Implementation, Analysis and Application of Retroactive Data Structures

Suneeta Agarwal
Professor, Computer Science and Engineering Department
Motilal Nehru National Institute of Technology
Allahabad, India 211004

Prakhar Panwaria
Ex-Student, Computer Science and Engineering Department,
Motilal Nehru National Institute of Technology
Allahabad, India 211004

***Abstract* — Retroactive data structures, introduced by Demaine, Iacono and Langerman [1], represent a model through which operations can be performed on a data structure in past time. Here, in this paper, we focus on the data structures for which retroactivity hasn't been implemented yet and the enhancements in existing Retroactive data structures. Besides going through the data structures used for their implementation, we also analyze their performance and demonstrate their usefulness through real life use-cases and applications.**

***Keywords* — data structure, reteroactive data structure**

I. INTRODUCTION

Talking generically, a data structure is created using two operations – *Insert(x)* and *Delete(x). Insert(x)* operation inserts '*x*' value in the data structure, whereas *Delete(x)* operation removes '*x*' value from the data structure. While creating such data structure, what if a value is mistakenly inserted or deleted at a particular instance in past? How do we regenerate that data structure with correct values in present? There are generally two solutions that are followed. First is performing 'roll back' operation on the current state of data structure, where we roll back all the operations performed on the data structure just before that particular instance and then again perform the subsequent operations but, this time, without that faulty operation. But, this is considered as an inefficient and time consuming solution. Second solution is to use 'Persistent Data Structures', where whenever we perform an operation on the data structure, we maintain its different versions so that we may easily reach the version of the data structure created through a wrong operation and then follow up with subsequent operations to finally get the desired data structure. But, the solution is considered inefficient from space point of view.

Retroactive data structures aim to solve this problem in both the dimensions by maintaining the sequence of operations performed on that data structure in another data structure, and use it to evaluate the actual state of the concerned data structure if a wrong operation was performed on it in past time.

A. *Motivation*

Retroactive data structures find a vast scope of application as an error correction technique in real life scenarios. This paradigm can also be used in the situations where we might want to find out the state of the system with different sequences of operations performed on it in the past. Going through the study done by Demaine, Iacono and Langerman [1] around Retroactive data structures, the concept and its large scope of application intrigued us to perform a deep research and analysis on the subject and apply the idea in the very basic scenarios. Thus, in this paper, we try to implement retroactive data structures using different algorithms and also try to develop retroactivity in other data structures. To achieve the main goal, the following have to be accomplished:

- Good understanding of the existing data structures.
- Understanding the retroactive versions of these data structures.
- Comparison with existing techniques, like persistent data structures and rollback methods.
- Selecting the best data structures for achieving retroactivity efficiently.

B. *Important Terms*

The following terms are important to understand the subject discussed later in the paper:

1. *Partially Retroactive:* A data structure is considered as 'Partially Retroactive' if we can perform an operation on that data structure in the past and get its corresponding result in the present.
2. *Fully Retroactive:* A data structure is considered as 'Fully Retroactive' if along with having properties of Partially Retroactive data structure, we are also able to answer any queries to the data structure made in the past.

C. *Outline*

The paper follows with the description of data structures used to implement retroactivity in Binary Search Tree and Hash in Section 2. In Section 3, we try to compare the existing data structure used for implementing Fully Retroactive Union-Sameset with our proposed data structure and analyze their performance results. Section 3 discusses the applications and different use-cases of retroactivity in data structures.

II. Data Structures Implemented

Consistency Note: We are assuming that at no point of time, the retroactive data structure is inconsistent. All the checks regarding consistency can be implemented very easily.

A. *Fully Retroactive Binary Search Tree*

Following are the details about the supported operations and the data structure used to implement Fully Retroactive BST.

Normal Operations Supported:
Update operation:

1. *insert (x):* insert an element in the binary search tree.

Query operations:

1. *find_root ():* returns the root of the binary search tree.
2. *structure ():* returns the whole structure of the binary search tree.
3. *search (x):* searches an element 'x' within the binary search tree.

Retroactive Operations Supported:
Update operations:

1. *insert_operation (insert(x), t):* inserts the *insert(x)* operation at time 't', i.e. it inserts an element '*x*' in the binary search tree at time 't'.
2. *delete_operation (t):* deletes the *insert(x)* operation at time 't', i.e. it deletes an element '*x*', which had been inserted at time 't', from the binary search tree.

Query operations

1. *find_root (t):* returns the root of the binary search tree at time 't'.
2. <u>*structure (t):*</u> returns the whole structure of the binary search tree at time '*t*'.
3. *search (x,t):* finds out whether an element '*x*' is present in the binary search tree at time '*t*'.

As mentioned earlier, the fully retroactive data structure will not only allow queries at the present time but also at past time, and also the updates of operations at any point of time.

Data Structure Used:

- Binary Search Tree, in which all nodes are linked through doubly linked list.
- Each node of the tree includes following fields:
  - *v*: value of the node.
  - *t*: time at which node is inserted.
  - *lc:* pointer to Left Child.
  - *rc:* pointer to Right Child.
  - *ls:* pointer to Left-Hand-Side Node of doubly linked list.
  - *rs:* pointer to Right-Hand-Side Node of doubly linked list.
  - *par:* pointer to Parent Node.

insert (x) operation:
This is the main update operation used to create the binary search tree. The tree structure can be created using two methodologies:

1. *Preserving the structure:*
   The binary search tree can be created while preserving the structure. By preserving the structure, we mean to say that suppose if an element '*x*' is inserted at any time '*t*' using *insert_operation(Insert(x), t),* the structure is created as if in reality the binary search tree was created using the *insert(x)* operation at time '*t*'. So, '*x*' should be present in the exact position like if it would have been inserted at time '*t*' using *insert(x)* operation instead of retroactive operation *insert_operation(Insert(x), t).*

2. *Without preserving the structure:*
   Another method of creating the binary search tree retroactively is without preserving its structure, i.e. if an element is inserted at any time 't' using retroactive operation *insert_operation(Insert(x), t)* , it can be inserted as if it is inserted at the present time, however the time field of the node inserted will hold the value equal to time '*t*', since in binary search tree there is no difference where we insert a value in it if we are just concerned with its 'searching' operation.

Average Case Complexity:
After implementation, we observed the following average case complexities for different retroactive operations as:

Operations	**Preserving the structure**	**Without Preserving the structure**
insert_operation (Insert(x), t)	*O(log n)*	*O(log n)*
delete_operation (t)	*O(log n)*	*O(log n)*
find_root (t)	*O(1)*	*O(1)*
search (x,t)	*O(log n)*	*O(log n)*

* '*n*' is the total number of elements in the binary search tree.

B. *Fully Retroactive Hash*

Following are the details about the supported operations and the data structure used to implement Fully Retroactive BST.

Normal Operations Supported:
Update operations:

1. *insert (x):* inserts an element in the hash.
2. *delete (x):* deletes an element in the hash.

Query operation:

1. *search (x):* searches an element 'x' within the hash.

Retroactive Operations Supported:
Update operations:

1. *insert_operation (insert(x), t):* It inserts the *insert(x)* operation at time '*t*', i.e. it inserts an element '*x*' in the Hash at time '*t*'.
2. *delete_operation (t):* It deletes the *insert(x)* operation at time '*t*', i.e. it deletes an element '*x*', which had been inserted at time '*t*', from the Hash.

Query operation:

1. *search (x,t):* It find out whether an element '*x*' is present in the hash at time '*t*'.

As mentioned earlier, the fully retroactive data structure will not only allow queries in the present time but also in past time, and also updating of data structure at any point of time.

Data Structure Used:

- AVL Tree, which contains link to nodes that are deleted.
- Hash table in which each node contains extra time variable

Average Case Complexity:
After implementation, we observed the following average case complexities for different retroactive operations as:

Operations	Retroactive Hash
insert_operation(Insert(x), t)	*O(k)*
delete_operation(t)	*O(k)*
search(x,t)	*O(k)*

* '*k*' is the max number of collisions.

III. Data Structures Enhanced

A. *Fully Retroactive Union-Sameset*

The fully retroactive Union-Sameset will not only allow queries at the present time but also at past time, and also the updates of operations at any point of time.

Data Structure Used:

- Doubly linked list, storing sequence of operations sorted by time.
- Hash Table of pointers, pointing to different nodes of the forest.
- Tree(s), one or more than one (created using *Rank Heuristic Method* and *Path Compression Method* [6]).

Query operations:

1. *sameset (x,y):* determines whether '*x*' and '*y*' are in the same tree or not.

sameset (x,y) operation:
The Union-Find structure can be made fully retroactive by replacing the *find(x)* operation by a *sameset (x, y)* operation, since *find(x)* can give its result according to the data structure built at present. We can query about the data structure at past time using *sameset (x, y)*. In order to support retroactive operations, we add to each edge the time at which it was created. Hence, to determine whether two nodes are in the same set at time '*t*', we just have to verify that the maximum edge time on the path from '*x*' and '*y*' to their respective roots is no larger than '*t*'.

Link-Cut Tree:
E. D. Demaine, J. Iacono, and Langerman [1] proposed that Link-Cut Trees of D. D. Sleator and R. E. Tarjan [4] can be used as an efficient data structure, to provide full retroactivity**,** to maintain the forest, in *O(log n)* time per operation. This data structure maintains a forest and support the creation and deletion of nodes, edges, and the changing of the root of a tree.

Comparison of our data structure with Link-Cut Tree:

1. Accessing of elements by Link-Cut tree takes *O(log n)* time, whereas by the data structure used in the project takes *O(1)* time.
2. In addition to this, the Link-Cut tree is a standard binary tree and has height of '*log n'*, where '*n'* being the no. of elements in the tree, thus finding the root takes *O(log n)* time, whereas here, root can be found initially in *O(log n)* time, and later in *O(1)* time, because of the path compression during the first query.
3. Here, we can have limited no. of nodes since we have used hash table, but there is no such kind of restriction in Link-Cut Tree.

Complexity Analysis:
We compared the complexities using both the structures and here is what we analyzed:

Operations	Our Data Structure	Link-Cut Tree
Accessing element in tree	*O(log n)*	*O(log n)*
Past time query: *sameset (x)*	*O(log n)*	*O(log n)*
Overall Time Complexity	***O(log n)***	***O(log n)***
Overall Space Complexity	**$O(n^2)$**	**$O(n^2)$**

* 'n' is the total number of elements in the forest.

IV. Applications of Retroactivity

Here, we present multiple applications of retroactivity in various data structures and how they could be used in various real life scenarios.

A. *Retroactive Queue: Breadth First Traversal in a Graph*

Problem:
Suppose, we realize after traversing the graph in breadth first manner, that a node which did not exist was considered to be in graph, and we want to write the corrected traversal.

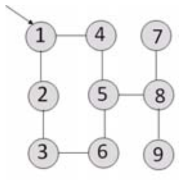
Normal Solutions:

1. Redo the whole traversal
2. Rollback to the point where node was traversed.

Both the solutions have linear complexity and are inefficient, especially in the cases where the number of nodes affected is significantly less than the total number of nodes in the graph. Therefore, we come up with the retroactive solution.

Retroactive solution:
The operations which would be affected by non-existence of that node, only those need to be, retroactively modified.

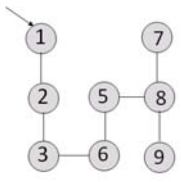
Example: Consider the following graph. Normal Breadth First Traversal carried is as follows:



QUEUE	OPERATIONS
1	enque(1)
2 4	dequeue (1)→enque(2), enque(4)
4 3	dequeue (2)→enque(3)
3 5	dequeue (4)→enque(5)
5 6	dequeue (3)→enque(6)
6 8	dequeue (5)→enque(8)
8	dequeue (6)
7 9	dequeue (8)→enque(7), enque(9)
9	dequeue (7)
	dequeue (9)

Figure 1. Incorrect Graph

Now, suppose that node 4 was not present in the graph, then:



QUEUE	OPERATIONS
1	enque(1)
2 4	dequeue (1)→enque (2), enque (4)
4 3	dequeue (2)→enque (3)
3 5	dequeue (4)→enque (5)
5 6	dequeue (3)→enque (6)
6 8	dequeue (5)→enque (8)
8	dequeue (6)
7 9	dequeue (8)→enque (7), enque (9)
9	dequeue (7)
	dequeue (9)

Figure 2. Original and Correct Graph

Only the underlined operations need to be deleted and inserted again at proper places, rather than redoing the whole traversal. By using the fully retroactive implementation of queue, these changes are made.

After changing, the result is:

QUEUE	OPERATIONS
1	enque (1)
2	dequeue (1)→enque (2)
3	dequeue (2)→enque (3)
6	dequeue (3)→enque (6)
5	dequeue (6)→enque (5)
8	dequeue (5)→enque (8)
7 9	dequeue (8)→enque (7), enque (9)
9	dequeue (7)
	dequeue (9)

Thus, the retroactive queue avoids the redoing of all the operations, and only alters the required operations.

Steps involved in implementing above application:

- Traverse the graph using retroactive queue instead of normal queue
- While traversing the graph for first time, store timings of *enque()* and *dequeue()* operations of every node
- Later, when some earlier node is to be removed, identify that node as target node
- Flag all the enque and deque operations depending on the deletion of target node (the storage created in 2nd step and concepts of retroactive queue will be used in the process)
- One by one, resolve the dependencies of all the flagged nodes
- Reposition the operations consistently
- Identify and eliminate the infinite loop of dependencies, which is formed in some cases
- The resulting structure gives the corrected breadth first traversal of graph

Complexity analysis:

Let, n = number of nodes in the graph, and
m = number of *enque*() operations, whose position will be changed in the retroactive queue.
The time complexity estimated is $O(m \log n)$

Explanation of m:
In real life applications, for example, a large scale railway network, number of nodes is extremely large, and removal of few nodes affects a small part of network. That is, in real life applications, where retroactive approach is applied, m is very small in comparison to n. Therefore, retroactive solution will be of time complexity $O(\log n)$ when $m << n$. This will be the case when number of nodes deleted from graph is small.

For the above case in consideration, overall comparison of complexities is as follows

Technique	Space Complexity	Time Complexity
Rollback	O(n)	O(n)
Persistent	O(n^2)	O(n)
Retroactive	O(n)	O(log n)

B. Retroactive Queue: Reducing Effect of Network Delay in Client Server Architecture

In a standard client-server model, the server can be seen as holding a data structure, and clients send update or query operations. When the order of the requests is important (e.g., Internet auctions), the users can send a timestamp along with their requests. The server should execute the operations in the order they were sent. If a request is delayed by the network, it should be retroactively executed at the appropriate time in the past.

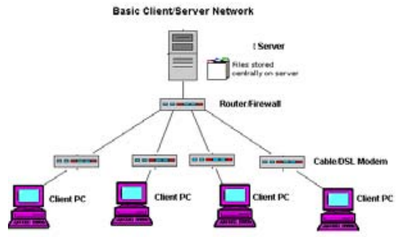


Figure 3. Basic Client-Server Architecture (Ref. [10])

To achieve this, partially retroactive queue is maintained by the server for handling requests from clients. So that the requests are in order they were sent, rather than in order, they were received.

C. *Retroactive Union-Sameset: Recovery of Weather-Forecasting Data*

In places where, data pertaining to weather is taken from different weather stations and is finally reported to a central computer. Various stations are placed in small groups, thus an average of the data is computed. Small groups are then combined to form larger groups, and now an average of the data is taken and so-on until we are left with only single group. Final data evaluated is, thus, analyzed and various statistics are drawn out of it.

1) *Problem 1: Suppose, we come to know that one of the weather station got malfunctioned at time 't' in the past.*

Normal Solution: Rollback to the point where station got malfunctioned.
The solution has linear complexity and is inefficient, especially in the cases where the number of stations affected is significantly less than the total number of stations. Therefore, we come up with the retroactive solution.

Retroactive Solution: The stations which would have been affected by malfunctioning of that station, only data of those needs to be retroactively modified. That particular group of affected stations can be identified by retroactive query, *sameset(x,y).*

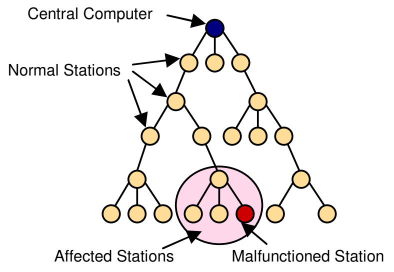


Figure 4. Group of Weather-Stations with one Malfunctioned

Thus, data can be recovered and it would indicate as if the malicious operation never occurred.

2) *Problem 2: Suppose, we come to know that one of the weather station was missed to be considered before.*

Normal Solution: Rollback to the point where station was missed to be considered.
The solution has linear complexity and is inefficient, especially in the cases where the number of stations affected is significantly less than the total number of stations. Therefore, we come up with the retroactive solution.

Retroactive solution: The stations whose data would have been affected by neglecting that station, only data of those need to be retroactively modified. That particular group of affected stations can be identified by retroactive operation, *insert (Union(x,y),t).*

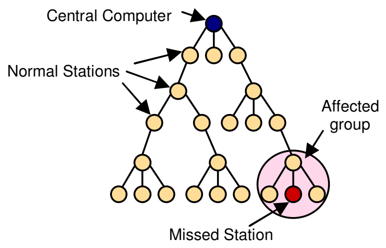


Figure 5. Group of Weather-Stations with one Missed

The data from the missed station can be used to evaluate final data again by applying retroactively adding the station into the group of stations. Thus, data can be analyzed again and it would indicate as if that station was already under consideration.

ACKNOWLEDGEMENT

We thank Daniel Sleator for his helpful pointers on Link-Cut Trees [4] and Splay Trees [3], Jeff Erickson for his lectures [8] on Link-Cut Trees. And, a special thanks to Harshit Gupta, Konark Gill and Abhinav Shrivastava for their helpful and healthy discussions and comments.

REFERENCES

[1] Demaine, E. D., Iacono, J., and Langerman. "Retroactive data structures". Journal of the Association for Computing Machinery, 281-290, 2004.
[2] Fleischer, R. "A simple balanced search tree with O(1) worst-case update time". Int. J. Found. Comput. Sci. 7, 2, 137–149, 1996.
[3] Sleator, D. D. and Tarjan, R. E. "Self-adjusting binary search trees Journal ACM 32 (1985), pp. 652-686.
[4] Sleator, D. D. and Tarjan, R. E. "A data structure for dynamic trees." J. Comput. Syst. Sci. 26, 3, 362–381, 1983.
[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. "Introduction to Algorithms". Copyright © 2001 by The Massachusetts Institute of Technology, Second edition . Tata McGraw-Hill, 2001.
[6] Sylvain Conchon, Jean-Christophe Filliˆatre, "A Persistent Union-Find Data Structure", Workshop on ML 2007.
[7] Horowitz E., Sahni S., and Anderson-Freed S. "Fundamentals of Data Structures" in C. W.H. Freeman and Company, 1998.
[8] http://granmapa.cs.uiuc.edu/~jeffe/teaching/data-structures/notes/07-linkcut.pdf
[9] http://www.boost.org/doc/libs/1_37_0/libs/graph/doc/incremental_components.html
[10] http://www.onsitepcdoc.com/network.html