we have already talked about one such application: evaluating an arithmetic expression, which is an example of using stacks and queues.
- In a lecture we also mentioned some applications of stacks and queues (finding the exit in a maze, card game).
- We have also talked about Huffman encoding, which is an application that uses binary trees and a priority queue.
- Now we are going to talk about another application: bracket matching

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

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.

(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 1f 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.

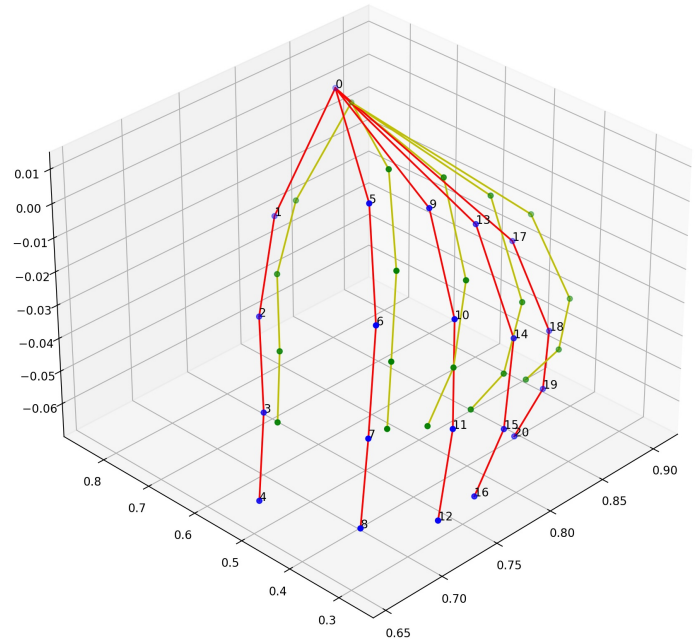
Example 3

- 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
```

- Bachelor exam subject a while ago: implement a recursive algorithm for merging two sorted arrays.

- On the next slide you will see three possible implementations in C++ (using vector and specific operations on vector). What is the complexity of these approaches?

```cpp
vector<string> interclasare1(vector<string> v1, vector<string> v2) {
    if (v1.size() == 0) {
        return v2;
    }
    if (v2.size() == 0) {
        return v1;
    }
    vector<string> res;
    if (v1[0] < v2[0]) {
        res.push_back(v1[0]);
        v1.erase(v1.begin());
        auto v3 = interclasare1(v1, v2);
        res.insert(res.end(), v3.begin(), v3.end());
    }
    else {
        res.push_back(v2[0]);
        v2.erase(v2.begin());
        auto v3 = interclasare1(v1, v2);
        res.insert(res.end(), v3.begin(), v3.end());
    }
    return res;

}
```

```cpp
vector<string> interclasare2(vector<string> v1, vector<string> v2) {
    if (v1.size() == 0) {
        return v2;
    }
    if (v2.size() == 0) {
        return v1;
    }
    vector<string> res;
    reverse(v1.begin(), v1.end());
    reverse(v2.begin(), v2.end());
    if (v1[v1.size()-1] < v2[v2.size()-1]) {
        res.push_back(v1[v1.size() - 1]);
        v1.pop_back();
    }
    else {
        res.push_back(v2[v2.size()-1]);
        v2.pop_back();
    }
    reverse(v1.begin(), v1.end());
    reverse(v2.begin(), v2.end());
    auto v3 = interclasare2(v1, v2);
    res.insert(res.end(), v3.begin(), v3.end());
    return res;

}
```

```
vector<string> interclasare3(vector<string> v1, vector<string> v2) {
    if (v1.size() == 0) {
        return v2;
    }
    if (v2.size() == 0) {
        return v1;
    }
    vector<string> res;
    string val;
    if (v1[v1.size() - 1] > v2[v2.size() - 1]) {
        val = v1[v1.size() - 1];
        v1.pop_back();
    }
    else {
        val = v2[v2.size() - 1];
        v2.pop_back();
    }
    res = interclasare3(v1, v2);
    res.push_back(val);
    return res;

}
```

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

- 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 100 minutes.

Exam organization

- Problems will be divided into 3 parts:
  - **PART A:** 4 multiple choice questions (0.4 points each)
  - **PART B:** 3 drawing questions (1 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. (1.4 p) +
  - two problems (with different difficulty and points) out of which you need to solve one (1.75p and 3p)
- 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!

- Time management is important!
- Terminology is important. Example from previous year:
  - Questions: *What data structure would you use for ...?*
  - Answer (from an actual exam paper): *"I would use an ADT Sorted Bag data structure with key value pairs"*