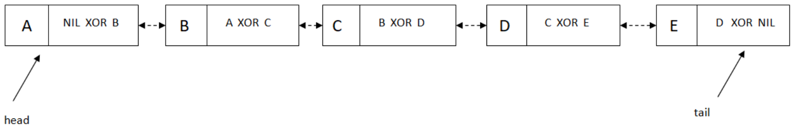
XOR Linked List

- Doubly linked lists are better than singly linked lists because they offer better complexity for some operations. They are also more flexible, since they can be traversed in both directions.
- Their disadvantage is that they occupy more memory, because you have two links to memorize, instead of just one.
- A memory-efficient solution is to have a *XOR Linked List*, which is a doubly linked list (we can traverse it in both directions), where every node retains one single link, which is the XOR of the previous and the next node.

XOR Linked List - Example



- How do you traverse such a list?
  - We start from the head (but we can have a backward traversal starting from the tail in a similar manner), the node with A
  - The address from node A is directly the address of node B (NIL XOR B = B)
  - When we have the address of node B, its link is A XOR C. To get the address of node C, we have to XOR B's link with the address of A (it is the previous node we come from): A XOR C XOR A = A XOR A XOR C = NIL XOR C = C

XOR Linked List - Representation

- We need two structures to represent a XOR Linked List: one for a node and one for the list

```
XORNode:
    info: TElem
    link: ↑ XORNode
```

```
XORList:
    head: ↑ XORNode
    tail: ↑ XORNode
```

XOR Linked List - Traversal

```
**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: $\Theta(n)$

XOR Linked List - addToBeginning

- How can we add an element to the beginning of the list?

```
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: $\Theta(1)$

Skip Lists

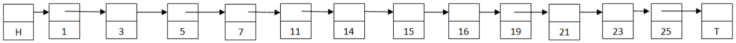
- Assume that we want to memorize a sequence of sorted elements. The elements can be stored in:
    - dynamic array
    - linked list (let's say doubly linked list)
- We know that the most frequently used operation will be the insertion of a new element, so we want to choose a data structure for which insertion has the best complexity. Which one should we choose?

Skip Lists

- We can divide the insertion operation into two steps: *finding where to insert* and *inserting the element*
    - For a dynamic array finding the position can be optimized (binary search $O(log_2 n)$), but the insertion is $O(n)$
    - For a linked list the insertion is optimal ($\Theta(1)$), but finding where to insert is $O(n)$

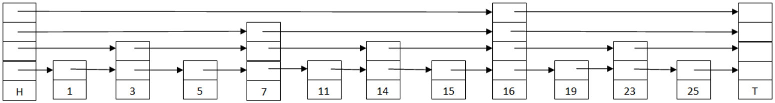
Skip List

- A skip list is a data structure that allows *fast search* in an ordered linked list.
- How can we do that?



- Starting from an ordered linked list, we add to every second node another pointer that skips over one element.
- We add to every fourth node another pointer that skips over 3 elements.
- etc.

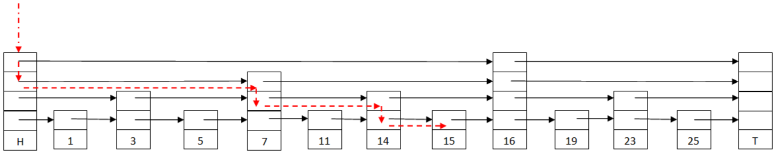
Skip List



- H and T are two special nodes, representing *head* and *tail*. They cannot be deleted, they exist even in an empty list (we need them to keep the *height* of the list).

Skip List - Search

- Search for element 15.



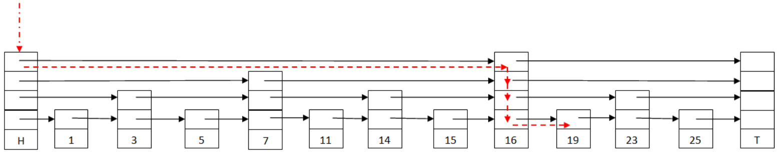
- Start from head and from highest level.
- If possible, go right.
- If cannot go right (next element is greater), go down a level.

Skip List

- Lowest level has all n elements.
- Next level has $\frac{n}{2}$ elements.
- Next level has $\frac{n}{4}$ elements.
- etc.
- $\Rightarrow$ there are approx $log_2 n$ levels.
- From each level, we check at most 2 nodes.
- Complexity of search: $O(log_2 n)$

Skip List - Insert

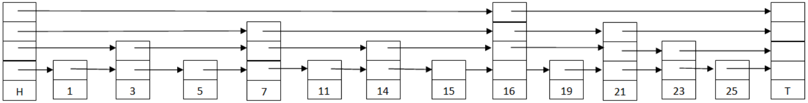
- Insert element 21.



- How *high* should the new node be?

Skip List - Insert

- *Height* of a new node is determined *randomly* with a method called *coin flip*. This method guarantees that approximately half of the nodes will be on level 2, a quarter of them on level 3, etc.



- Assume we randomly generate the height 3 for the node with 21.

Skip List

- Skip Lists are *probabilistic* data structures, since we 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).
- However, the structure (i.e. heights of nodes) is independent of the order in which the elements are inserted, so there is no *bad sequence of insertion*.
- In practice, they function well.

Skip List representation ideas

- How would you represent a skip list?
- Since this is a linked list, we need a structure for a node. What do we have in a node?
- There are two approaches for defining a node:
  - Each node has an array of pointers (so that you can have several *next* pointers for different levels)
  - Each node has one single *next* pointer and it has a *down* pointer, pointing to the same node one level below.
- Another important issue is to decide whether we will have singly or doubly linked lists. Or maybe a combination: bottom level doubly linked list, higher levels singly linked.

Skip List implementation ideas

- It is possible to keep special *head* and *tail* nodes, which always have the maximum possible height.
- While insertion and deletion seems quite straightforward, for both operations, potentially, several linked lists need to be modified. In order to be able to do so, we, potentially, need to store somewhere the nodes that we pass during the search for the position. Even in this case, there might be nodes involved, which are not on this path.
- For example, in the previously used figure, if we want to remove 16, we will immediately find it, as we start from the *head* node on the highest level. However, when it is removed, pointers of nodes 7, 14 and 15 will need to be changed.

Disadvantages of skip lists

- One disadvantage of skip lists is that they use extra space to memorize the extra links, for N elements we need 2N nodes.
- In the basic version, skip lists are unidirectional (they are singly linked lists).
- Adjacent nodes in a skip list might be stored in totally different regions of the memory, so skip lists are less optimized for caching. This disadvantage can be reduced a little, if the skip list node contains an *array* of pointers (we can benefit from the cache at least when descending a level).