MODIFIED APPROACH TO DELETION IN BINARY SEARCH TREES : LAZY DELETION

SUBMITTED BY:

PRANOY DEV---16BEC0098

pranoy.dev2016@vitstudent.ac.in

UNDER THE GUIDANCE OF:

PROF. GAYATHRI P.

Vellore Institute of Technology

CONTENTS

1) ABSTRACT

This paper describes algorithms for lazy deletion on in Binary trees. There are published algorithms and pseudo code for searching and inserting keys, but deletion, due to its greater complexity and perceived lesser importance, is glossed over completely or least an exercise to the reader. To remedy this situation on, we provide a well-documented algorithm and pseudo-code for deletion, their relation on to search and insertion on algorithms, and a reference to a freely available, complete Binary search tree library written in the C++ programming language.

We will be able to compare the existing algorithms with our chosen algorithm and discuss about the advantages and disadvantages of the particular algorithm.

2) INTRODUCTION

There has been some research on the acceptability of relaxing the constraint of minimum node size to reduce the number of so-called unsafe tree operations, i.e., those which contain node splitting and merging . [2]The debate has culminated in analysis of a weaker form of the deletion algorithm which we call lazy deletion, that imposes no constraints on the number of entries left in the nodes, allowing them to empty completely before simply removing them.[1] According to , most database system implementations of binary search trees have adopted this approach. [3]

Its most effective use is when it is vital to allow concurrent access to the tree , [4] and excessive splitting and merging of nodes would restrict concurrency, derives some analytic solutions calculating memory utilization for Binary search trees under a mix of insertions and lazy deletions, based on previous research which considered insertions only .[5][6][7] The simulations in support its analysis to show

that in typical situations, where deletions don't outnumber insertions in the mix of operations, the tree nodes will contain acceptable percentages of entries.

One of the work's assumptions is that the keys and tree operations are chosen uniformly from a random distribution. [9][7]This assumption is unreasonable in certain realistic situations such as one described below. Allowing interior nodes with only a single pointer to exist in a B+ -tree creates the possibility for arbitrarily unbalanced trees of any height, which are virtually empty, and in which access times have degenerated from the logarithmic bound Binary search trees are meant to guarantee to a worst case unbounded access time.[10]

Methods that can be used for the Lazy deletion tree are –

Linked list , Binary search tree ,Array and pointers.

KEYWORDS: Binary search trees , lazy deletion , time complexity

3) <u>LITERATURE REVIEW</u>

There have been some researches on the minimum node size mechanisms like merging and splitting. So we came up with lazy deletion tree mechanism where the entries in the node are removed completely before the node is removed.

Most of the databases are using this concept for repetitive access. The methods like merging and splitting of nodes are not concurrent access friendly. [5][6]Now there are researches going on both mix of insertion and deletion by lazy tree concept where they can be able to solve many different situations. This assumption is not good for some kind of situations like where interior nodes with only single pointer may have unbalanced tree of any height, which is virtually empty. [6][7][8]The access time is degenerated as it is guaranteed for worst time case accessibility. Nodes are not removed until they are completely removed. So this method does not work all the possibilities.

The dynamic data structure management technique called hashing with lazy deletion is studied.[4][5][9] A table managed is built by a sequence of insertions and deletions of items.

When hashing with lazy deletions, one does not delete items as soon as possible but keeps more items in the data structure than would be the case with immediate-deletion strategies.

This deferral allows the use of a simpler deletion algorithm, leading to a lower overhead—in space and time. It is of interest to know how much extra space is used. [10][4]This paper investigates the maximum size and the excess space used , under general probabilistic assumptions, by using the methodology of queueing theory.

In particular, for the Poisson arrivals and general lifetime distribution of items, the excess space does not exceed the number of buckets . As a byproduct of the analysis, the limiting distribution of the maximum queue length in an infinity queueing system is also derived. [11][12]The results generalize previous work in this area.

4) PROPOSED METHOD

A lazy-deletion binary search tree is a binary search tree where erased objects are simply tagged as erased while the nodes themselves remain in the tree. Occasionally, a member function can be called to clean up all erased nodes at once. Almost all functions will be implemented by calling the corresponding function on the root nodes. There will be the following major fuction used:

1) bool empty()--Return true if the tree is empty (the size is 0).

2) int size() --Returns the number of nodes in the tree excluding erased nodes

3) int height()--Returns the height of the tree including nodes tagged as erased.

4) bool member(Type const &obj)-- Return true if the argument is in the tree and not tagged as erased and false otherwise.

5) Type front()--Return the minimum non-erased object of this tree by calling front on the root node.

6) Type back()--Return the maximum non-erased object of this tree by calling back on the root node.

7) bool insert(Type const &)--Insert the new object into the tree: If the object is already in the tree and not tagged as erased, return false and do nothing; if the object is already in the tree but tagged as erased, untag it and return true.

8) bool erase()--Removes the object from the tree: If the object is not already in the tree, return false; if the object is in the tree but tagged as erased, return false.

9) void clear()--Delete all the nodes in the tree.

10) void clean()--Delete all nodes tagged as deleted within the tree.

5) PSUEDOCODE AND ALGORITHM:

1) Define lazy deletion node
2) Include algorithm and utility header file and define null pointer
3) Define lazy deletion tree class and include element , left tree pointer , right tree pointer as private members
4) Also include retrieve , clear , clean ,erase , insert , height function and introduce lazy deletion tree friend function.
5) **Taking input** as the node number or element.
6) If initially left and right tree ptr == 0 or nullptr return erase function as false
7) If left or right ptr exists , check for their value and return that using retrieve function
8) **For height at each level** , check for max among height of left and right ptr and return 1 + max(left-ptr and left-ptr) else if ptr=0 return false
9) At each step if retrieve()==element and !erased return true
10) Now if obj < retrieve() , return left()->member(element)
11) Else return right()->member(element)
12) Then ptr->data==0 and tagged erased
13) **Now for insertion** , Take element as input
14) If ptr==0 ; ptr->data=element;return true
15) Else if retrieve()==element ; if erased return true else false
16) Now if element<retrieve() and if left()==0 , left tree ptr==new deletion node else left->insert(element)
17) Else ; if right()==0 ; , right tree ptr==new deletion node else right->insert(element)
18) **For erase function** : take input as element to be erased
19) If ptr==0 ; return false
20) Else if retieve()==element and if !erased ; retrn erased==true else return false
21) Else if element < retrieve() ; return left()->erase(element)
22) Else return right()->erase(element)
23) **Now for the clear function** ; if ptr==0 return false
24) else if left() != nullptr then left()->clear and same for right.

25) delete ptr;
26) **now calling clean function**
27) if ptr==0 ; return false

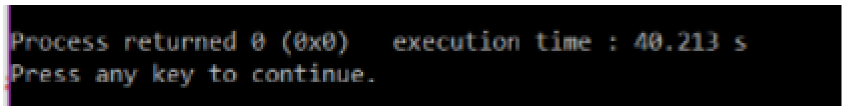
```
28) Else if erased and (right()->front).second
29) Element = (right()->front).first and return erased as false
30) Else if erased and (left()->back).second
31) Element = (left()->back).second and return erased as false
32) Left()->erase(element)
33) Else execute left()->clean(left_tree) and right()->clean(right_tree)
34) If erased then ptr == 0 and delete ptr;
35) End if condition satisfies
```

6) RESULTS , COMPARISON AND DISCUSSION:

Result – compilation time for both are 0 minutes and 0 seconds

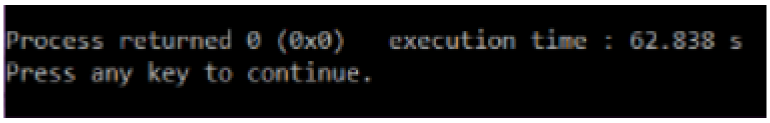
- Running time

  For lazy deletion :



```
Process returned 0 (0x0)   execution time : 40.213 s
Press any key to continue.
```

For hibbard or normal deletion :



```
Process returned 0 (0x0)   execution time : 62.838 s
Press any key to continue.
```

Result – The run time for deletion using lazy method is 40.213 seconds and for the normal deletion or hibbard deletion it is 62.838 seconds . Thus proving the algorithm using lazy deletion is better and efficient .

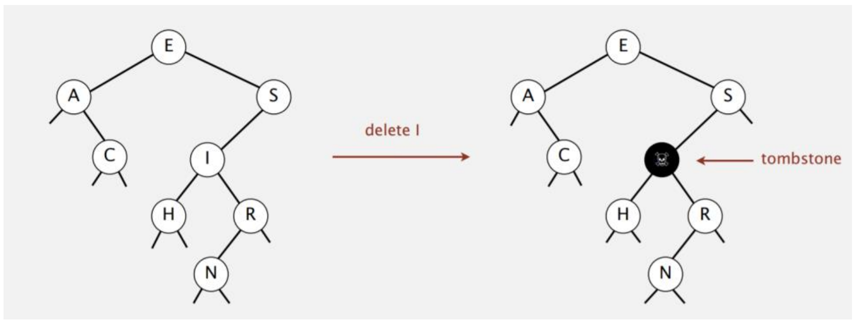
1) Using lazy deletion method :

To remove a node with a given key:

-Set its value to null.

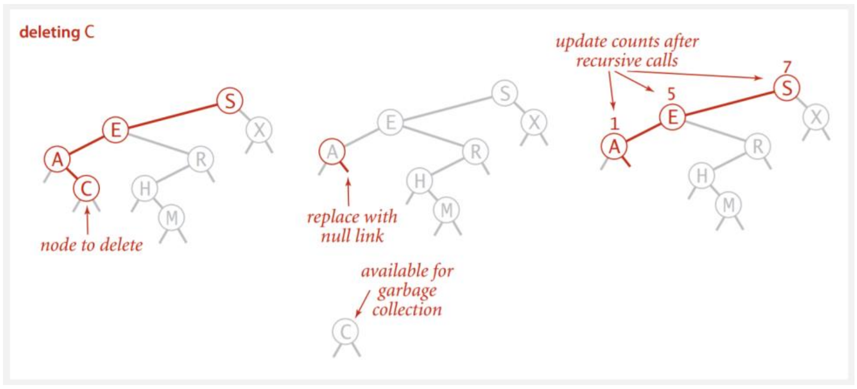
-Leave key in tree to guide search

- in N' per insert, search, and delete (if keys in random order), where N' is the number of key-value pairs ever inserted in the BST.

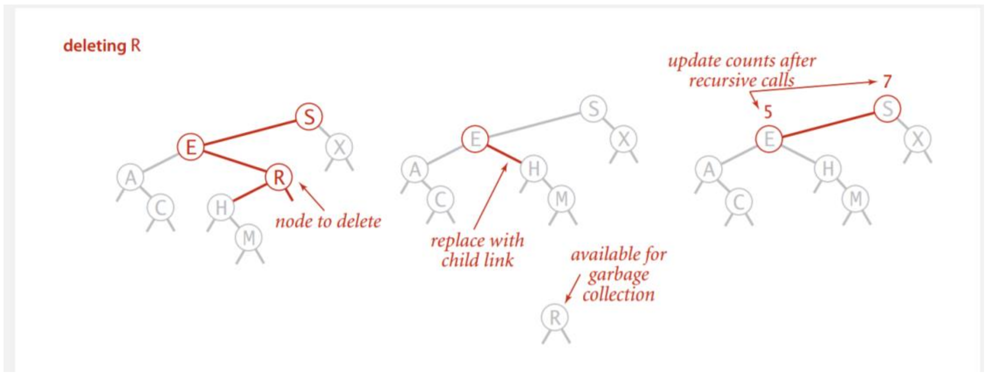


2) **Normal deletion:**

Case 0: [0 children] Delete t by setting parent link to null.


deleting C
node to delete
replace with null link
available for garbage collection
update counts after recursive calls
7
5
1

Case 1: [1 child] Delete t by replacing parent link.

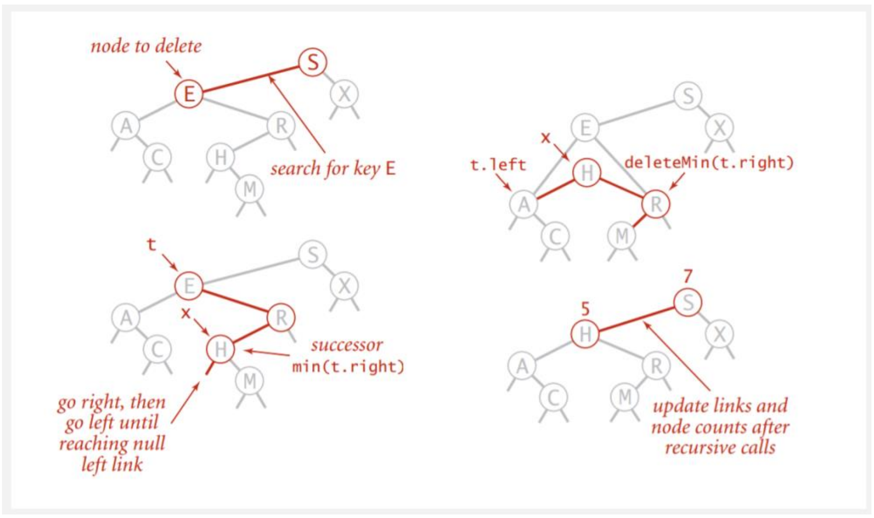

deleting R
node to delete
replace with child link
available for garbage collection
update counts after recursive calls
5
7

Case 2: [2 children]

-Find successor x of t.

-Delete the minimum in t's right subtree.

-Put x in t's spot.


node to delete
search for key E
t
x
successor
min(t.right)
go right, then go left until reaching null left link
t.left
x
deleteMin(t.right)
5
7
update links and node counts after recursive calls

But it gives us the Unsatisfactory solution**.**

7) CONCLUSION:

Thus we find that lazy deletion method is a better way of deletion than the normal one due to the reduced complexity and time taken to run of the program.

8) REFERENCES:

1) R. A. Baeza-Yates. Expected behaviour of B+ - trees under random insertions. Acta Informatica, 1989.

2) T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. MIT Press, Cambridge MA, 1990.

3) D. Comer. The ubiquitous B-tree. ACM Computing Surveys,1979.

4) R. Elmasri and S. B. Navathe. Fundamentals of Database Systems. Benjamin / Cummings, Redwood City CA, second edition, 1994.

5) M. J. Folk and B. Zoellick. File Structures. Addison{Wesley, Reading MA, second edition, 1992.

6) J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kauman, 1993.

7) L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In Proceedings of the 19th Annual Symposium on Foundations of Computer Science,1978.

8) T. Johnson and D. Shasha. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. Journal of Computer and System Sciences,1993.

9) T. Johnson and D. Shasha. The performance of current B-tree algorithms. ACM Transactions on Database systems,1993.

10) D. E. Knuth. The Art of Computer Program- ming: Volume III. Addison{Wesley, Reading MA, 1973.

11) P. Lividas. File Structures: Theory and Practice. Prentice Hall, Englewood Clis NJ, 1990.

12) I. Oliver. Programming Classics: Implementing the World's Best Algorithms.

Prentice Hall, England.