# 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 ∊ [0, N-1]
- We have to sort S based on the keys.

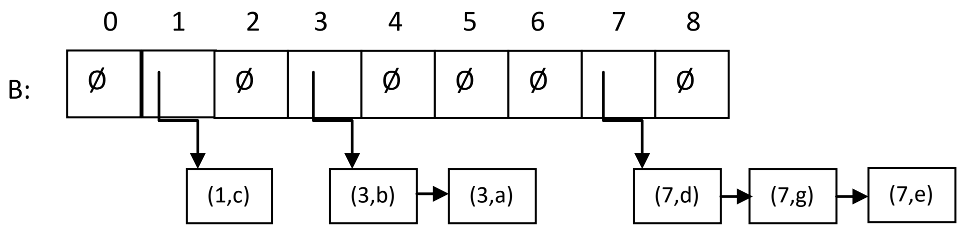
For example:

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

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.



Assume that the sequence is already implemented, and it has the following operations:

- empty (sequence): boolean
- first (sequence): TPosition
- remove (sequence, TPosition): element
- insertLast(sequence, element)
- Obs: element in our case will be a pair (k, v)

```
Subalgorithm BucketSort(S, N) is:
//define array B of dimension N
    While ¬ empty (S) execute:
        p ← first (S)
        (k, v) ← remove (S, p)
        insertLast (B[k], (k,v))
    end-while
    for i ← 0, N-1, execute:
        While ¬ empty (B[i]) execute:
            p ← first (B[i])
            (k, v) ← remove (B[i], p)
            insertLast (S, (k,v))
        end-while
    end-for
end-subalgorithm
```
Complexity: $\Theta(N + n)$

Observations:

- 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

d-tuple $(x_1, x_2, ..., x_d)$

$(x_1, x_2, ... ,x_d) < (y_1, y_2, ..., y_d) \Leftrightarrow x_1 < y_1 \lor (x_1=y_1 \land ((x_2,..., x_d) < (y_2, ..., y_d)))$

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

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

We will use:

- $R_i$ – a relation that can compare 2 tuples considering the i<sup>th</sup> dimension.
- *stableSort(S, r)* – a stable sorting algorithm that uses a relation to compare the elements.

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

```
Subalgorithm LexicographicSort(S, R, d) is:
    For i ← d, 1, -1, execute:
        stableSort(S, $R_i$)
    end-for
end-subalgorithm
```
Complexity: $\Theta$ (d * T(n))
where T(n) – complexity of the stableSort algorithm

Ex. (7, 4, 6) (5, 1, 5) (2, 4, 6) (2, 1, 4) (3, 2, 4)

| | |
|---|---|
| Sort based on dimension 3: | (2, 1, 4) (3, 2, 4) (5, 1, 5) (7, 4, 6) (2, 4, 6) |
| Sort based on dimension 2: | (2, 1, 4) (5, 1, 5) (3, 2, 4) (7, 4, 6) (2, 4, 6) |
| Sort based on dimension 1: | (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: $\Theta$ (d * (n + N))

2. Indexed List

- TPosition – Integer – index of the element from the list

Analyze the complexity of parsing all the elements of an indexed list

a) Using an iterator

```
Subalgorithm iterateWithIterator(l) is:
//pre: l is a list
    iterator(l, it)
    while valid(it) execute:
       getCurrent (it, e)
       @ do something with e
       next(it)
    end-while
 end-subalgorithm
```

b) Using indexes

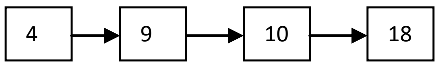
```
Subalgorithm iterateWithIntegerPos(l) is:
//pre: l is a list
   for i ← 1, n execute
       element(l, i ,e)
       @do something with e
   end-for
end-subalgorithm
```

I. Representation on an array:
   a. Θ(n)
   b. Θ(n)

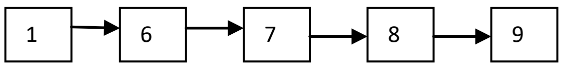
II. Representation on a linked list:
   a. Θ(n)
   b. Θ($n^2$)

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

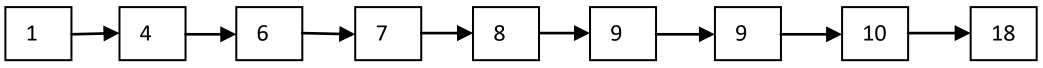
L1.



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).

```
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
    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 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: Θ(n + m)
n – length of L1
m – length of L2