Self Organising List using Move-To-Front method using Doubly Linked List

1. Efficiency:

- MTF offers constant-time access, insertion, and deletion operations, ensuring efficient performance.

- DLL provides efficient traversal and pointer manipulation, supporting MTF reorganization operations with a time complexity of O(n).

2. Temporal Locality:

- MTF leverages temporal locality by prioritizing recently accessed elements, leading to improved average access time.

- Doubly Linked List allows for easy adjustment of pointers to move elements to the front, facilitating MTF reorganization without significant overhead.

3. Simplicity:

- MTF is straightforward to implement and understand, with minimal overhead compared to more complex methods like Count.

- Doubly Linked List offers simplicity in insertion, deletion, and traversal, making it well-suited for MTF implementation without sacrificing efficiency.

4. Comparison with other methods:

- Transpose and Count methods, while offering some benefits, may not provide as significant improvements in average access time as MTF.

- Doubly Linked List outperforms other data structures like Binary Search Trees, Queues, or Stacks in supporting MTF reorganization efficiently.

5. Conclusion:

- With its balance of efficiency, simplicity, and effectiveness in leveraging temporal locality, the Move-to-Front method using a Doubly Linked List emerges as the optimal choice for implementing a self-organizing list.

- Therefore the Doubly Linked List with the Move-to-Front method remains the best-suited approach for efficiently maintaining access patterns and optimizing average access time in a wide range of scenarios.

Algorithm 1: Insertion of an element into the SOL
Input: head, num: The integer value to be inserted into the linked list.
Output: true (if the insertion is successful) / false (if memory allocation fails).

```
1. Create a new node and initialize the new node.
2. newnode->data <- num
3. newnode->next <- nullptr
4. If head is nullptr (the list is empty)
     a. newnode->prev <- nullptr
     b. head <- newnode
     c. tail <- newnode
     d. Return true
5. Otherwise (insert at the end of non-empty list):
     a. newnode->prev <- tail
     b. tail->next <- newnode
     c. tail <- newnode
     d. Return true
```

Algorithm 2: Deletion of an element into the SOL
Input: head, num: The integer value to be deleted from the linked list.
Output: true (if the deletion is successful) / false (if the element is not found in the list or the list is empty).

```
1. Initialize a temporary node temp <- head
2. Repeat until temp is not nullptr
     a. If temp->data is num
          i. If temp is head
               1. head <- head->next
               2. If head is not nullptr
                    a. head->prev <- nullptr
               3. Otherwise
                    a. tail is nullptr
          ii. Otherwise if temp is tail
               1. tail is tail->prev
               2. If tail is not nullptr
                    a. tail->next <- nullptr
               3. Otherwise
                    a. head <- nullptr
          iii. Otherwise
               1. temp->prev->next <- temp->next
               2. temp->next->prev <- temp->prev
          iv. Delete temp
          v. Return true
```

```
        b. temp <- temp->next
3. Otherwise
        a. Return false
```

Algorithm 3: MTF (Move To Front) – move an element to the front
Input: head, temp: A pointer to the node that needs to be moved to the front of the linked list.
Output: The linked list is modified such that the node originally pointed to by temp becomes the new head node.

```
1. If temp is head
        a. Return
2. Otherwise if temp is tail
        a. tail <- temp->prev
        b. tail->next <- nullptr
3. Otherwise
        a. temp->next->prev <- temp->prev
4. temp->prev->next <- temp->next
5. temp->prev <- nullptr
6. head->prev <- temp
7. temp->next <- head
8. head <- temp
```

Algorithm 4: Access an element in the SOL
Input: head, num: The integer value to be searched for in the linked list.
Output: true (if the element with the value num is found in the list) / false (if the element is not found in the list).

```
1. Initialize a pointer variable target <- head
2. Repeat until target is not nullptr
        a. If target->data is num
                i. Call function movetofront(target)
                ii. Return true
        b. target <- target->next
3. Otherwise
        a. Return false
```

Algorithm 5: Display elements of the SOL
Input: head
Output: Prints the contents of the self-organizing list to the console. If the list is empty, an appropriate message is displayed.

```
1. Initialize a temporary node temp <- head
2. If head is nullptr
        a. Print 'The list is empty'
3. Otherwise
        a. Print 'Contents of the self organizing list:'
        b. Repeat until temp is not nullptr
                i. Print temp->data
```

```
ii. temp <- temp->next
```

<u>FUNCTIONAL REQUIREMENTS:</u>

1. insertion()
2. deletion()
3. movetofront()
4. access()
5. display()

<u>TIME COMPLEXITY ANALYSIS FOR EACH ALGORITHM:</u>

<u>1. insertion() – O(1) (Constant Time)</u>

$f(n) = 9$
$g(n) = 1$ and $c = 10$

n	Cost of f(n)	Cost of c.g(n)
0	9	10
1	9	10

Therefore $n_0 = 0$ for which $f(n) <= c.g(n)$ is true for all $n_0 < n$.
Therefore $f(n) = O(n)$

<u>Analysis summary:</u>

- Best Case: O(1) (Constant Time)
- Average Case: O(1) (Constant Time)
- Worst Case: O(1) (Constant Time)

<u>Explanation:</u>

The provided insertion method specifically inserts new nodes at the end of the linked list. This characteristic leads to constant time complexity in all cases:

- Best Case (O(1)): Even if the list is empty (head == nullptr), only a few constant time operations are needed to set the new node as both head and tail.
- Average Case (O(1)): Regardless of the initial list state (empty or non-empty), inserting at the end always involves a fixed number of pointer updates, making the average time complexity constant.
- Worst Case (O(1)): The worst case scenario, where the list might be very large, doesn't affect the time complexity. Since we're only manipulating pointers at the end of the list for insertion, the number of operations remains constant, resulting in O(1) complexity.

2. deletion()– O(n)

$f(n) = 9n+3$
$g(n) = n$ and $c = 10$

n	Cost of f(n)	Cost of c.g(n)
0	3	0
1	12	10
2	21	20
3	30	30
4	39	40

Therefore $n_0 = 3$ for which $f(n) <= c.g(n)$ is true for all $n_0 < n$.
Therefore $f(n) = O(n)$

Analysis summary:

- Best Case: O(1) (Constant Time)
- Average Case: O(n) (Linear Time)
- Worst Case: O(n) (Linear Time)

Explanation:

The time complexity of the deletion method depends on how long it takes to find the target element in the list.

- Best Case (O(1)): This occurs when the target element is at the head of the list. In this case, the first iteration of the loop finds the element, and the deletion process involving pointer updates and memory deallocation takes constant time.
- Average Case (O(n)): On average, the element might be located somewhere in the middle of the list. The loop needs to iterate potentially through all n elements in the worst case to reach the target. The average number of iterations might be less than n but still proportional to the list size.
- Worst Case (O(n)): The worst case scenario happens when the target element is at the end of the list or isn't present in the list at all. The loop iterates through the entire list (n elements) before exiting without finding a match.

3. movetofront()– O(1) (Constant Time)

$f(n) = 9$
$g(n) = 1$ and $c = 10$

n	Cost of f(n)	Cost of c.g(n)
0	9	10
1	9	10

Therefore $n_0 = 0$ for which $f(n) <= c.g(n)$ is true for all $n_0 < n$.

Therefore f(n) = O(1)

Analysis summary:

- Best Case: O(1) (Constant Time)
- Average Case: O(1) (Constant Time)
- Worst Case: O(1) (Constant Time)
-

Explanation:

- Best Case (O(1)): This occurs when temp is already the head of the list. In this case, the first check (temp == head) leads to an immediate return, requiring only a constant time comparison.
- Average Case (O(1)): Even for a non-special case (where temp is not the head or tail), the pointer updates and assignments are all constant time operations. The average time complexity remains constant because the method performs the same set of operations irrespective of the initial position of temp within the list.
- Worst Case (O(1)): The worst case also results in constant time complexity. Regardless of the list's size or temp's position, the method involves a fixed number of pointer manipulations and assignments, leading to O(1) complexity.

4. access()– O(n)

f(n) = 5n+3
g(n) = n and c = 6

n	Cost of f(n)	Cost of c.g(n)
0	3	0
1	8	6
2	13	12
3	18	18
4	23	24

Therefore n_0 = 3 for which f(n) <= c.g(n) is true for all n_0 < n .
Therefore f(n) = O(n)

Analysis summary:

- Best Case: O(1) (Constant Time)
- Average Case: O(n) (Linear Time)
- Worst Case: O(n) (Linear Time)
-

Explanation:

- Best Case (O(1)): This occurs when the target element is at the head of the list. In this case, the first iteration of the loop finds the element, and the subsequent call to movetofront has constant time complexity (as analyzed previously). The overall time complexity becomes constant (O(1)).

- Average Case (O(n)): On average, the element might be located somewhere in the middle of the list. The loop needs to iterate potentially through all n elements in the worst case to reach the target. The average number of iterations might be less than n but still proportional to the list size, leading to an average time complexity of O(n).
- Worst Case (O(n)): The worst case scenario happens when the target element is at the end of the list or isn't present in the list at all. The loop iterates through the entire list (n elements) before exiting without finding a match. The additional call to movetofront (which is O(1)) doesn't affect the dominant factor, which is the search loop.

5. display()– O(n)

f(n) = 3n+5
g(n) = n and c = 4

n	Cost of f(n)	Cost of c.g(n)
0	5	0
1	8	4
2	11	8
3	14	12
4	17	16
5	20	20
6	23	25

Therefore n_0 = 5 for which f(n) <= c.g(n) is true for all n_0 < n .
Therefore f(n) = O(n)

Analysis summary:

- Best Case: O(n) (Linear Time)
- Average Case: O(n) (Linear Time)
- Worst Case: O(n) (Linear Time)

Explanation:

- Best Case (O(n)): Even if the list has only one element (head), the loop needs to iterate once to print that element's data. This results in linear time complexity (O(n)).
- Average Case (O(n)): There's no scenario where the loop iterates less than the number of elements (n) in the list. On average, the loop iterates through all elements, leading to O(n) complexity.
- Worst Case (O(n)): The worst case also exhibits linear time complexity. Regardless of the data arrangement in the SOL (random, sorted, etc.), the loop needs to visit all elements once to print their data, resulting in O(n) complexity.

<u>APPLICATIONS:</u>

Some common applications of SOLs:

1. <u>Cache Management:</u>

- In computer systems, caches store frequently accessed data from main memory to improve retrieval speed. SOLs can be employed in cache management to prioritize data items that are accessed more often. By moving recently accessed data to the front of the list, subsequent retrievals become faster.

2. <u>Recently Used Items (RUIs) Lists:</u>

- Many applications maintain lists of recently used items (RUIs) to provide users with quick access to previously interacted-with elements. SOLs can be used for these RUI lists, ensuring that the most recently used items are at the beginning for faster retrieval. This can be seen in features like recently opened files or browsing history.

3. <u>Branch Prediction in Processors:</u>

- Modern processors employ branch prediction to speculate on which branch (true or false) of a conditional statement is more likely to be taken. SOLs can be used to store branch history information, with frequently taken branches positioned closer to the front for faster prediction decisions.

4. <u>Network Routing:</u>

- In networking scenarios, routers maintain routing tables to determine the best path for data packets. SOLs can be used in these routing tables to prioritize frequently used routes, potentially improving network performance.

5. <u>Database Query Optimization:</u>

- Database systems can benefit from SOLs to optimize query execution. By keeping frequently accessed data structures or query patterns at the forefront of an SOL, subsequent queries referencing that data might experience faster execution times.

6. <u>Spell Checkers and Autocomplete:</u>

- Spell checkers and autocomplete features rely on maintaining a dictionary of words. SOLs can be used for these dictionaries, keeping frequently misspelled words or user-typed prefixes closer to the beginning for faster suggestion retrieval.

7. <u>Intrusion Detection Systems (IDS):</u>

- IDS monitor network traffic for suspicious activity. SOLs can be used to store patterns of known attacks or suspicious IP addresses. By prioritizing recently encountered patterns, the IDS might be able to identify potential threats more efficiently.

8. <u>Web Server Request Logging:</u>

- Web servers often log access requests. SOLs can be used to maintain these logs, keeping recently accessed pages or user information closer to the front for faster retrieval during analysis or troubleshooting.