# DATA STRUCTURES

## LECTURE 13

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

# In Lecture 12...

- Binary Trees
- Binary Search Trees

# Today

- Huffman encoding
- Parenthesis matching
- Linked hash table
- Exam info

- Starting from an initially empty Binary Search Tree and the relation $\leq$, insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.

# Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation $\leq$, insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.
- How would you count how many times the value 5 is in the tree?

# Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation $\leq$, insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.
- How would you count how many times the value 5 is in the tree?
- Remove 3 (show both options)

- Starting from an initially empty Binary Search Tree and the relation $\leq$, insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.
- How would you count how many times the value 5 is in the tree?
- Remove 3 (show both options)
- How would you count now how many times the value 5 is in the tree now?

# 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).

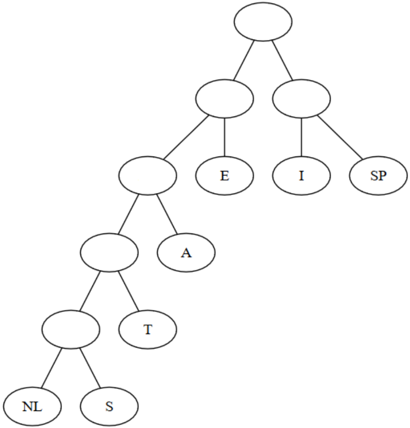
- When building the Huffman encoding for a message, we first have to compute the frequency of every character from the message, because we are going to define the codes based on the frequencies.

- Assume that we have a message with the following letters and frequencies

| Character | a | e | i | s | t | space | newline |
|---|---|---|---|---|---|---|---|
| Frequency | 10 | 15 | 12 | 3 | 4 | 13 | 1 |

- 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.

# 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:
  - NL - 00000
  - S - 00001
  - T - 0001
  - A - 001
  - E - 01
  - I - 10
  - SP - 11
- In order to encode a message, just replace each character with the corresponding code

- Assume we have the following code and we want to decode it: 011011000100010011100100000
- 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: E I SP T T A SP I E NL

# Delimiter 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

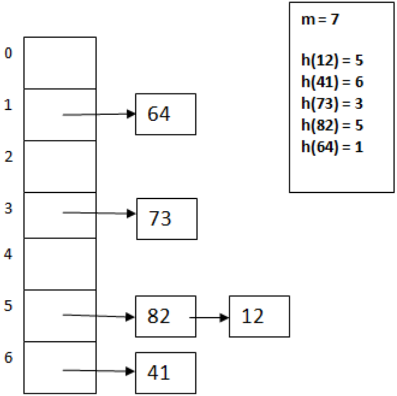
- How can we extend the previous idea so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
  - Open brackets that are never closed
  - Closed brackets that were not opened
  - Mismatch
- Keep count of the current position in the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

# Linked Hash Table

# Linked Hash Table

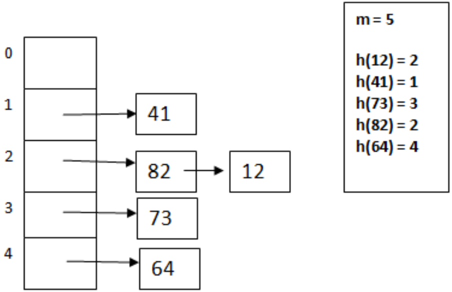
- Assume we build a hash table using separate chaining as a collision resolution method.
- We have discussed how an iterator can be defined for such a hash table.
- When iterating through the elements of a hash table, the order in which the elements are visited is *undefined*
- For example:
  - Assume 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

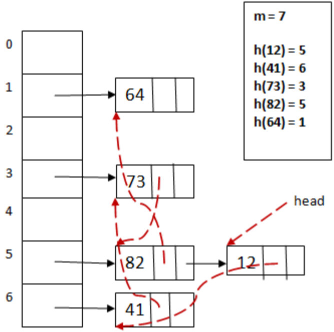
# Linked Hash Table

- 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.




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

- 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).

- 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.

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

<u>Node:</u>
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*

<u>LinkedHT:</u>
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...
```

```
    //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
```

# Conclusions

- During the semester we have talked about the most important containers (ADT) and their main properties and operations
  - Bag, Set, Map, Multimap, List, Stack, Queue and their sorted versions
- 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, Binary Search Tree

# Conclusions

- You should be able to identify the most suitable container for solving a given problem:

# Conclusions

- You should be able to identify the most suitable container for solving a given problem:
- Example: *You have a type 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?

# Conclusions

- When you use containers existing in different programming languages, you should have an idea of how they are implemented and what is the complexity of their operations:

- When you use containers existing in different programming languages, you should have an idea of how they are implemented and what is the complexity of their operations:
- Consider the following algorithm (written in Python):

```
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*?

# Conclusions

- 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.*

- 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.

- 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 an implementation provided by a student. 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--;
  }
}
```