DSA – Seminar 4

1. Sort Algorithms

A. BucketSort

- We are given a sequence S, formed of *n* pairs (key, value), keys are integer numbers from an interval ϵ [0, N-1]
- We have to sort S based on the keys.

For example:

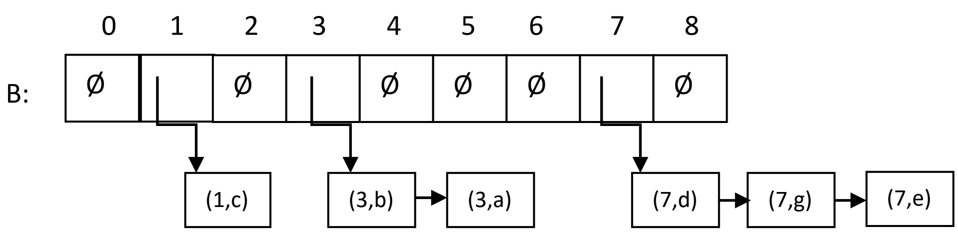
S: (7, d) (1, c) (3, b) (7, g) (3, a) (7, e) =>

(1, c) (3, b) (3, a) (7, d) (7, g) (7, e) N = 9

- What to do when we have equal keys (ex. (3, b), (3, a))? Requirement is to *sort by the keys*, so we will not compare the values. If the keys are equal, the two pairs are equal and they could be in any order (so we could have (3,a) and then (3, b) or (3,b) and then (3,a)) the sequence would be sorted. We will go with a version in which in case of equal keys we will keep the pairs in the order in which they are initially in the sequence. But this is our decision, the sequence would be sorted without this decision as well.

Idea:

- Use an auxiliary array, B, of dimension N, in which each element is a sequence.
- Each pair will be placed in B in the position corresponding to the key (B[k]) – and will be deleted from S.
- We parse B (from 0 to N-1) and move the pairs from each sequence from each position of B to the end of S.



What is a Sequence?

- Assume that the ADT Sequence is already implemented, and it has the following operations (we assume they all run in Θ(1) complexity):
  - empty (sequence): boolean
  - first (sequence): element
  - remove First(sequence)
  - insertLast(sequence, element)

Obs1.: element in our case will be a pair (k, v)
Obs2.: What data structure should we use if we wanted to implement *sequence* in order to get the Θ(1) complexity for the operations?

```
Subalgorithm BucketSort(S, N) is:
//define array B of dimension N
    While ¬ empty (S) execute:
        (k, v) ← first (S)
        removeFirst (S)
        insertLast (B[k], (k,v))
    end-while
    for i ← 0, N-1, execute:
        While ¬ empty (B[i]) execute:
            (k, v) ← first (B[i])
            removeFirst (B[i])
            insertLast (S, (k,v))
        end-while
    end-for
end-subalgorithm
```

Complexity: Θ(N + n)

<u>Observations:</u>

- Keys must be natural numbers (we are using them as indexes)
- In our implementation, the relative order of the pairs that have the same key will not change -> we call such sorting algorithms *stable.*

B. Lexicographic Sort

Elements to be sorted are d-dimensional: $(x_1, x_2, ..., x_d)$ – d-tuples.
How do we compare two such tuples?
$(x_1, x_2, ... ,x_d) < (y_1, y_2, ..., y_d) \Leftrightarrow x_1 < y_1 \vee (x_1=y_1 \wedge ((x_2,..., x_d) < (y_2, ..., y_d)))$

- We compare the first dimension, if they are equal then the 2nd and so on...

We are given a sequence S of tuples. We have to sort S in a lexicographic order.

In the implementation we will use:

- R_i – a relation that can compare 2 tuples considering the i^{th} dimension (and we have a relation for every dimension: $R_1, R_2, ..., R_d$).
- *stableSort(S, r)* – a stable sorting algorithm that uses a relation *r* to compare the elements.

The lexicographic sorting algorithm will execute StableSort *d* times (once for every dimension).

```
Subalgorithm LexicographicSort(S, R, d) is:
//S- input sequence
//d - number of dimensions
//R - the set of all relations
    For i ← d, 1, -1, execute:
        stableSort(S, Ri)
    end-for
end-subalgorithm
```

Complexity: Θ (d * T(n))
where T(n) – complexity of the stableSort algorithm

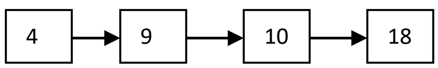
Ex. d = 3
(7, 4, 6) (5, 1, 5) (2, 4, 6) (2, 1, 4) (3, 2, 4)
Sort based on dimension 3 (last digit): (2, 1, 4) (3, 2, 4) (5, 1, 5) (7, 4, 6) (2, 4, 6)
Sort based on dimension 2 (middle digit): (2, 1, 4) (5, 1, 5) (3, 2, 4) (7, 4, 6) (2, 4, 6)
Sort based on dimension 1 (first digit): (2, 1, 4) (2, 4, 6) (3, 2, 4) (5, 1, 5) (7, 4, 6)

C. Radix Sort

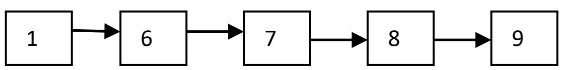
- A variant of the lexicographic sort, which uses as a stable sorting algorithm Bucketsort → every element of the tuples has to be a natural number from some interval [0, N-1].
- Complexity: Θ (d * (n + N))

2. Write a subalgorithm to merge two sorted singly-linked lists. Analyze the complexity of the operation.

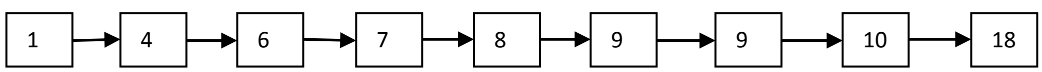
L1.



L2.



Result:



Representation:
<u>Node:</u>
info: TComp
next: ↑Node

<u>List:</u>
head: ↑Node
//possibly a relation, but then we have to make sure that the two lists contain the same relation.

a. We do not destroy the two existing lists: the result is a third list (we have to copy the existing nodes) – similar to what we do when we merge two arrays.

```
subalgorithm merge (L1, L2, LR) is:
    currentL1 ← L1.head
    currentL2 ← L2.head
    headLR ← NIL //the first node of the result
    tailLR ← NIL //the last node, needed because we add nodes to the end and
without this, we had to go through the already built list every time we add a
new node
    while currentL1 ≠ NIL and currentL2 ≠ NIL execute
        allocate(newNode)
```

```
        [newNode].next ← NIL
        if [currentL1].info  < [currentL2].info then
            [newNode].info ← [currentL1].info
            currentL1 ← [currentL1].next
        else
            [newNode].info ← [currentL2].info
             currentL2 ← [currentL2].next
        end-if
        if headLR = NIL then
            headLR ← newNode
            tailLR ← newNode
        else
            [tailLR].next ← newNode
             tailLR ← newNode
        end-if
    end-while
      //one of the currentNodes is NIL, we will keep the other one in a
      //separate variable, to write the following while loop only once
    if currentL1 ≠ NIL then
         remainingNode ← currentL1
    else
        remainingNode ← currentL2
    end-if
    while remainingNode ≠ NIL execute
        alocate (newNode)
        [newNode].next ← NIL
        [newNode].info ← [remainingNode].info
        remainingNode ← [remainingNode].next
        if headLR = NIL then
            headLR ← newNode
            tailLR ← newNode
        else
            [tailLR].next ← newNode
             tailLR ← newNode
        end-if
    end-while
    LR.head ← headLR
end-subalgorithm
```

Complexity: Θ(n + m)

n – length of L1
m – length of L2

b. We do not keep the two existing lists, the result will contain the existing nodes (but the links are changed)

```
subalgorithm merge (L1, L2, LR) is:
    currentL1 ← L1.head
    currentL2 ← L2.head
    headLR ← NIL //the first node
    tailLR ← NIL //the last node, needed because we add nodes to the end

    while currentL1 ≠ NIL and currentL2 ≠ NIL execute
      //chosenNode will be the actual node we take from a list
```

```
        if [currentL1].info  < [currentL2].info then
            chosenNode ← currentL1
            currentL1 ← [currentL1].next
        else
            chosenNode ←  currentL2
             currentL2 ← [currentL2].next
        end-if
        [chosenNode].next ← NIL
        if headLR = NIL then
            headLR ← chosenNode
            tailLR ← chosenNode
        else
            [tailLR].next ← chosenNode
             tailLR ← chosenNode
        end-if
    end-while
    if currentL1 ≠ NIL then
         remainingNode ← currentL1
    else
        remainingNode ← currentL2
    end-if
      //no need for a loop, just attach every remaining node (starting from
      //remainingNode) to the beginning/end of list. Since this is the last
      //instruction, the value of tailLR does not need to be updated.
    if headLR = NIL then
            headLR ← remainingNode
    else
            [tailLR].next ← remainingNode
    end-if
    LR.head ← headLR
    L1.head ← NIL //make sure you have no nodes left in the lists
    L2.head ← NIL
end-subalgorithm
```

Complexity:

- We no longer need to parse both lists completely, we only have the while loop which is executed as long as both lists have elements. This might be the shortest list (consider a list having values 2 and 3 and another having values 1 to 100) or the longer list (consider a list having values 1 and 100 and another one having values 2 to 99). These situations will give us the best and worst case complexity:

Best case: Θ(min(n, m))
Worst case: Θ(m + n)
Total complexity: O(m + n)

n – length of L1
m – length of L2

In our implementation the lists can have duplicates: we can have the same element several times in one of the lists (we could have for example two nodes with value 6 in the second list on the example) and we can have the same element in both lists. If something like this happens, the resulting list will contain all the elements, with duplicates.
An interesting version of this problem is achieved if we consider that the input lists are not allowed to contain duplicates and we do not want to have duplicates in the result list either. In this case, since the

input lists do not contain duplicates, the only way to have duplicates is when we have the same element in both lists.
When you create a copy of the nodes (version a), this is simple: if the info from the two current nodes is equal, create only one copy, but progress with both current nodes.
However, when you are reordering the nodes (version b), this means to actually delete from the list those nodes with are equal.