DS – Seminar 3 - Bucket Sort, Merging SSLL

- **Content**:
  - Sorting algorithms
    - Bucket Sort
    - Lexicographic Sort
    - Radix Sort
  - Merge two sorted dynamically allocated single linked lists

Sorting algorithms

A. BucketSort

- We are given a sequence S, formed of *n* pairs (key, value), where the keys are integer numbers from the 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)

N = 9

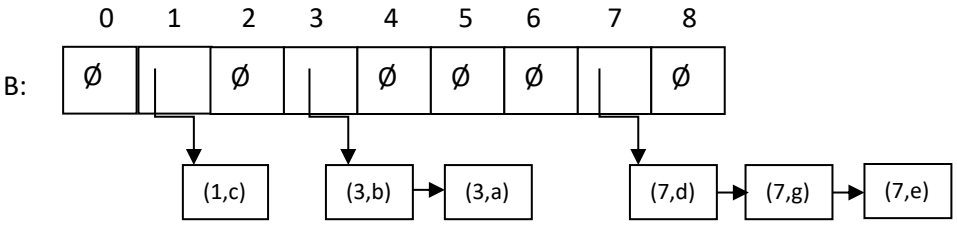
=> (1, c) (3, b) (3, a) (7, d) (7, g) (7, e)

- 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:

- An auxiliary array, B, of length N, in which each element is a sequence, is used.
- Each pair will be placed in B in the sequence at the position corresponding to the key (B[k]) – and will be deleted from S.

- B is traversed (from 0 to N-1) and move the pairs from each sequence at 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): <TKey, TValue>
  - removeFirst(Sequence)
  - insertLast(Sequence, <TKey, TValue>

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:
    //we suppose that the array B of length N is already allocated
    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)

Notes:

- Keys must be natural numbers (we are using them as indexes). If they are not, then we have to map them to valid indexes.
- In our implementation, the relative order of the pairs that have the same key will not changed. We call such sorting algorithms *stable.*

B. Lexicographical 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 lexicographical 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 LexicographicalSort(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
```

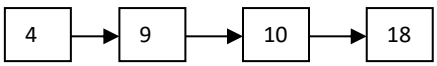
Complexity: Θ (d * T(n))
where T(n) – complexity of the stableSort algorithm

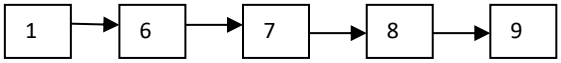
Ex. d = 3
(7, 4, 6) (5, 1, 5) (2, 4, 6) (2, 1, 4) (3, 2, 4)
Sort based on dimension 3 (last digit): (2, 1, 4) (3, 2, 4) (5, 1, 5) (7, 4, 6) (2, 4, 6)
Sort based on dimension 2 (middle digit): (2, 1, 4) (5, 1, 5) (3, 2, 4) (7, 4, 6) (2, 4, 6)
Sort based on dimension 1 (first digit): (2, 1, 4) (2, 4, 6)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

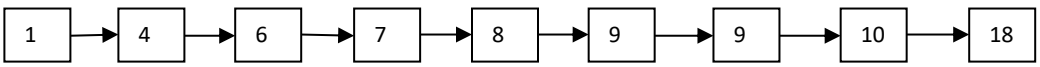
C. Radix Sort

- A variant of the lexicographical sort, which uses as a stable sorting algorithm BucketSort. For this, every element of the tuple has to be a natural number from an interval [0, N-1].
- Complexity: Θ (d * (n + N))

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

L1. 

L2. 

Result: 

Representation:

Node:
info: TComp
next: ↑Node

SSLL:
head: ↑Node
rel: TComp x TComp -> {True, False}

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
        newNode  ← allocate()
        [newNode].next ← NIL
        if L1.rel([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
        else
            [tailLR].next ← newNode
        end-if
        tailLR ← newNode
    end-while
      //one of the current nodes is NIL, so we will keep the other one in a
      //separate variable, to write the following while loop only once
    current ← currentL1
    if current = NIL then
        current ← currentL2
    end-if
    while current ≠ NIL execute
        alocate(newNode)
        [newNode].next ← NIL
        [newNode].info ← [current].info
        current ← [current].next
        if LR.head = NIL then
            LR.head ← newNode
        else
            [tailLR].next ← newNode

        end-if
```

```
        tailLR ← newNode
    end-while
end-subalgorithm
```

Complexity: Θ(n + m)

n – length of L1
m – length of L2