Lecture 6

- Linked and array representations. Combinations
  - LinkedList on Array
  - XOR LinkedList
  - Skip List
  - others […]

Lect. PhD. Lupsa Dana
Babes - Bolyai University
Computer Science and Mathematics Faculty
2021 - 2022

Some slides borrowed from: Lect. PhD. Onet-Marian Zsuzsanna

Previously, in Lecture 5

- List
  - ADT
    - List, IteratedList, IndexedList
    - SortedList
  - Singly Linked List
  - Doubly Linked List

Linked Lists on Arrays

<u>Problem</u>:

Implement linked list data structures, without the explicit use of pointers or memory addresses, simulating them by using arrays and array indexes.

We can define a linked data structure on an array, if we consider that the order of the elements is not given by their positions (indices) in the array, but by an integer number associated with each element, which shows the index of the next element in the array. Thus we have a singly linked list.

<u>E.g.</u>:

elems	46	78	11	6	59	19				
next	5	6	1	-1	2	4				

head = 3

Singly Linked List on Array: SLLA

Representation:

<u>SLLANode</u>:
- info: TElem
- next: Integer

<u>SLLA</u>:
- nodes: SLLANode []
- cap: Integer
- head: Integer
- firstFree: Integer

Use this one in the next examples:

<u>SLLA</u>:
- elems: TElem[]
- next: Integer[]
- cap: Integer
- head: Integer
- firstFree: Integer

<u>SLLAIterator</u>:
- list: SLLA
- current: Integer

SLLA - Search

function search (slla, elem) **is:**
//pre: slla is a SLLA, elem is a TElem
//post: return True is elem is in slla, False otherwise
    current $\leftarrow$ slla.head
    while current $\neq$ -1 **and** slla.elems[current] $\neq$ elem **execute**
        current $\leftarrow$ slla.next[current]
    end-while
    if current $\neq$ -1 **then**
        search $\leftarrow$ True
    else
        search $\leftarrow$ False
    end-if
end-function

Complexity: O(n)

SLLA – memory management inside array

Free space in array:

- free "nodes" are linked
- keep the first element (firstEmpty)

Complexity: ?

```
subalgorithm freeP (slla, pos) is:
  slla.next[pos] ←  slla.firstEmpty
  slla.firstFree ←  pos
end-subalgorithm
```

```
function allocateP(slla) is:
        if slla.firstFree = -1 then
            @ resize arrays: reallocate, copy elements, re-initialize free space
        end-if
        newFreePos ←  slla.firstFree
        slla.firstFree ←  slla.next [slla.firstFree]
        allocateP ← newFreePos
end-function
```

SLLA – memory management inside array

Initialize free space:

Complexity: ?

```
Subalg. initFreeSpace (slla) is:
    for i ← 1, slla.cap execute
        slla.next[i] ← i+1
    endfor
    sllla.next[sll.cap] ← -1
    slla.firstFree ← 1
End_subalgorithm
```

Assume:
1-based
indexing

Think about:
What other strategies to initialize free space can we use?

SLLA – memory management inside array

- Operation insertFirst

```
subalgoritm insertFirst(slla, elem) is:
      newPosition ← allocateP(slla)
      slla.elems[newPosition] ← elem
      slla.next[newPosition] ← slla.head
      slla.head ← newPosition
end-subalgorithm
```

Complexity: $\Theta(1)$ amortized (if we use Dynamic Array)

SLLA – operation deleteElement

•

See SLL deleteElement !

```
subalgorithm deleteElement(slla, elem) is:
    current ← slla.head
    prev ← -1
    while current ≠  -1 and slla.elems[current] ≠ elem execute
        prev ← current
        current ← slla.next[current]
    end-while
    if current ≠ -1 then
        if current = slla.head then
                slla.head ←  slla.next[slla.head]
        else
                slla.next[prev] ← slla.next[current]
        end-if
        freeP(current)
        deleteElement ← True
    else
        deleteElement ←  False
    end-if
end-subalgorithm
```

Complexity: O(n)

DLL on Array

- Same idea as in case of SLLA
- Here we discuss representation and only one operation (the same approach works for all operations)

Representation:

<u>DLLANode</u>:

info: TElem

next: Integer

prev: Integer

<u>DLLA</u>:

nodes: DLLANode []

cap: Integer

head: Integer

tail: Integer

firstFree: Integer

<u>DLLA</u>:

elems: TElem[]

next: Integer[]

cap: Integer

head: Integer

tail: Integer

firstFree: Integer

<u>DLLAIterator</u>:

List: DLLA

current: Integer

Use this one in the next examples

DLLA – operation insertLast

•

See DLL insertLast !

```
subalgorithm insertLast (dlla, elem) is:
    newPosition  ← allocateP()
    dlla.nodes[newPosition].info ← elem
    dlla.nodes[newPosition].next ← -1
    dlla.nodes[newPosition].prev ← dlla.tail
    if dlla.head = -1 then
            dlla.head ← newPosition
            dlla.tail ← newPosition
    else
            dlla.nodes[dlla.tail].next  ← newPosition
            dll.tail ← newPosition
    end-if
end-subalgorithm
```

Complexity: Θ(1) amortized

Linked Lists on Arrays

We discussed:

- Representation
- Memory management inside the array
- A few operations

    just to see how to approach the implementation

XOR Linked Lists

XOR Linked List is equivalent to a doubly linked list,
where every node keeps one single link,
which is the XOR of the previous and the next node.

<u>XORNode</u>:
  info: TELem
  link: ↑ XORNode

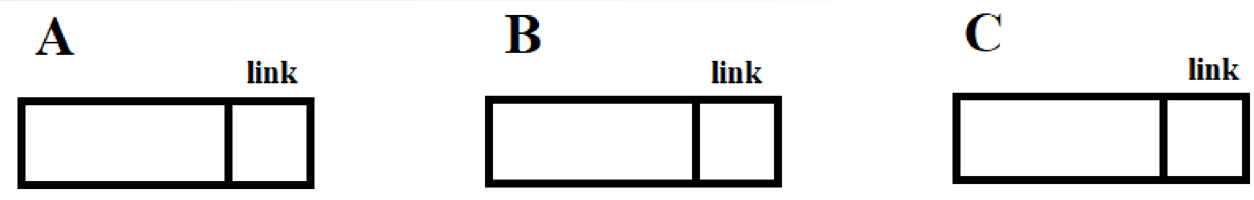
<u>XORList</u>:
  head: ↑ XORNode
  tail: ↑ XORNode

Use dynamic representation

XOR link

XOR has the following properties:

- A XOR A = 0
- A XOR 0 = X
- A XOR B = B XOR A



link(B) = addr(A) XOR addr(C)

- addr is the function that indicates the address of a specific node

link(B) XOR addr(A) =

= addr(A) XOR addr(C) XOR addr(A)

= addr(A) XOR addr(A) XOR addr(C)

= 0 XOR addr(C)

= addr(C)

To implement XOR linked list,
the language should support conversion between pointers and integers.
<u>e.g.:</u> It is supported by C / C++ and it is not supported by Java

```
subalgorithm printListForward(xorl) is:
//pre: xorl is a XORList
//post: true (the content of the list was printed)
  prevNode ← NIL
  currentNode ← xorl.head
  while currentNode ≠ NIL execute
    write [currentNode].info
    nextNode ← prevNode XOR [currentNode].link
    prevNode ← currentNode
    currentNode ← nextNode
  end-while
end-subalgorithm
```

Complexity: $\Theta(n)$

```
subalgorithm addToBeginning(xorl, elem) is:
//pre: xorl is a XORList
//post: a node with info elem was added to the beginning of the list
   newNode ← allocate()
   [newNode].info ← elem
   [newNode].link ← xorl.head
   if xorl.head = NIL then
      xorl.head ← newNode
      xorl.tail ← newNode
   else
      [xorl.head].link ← [xorl.head].link XOR newNode
      xorl.head ← newNode
   end-if
end-subalgorithm
```

Complexity: $\Theta(1)$

Think about:

- What do we store in **head** of the list?

  What is the **link value** of node **head**?

- How do we traverse the list?
- Forward Iterator: representation?

- Implement addToBeginning
- Implement insertAfter
  - In case of a DLL, when we have the address of a node, we can add an element after it. Can we do the same in case of XOR lists?

Skip Lists

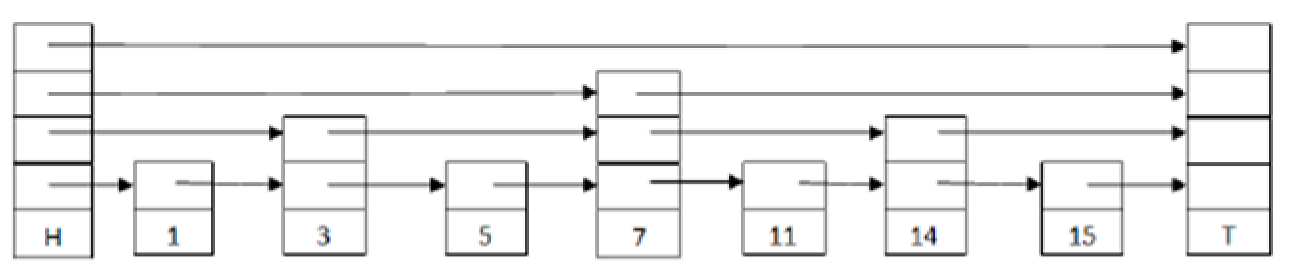
Skip lists are sorted linked lists, augmented to make the search faster

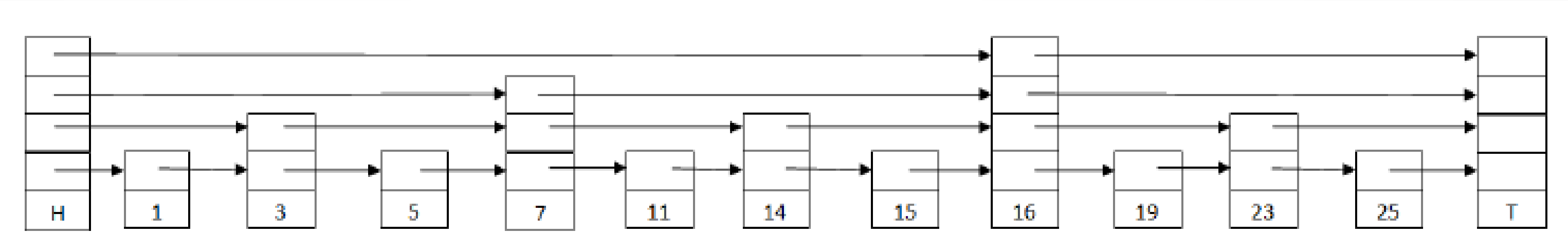
- they provide a way to keep a list of elements sorted
- support search, insert, and delete operations in an expected time of O(log n).

  We say an expected time rather than a guaranteed time because skip lists rely on a *probabilistic* algorithm to keep the list elements sorted.

Skip Lists

e.g.:



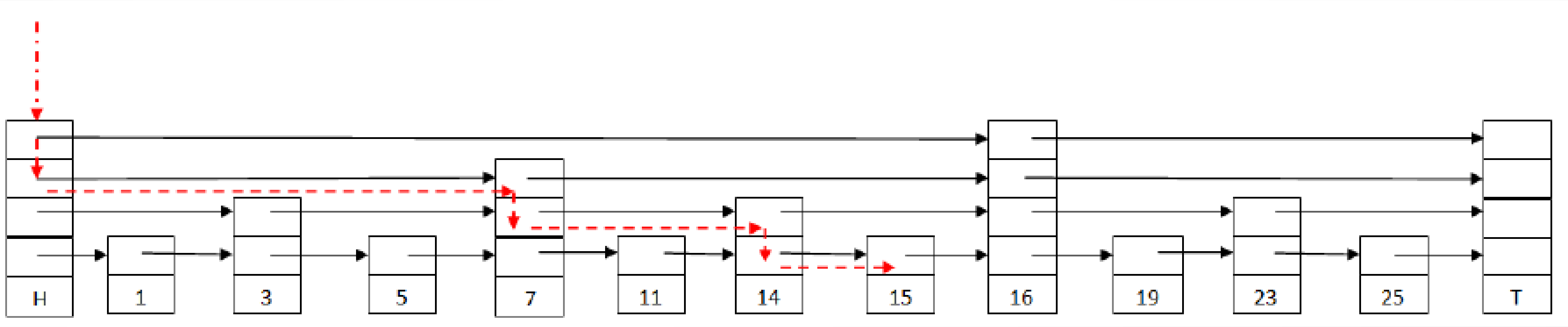


Source: Wikipedia

Skip Lists

- H and T are two special nodes, representing head and tail.
- They cannot be deleted, they exist even in an empty list.

- Search for element 15



- Start from head and from highest level.
- If possible, go right.
- If cannot go right (next element is greater), go down a level.

Skip Lists

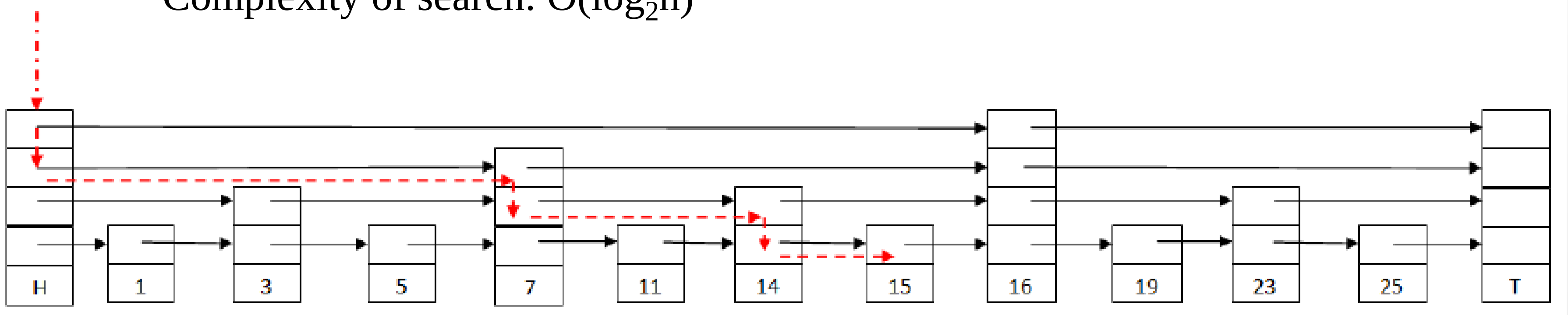
What do we want?

- Lowest level has all n elements.
- Next level has n/2 elements.
- Next level has n/4 elements.

  … etc.

- There are approx $\log_2 n$ levels. From each level, we check approx. at most 2 nodes.

  Complexity of search: $O(\log_2 n)$



Probabilistic data structure

How do we create it?

- Decide randomly the height of a newly inserted node.
- There might be a worst case, where every node has height 1

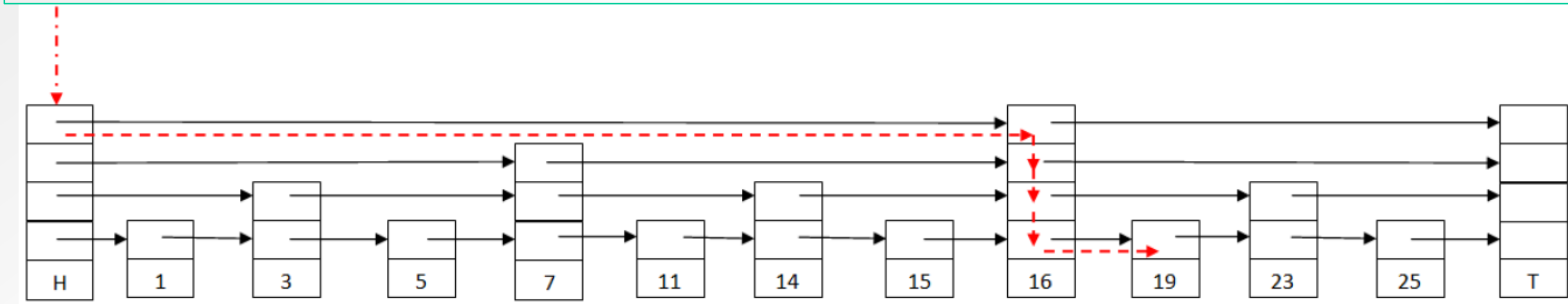
  (so it is just a linked list).

Skip Lists

e.g.:

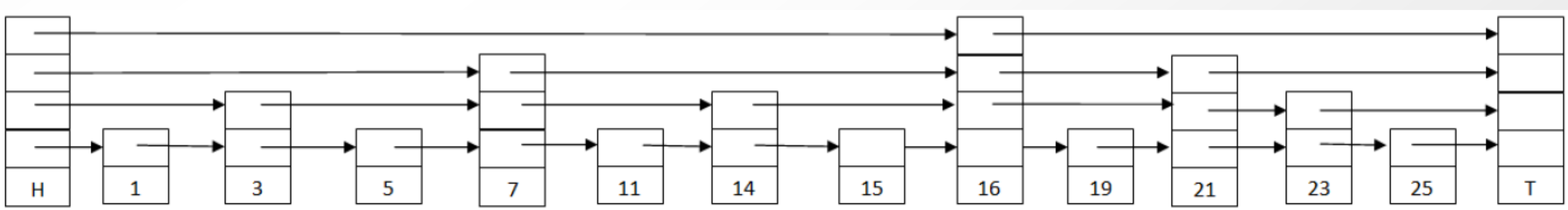
Insert element 21

- search



- generate the height for the node randomly, but in such a way that approximately half of the nodes will be on level 2, a quarter of them on level 3, etc.

  Assume: generated height is 3
- insert



Skip Lists

- Skip lists support search, insert, and delete operations in an expected time of O(log n). We say “an expected” time rather than a guaranteed time because skip lists rely on a *probablistic* algorithm to keep the list elements sorted.
- The time complexity of search, insert and delete can become O(n) in WC, but they are unlikely to occur.
- They use O(n) extra space.

<u>Remark</u>:
There is an implementation ConcurrentSkipListMap in Java
that uses a concurrent variation of SkipList data structure
providing log(n) time cost for insertion, removal, update, and access operations.