Option #2: Retroactive Search Trees

Scott Miner

Colorado State University – Global Campus

Abstract

Tree data structures are some of the most popular in computer science. Retroactive data structures prove useful in various scenarios, including resolving and maintaining simple errors, security breaches, online protocols, and version control software. This study aims to investigate the performance differences between implementations of partially and fully retroactive binary search trees (BSTs) and their non-retroactive counterparts, along with simple rollback solutions. First, sets of unique, pseudo-random integers were generated to test the performances of each of the BST implementations using update and query operations. Then, the performance times of each of the implementations were compared using the nonparametric Kruskal-Wallis test (KWt) with a *p*-value set to .05. Finally, Dunn's test and the Bonferroni correction method were used to perform pairwise comparisons between implementations with a *p*-value set to .0125. The results indicate that the fully retroactive BST performs significantly slower than all other implementations, whereas the non-retroactive BST performs significantly faster than all other implementations when increasing the number of test elements from 10 to 100. Furthermore, the study found no threshold where the fully retroactive solution surpasses the performance of the simple rollback solution since the simple rollback solution always outperforms the complex solution. Therefore, retroactive data structures should be implemented on an as-needed basis, according to the minimum requirements of the scenario, since the performance times of all BSTs differed significantly from each other when performing 100 tests on structures containing 100 unique pseudo-random integers.

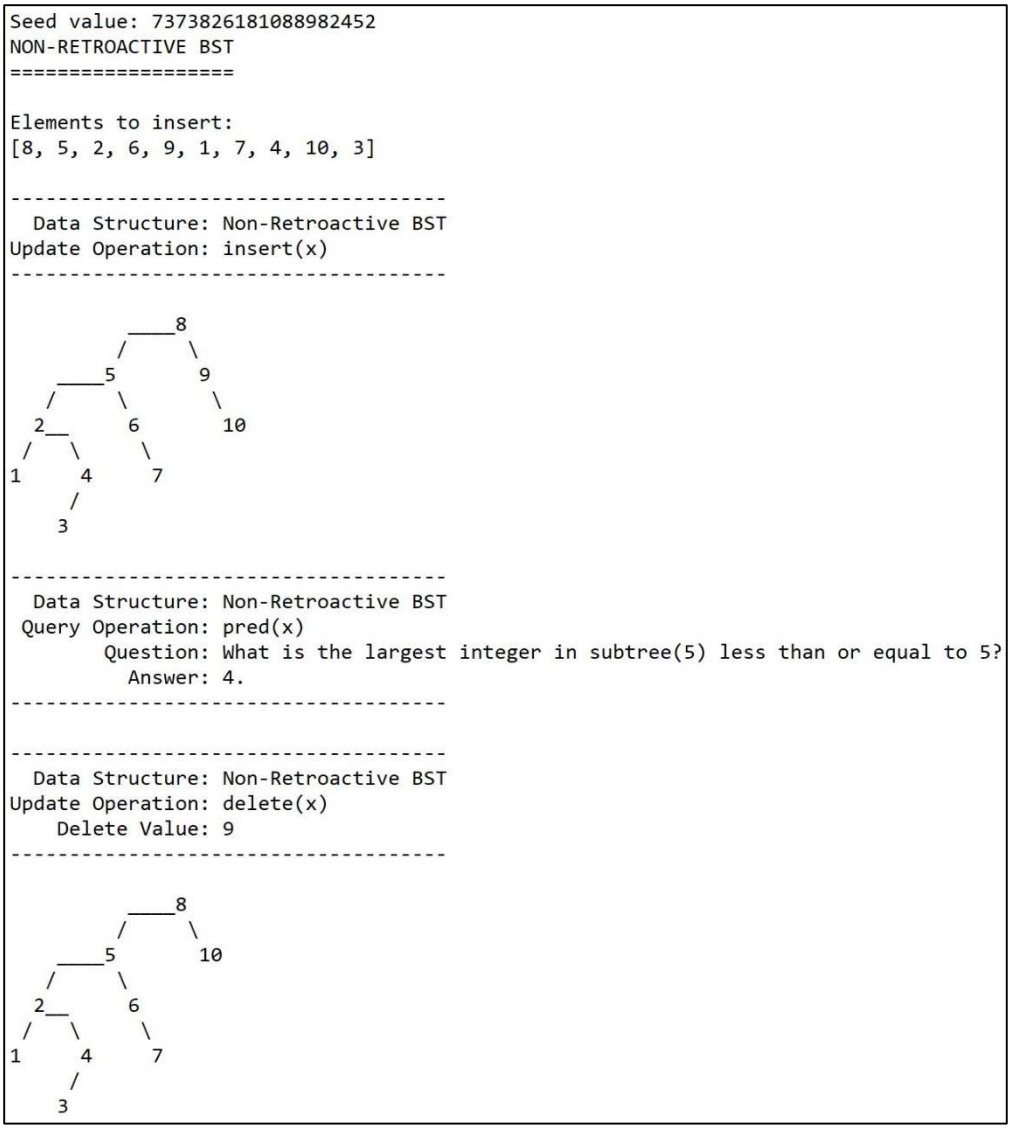


Figure 1. Update and query operations performed on a non-retroactive BST

```
PARTIALLY-RETROACTIVE BST
==================

Elements to insert:
[8, 5, 2, 6, 9, 1, 7, 4, 10, 3]

------------------------------------------------
  Data Structure: Partially-Retroactive BST
Update Operation: insertAgo(x, timeAgo=0) (now)
------------------------------------------------

          ____8
         /     \
    ____5       9
   /     \       \
  2__     6       10
 /   \     \
1     4     7
     /
    3

-----------------------------------------
  Data Structure: Partially-Retroactive BST
Update Operation: deleteAgo(timeAgo)
        Time Ago: 9
-----------------------------------------

    ____5
   /     \
  2__     6__
 /   \       \
1     4       9
     /       / \
    3       7   10

-----------------------------------------
  Data Structure: Partially-Retroactive BST
Update Operation: insertAgo(x, timeAgo)
    Insert Value: 18
            When? 9 operation(s) ago.
-----------------------------------------

          _________18
         /
    ____5
   /     \
  2__     6__
 /   \       \
1     4       9
     /       / \
    3       7   10

-----------------------------------------
  Data Structure: Partially-Retroactive BST
 Query Operation: pred(x)
               X: 18
        Question: What is the largest integer in the subtree(18) less than or equal to 18?
          Answer: 10.
-----------------------------------------
```

Figure 2. Update and query operations performed on a partially retroactive BST

```
FULLY-RETROACTIVE BST
====================

Elements to insert:
[8, 5, 2, 6, 9, 1, 7, 4, 10, 3]

------------------------------------------------
  Data Structure: Fully-Retroactive BST
Update Operation: insertAgo(x, timeAgo=0) (now)
------------------------------------------------

          ____8
         /     \
    ____5       9
   /     \       \
  2__     6       10
 /   \     \
1     4     7
     /
    3

-----------------------------------------
  Data Structure: Fully-Retroactive BST
 Query Operation: pred(x, timeAgo)
               X: 6
         timeAgo: 5 operation(s) prior.
        Question: What is the largest integer in the subtree(6) less than or equal to 6?
          Answer: 6.
-----------------------------------------
Tree 5 iterations ago:

    __8
   /   \
  5     9
 / \
2   6

-----------------------------------------
  Data Structure: Fully-Retroactive BST
Update Operation: deleteAgo(timeAgo)
        Time Ago: 9 operation(s) ago.
-----------------------------------------

    ____5
   /     \
  2__     6__
 /   \        \
1     4       9
     /       / \
    3       7   10

-----------------------------------------
  Data Structure: Fully-Retroactive BST
Update Operation: insertAgo(x, timeAgo)
    Insert Value: 18
            When? 9 operation(s) ago.
-----------------------------------------

          _________18
         /
    ____5
   /     \
  2__     6__
 /   \        \
1     4       9
     /       / \
    3       7   10

-----------------------------------------
  Data Structure: Fully-Retroactive BST
 Query Operation: pred(x, timeAgo)
               X: 6
         timeAgo: 9 operation(s) prior.
        Question: What is the largest integer in the subtree(6) less than or equal to 6?
          Answer: None.
-----------------------------------------
Tree 9 iterations ago:

18
```

Figure 3. Update and query operations performed on a fully retroactive BST

```
SIMPLE ROLLBACK SOLUTION
========================

Elements to insert:
[8, 5, 2, 6, 9, 1, 7, 4, 10, 3]

------------------------------------------------
  Data Structure: Simple Rollback Solution
Update Operation: insertAgo(x, timeAgo=0) (now)
------------------------------------------------

          ____8
         /     \
    ____5       9
   /     \       \
  2__     6       10
 /   \     \
1     4     7
     /
    3

-----------------------------------------
  Data Structure: Simple Rollback Solution
Update Operation: deleteAgo(timeAgo)
        Time Ago: 9 operation(s) ago.
-----------------------------------------

    ____5
   /     \
  2__     6__
 /   \       \
1     4       9
     /       / \
    3       7   10

-----------------------------------------
  Data Structure: Simple Rollback BST
Update Operation: insertAgo(x, timeAgo)
    Insert Value: 18
            When? 9 operation(s) ago.
-----------------------------------------

          _________18
         /
    ____5
   /     \
  2__     6__
 /   \       \
1     4       9
     /       / \
    3       7   10

-----------------------------------------
  Data Structure: Simple Rollback BST
 Query Operation: pred(x, timeAgo)
               X: 6
         timeAgo: 9 operation(s) prior.
        Question: What is the largest integer in the subtree(6) less than or equal to 6?
          Answer: None.
-----------------------------------------
Tree 9 iterations ago:

18
```

Figure 4. Update and query operations performed using a simple rollback solution

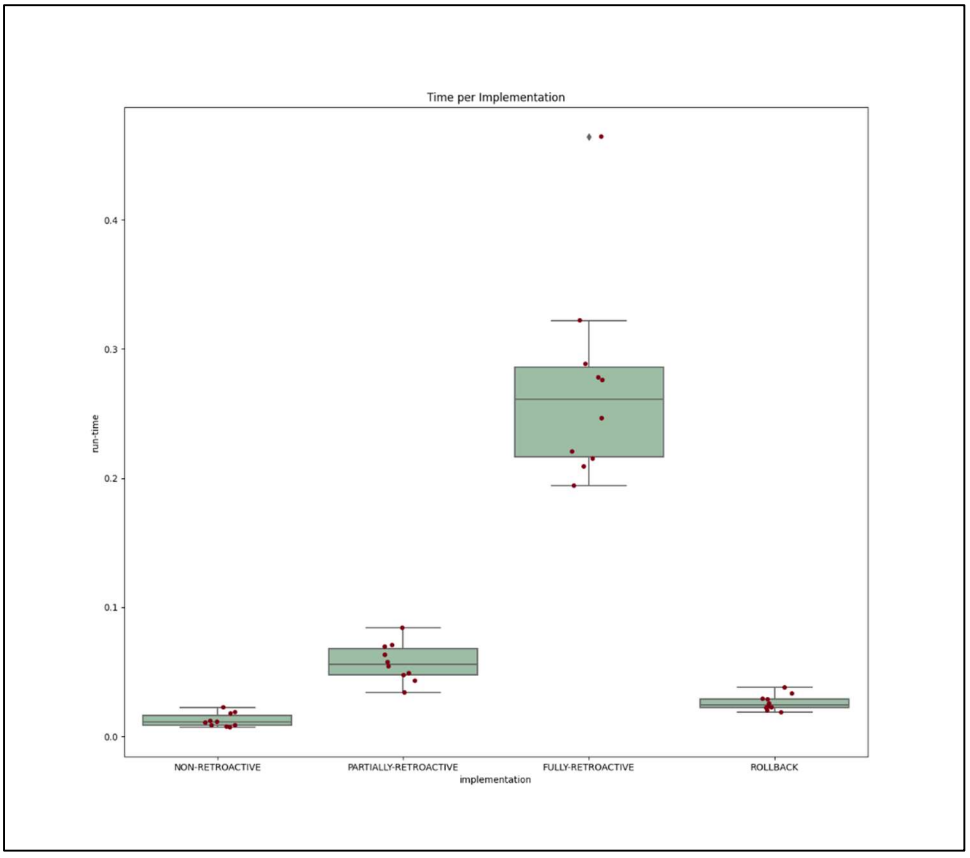


Figure 5. Boxplot showing runtimes by BST implementation after running ten tests per implementation on structures containing ten elements

	Time			
Classification	NON-RETROACTIVE	ROLLBACK	PARTIALLY-RETROACTIVE	FULLY-RETROACTIVE
Test #				
1	0.011298	0.023054	0.069489	0.209051
2	0.008605	0.037893	0.057515	0.464513
3	0.022498	0.022454	0.070725	0.275994
4	0.018775	0.029153	0.084079	0.288475
5	0.008603	0.018647	0.048938	0.215129
6	0.007070	0.020154	0.033999	0.246428
7	0.007542	0.028706	0.054267	0.220614
8	0.017745	0.033211	0.047455	0.194181
9	0.011974	0.022114	0.063182	0.278033
10	0.010690	0.025494	0.043069	0.322185

Kruskal Wallis:
H-statistic: 35.73804878048779
P-value: 8.506788751379051e-08

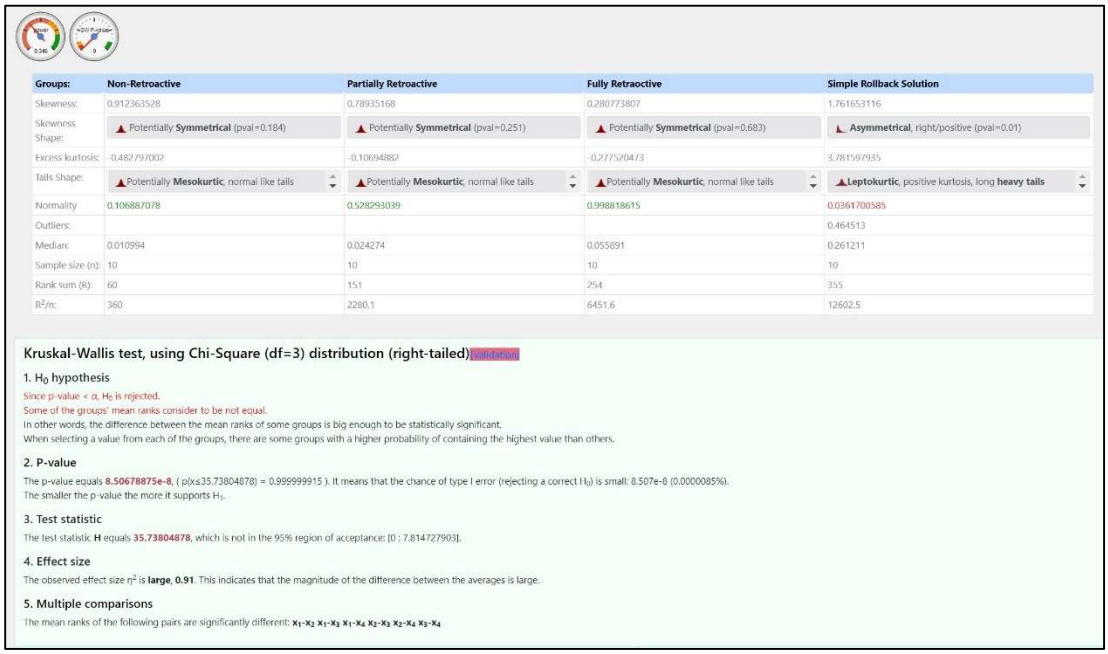
Dunn's Test

	1	2	3	4
1	1.000000e+00	0.001240	1.005276e-07	0.490539
2	1.240139e-03	1.000000	3.202626e-01	0.292953
3	1.005276e-07	0.320263	1.000000e+00	0.000572
4	4.905387e-01	0.292953	5.724791e-04	1.000000

Figure 6. Performance times and the results of the KWt and Dunn's test performed on sample data containing ten records

```
  File "C:\Users\Scott\OneDrive\Documents\Komodo\BinarySearchTrees\main.py", line 18, in insertVal
    bst.insert(Node(value))
  File "C:\Users\Scott\OneDrive\Documents\Komodo\BinarySearchTrees\Node.py", line 4, in __init__
    def __init__(self, key, left=None, right=None):
RecursionError: maximum recursion depth exceeded in comparison
```

Figure 7. Maximum recursion depth error received when running tests on lists with thousands of elements



Groups:	Non-Retroactive	Partially Retroactive	Fully Retroactive	Simple Rollback Solution
Skewness:	0.912363528	0.78935168	0.280773807	1.761653116
Skewness Shape:	Potentially **Symmetrical** (pval=0.184)	Potentially **Symmetrical** (pval=0.251)	Potentially **Symmetrical** (pval=0.683)	**Asymmetrical**, right/positive (pval=0.01)
Excess kurtosis:	-0.482797002	-0.10694882	-0.277520473	3.781597935
Tails Shape:	Potentially **Mesokurtic**, normal like tails	Potentially **Mesokurtic**, normal like tails	Potentially **Mesokurtic**, normal like tails	**Leptokurtic**, positive kurtosis, long **heavy tails**
Normality	0.106887078	0.528293039	0.998818615	0.0361700585
Outliers:				0.464513
Median:	0.010994	0.024274	0.055891	0.261211
Sample size (n):	10	10	10	10
Rank sum (R):	60	151	254	355
R^2/n:	360	2280.1	6451.6	12602.5

Kruskal-Wallis test, using Chi-Square (df=3) distribution (right-tailed) Validation

1. H_0 hypothesis

Since p-value < α, H_0 is rejected.
Some of the groups' mean ranks consider to be not equal.
In other words, the difference between the mean ranks of some groups is big enough to be statistically significant.
When selecting a value from each of the groups, there are some groups with a higher probability of containing the highest value than others.

2. P-value

The p-value equals **8.50678875e-8**, (p(x≤35.73804878) = 0.999999915). It means that the chance of type I error (rejecting a correct H_0) is small: 8.507e-8 (0.000008%).
The smaller the p-value the more it supports H_1.

3. Test statistic

The test statistic **H** equals **35.73804878**, which is not in the 95% region of acceptance: [0 : 7.814727903].

4. Effect size

The observed effect size η^2 is **large**, **0.91**. This indicates that the magnitude of the difference between the averages is large.

5. Multiple comparisons

The mean ranks of the following pairs are significantly different: **x_1-x_2 x_1-x_3 x_1-x_4 x_2-x_3 x_2-x_4 x_3-x_4**

Figure 8. Results from an online KWt calculator. The *p*-value and *H*-statistic match those provided by the Python code.

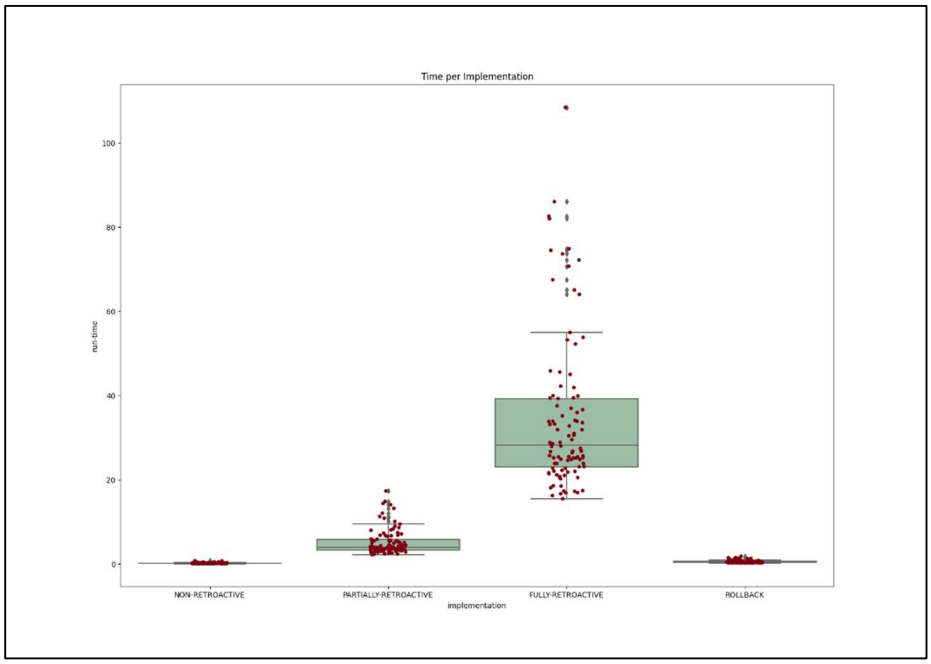


Figure 9. Boxplot showing runtimes by BST implementation after running 100 tests per implementation on structures containing 100 elements

	Time			
Classification	NON-RETROACTIVE	ROLLBACK	PARTIALLY-RETROACTIVE	FULLY-RETROACTIVE
Test #				
1	0.103481	0.398162	4.179760	22.733958
2	0.107315	0.347511	3.831664	20.551910
3	0.110468	0.505651	2.829023	21.830663
4	0.146628	0.361726	3.875442	22.253052
5	0.115901	0.425218	3.594254	25.504088
6	0.108112	0.289409	3.742564	33.888537
7	0.136349	0.393518	2.959354	21.058412
8	0.153329	0.355077	3.909374	39.984921
9	0.123077	1.356258	3.621432	32.784217
10	0.390497	0.486129	6.839710	42.233381
11	0.146450	0.527255	3.745661	25.250067
12	0.184316	0.450938	3.564196	25.304465
13	0.151296	0.313260	3.641733	24.892731
14	0.117013	0.441042	3.668318	23.093529
15	0.118310	0.379611	3.527807	22.776772
16	0.131359	0.409770	3.973046	22.052119
17	0.123160	0.333415	3.447089	21.637767
18	0.099008	0.371008	3.020129	26.846866
19	0.152848	0.364496	3.403727	20.778828
20	0.109912	0.414202	2.959605	20.529229
21	0.114426	0.326880	2.959132	21.166039
22	0.121818	0.809201	3.385784	25.466010
23	0.255428	0.349990	5.079049	31.906841
24	0.111362	0.485781	3.940809	23.882293
25	0.216781	0.394071	8.102763	18.590130
26	0.098957	0.280199	2.408087	17.254802
27	0.086003	0.277625	2.378359	16.903548
28	0.081875	0.515548	2.515074	24.834161
29	0.143254	0.778709	3.631358	39.890688
30	0.221379	0.735310	6.739486	30.990585
31	0.308762	0.448927	4.007685	23.880648
32	0.132008	0.434049	6.709766	25.170788
33	0.136285	0.377153	3.122664	21.430866
34	0.127262	0.315323	3.243563	33.228804
35	0.139615	0.392374	3.018759	22.002673
36	0.109524	0.267824	2.850865	20.245549
37	0.091321	0.395390	2.882835	24.817138
38	0.100041	0.369910	2.702601	28.565129
39	0.143942	0.262849	4.540518	23.867183
40	0.105217	0.264612	2.710801	17.316591
41	0.118804	0.294993	2.641319	17.426963
42	0.089174	0.572782	2.480541	18.508947
43	0.263677	0.255212	3.880299	26.409629
44	0.100098	0.270434	2.335418	16.669102
45	0.105302	0.241345	2.348880	16.253653
46	0.096208	0.304238	2.764486	18.114327
47	0.086454	0.225019	2.232034	15.506077
48	0.087677	0.411457	2.794089	25.446578
49	0.101854	0.459029	2.537036	16.953494
50	0.108387	0.337090	3.439247	23.111114
51	0.156110	0.544900	3.643223	25.167734
52	0.224511	1.248350	11.284156	86.046190
53	0.741612	1.082119	13.220570	74.488663
54	0.483338	0.853146	14.084218	74.885774
55	0.375041	0.860546	6.866017	70.720532
56	0.288956	1.248451	8.989229	65.069004
57	0.482865	0.756691	12.066663	73.663414
58	0.254553	0.667715	7.143935	45.583811
59	0.216796	0.721795	5.893560	36.650323
60	0.222908	0.845048	5.258036	34.123446
61	0.281617	0.623146	5.955410	33.858682
62	0.261750	0.550851	5.643147	33.892669
63	0.169569	0.518825	5.114397	35.190659
64	0.192306	0.503528	4.937439	35.974269
65	0.204526	0.486952	5.578598	28.475193
66	0.172469	0.491296	5.568201	30.583452
67	0.138999	1.490995	5.617993	39.476234
68	0.505312	0.509789	9.496652	27.463042
69	0.194204	0.593028	4.289383	33.573304
70	0.243241	1.331960	10.116203	64.029059
71	0.286819	0.596836	10.875900	54.981418
72	0.227587	0.632803	7.318051	52.264364
73	0.348449	0.516181	4.252371	45.871744
74	0.148222	0.406061	3.321900	28.360598
75	0.161546	0.539823	3.450083	28.046842
76	0.169238	0.762410	3.274083	39.297847
77	0.315936	0.660607	7.367217	53.217336
78	0.229688	1.519163	7.435423	82.584254
79	0.470760	0.939227	14.381590	67.503773
80	0.327844	0.743219	8.335183	53.816369
81	0.343512	0.702858	8.015984	41.914367
82	0.181246	0.525097	5.551205	39.418832
83	0.172325	0.449603	6.559800	36.981154
84	0.155737	0.467393	4.135851	28.878311
85	0.180320	0.452705	4.484021	27.932568
86	0.215586	0.387439	4.361834	26.694895
87	0.115303	0.425388	3.587776	25.747840
88	0.139838	0.506743	3.885579	29.534161
89	0.194029	0.469878	3.939263	26.810917
90	0.167958	0.444447	3.659461	25.703355
91	0.133437	0.417480	3.485067	24.637067
92	0.133424	0.587334	4.453469	33.138755
93	0.160685	0.352921	5.737712	30.437029
94	0.153098	0.549077	3.887711	28.824524
95	0.177632	0.487059	5.080324	31.892227
96	0.154981	0.568770	4.324947	37.577288
97	0.199744	0.825975	5.356853	45.046475
98	0.312935	1.856136	8.647097	82.003976
99	0.775482	1.049103	17.353670	108.441355
100	0.278979	1.335422	14.889067	72.184829

Figure 10. Performance times per implementation after running 100 tests on structures containing 100 elements

```
Kruskal Wallis:
H-statistic: 366.0370623441397
P-value: 5.0228021400390637e-79

Dunn's Test
              1             2             3             4
1  1.000000e+00  7.690852e-32  1.150935e-71  3.383908e-07
2  7.690852e-32  1.000000e+00  6.071411e-09  6.139421e-10
3  1.150935e-71  6.071411e-09  1.000000e+00  1.829081e-35
4  3.383908e-07  6.139421e-10  1.829081e-35  1.000000e+00
```

Figure 11. The results of the KWt and Dunn's test performed on sample data containing 100 records

Contents

Option #2: Retroactive Search Trees

For his portfolio project in CSC506: Design and Analysis of Algorithms, the student implements partially and fully retroactive binary search trees (BSTs) in the Python programming language. The update operations to the non-retroactive BSTs are *insert(x)* and *delete(x)*. The query operation, *pred(x)*, returns the largest element in the subtree ≤ x, also known as its predecessor. This paper defines appropriate interfaces to partially and fully retroactive BSTs, implements the interfaces, verifies their correctness by testing them against simple rollback solutions, and compares the performances of these different implementations.

Introduction

Tree data structures are among the most widely used organization structures in computer science (Kundu & Bertino, 2008). For example, Greenblatt *et al.* (1967) write that chess programs use tree data structures, also known as game trees, to maintain all positions a program searches through to find a player's next move. On May 11, 1997, IBM's supercomputer, Deep Blue, achieved one of the oldest challenges in computer science, to create the first World Champion-class chess computer, by beating the World Chess Champion Garry Kasparov in a six-game chess match (Feng-Hsiung Hsu, 1999). Deep Blue used tree data structures and the minimax search algorithm to achieve this feat, which today's leading chess engines still use (Cardy, 2017).

Computers also use tree data structures to manage eXtended Markup Language (XML) data. Since its inception in 1998, XML has become a ubiquitous language for data interoperability purposes in various domains and the main data exchange format of the web (Hachicha & Darmont, 2013; Maneth *et al.*, 2008). For instance, Kundu and Bertino (2008) write that hospitals store their XML-based patient healthcare records in tree schemas. The tree's

root, the *HealthRecord,* contains edges creating paths to child nodes that contain each patient's contact information and medical history. Thus, computers can use tree data representations of XML documents to query XML data efficiently (Hachicha & Darmont, 2013).

Binary Trees (BTs) allow for the rapid retrieval of data and are among the most popular data structures for retrieving information by its "name" (Davis, 1987; Knuth, 1971). Lysecky and Vahid (2019) describe BTs as data structures with nodes containing up to two children. Contrast this with list data structures, where nodes have up to only one successor. A binary search tree (BST) is an especially useful form of a BT with an ordering property stating that any node's left subtree key is ≤ the node's key and any node's right subtree key is ≥ the node's key.

Today's computers process millions of transactions daily. Demaine *et al.* (2007) describe several scenarios where retroactive data structures prove useful. For example, consider the scenario where several weather stations send temperature data to centralized computers, and a station's sensors have begun to malfunction. Retroactive data structures help remove the malfunctioning sensor's data and change its secondary effects, including any incorrect ranges, averages, and weather predictions. If, on the other hand, a particular station's transmission system has become damaged and its data is later recovered, we can observe the effects of retroactively entering this data on past and present weather predictions.

Further, Demaine *et al.* (2007) write that retroactive data structures are useful for maintaining online auctions. For example, in an online auction, if requests from client machines are delayed by network traffic, the requests can be retroactively executed in their sent order if they contain timestamps. Another example where retroactive data structures prove useful is in solving more complex problems. Garg (2018), for instance, used retroactive priority queues to solve the dynamic shortest path problem, a generic problem with applications in computer

science, management systems, operations research, and artificial intelligence. Garg used retroactive priority queues to dynamize the static Dijkstra algorithm, which uses non-retroactive priority queues.

Research Hypothesis

This paper aims to answer the following research questions:

- Which BST performs the fastest when (a) inserting ten unique pseudo-random integers between 1 and 10, inclusive, into the BST, (b) deleting an element from the BST, (c) inserting a new element into the BST, and (d) querying the BST for a given predecessor over ten iterations? How do these results change after increasing the number of elements and iterations to 100?
- At what threshold does the complex solution, the fully retroactive BST, begin to surpass the simple rollback solution?

The null hypothesis for the first question states that the distributions of performance times amongst the BSTs are the same. On the other hand, the alternate hypothesis states that the distributions of performance times are different amongst the BSTs. In other words, one or more implementations perform better than the others. The null hypothesis for the second question says there is no threshold where the complex solution surpasses the simple rollback solution, whereas the alternate hypothesis states there is a threshold where the complex solution surpasses the simple rollback solution.

Objectives

This study aims to identify the differences amongst the distributions of performance times for non-retroactive BSTs, partially retroactive BSTs, fully retroactive BSTs, and simple rollback solutions using update and query operations in the Python programming language.

Additionally, the study aims to determine whether there is a threshold at which the fully retroactive BST outperforms the simple rollback solution.

Overview of Study

The remainder of this paper is divided into the following sections: (a) literature review, presenting an overview of past research into retroactive data structures; (b) research design, outlining the study's methodology, methods, limitations, and ethical concerns; (c) findings, answering the study's primary research questions; (d) conclusion, summarizing the research process; and (e) recommendations, providing suggestions for future research.

Literature Review

We can better understand retroactive data structures by exploring them alongside ephemeral and persistent data structures. Driscoll *et al.* (1989) defined ephemeral data structures as the "ordinary" kind, where any change to a structure destroys its original version, leaving only the new one. Persistent data structures, on the other hand, allow access to multiple versions. A partially persistent data structure allows modifications to the current version and access to any past version, whereas a fully persistent data structure allows access and modifications to all versions. Numerous researchers have created partially and fully persistent forms of classic data structures, including stacks, queues, and search trees (Hood & Melville, 1980; Myers, 1983; Sarnak & Tarjan, 1986).

Demaine *et al.* (2007) introduced retroactive data structures, which are both similar to and different from persistent data structures. The structures are similar because they both deal with the concept of time. However, persistent data structures treat historical versions as unchangeable archives, whereas retroactive data structures allow changes directly to previous versions. Thus, changes to a retroactive data structure's operational history affect all operations

between the time of modification and the present. In addition, *partially retroactive* data structures allow for only updating the past and querying the present, whereas *fully retroactive* data structures allow for updating the past and querying both the past and present. Finally, Demaine *et al.* presented efficient retroactive data structures for queues, deques, and priority queues. This author is not aware of any studies in the literature that compare performances of retroactive BSTs, which is the primary objective of this study.

Research Design

This study uses an experimental design to investigate the impact that the independent variable, the solution-type (e.g., non-retroactive BST, partially retroactive BST, fully retroactive BST, or rollback solution), has on the dependent variable, the length of time each solution takes to complete the update and query operations.

Methodology

O'Leary (2017) describes two research methodologies: quantitative and qualitative. Quantitative research is hypothesis-driven, relying on numerical data and assumptions tied to positivism. Positivism holds that true knowledge is best pursued by the scientific method. On the other hand, qualitative research is inductive rather than deductive and relies on assumptions tied to subjectivism, which emphasizes the subjective aspects of personal experience. As such, this study employs a quantitative research approach.

Methods

To conduct this study, the researcher needed to define interfaces for partially and fully retroactive BSTs, implement the interfaces along with their non-retroactive counterparts, and compare their performance with simple rollback solutions using the Python programming language. The interfaces for the insert, delete, and query operations of the partially retroactive

BST are *insertAgo(x, t = 0), deleteAgo(t),* and *pred(x)*, where *x* corresponds to the element being inserted or searched for and *t* to a time in the structure's operational history.

In the context of the insert operation, the *t* argument represents the index *before* which the operation is inserted. For example, the index "0" corresponds to the present and "1" to the most recent operation performed on the structure. Therefore, to insert an element at the root of the BST, we insert it $N - 1$ operations prior, where N represents the number of items in the structure. On the other hand, the delete operation deletes an element at a specified time, *t*, in the data structure's operational history. The interface for the fully retroactive BST is identical to that of the partially retroactive version, except the query operation, *pred(x, t),* accepts the additional *t* parameter since fully retroactive data structures allow for querying the past. In addition, the simple rollback solution contains an additional method, *constructTree(s)*, where the optional *s* parameter allows the rollback solution to mimic the fully retroactive BST's capability to query the past.

To implement the partially and fully retroactive BSTs, the researcher found an incomplete project on GitHub aiming to "turn known algorithms for various types of retroactive data structures into *implementations*" (Voss, 2014/2021). Voss (n.d.) writes that the general method to transform any non-retroactive data structure into a partially retroactive version requires O(*r*) time, where *r* represents the number of previous operations performed on the data structure. Thus, the general method to transform any structure into a partially retroactive solution is to store the structure's current and initial states, along with the structure's operational history, so that these operations can be retroactively performed on the structure's initial state each time an element is inserted into or removed from the structure.

Moreover, Voss (n.d.) writes that the general method to turn any partially retroactive structure into a fully retroactive solution requires O(m) time, where m represents the total number of retroactive updates performed thus far. Thus, the general method to transform any structure into a fully retroactive implementation requires storing a sequence of partially retroactive data structures. By querying this sequence of versions, we can query the structure's past. There is also a more efficient implementation that requires $O(\sqrt{m})$ time and uses persistent data structures.

To implement partially and fully retroactive BSTs, the researcher needed to modify Voss's (2014/2021) code since it only contained retroactive implementations of queues, stacks, deques, and priority queues. Also, the researcher needed to implement the BST query operation, *pred(x)*, which returns the largest element in the subtree ≤ x. Thus, to find a subtree's predecessor, the algorithm begins by searching for the node in the BST. If the algorithm does not find the node, it returns "None." Otherwise, if the algorithm finds a node with no children, it returns the node as the predecessor. If, on the other hand, the algorithm finds a node with a left child, it returns the left child if the left child does not have a right child. However, if the left child has a right child, the algorithm traverses each right child node until it reaches a leaf, which it returns as the largest element in the subtree ≤ x.

As mentioned, the simple rollback solution contains an additional method, *constructTree(s),* which takes an optional argument, *s*, that mimics the past querying behavior of the fully retroactive BST. Therefore, the simple rollback solution works by deleting all nodes from the current BST and reconstructing the tree from scratch each time the *contstructTree()* method is called, inserting each element from a list representative of the structure's operational

history. Figures 1 – 4 display the update and query operations performed on the different BST implementations.

For instance, Figure 3 shows the results of inserting ten unique pseudo-random integers into the fully retroactive BST, deleting an integer, *i,* inserted *x* operations prior, and inserting a new value, *j*, at this point in time. All subsequent query operations performed on the data structure show this newly inserted value. Because the non-retroactive implementation has no concept of time, the integer, *x,* rather than a value corresponding to a specific time, *t*, is deleted from the data structure. The program performed these operations on all four data structures over ten iterations. In each iteration, a new set of ten unique pseudo-random integers were generated. The program tracked the time that each data structure took to perform these operations. Figure 6 shows the results over ten iterations, and Figure 5 shows a boxplot of this data, which is the runtime of each test per implementation.

After these results were generated, the program performed the Kruskal-Wallis *H* test (KWt), a nonparametric statistical test to compare independent samples from several populations, to determine whether there were any statistical differences between the performance times of the implementations using a significance level (α) of 0.05 (Vargha & Delaney, 1998). Next, Dunn's test, the appropriate nonparametric pairwise multiple comparison procedure to run when rejecting the results of a KWt, was run to determine which data structures had significantly different performance times. The program uses the Bonferroni correction method to reduce the likelihood of rejecting a correct null hypothesis by dividing α by the number of tests performed, corresponding to 0.05/4 in this scenario and an alpha level of 0.0125 (Dinno, 2015). Finally, the program performed the same operations and tests 100 times on lists containing 100 elements, increasing the number of tests and elements tenfold. Figures 9 – 11 show the performance times

for each iteration per data structure containing 100 elements, along with the KWt and Dunn's test results, in both tabular and boxplot formats.

Limitations

This study used general algorithmic approaches to transform non-retroactive BSTs into their partially and fully retroactive counterparts. These general approaches use something of a rollback solution in and of themselves since they store a structure's initial state and its current state, along with the structure's operational history. Every time a new element is inserted into or removed from the structure, its operational history is updated and reperformed on its initial state. Because this implementation resembles a simple rollback solution yet efficiently stores only two states of the data structure, it became difficult to conceive a "simpler" rollback solution. Ultimately, the simple rollback solution included the additional *constructTree()* method, which took as an argument a list of elements representative of the data structure's operational history.

Also, all tests in this study were performed on the researcher's personal computer, which has 16.0 GB of RAM and runs Windows 10 64-bit OS on an Intel Core i7-865OU processor with speeds of 1.90 GHz and 2.11 GHz. Thus, the program was limited in the number of elements that could be inserted into the BSTs. For instance, Figure 7 shows an error message received when running tests on lists containing thousands of elements. To overcome this challenge, the researcher reduced the number of elements inserted into the data structures from thousands to hundreds. Consequently, this study provides significant insight into how non-retroactive and retroactive BSTs, along with simple rollback solutions, compare when performing nearly identical operations on sets of pseudo-random integers, an area of research up until this time unexplored in the literature.

Ethical Considerations

Because this study did not use human participants, ethical issues of informed consent, anonymity, confidentiality, and privacy do not apply. Additionally, to confirm the validity of the program's output, the researcher used online calculators. For instance, Figure 8 shows the results of inputting the performance times output by the program into an online KWt calculator (*Kruskal-Wallis Test Calculator - One-Way ANOVA on Ranks*, n.d.). The *p*-value and *H*-statistic returned by the online calculator match those output by the program.

Findings

The boxplots in Figures 5 and 9 show that the fully retroactive implementation ($\bar{x} = 34.6$; $n = 100$) performed the slowest over 10 and 100 tests, respectively. Conversely, the non-retroactive structure ($\bar{x} = 0.2$; $n = 100$) performed the fastest each time. The simple rollback solution ($\bar{x} = 0.57$; $n = 100$) performed second fastest and slightly slower than the non-retroactive solution, and the partially retroactive solution ($\bar{x} = 5.21$; $n = 100$) performed the third fastest. Were these results statistically significant? Figure 6 shows the output of the KWt test performed on the sample data containing ten records ($H = 35.73$; $p < 0.05$), indicating statistically significant differences between the mean ranks of some of the implementation performance times. Therefore, we reject the null hypothesis for the first research question and conclude that the distributions of performance times amongst the BSTs are different.

Between which implementations do these significant differences lie? The numbered column headings output from Dunn's test correspond to the ordering of implementations shown in the boxplots. The performance times of the non-retroactive implementation differed significantly from both the partially ($p = 0.002$) and fully retroactive implementations ($p < 0.001$), but not from the simple rollback solution ($p = 0.49$), most likely because the simple rollback solution resembled the non-retroactive implementation with an additional

constructTree() method. The partially retroactive structure differed significantly from the non-retroactive structure ($p = 0.002$) but not from the fully retroactive structure ($p = .32$) or from the simple rollback solution ($p = .29$). Moreover, the fully retroactive solution differed significantly from the simple rollback solution ($p < 0.001$) and all other solutions. On the other hand, the results of Dunn's test performed on the sample containing 100 records show significant differences between the performance times of all implementations.

Therefore, to answer the second research question, we accept the null hypothesis and conclude there is no threshold where the performance of the complex solution surpasses that of the simple rollback solution in this scenario since the complex solution always performed significantly worse than the simple rollback solution. This is most likely because the rollback solution did not incorporate the full functionality of the retroactive solution and was merely used to verify the correctness of the retroactive implementations. As a result, the fully retroactive implementation should only be used when its capabilities are specifically needed, and retroactive querying is of the utmost importance. Furthermore, as a general rule of thumb, only the structure that meets the minimum requirements for a given situation should be used since the performance times of all structures differed significantly from each other when comparing the results of 100 tests using 100 elements.

Conclusion

In conclusion, this paper presented implementations of partially and fully retroactive BSTs, along with their non-retroactive counterparts. The update operations performed on the non-retroactive BSTs were *insert(x)* and *delete(x)*, whereas the query operation was *pred(x)*, which returned the largest element in the subtree $\leq x$, also known as its predecessor. This paper defined appropriate interfaces to partially and fully retroactive BSTs and detailed the

implementations of these interfaces. Additionally, the correctness of the implementations was verified by comparing their results to simple rollback solutions.

Finally, this paper compared the performance times of the data structures when performing update and query operations on sets of pseudo-random integers containing 10 and 100 elements. The fully retroactive data structure took significantly more time than all the other data structures when performing these operations over 10 and 100 iterations. The non-retroactive data structure performed these operations the fastest. Because the complex solution always performed significantly slower than the simple rollback solution, this study did not establish a threshold for when the complex solution outperformed the simple rollback solution.

Recommendations

Voss (n.d.) writes that the general transformation from partial to full retroactivity requires O(*m)* overhead, where *m* corresponds to the total number of retroactive updates performed on the data structure thus far. However, there is a more efficient implementation that requires O($\sqrt{m}$) overhead and uses persistent data structures. Recommendations for future research include comparing the performance of the fully retroactive BST that takes O(*m)* time with that which takes O($\sqrt{m}$) time.

Also, Demaine *et al.* (2007) write that achieving efficient retroactivity for data structures with a strong dependency between operations requires developing new technologies. Therefore, it may also be worthwhile to determine how and if other data structures, such as hash tables or linked lists, could be used to develop efficient retroactive BST solutions. Finally, researchers may wish to investigate how retroactive BSTs compare to other retroactive data structures and how the algorithms perform under different conditions, using more items and different sequences of operations.

References

Cardy, D. (2017, August 7). *Game Trees & Minimax | Dan Cardy*. https://www.cardy.net/posts/game-trees/

Davis, I. J. (1987). A locally correctable AVL tree. *Digest of Papers: The Seventeenth International Symposium on Fault-Tolerant Computing*, 85–88.

Demaine, E. D., Iacono, J., & Langerman, S. (2007). Retroactive data structures. *ACM Transactions on Algorithms*, *3*(2), 13. https://doi.org/10.1145/1240233.1240236

Dinno, A. (2015). Nonparametric Pairwise Multiple Comparisons in Independent Groups using Dunn's Test. *The Stata Journal*, *15*(1), 292–300. https://doi.org/10.1177/1536867X1501500117

Driscoll, J. R., Sarnak, N., Sleator, D. D., & Tarjan, R. E. (1989). Making data structures persistent. *Journal of Computer and System Sciences*, *38*(1), 86–124.

Feng-Hsiung Hsu. (1999). IBM's Deep Blue Chess grandmaster chips. *IEEE Micro*, *19*(2), 70–81. https://doi.org/10.1109/40.755469

Garg, D. (2018). Dynamizing Dijkstra: A solution to dynamic shortest path problem through retroactive priority queue. *Journal of King Saud University-Computer and Information Sciences*.

Greenblatt, R. D., Eastlake III, D. E., & Crocker, S. D. (1967). The Greenblatt chess program. *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, 801–810.

Hachicha, M., & Darmont, J. (2013). A Survey of XML Tree Patterns. *IEEE Transactions on Knowledge and Data Engineering*, *25*(1), 29–46. https://doi.org/10.1109/TKDE.2011.209

Hood, R. T., & Melville, R. C. (1980). *Real-time queue operations in pure LISP*. Cornell University.

Knuth, D. E. (1971). Optimum binary search trees. *Acta Informatica*, *1*(1), 14–25.

Kruskal-Wallis test calculator—One-way ANOVA on ranks. (n.d.). Retrieved June 5, 2021, from https://www.statskingdom.com/kruskal-wallis-calculator.html

Kundu, A., & Bertino, E. (2008). Structural signatures for tree data structures. *Proceedings of the VLDB Endowment*, *1*(1), 138–150.

Lysecky, R., & Vahid, F. (2019). *Data Structures Essential: Pseudocode with Python Examples*. Zyante Inc. (zyBooks.com).

Maneth, S., Mihaylov, N., & Sakr, S. (2008). XML tree structure compression. *In: Xantec*, 243–247.

Myers, E. W. (1983). An applicative random-access stack. *Information Processing Letters*, *17*(5), 241–248.

O'Leary, Z. (2017). The essential guide to doing your research project (3rd edition). SAGE Publications.

Sarnak, N., & Tarjan, R. E. (1986). Planar point location using persistent search trees. *Communications of the ACM*, *29*(7), 669–679.

Vargha, A., & Delaney, H. D. (1998). The Kruskal-Wallis test and stochastic homogeneity. *Journal of Educational and Behavioral Statistics*, 170–192.

Voss, C. S. (n.d.). *Specific Implementations—Python Retroactive Data Structures*. Retrieved May 30, 2021, from https://python-retroactive-data-structures.readthedocs.io/en/latest/specifics/

Voss, C. S. (2021). *Csvoss/retroactive* [Python]. https://github.com/csvoss/retroactive (Original work published 2014)