

Lecture 5

- 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
2022 - 2023

Previously, in Lecture 4

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

Linked Lists on Arrays

Problem:

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.

E.g.:

elems	46	78	11	6	59	19				
next	5	6	1	-1	2	4				
head = 3										

Singly Linked List on Array: SLLA

Representation:

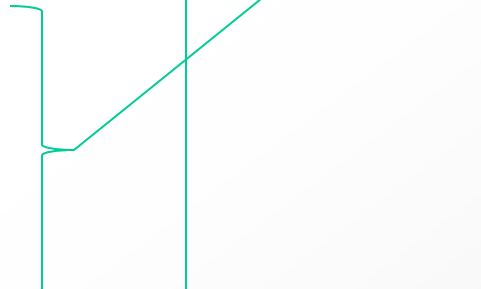
We will use this representation in the next examples:

SLLANode:

info: TElem
next: Integer


SLLA:

nodes: SLLANode []
cap: Integer
head: Integer
firstFree: Integer



SLLA:

elems: TElem[]
next: Integer[]
cap: Integer
head: Integer
firstFree: Integer



SLLAIterator:

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 – insert

- Operation insertFirst

```
subalgorithm 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)

Think about:

What strategies to manage free space can we use?

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:

```
Subalg. initFreeSpace (slla) is:  
  for i  $\leftarrow$  1, slla.cap execute  
    slla.next[i]  $\leftarrow$  i+1  
  endfor  
  slla.next[slla.cap]  $\leftarrow$  -1  
  slla.firstFree  $\leftarrow$  1  
End_subalgorithm
```

Complexity: ?

Assume:
1-based
indexing

SLLA – operation deleteElement

See SLL deleteElement !

subalgorithm deleteElement(slla, elem) is:

current \leftarrow slla.head

prev \leftarrow -1

while current \neq -1 and slla.elems[current] \neq elem execute

prev \leftarrow current

current \leftarrow slla.next[current]

end-while

if current \neq -1 then

if current = slla.head then

slla.head \leftarrow slla.next[slla.head]

else

slla.next[prev] \leftarrow slla.next[current]

end-if

freeP(current)

deleteElement \leftarrow True

else

deleteElement \leftarrow 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:

DLLANode:

info: TElem
next: Integer
prev: Integer

DLLA:

nodes: DLLANode []
cap: Integer
head: Integer
tail: Integer
firstFree: Integer

DLLA:

elems: TElem[]
next: Integer[]
cap: Integer
head: Integer
tail: Integer
firstFree: Integer

DLLAIterator:

List: DLLA
current: Integer



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: $\Theta(1)$
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.

XORNode:

info: TE Lem

link: \uparrow XORNode

XORList:

head: \uparrow XORNode

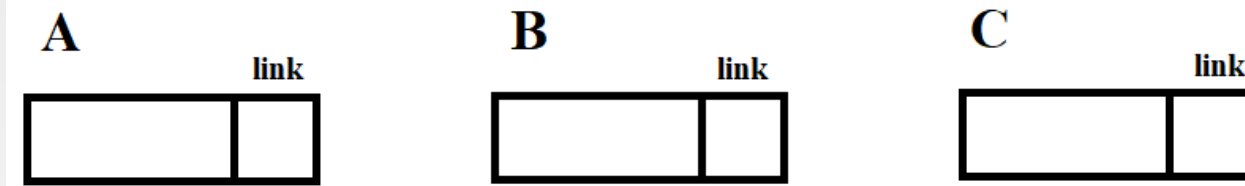
tail: \uparrow XORNode

Use dynamic representation
for XOR linked list

XOR link

XOR has the following properties:

- $A \text{ XOR } A = 0$
- $A \text{ XOR } 0 = A$
- $A \text{ XOR } B = B \text{ XOR } A$



$$\text{link}(B) = \text{addr}(A) \text{ XOR } \text{addr}(C)$$

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

$$\begin{aligned}\text{link}(B) \text{ XOR } \text{addr}(A) &= \\ &= \text{addr}(A) \text{ XOR } \text{addr}(C) \text{ XOR } \text{addr}(A) \\ &= \text{addr}(A) \text{ XOR } \text{addr}(A) \text{ XOR } \text{addr}(C) \\ &= 0 \text{ XOR } \text{addr}(C) \\ &= \text{addr}(C)\end{aligned}$$

To implement XOR linked list,
the language should support conversion between pointers and integers.
e.g.: 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 \leftarrow NIL

currentNode \leftarrow xorl.head

while currentNode \neq NIL **execute**

write [currentNode].info

 nextNode \leftarrow prevNode XOR [currentNode].link

 prevNode \leftarrow currentNode

 currentNode \leftarrow 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 \leftarrow allocate()

[newNode].info \leftarrow elem

[newNode].link \leftarrow xorl.head

if xorl.head = NIL **then**

 xorl.head \leftarrow newNode

 xorl.tail \leftarrow newNode

else

 [xorl.head].link \leftarrow [xorl.head].link XOR newNode

 xorl.head \leftarrow 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

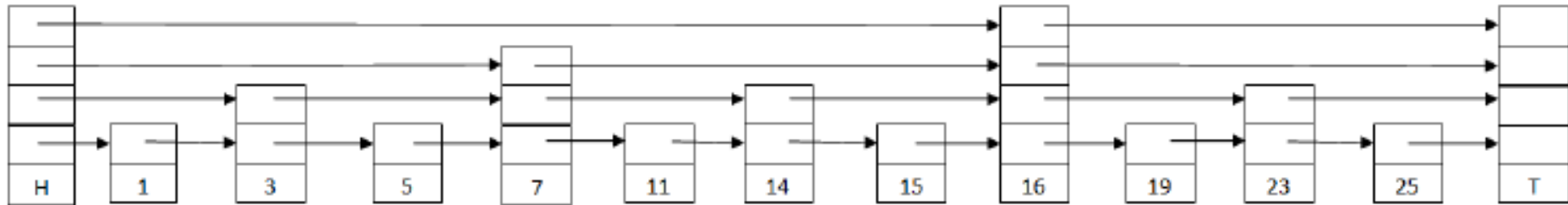
Skip lists are sorted linked lists, augmented to make the search faster

- they 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 add the skip pointers.

Skip Lists

e.g.:

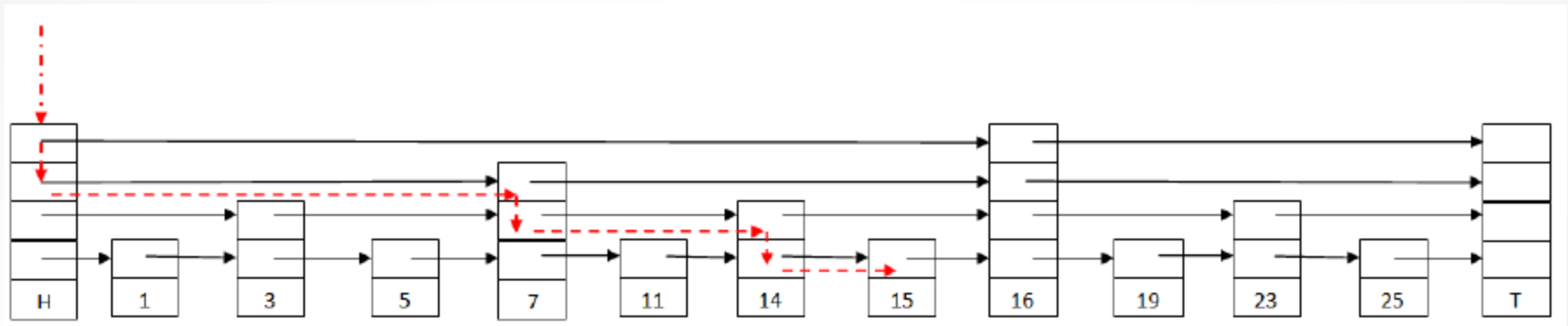


How many levels?

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



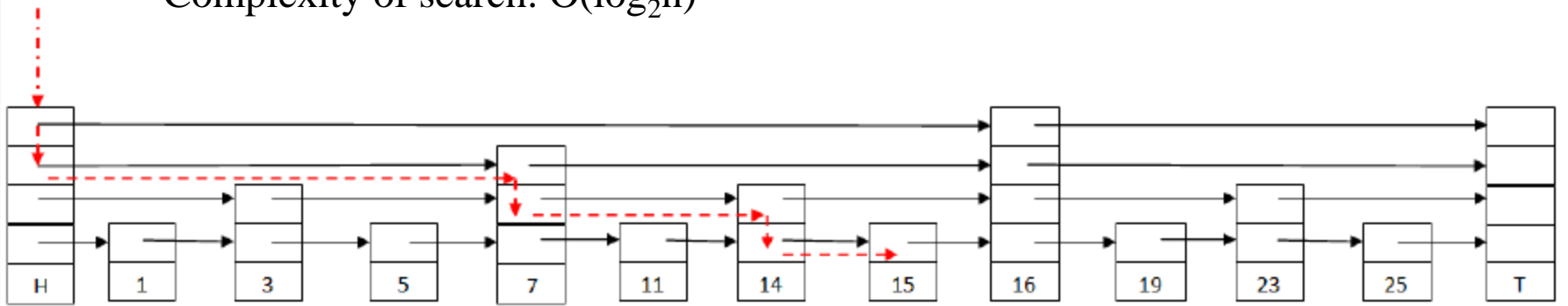
- Start from head and from highest level.
- If possible, go right.
- If cannot go right (next element is greater), go down a level.

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?

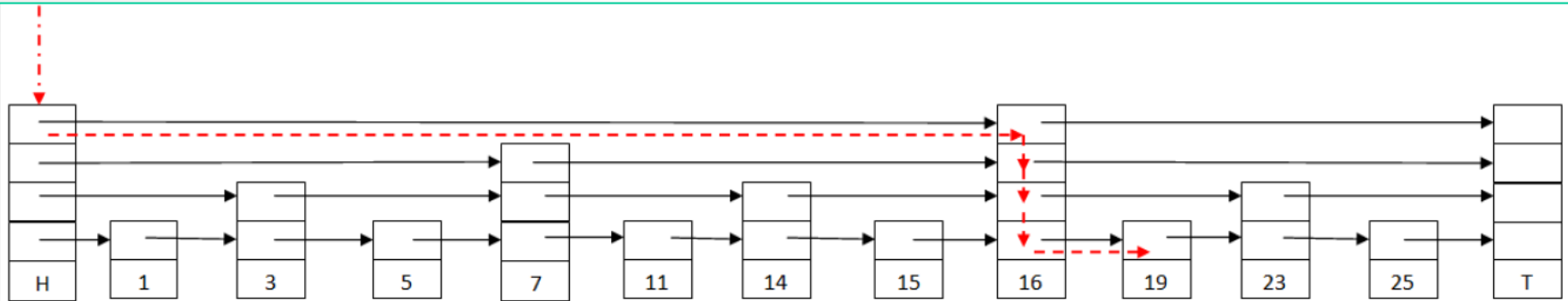
- Decide randomly the height of a newly inserted node.
- There might be a worst case, where every node has height 1 (so it is just a linked list).

Skip Lists

e.g.:

Insert element 21

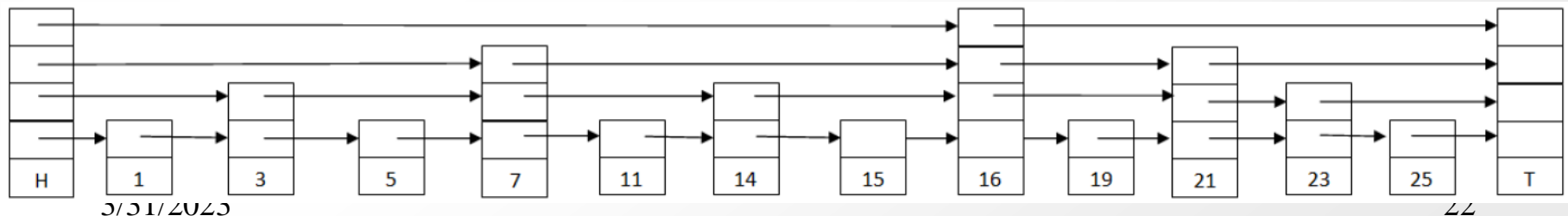
- search



- 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 *probabilistic* algorithm to set the skip pointers.
- 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.

Remark:

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.

Other Linked Structures

- Multiple Values per node

ULNode:

next: ↑ **ULNode**

elemCount: Integer

elemData: TElem [MAX]

Terminology: **unrolled linked list**

Other Linked Structures

- Multiple Links per node

Node:

info: TElem

links: List< \uparrow *Node*> *//array, record, linked*

Terminology: General Linked Lists
 Multiply Linked List
 Multi-Linked Lists

Multidimensional arrays

Matrices

e.g.: `matrix : TElem[3][5]`

C++ : `TElem matrix[3][5];`

access: `matrix[1][3]`

(the second element vertically and fourth horizontally)

Jagged arrays:

many rows of varying lengths

Jagged arrays

Example, in C++, use arrays of pointers:

```
typedef TElem* array1dim;  
typedef array1dim* array2dim;  
array1dim x,y;  
x= new TElem[7]; x[0]=15; x[1]=5;  
y= new TElem[2]; y[0]=0; y[1]=1;  
array2dim z;  
z= new array1dim[2];  
z[0]=x;  
z[1]=y;  
cout << z[0][1] << endl;
```

What can you say about creating a 2-dim array by using STL Vector ?

```
vector<vector<TElem>> arr
```