VNUHCM - University of Science

Faculty of Information Technology

PROJECT REPORT

LAB 3: SORTING

Theory Lecturer:	Mr. Nguyen Thanh Phuong	
Practical Instructors:	Mr. Bui Huy Thong	
	Ms. Nguyen Ngoc Thao	
Group Members:	Ho Thanh Nhan	21127122
	Nguyen Tien Bach	21127223
	Tran Thanh Quy	21127411

Course: Data Structure and Algorithms

Term: 3 - Academic Year: 2021 - 2022

TABLE OF CONTENTS

1 Self-organizing list

1.1 Background

1.1.1 Introduction

The simplest implementation of a self-organizing list is as a linked list, and thus, while being efficient in random node inserting and memory allocation, it suffers from inefficient accesses to random nodes. A self-organizing list reduces the inefficiency by dynamically rearranging the nodes in the list based on access frequency.

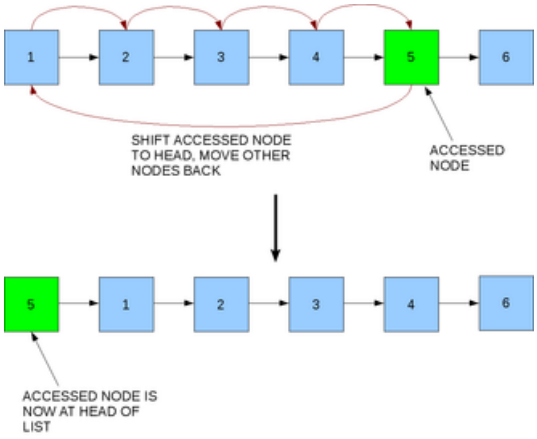

Figure 1: Self-organizing list

1.1.2 History

The concept of self-organizing lists has its roots in the idea of active organization of records in files stored on disks or tapes. One frequently cited discussion of self-organizing files and lists is that of Knuth.

1.1.3 Applications

The aim of a self-organizing list is to improve the efficiency of linear search by moving the most frequently accessed items towards the head of the list.

Application:

1. Language translators like compilers and interpreters use self-organizing lists to maintain symbol tables during compilation or interpretation of program source code.

2. These lists are also used in artificial intelligence and neural networks, as well as self-adjusting programs.

3. The algorithms used in self-organizing lists are also used as caching algorithms, as in the case of the LFU algorithm.

1.2 Variations

There are three popular techniques for rearranging nodes in a self-organizing list:

1. Move to front method (MTF)
2. Count method
3. Transpose method

1.2.1 Move to front method (MTF)

Any node or element requested is moved to the front of the list.

Pros

1. This method is easily implemented and does not require any extra memory or storage
2. This method easily adapts to quickly changing access patterns. Even if the priorities of the nodes change dynamically at runtime, the list will reorganize itself very quickly in response.

Cons

1. This method may prioritize infrequently accessed nodes: for example, if an uncommon node is accessed even once, it is moved to the head of the list and given maximum priority even if it is not going to be accessed frequently in the future. These 'over rewarded' nodes clog up the list and lead to slower access times for commonly accessed elements.
2. This method, though easily adaptable to changing access patterns, may change too frequently. Basically, this technique leads to very short memories of access patterns whereby even an optimal arrangement of the list can be disturbed immediately by accessing an infrequent node in the list.

Pseudo-code of MTF method:

```
At the t-th item selection:
    if item i is selected:
        move item i to head of the list
```

1.2.2 Count method

Every node counts the number of times it has been searched for. Every node keeps a separate counter variable which is incremented every time it is called. The nodes are then rearranged according to decreasing count.

Pros

1. Reflects more realistically the actual access pattern.

Cons

1. Must store and maintain a counter for each node, thereby increasing the amount of memory required.

2. Does not adapt quickly to rapid changes in the access patterns.
   For example: if the count of the head element is say A is 100 and for any node after it say B is 40. now, even if B becomes the new most commonly accessed element, it must still be accessed at least (100 - 40 = 60) times before it can become the head element.

Pseudo-code of Count method:

```
init: count(i) = 0 for each item i
    At t-th item selection:
        if item i is searched:
            count(i) = count(i) + 1
            rearrange items based on count
```

1.2.3 Transpose method

This method includes exchanging an accessible node for its predecessor. As a result, whenever a node is accessed, its priority is increased by swapping it with the node in front, unless it is the head node. This technique, like the previous one, is simple to construct and takes up little space, and it is more likely to retain frequently used nodes near the top of the list.

Pros

1. Easy to implement and requires little memory.

2. More likely to keep frequently accessed nodes at the front.

Cons

1. More cautious than move to front. It will take many accesses to move the element to the head of the list.

Pseudo-code of Transpose method:

```
At the t-th item selection:
    if item i is selected:
        if i is not the head of list:
            swap item i with item (i - 1)
```

1.3 Step-by-step Descriptions

A self-organizing linked list that already has some elements:

Figure 2: A self-organizing list

Item insertion - insert by index:
Insert 80 to index 2:

- Step 1: Find the corresponding position to insert.

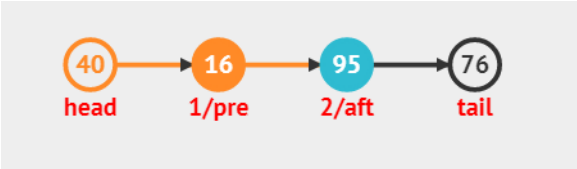

Figure 3: Find the position to insert.

- Step 2: Create new node contain the value need to be inserted.

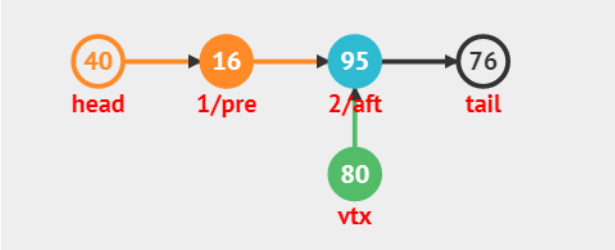

Figure 4: Create new node.

- Step 3: Point the pointer of new node to the node at index 2. Then move the pointer of index 1 to the new node.

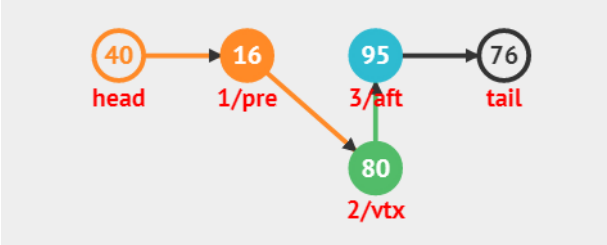

Figure 5: Assign new node to the list.

- After insert:

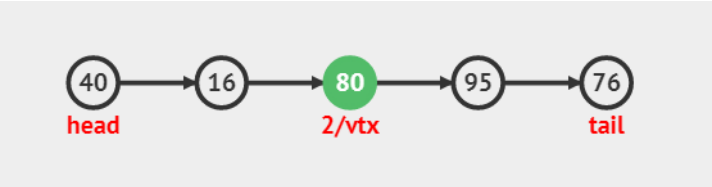

Figure 6: After insert.

The Count Method is always chosen to add to the tail of the list, cause it's the item which has the least time being searched.

Item removal - remove head:

- Step 1: Assign head to temporary variable.

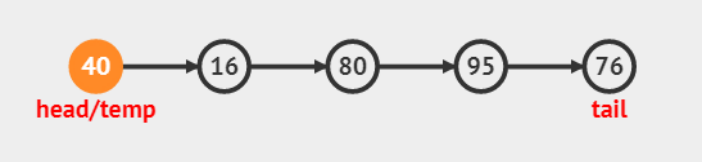

Figure 7: Assign head to temp.

- Step 2: Assign head to head→next then delete temporary variable.

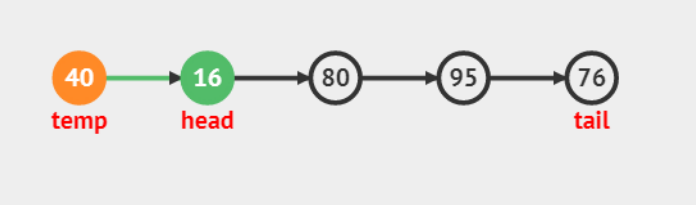

Figure 8: Move head to next position and delete tmp.

- After remove head:

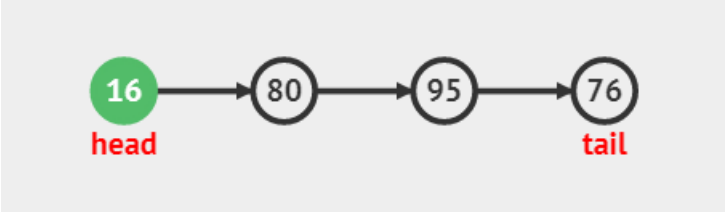

Figure 9: After delete.

1.4 Complexity Evaluations

The time complexity for each basic operation:

- Check empty of a linked list: Always run with constant time (O(1)) because the solution is checking the head(NULL or not).

- Get the number of items in a linked list; Build a linked list from given items; Remove all items from the linked list: Always run with O(n) because it must reach to all the items of the list.

1.4.1 Insert operation

With the Count method, the insertion is only to the tail of the list, because it is the most suitable way. So at first declaration of a structure, we add a "node" call Tail for easier connection. With the Move-To-Front method and Transpose method, we can choose where to insert a new element (because it doesn't depend on the number of time each element appears. And to insert an element to a position that we want, it will start from the head and reach to the tail. There is a conclusion about the above evaluations:

- Best case: The best case is O(1). (This is a case that always happen in Count method. With Move-To-Front method and Transpose method, the best case happens when we choose to add new element to the head (position: 0)).

- Worst case:

  + With Count method: Worst case is still O(1) because it usually adds new element to the tail.

  + With Move-To-Front method and Transpose method: Worst case is O(n) when the new element was being chosen to add to the position behind the last "node" of the list.

1.4.2 Delete operation

- Best case: The best case of 3 methods is O(1). It happens when the node chosen to remove is standing at the head of the list.

- Worst case: The worst case of 3 methods is O(n), because before removing a "node", the first step is searching for the "node" and it will run in linear time. So the worst case is O(n).

1.4.3 Search operation

1. **Best case:** The node to be searched is one which has been commonly accessed and has thus been identified by the list and kept at the head. This will result in a near constant time operation. In the best case, accessing an element is an $O(1)$ operation.

2. **Worse case:** The element to be found is at the very end of the list, and hence n comparisons must be conducted to reach it. As a result, the worst-case running time of a linear search on the list is $O(n)$.

3. **Average case:** It can be shown that in the average case, the time required to a search on a self-organizing list of size n is:
   $Tavg = 1 * p(1) + 2 * p(2) + ... + n * p(n)$
   where $p(i)$ is the probability of accessing the ith element in the list, thus also called the access probability. If the access probability of each element is the same ($p(1) = p(2) = p(3) = ... = p(n) = 1/n$) then the ordering of the elements is irrelevant and the average time complexity is given by:
   $Tavg = \frac{1+2+3+...+n}{n} = \frac{n+1}{2}$
   $\Rightarrow$ Average case running time is $O(n)$

1.5 Similarities and dissimilarities to a regular linked list

1.5.1 Similarities

- The implementation and methods of a self-organizing list are identical to those of a standard linked list.

- A self-organizing list has the basic operations: check empty, insert, delete, and search.

1.5.2 Dissimilarities

The linked list and the self-organizing list differ only in terms of the organization of the nodes.

Main difference: after access 1 node:

- Singular linked list: The order of the nodes in the linked list remains unchanged.

- The self-organizing list: The order of nodes is updated after each access.

2 XOR linked list

2.1 Background

2.1.1 Introduction

An XOR linked list is a type of data structure used in computer programming. It takes advantage of the bitwise XOR operation to decrease storage requirements for doubly linked lists. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of the addresses of the previous and next nodes.

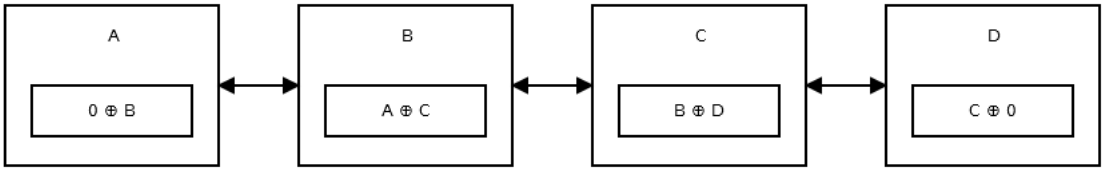

Figure 10: XOR linked list.

2.1.2 History

Knuth introduced the concept of self-organizing lists in the Linear Linked Structures chapter in 2009.

2.1.3 Applications

Because it is memory efficient and employs bitwise XOR to save space for one address, it can be used to replace doubly linked lists.

2.2 Variations

- The underlying principle of the XOR linked list can be applied to any reversible binary operation.

- Replacing XOR by addition or subtraction gives slightly different, but largely equivalent, formulations:

2.2.1 Addition linked list

This kind of list has exactly the same properties as the XOR linked list, except that a zero-link field is not a "mirror". The address of the next node in the list is given by subtracting the previous node's address from the current node's link field.

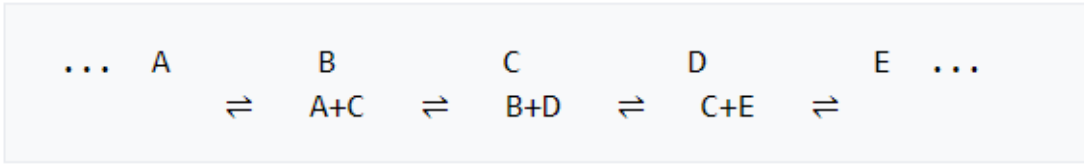

Figure 11: Subtraction linked list.

2.2.2 Subtraction linked list

This kind of list differs from the standard "traditional" XOR linked list in that the instruction sequences needed to traverse the list forward are different from the sequences needed to traverse the list in reverse.

The address of the next node, going forward, is given by adding the link field to the previous node's address; the address of the preceding node is given by subtracting the link field from the next node's address.

The subtraction linked list is also special in that the entire list can be relocated in memory without needing any patching of pointer values, since adding a constant offset to each address in the list will not require any changes to the values stored in the link fields. (See also serialization.) This is an advantage over both XOR linked lists and traditional linked lists.

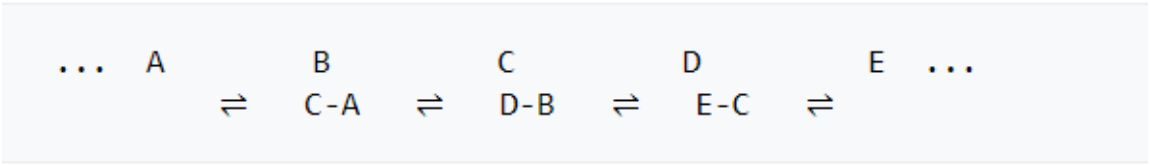

Figure 12: Subtraction linked list.

2.2.3 Binary search trees (BSTs)

The same concept of XOR list can be applied to XOR BSTs, allowing 2 directional traversal in the BSTs

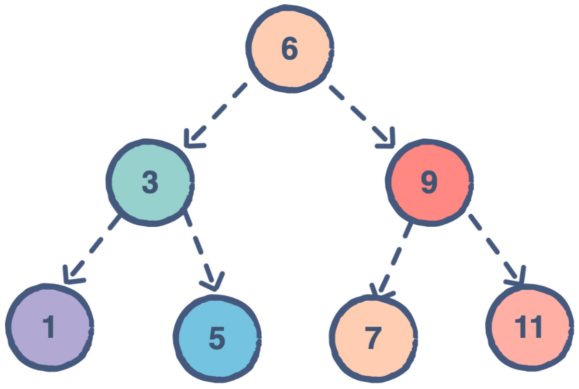

Figure 13: Binary search tree.

2.3 Step-by-step Descriptions

An XOR linked list that already has some elements:

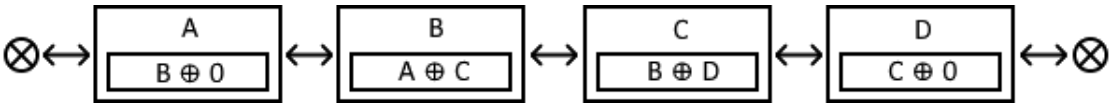

Figure 14: XOR list.

Insert by index:

- Step 1: Find the corresponding position to insert.

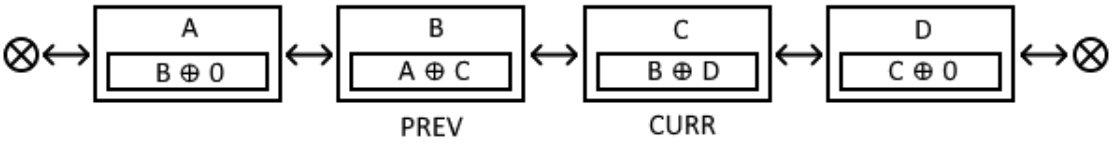

Figure 15: Find position.

- Step 2: Create new node with its link set to the XOR of the address of nodes PREV(B) and CURR(B).

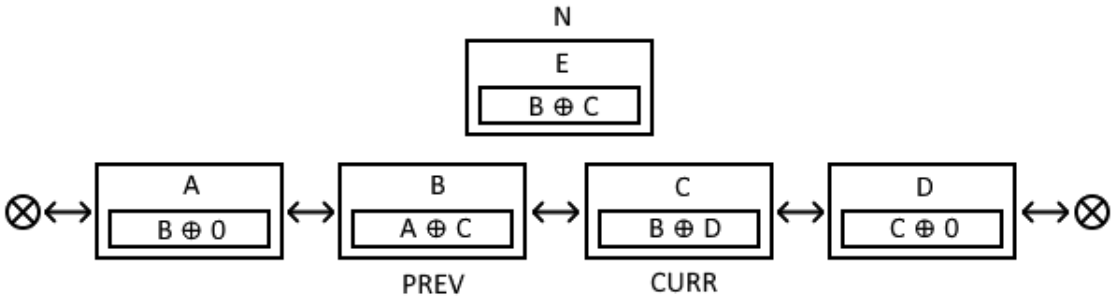

Figure 16: Create new node.

- Step 3: XOR the link of PREV(B) and CURR(C) to each another node's address to negates them.

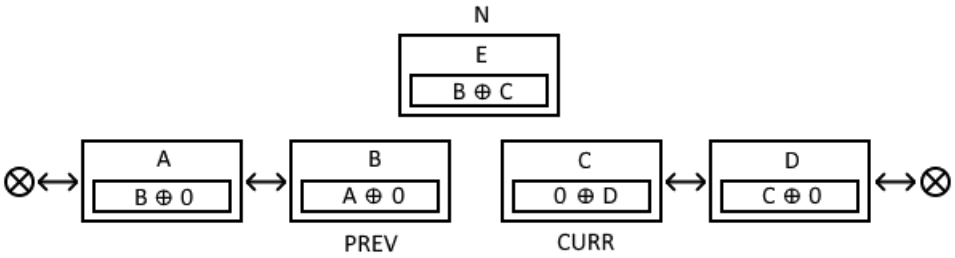

Figure 17: Link the new node to the list.

- Step 4: XOR the link in of PREV(B) and CURR(C) for the address of N(E).

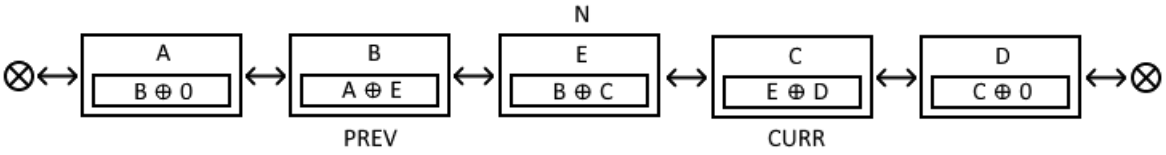

Figure 18: After the insertion process.

Delete by index:

- Step 1: Find the corresponding postion node need to be deleted.

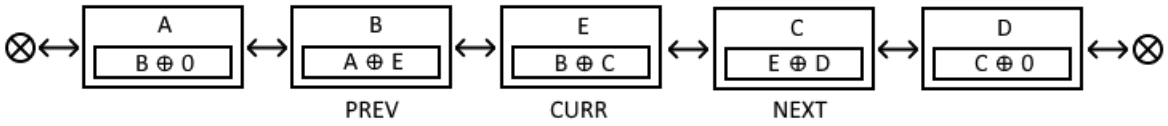

Figure 19: Find index.

- Step 2: XOR the link of PREV and NEXT to the address of CURR(E).

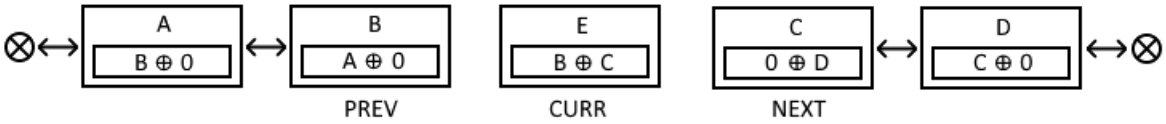

Figure 20: Delete.

- Step 3: XOR the link in PREV(B) and NEXT(C) to each another node's address.

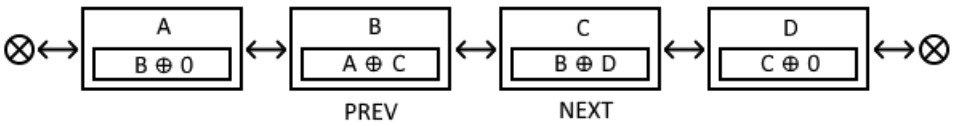

Figure 21: After the delete process.

Delete head of the list:

- Step 1: Locates heading node for deletion(A) and the new heading node(B)

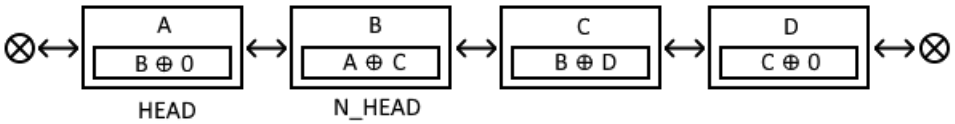

Figure 22: List in question for deletion operation.

- Step 2: XOR the link of the new head (B) to the current heading node (A).

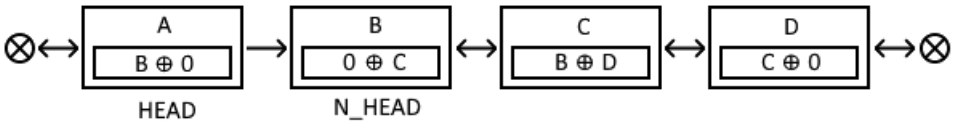

Figure 23: Change the link for the new heading node.

- Step 3: Set B to be the new heading node.

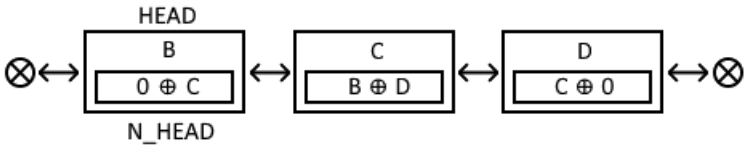

Figure 24: After the delele head process.

2.4 Complexity Evaluations

2.4.1 Insertion/Deletion operation(s)

- Best case: O(1) insertion/deletion at the head of the list.
- Worst case: O(n) insertion/deletion at the end(tail) of the list as we have to iterate though the whole list.

2.4.2 Count the number of nodes in the list

- O(n) in all cases for we always have to iterate though the whole list.

2.4.3 Check whether the list is empty

- O(1) in all cases, as we have to check the pointed head node only.

2.5 Similarities and dissimilarities to a regular linked list

2.5.1 Similarities

- An XOR linked list is a modified version of a doubly linked regular list. Each node is just a "pointer" instead of two. That "pointer" includes the XOR of the following pointer and the previous pointer.
- The implementation and methods of an XOR list are similar to those of a standard linked The XOR linked list allows two-way traversal.list.

2.5.2 Dissimilarities

1 Nodes traversal:

- The Xor linked list allows traversing nodes in 2 directions.
- The standard linked list can only traverse nodes in one direction.

2 Nodes values:

- Every node of a Xor linked list stores the XOR of the addresses of the previous and next nodes.
- Every node of a standard linked list stores the normal values and addresses of the previous and next nodes.

3 References

List of references:

1. Get the xor function.
2. Background and variations of self-organizing list.
3. Background and variations of XOR list.
4. Complexity of search function of self-organizing list.
5. Similarities of XOR list.
6. Visualizations of list examples.

4 Notes

Source code of self-organizing list must run on console.