Binary Search Trees, Red-Black Trees & B-Trees

Tom Lancaster

Spring 2019

CS 584 - Algorithm Design & Analysis

Professor Daniel Leblanc

Final Project

Introduction

Trees are an ubiquitous data structure in computer science and the software industry. They store values such that they provide efficient ways to lookup, remove, add and sort. These data structures are essential to many applications and programs. I have chosen to write and do performance tests on three different types of trees - binary search tree, red-black trees and B-trees. Binary search trees “serve as the basis for many data structures” including red-black trees and B-trees.[1] Red-black trees are self-balancing binary search tree with each node containing an extra bit that is “red” or “black” to help ensure approximate balance during insert and delete operations. They are commonly used in computational geometry and for the Completely Fair Scheduler used by current Linux kernels.[2] B-trees (the *B* in B-trees has never been established) are similar to red-black trees, but are specifically designed to be stored on disks.They are commonly used in database and file systems including Apple's file system HFS+, Microsoft's NTFS, AIX’s jfs2 and some Linux file systems, such as btrfs and Ext4. [3] This means you have been using the B-trees data structures every time you use a computer! These data structures have important everyday applications and are beneficial for any computer scientist to study.

By doing a compare and contrast of the different trees, I am better able to understand the “under the hood” parts of database and file systems, and why the creators of these systems chose to implement these type of data structures. The process of going through the pseudocode, writing the code, and running various tests on the three tree data structures helps me understand them more intimately. Also, since I am pursuing the database course track, it prepares me for my future studies.

Algorithms

The most important operations on a tree are: insertion, search and deletion. I have chosen to not go into detail about binary search trees since this is a known concept to graduate students. I will be providing the pseudocode for the red-black trees and B-trees’ insert, search and delete algorithms.

Red-black trees are a binary search tree with either a red or black color attribute in each node that helps maintain balance within the tree.
The red-black properties are as follows:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes. [1]

Red-Black Trees - Insert

Note: See Appendix B for the pseudocode of the supporting functions of these operations.

Insert Pseudocode

- Let *T* be the tree that contains the root and sentinel key value *nil*.
- Let *a* be the preceding node to the inserted one
- Let *b* be a node containing a new value to insert
- Let *r* be a pointer to the current node in the tree

```
InsertRB (T, b)
    a = T.nil
    r = T.root
    while r ≠ T.nil
        a = r
        if b.key < r.key
            r = r.left
        else r = r.right
    b.p = a
    if a == T.nil
        T.root = b
    elseif b.key < a.key
        a.left = b
    else a.right = b
    b.left = T.nil
    b.right = T.nil
    b.color = RED
    InsertColorFix(T,b) [1]
```

The red-black tree insertion works similar to the binary search tree insertion. A pointer *r* goes through the tree comparing the new insert value *b* to other values in the tree. Once the proper empty node location based on the key values is found on the tree, the inserted node replaces the empty node and the left and right pointers are connected to *T.nil*. The inserted node is also given the color red (since both children are black), and the *InsertColorFix* function is called to fix any colors outside of the red-black properties. The insertion operation runs in *O*(lg *n*) because the supporting *InsertColorFix* function takes a total of *O*(lg *n*) time.

Red-Black Trees - Search

The search operation of the red-black tree is the same as a regular binary search tree. (Please see Appendix A for the binary search tree search operation pseudocode.) The program traverses the tree recursively by comparing the value of the search key and the current node. It either finds and returns the node with the search key value or returns *nil* if the value is not in the tree. It runs in *O*(lg *n*) since it traversing the height of a balanced tree.[1]

Red-Black Trees - Delete

Delete Pseudocode

- Assume same *T* and node class as previous functions
- Let *d* be the node to delete
- Let *c* be the node to shift
- Let *p* be a pointer to the predecessor of a node

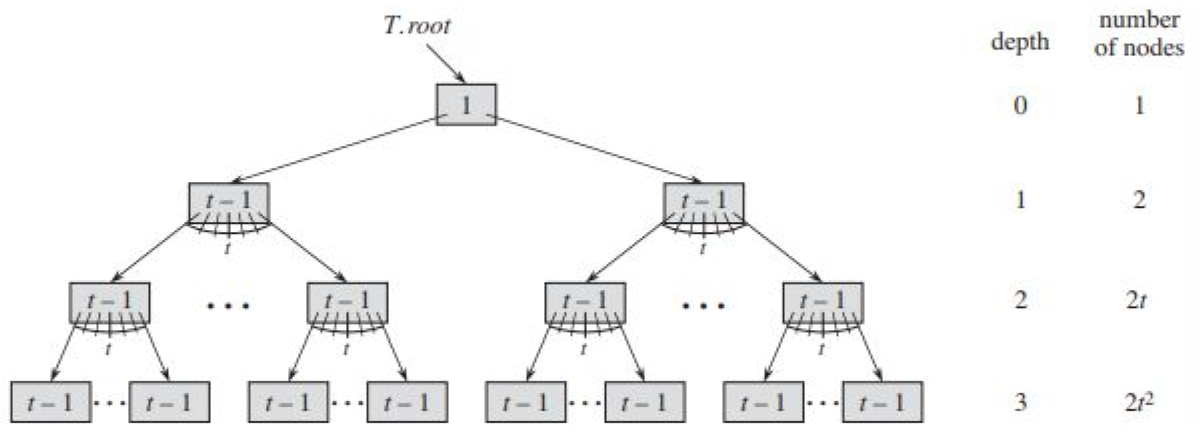
```
DeleteRB (T, d)
    c = d
    c-color-original = c.color
    if d.left == T.nil
        x = d.right
        ShiftRB(T, d, d.right)
    elseif d.right == T.nil
        x = d.left
        ShiftRB(T, d, d.left)
    else c = MinimumNode(d.right)
        c-color-original = c.color
        x = c.right
        if c.p == d
            x.p = c
        else ShiftRB(T, c, c.right)
            c.right = d.right
            c.right.p = c
        ShiftRB(T, d, c)
        c.left = d.left
        c.left.p = c
        c.color = d.color
    if c-color-original == BLACK
        DeleteColorFix(T,x) [1]
```

The DeleteRB function is similar to a binary search tree deletion, but becomes complicated because of the red-black properties. There are three deletion possibilities: delete a leaf, delete a node with 1 child, or delete a node with 2 children. Once one of these deletions take place, the appropriate rearranging of the nodes occurs in the *ShiftRB* function, and the color properties are maintained in the *DeleteColorFix* function. The deletion operation runs in *O*(lg *n*) because of the *DeleteColorFix* function traversing through the tree. [1]

B-trees are rooted trees with the following properties:

1. Every node *x* has *n* number of keys stored in nondecreasing order. Each leaf contains a boolean value to determine whether *x* is a leaf (TRUE) or an internal node (FALSE).

2. Each internal node contains x.n+1 pointers to its children except for leaf nodes.
3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then $k1 \leq x.key_1 \leq k2 \leq x.key_2 \leq \ldots x.key_{x.n} \leq k_{x.n+1}$
4. All leaves have the same depth, which is the tree's height *h*.
5. Nodes have lower and upper bounds on the number of keys they can contain which is expressed by a fixed integer *minimum degree* of the B-tree *t* less than or equal to 2.
    a. Every node other than the root must have at least *t* - 1 keys. Every internal node other than the root thus has at least *t* children. If the tree is nonempty, the root must have at least one key.
    b. Every node may contain at most 2*t* -1 keys. Therefore, an internal node may have at most 2*t* children. We say that a node is full if it contains exactly 2*t* - 1 keys. [1]



B-tree with t children and t - 1 keys [1]

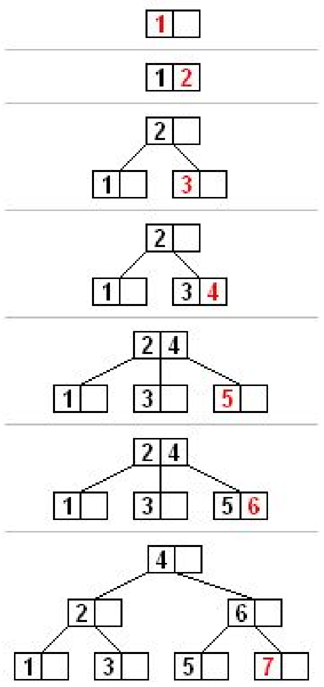
B-Trees - Insert

Note: See Appendix C for the pseudocode of the supporting functions of these operations.

Insert Pseudocode

- Assume a tree was created using the *B-Tree-Create*(T) function
- Assume the root of the B-tree is in main memory
- The node class consists of *n* number of keys, k_i value of *i* key, c_i child pointer to *i* child node, and *leaf* boolean value for whether the node is a leaf or not
- Let *T* be the tree that contains the root and node *x*.
- *Allocate-Node*( ) function allocates one disk page to be used as a new node in *O*(1) time.
- *B-Tree-Split-Child(x, i)* function splits a full child c_i and adjust *x* so that it has an additional child.
- *B-Tree-Insert-Nonfull(x, k)* function inserts key *k* into nonfull node *x* which assumes it is not full when called.

```
InsertB (T, k)
    r = T.root
    if r.n == 2t-1
        s = Allocate-Node( )
        T.root = s
        s.leaf = FALSE
        s.n = 0
        $s.c_1$ = r
        B-Tree-Split-Child(s, 1)
        B-Tree-Insert-Nonfull (s, k)
    else
        B-Tree-Insert-Nonfull(r, k) [1]
```



Insertion example

The B-Tree insert operation begins by checking whether the root is full. If it is, then a new disk page is allocated using the *Allocate-Node* function and it splits the root (which is the only way to increase the height of a b-tree.) Afterwards, through the *B-Tree-Insert-Nonfull*, function it compares the insert node key value to other key values, finds the appropriate node to insert the value in, and does more comparisons until it reaches its proper location. If the root isn't full, then just *B-Tree-Insert-Nonfull* function is called. Although the height of the tree grows *O*(log *n*), the logarithm's base is *t* and therefore, is much more efficient thanks to the multiple keys and children in the tree. [1]

B-Trees - Search

Search Pseudocode

- Assume the root of the B-tree is in main memory
- Assume same *T* and node class as previous functions
- Let *x* be the pointer to some object
- Let *k* be the key value to find
- *Disk-Read*(*x*) is a function that accesses and/or modifies the attributes of *x*

```
SearchB (x, k)
    i = 1
    while i $\leq$ x.n and k > $x.key_i$
        i = i + 1
    if i $\leq$ x.n and k == $x.key_i$
```

```
        return (x, i)
    elseif x.leaf
        return NIL
    else Disk-Read(x, $c_i$)
        return SearchB ($x.c_i$, k)
```

The search operation in a B-tree is similar to a binary search tree's search operation, but in this algorithm, multiway branching decisions are made. It begins by finding the smallest index *i* in the tree and checks to see if this internal branch contains the key. If it does, then it returns the node *x* and index *i*. If cannot find the key in this internal branch and it is on leaf node, then it returns NIL. Otherwise, it continues checking the index against the child pointers and going down the tree until the key is found or it returns NIL. The search operation runs in $O(log_t n)$ since the height of the tree is less than other tree types due to multiple keys and children.

B-Trees - Delete

Delete Pseudocode

- Assume same *T* and node class as previous functions
- Let *x* be the pointer to some object
- Let *k* be the key value to find
- *Preceding*(x) returns the left child of key *x*
- *Successor*(x) returns the right child of key *x*
- *Move-Key*(*k, n1, n2*) moves key *k* from node *n1* to node *n2*
- *Merge-Nodes*(*n1, n2*) merges the keys of nodes *n1* and *n2* into a new node
- *Find-Predecessor*(*n, k*) returns the key preceding key *k* in the child of node *n*
- *Find-Successor*(*n, k*) returns the key succeeding key *k* in the child of node *n*
- *RootKey* finds the root key of the tree
- *Remove-Key*(*k, n*) deletes key *k* from node *n*. *n* must be a leaf node

```
DeleteB (x, k)
    if x.leaf ≠ TRUE
        y = Preceding(x)
        z = Successor(x)
        if y.n > t - 1
            l = Find-Predecessor(k, x)
            Move-Key(l, y, x)
            Move-Key(k, x, z)
            DeleteB(k, z)
        elseif z.n > t - 1
            l = Find-Successor(k, x)
```

```
                    Move-Key(l, z, x)
                    Move-Key(k, x, y)
                    DeleteB(k, y)
                else
                    Move-Key(k, x, y)
                    Merge-Nodes(y, z)
                    DeleteB(k, y)
        else
                y = Preceding(x)
                z = Successor(x)
                w = x.p
                v = RootKey(x)
                if x.n > t-1
                    Remove-Key(k, x)
                elseif y.n > t-1
                    l = Find-Predecessor(w, v)
                    Move-Key(l, y, w)
                    l = Find-Successor(w, v)
                    Move-Key(l, w, x)
                    DeleteB(k, x)
                elseif w.n > t-1
                    l = Find-Successor(w, v)
                    Move-Key(l, z, w)
                    l = Find-Predecessor(w, v)
                    Move-Key(l, w, x)
                    DeleteB(k, x)
                else
                    s = Find-Sibling(w)
                    w' = w.p
                    if w'.n = t - 1
                        Merge-Nodes(w', w)
                        Merge-Nodes(w, s)
                        DeleteB(k, x)
                    else
                        Move-Key(v, w, x)
                        DeleteB(k, x) [4]
```

In the B-tree delete operation, the function deletes the key value *k* from a leaf or an internal node. If *k* is in a leaf, then it is simply deleted as long the node meets the minimum degree *t*. If *t* is not met, then nodes are merged. If *k* is in an internal node, then the node's children must be rearranged to ensure the B-tree's properties are maintained. It achieves this by calling itself recursively to move *k* down the tree until it reaches a leaf node where it is deleted. The delete operation runs in $O(log_t\, n)$ for the same reasons mentioned previously.

Big O Summary

Operation / Tree Type	**Binary Search Tree**	**Red-Black Trees**	**B-Trees**
Insertion	$O(n)$	$O(\log n)$	$O(\log_t n)$
Search	$O(n)$	$O(\log n)$	$O(\log_t n)$
Deletion	$O(n)$	$O(\log n)$	$O(\log_t n)$

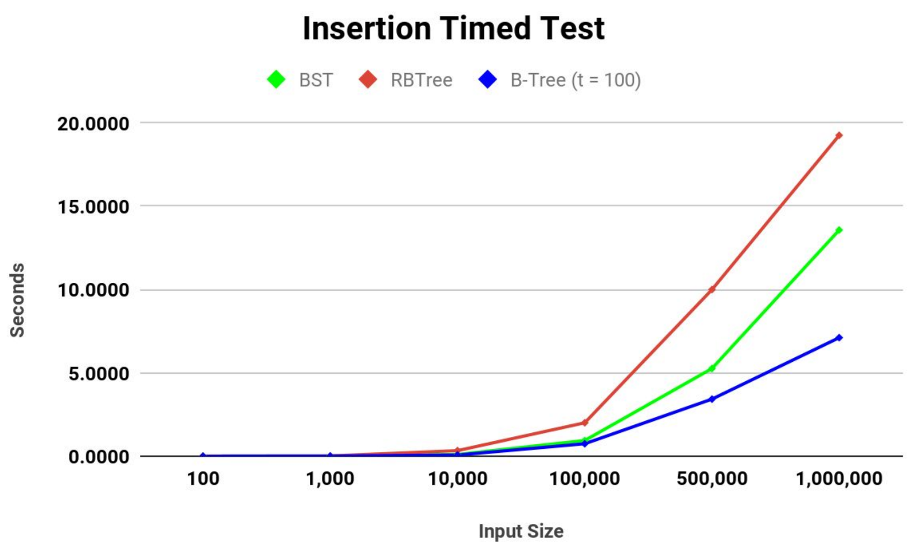
Experimental Procedures

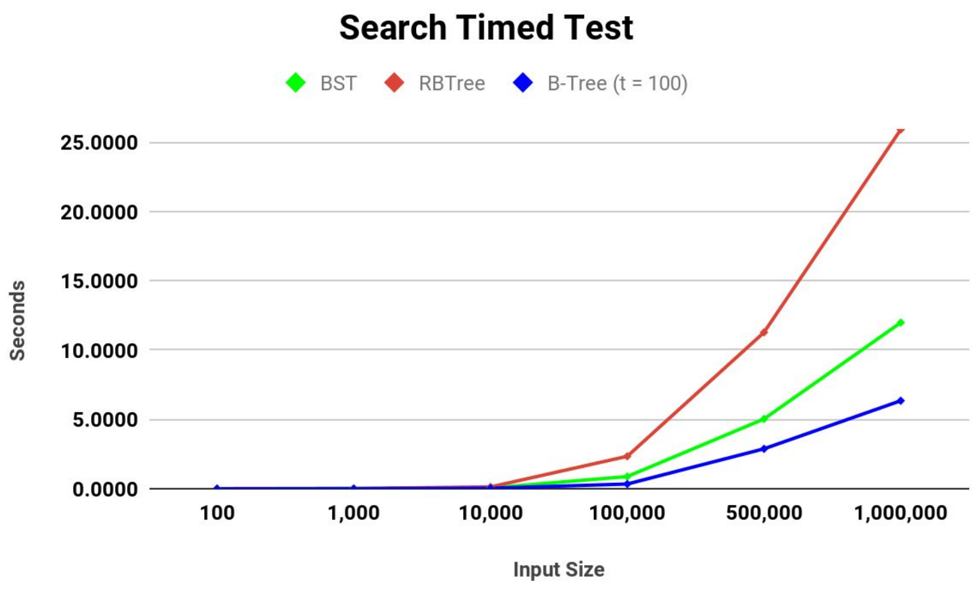
I did my tests in Python and used modified code of binary search trees, red-black trees and B-trees from various online resources. Four set of test procedures were used: the first only inserted *n* random integers into the tree. For the second test, I performed a search on all *n* values. The third test I deleted all *n* integers. The last test performs all three functions at once. For B-trees, I changed the previously mentioned minimum degree *t* by 4, 10, 100, and 1000 to find the effects it makes on our B-trees performance.

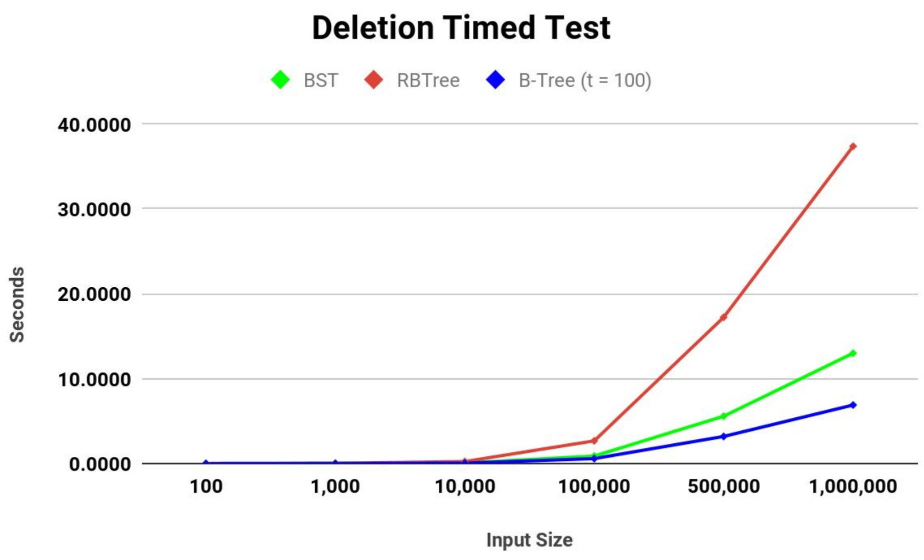
These four test sequences were performed for *n* from 100 to 1,000,000 with *n* growing exponentially after each test run. I ran each test cycle only once due to time constraints with the larger inputs.

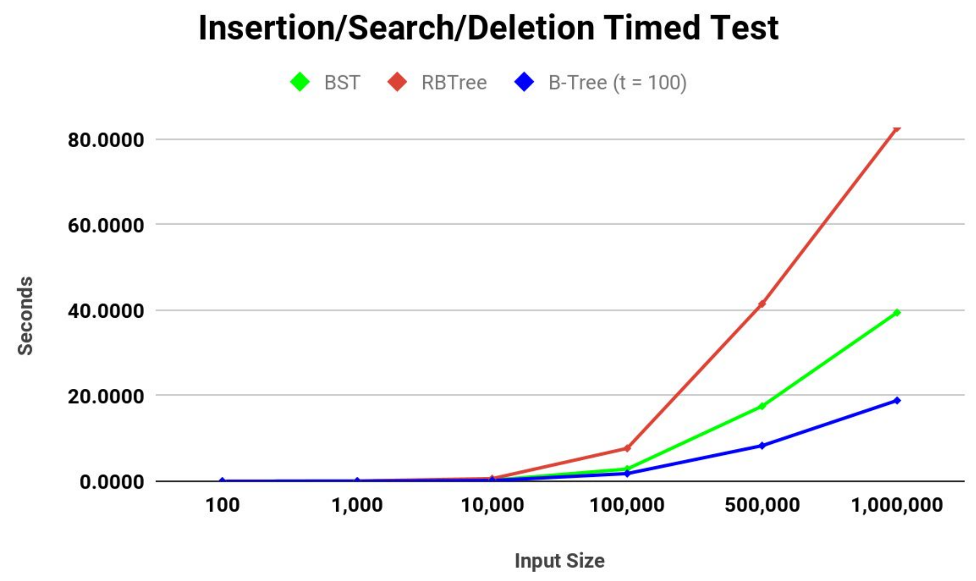
I took many steps to ensure the timed tests only tested the operation and not other parts of the code. First, I put the file I/O outside of the functions to be tested. Second, each search and deletion procedure required a pre-populated tree to exist in order to perform those operations. To make sure I didn't skew the results, I populated a tree *before* performing the search or deletion timed test. I also shuffled the numbers one more time so the program had to work harder to search for or delete numbers. Third, I used the Python library package *timeit* so that garbage collection and any other underlying processes wouldn't corrupt the results.

Results (Raw Data)









Observations/Conclusions

- As you can see from the graphs, the clear winner out of the three tree types as *n* grows exponentially is B-trees; hence why they are used for database and file systems. On all four tests, B-trees is either on par with other trees or is the clear winner as the input size increases. Considering databases and file systems handle *millions or billions* of files and records, these systems will want to use the most efficient data structure to maximize the speed. B-trees structure of containing nodes with multiple keys and children significantly lowers the height and therefore improves the efficiency of trees.

- Each tree type performs nearly the same until about 10,000 integers where a divergence in performance occurs.

- B-tree's performance increases as the minimum degree *t* grows until $t \geq 100$. After a minimum degree of 100, the performance is about the same.

- Red-black trees appears to be the least efficient, but these tests only account for randomized integer lists. When I did a quick separate insertion test on binary search trees using a *sorted* list instead of a randomized list, it failed to insert more than 994 values before interrupting the program. This occurred because the binary search tree uses a recursive call function which after 994 values became too costly of an operation. It also became degenerate or simply a linked list which nullifies the benefits of a tree. This is unacceptable considering how often large amounts of data is sorted nowadays. This makes red-black trees a more stable second option to B-trees.

- If you have less than 100,000 items of *random* data and would like to use a tree data structure, your best bet is a binary search tree since it is quick to code or implement. If it is sorted or greater 100,000 items, then a B-tree with a minimum degree $t \geq 100$ is the best tree structure to use.

Citations / References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to algorithms*, Cambridge (Inglaterra): Mit Press.
   Chapter 12: Binary Search Trees, Ch 13: Red-Black Trees, Ch 18: B-Trees

2. Anon. 2019. Red–black tree. (June 2019). Retrieved June 6, 2019 from https://en.wikipedia.org/wiki/Red–black_tree

3. Anon. 2019. B-tree. (May 2019). Retrieved June 6, 2019 from https://en.wikipedia.org/wiki/B-tree

4. A. Kaltenbrunner, L. Kellis, and D. Marti. Btrees.pdf. Retrieved June 6, 2019 from http://www.di.ufpb.br/lucidio/Btrees.pdf

5. Anon. 2019. Binary tree. (June 2019). Retrieved June 6, 2019 from https://en.wikipedia.org/wiki/Binary_tree

6. Anon. 2019. Binary search tree. (May 2019). Retrieved June 6, 2019 from https://en.wikipedia.org/wiki/Binary_search_tree

7. Anon. 2018. Binary Search Tree | Set 1 (Search and Insertion). (September 2018). Retrieved June 6, 2019 from https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

8. Anon. 2019. Binary Search Tree | Set 2 (Delete). (May 2019). Retrieved June 6, 2019 from https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/

9. Stanislav Kozlovski. 2018. Red-Black-Tree. (May 2018). https://github.com/stanislavkozlovski/Red-Black-Tree/blob/master/rb_tree.py#L7

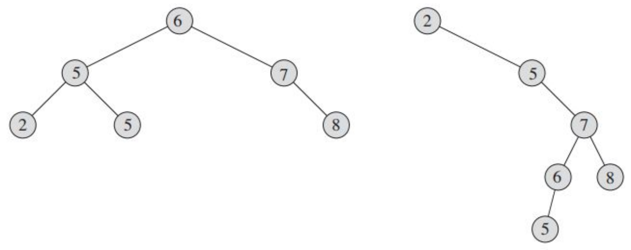
10. Martin Thoma. 2012. btree.py. (July 2012). https://gist.github.com/MartinThoma/3159687

Appendix A: Binary Search Trees

Binary search tree is a node-based tree data structure that orders all of the values or keys in such a way that it satisfies the binary-search-tree property:

> Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key $\leq$ x.key. If y is a node in the right subtree of x, then y.key $\geq$ x:key. (#)

There cannot be any duplicate keys in a binary search tree. They can come in different shapes and forms though. The figure below shows two different trees with the same keys.



(a) (b)

The following pseudocodes assume a node class that creates a new node with empty left and right pointers.

Binary Search Tree Operations in Pseudocode

```
InsertBST(root, node)
    if root is empty: //
        root = node
    else:
        if root.value < node.value
            if root.right is empty
                root.right = node
            else:
                InsertBST(root.right, node)
        else:
            if root.left is empty:
                root.left = node
            else:
                InsertBST(root.left, node)
```

```
SearchBST(node, sval)
    if node == NIL or sval == node.key
        return node
    if sval < node.key
        return SearchBST(node.left, sval)
    else return SearchBST(node.right, sval)
```

```
MinimumNode(node)
    while node.left ≠ NIL
        x = x.left
    return x


TreeShift(Tree,u,v)
    if u.parent == NIL
        Tree.root = v
    elseif u == u.parent.left
        u.parent.left = v
    else u.parent.right = v
    if v ≠ NIL
        v.parent = u.parent
```

```
DeleteBST(Tree, z)
    if z.left == NIL
        TreeShift(Tree, z, z.right)
    elseif z.right == NIL
        TreeShift(Tree, z, z.left)
    else y = MinimumNode(z.right)
        if y.parent ≠ z
            TreeShift(Tree, y, y.right)
            y.right.parent = y
        TreeShift(Tree, z, y)
        y.left = z.left
        y.left.parent = y
```

Appendix B: Red-Black Trees Support functions

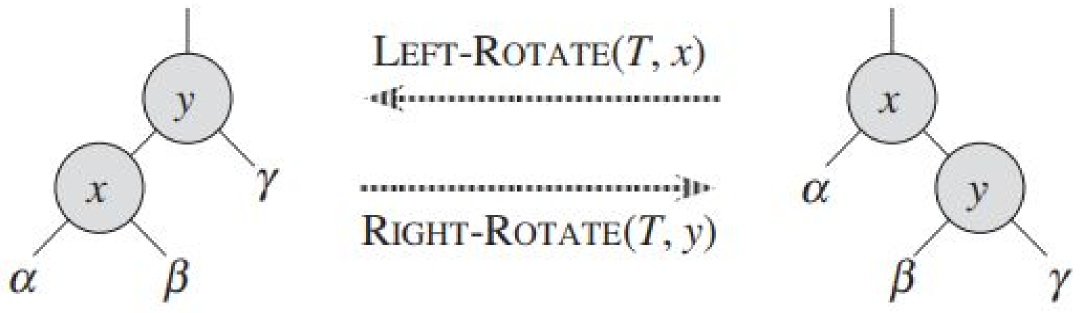
Supporting Insert Functions Pseudocode

- Let *T* be the tree that contains the root and sentinel key value *nil*.
- Let *a* be the node to be rearranged.
- The node class include *color* (red or black), *key* (value), *left* and *right* pointers, and *p* pointer (predecessor or parent).

```
LeftRearrange (T, a)
b = a.right
a.right = b.left
if b.left ≠  T.nil
    b.left.p = a
b.p = a.p
if a.p == T.nil
    T.root = b
elseif a == a.p.left
    a.p.left = b
else
    a.p.right = b
    b.left = a
    a.p = b
```

- Assume same *T* and node class as previous function.
- Let *b* be a node further down the tree.
- Let *a* be a different color node further up the tree.

```
InsertColorFix (T,  a)
while b.p.color == RED
    if b.p == b.p.p.left
        a == b.p.p.right
        if a.color == RED
            b.p.color = BLACK
            a.color = BLACK
            b.p.p.color = RED
            b = b.p.p
        else if b == b.p.right
              b = b.p
             LeftRearrange(T, b)
            b.p.color = BLACK
            b.p.p.color = RED
            RightRearrange(T, b.p.p)
    else
          (exchange code above for  “right”
    and “left” functions)
T.root.color = BLACK
```



Following the pseudocode and the visual from CLRS above, the node *a* (node *x* in the figure above) pointer structure switches spots on the tree with node *b* (node *y* in the figure above) by the *LeftRearrange* function or its inverse *RightRearrange* depending on the direction it needs to shift. This function runs in *O*(1) time. The InsertColorFix function maintains the red-black properties.

Supporting Delete Functions Pseudocode

- Assume same *T* and node class as previous functions
- Let *u* be the starting node to be rearranged
- Let *v* be the ending node to be arranged

```
ShiftRB (T, u, v)
    if u.p == T.nil
        T.root = v
    elseif u == u.p.left
        u.p.left = v
    else
        u.p.right = v
    v.p = u.p


MinimumNode(node)
    while node.left ≠ NIL
        x = x.left
    return x
```

- Assume same *T* and node class as previous functions
- Let *b* be the node of the color to fix
- Let *a* be the node to shift

```
DeleteColorFix(T, d)
    while d ≠ T.root and d.color == BLACK
        if d == d.p.left
            c = d.p.right
            if c.color == RED
                c.color = BLACK
                d.p.color = RED
                LeftRearrange(T, d.p)
                c = d.p.right
            if c.left.color == BLACK and
    c.right.color == BLACK
                c.color = RED
                d = d.p
            else if c.right.color == BLACK
                c.left.color == BLack
                c.color = RED
                RightRearrange(T, c)
                c = d.p.right
              c.color = d.p.color
              d.p.color = BLACK
              c.right.color = BLACK
              LeftRearrange(T, d.p)
              d = T.root
        else
(same as statement above but "right and "left"
exchanged)
    d.color = BLACK
```

Appendix C: B-Trees Supporting Functions

```
B-TREE-CREATE(T)
1  x = ALLOCATE-NODE()
2  x.leaf = TRUE
3  x.n = 0
4  DISK-WRITE(x)
5  T.root = x
```

B-TREE-CREATE requires $O(1)$ disk operations and $O(1)$ CPU time.

```
B-TREE-SPLIT-CHILD(x, i)
 1  z = ALLOCATE-NODE()
 2  y = x.c_i
 3  z.leaf = y.leaf
 4  z.n = t - 1
 5  for j = 1 to t - 1
 6      z.key_j = y.key_{j+t}
 7  if not y.leaf
 8      for j = 1 to t
 9          z.c_j = y.c_{j+t}
10  y.n = t - 1
11  for j = x.n + 1 downto i + 1
12      x.c_{j+1} = x.c_j
13  x.c_{i+1} = z
14  for j = x.n downto i
15      x.key_{j+1} = x.key_j
16  x.key_i = y.key_t
17  x.n = x.n + 1
18  DISK-WRITE(y)
19  DISK-WRITE(z)
20  DISK-WRITE(x)
```

```
B-TREE-INSERT-NONFULL(x, k)
 1  i = x.n
 2  if x.leaf
 3      while i ≥ 1 and k < x.key_i
 4          x.key_{i+1} = x.key_i
 5          i = i - 1
 6      x.key_{i+1} = k
 7      x.n = x.n + 1
 8      DISK-WRITE(x)
 9  else while i ≥ 1 and k < x.key_i
10          i = i - 1
11      i = i + 1
12      DISK-READ(x.c_i)
13      if x.c_i.n == 2t - 1
14          B-TREE-SPLIT-CHILD(x, i)
15          if k > x.key_i
16              i = i + 1
17      B-TREE-INSERT-NONFULL(x.c_i, k)
```