Analysis of Algorithms

BLG 335E

Project 3 Report

SELİN YILMAZ
yilmazsel21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 20.12.2024

1. Implementation

1.1. Data Insertion

1.1.1. Binary Search Tree - Data Insertion

Implementation and execution of functions in short looks like:

- `main` function takes the arguement 1 as csv file name. Calls the generate_BST_tree_from_csv function using that name.
- `generate_BST_tree_from_csv` starts to parse and read the lines. It decides the year of that game belongs to which decade. Compares that decade with the `currentDecade` (defined as global variable; used for both calling print_best_seller and finding decade change). If there exists a decade change (e.g. current decade is 1990 year is 2001) calls find_best_seller() function.

  It creates a `vector<string> n` vector. This vector holds the publisher_name, na_sales, eu_sales, and other_sales string data. It calles the function `insertValue(vector<string> n)` and provides the vector it created.
- `insertValue` creates a new node with those values and calls `BST_insert(root, newnode)` function.
- `BST_insert` checks if the provided node is already in the tree. If it is already in the tree, it updates the existing node. If it is not in the tree, it finds where to add it. Then, it returns the root.

Its complexity is $O(n)$ because it is unbalanced. If a sorted dataset had been given, the tree would have become a linked list.

It takes 53056 microseconds to insert all the data in the dataset into the tree.

1.1.2. Red-Black Tree - Data Insertion

Actually, it uses a very similar logic to the Binary Search Tree.

- `main` function takes argument 1 as CSV file name. Calls the generate_RBT_tree_from_csv function using that name.
- `generate_RBT_tree_from_csv` starts to parse and read the lines. It decides the year of that game belongs to which decade. Compares that decade with the `currentDecade` (defined as global variable; used for both calling print_best_seller and finding decade change). If there exists a decade change (e.g. current decade is 1990 year is 2001) calls find_best_seller() function.

It creates a `vector<string> n` vector. This vector holds the publisher_name, na_sales, eu_sales, and other_sales string data. It calles the function `insertValue(vector<string> n)` and provides the vector it created.

- `insertValue` creates a new node with those values. Every new node's color is set to RED (1). Then, calls `RB_insert(root, newnode)` function.
- `RB_insert` checks if the provided node is already in the tree. If it is already in the tree, it updates the existing node and returns the root. If it is not in the tree, it finds where to add it. Then calls the RB_insert_fixup(newnode) function.
- `RB_insert_fixup` basically checks a set of rules. Pseudocode for it is:

```
function RB_insert_fixup(ptr):
    while ptr.parent != NULL and ptr.parent.color == RED do:
        if ptr.parent == ptr.parent.parent.left then:
            uncle <- ptr.parent.parent.right
            if uncle != NULL and uncle.color == RED then:
                ptr.parent.color <- BLACK
                uncle.color <- BLACK
                ptr.parent.parent.color <- RED
                ptr <- ptr.parent.parent
            else:
                if ptr == ptr.parent.right then:
                    ptr <- ptr.parent
                    RB_Left_Rotate(ptr)
                ptr.parent.color <- BLACK
                ptr.parent.parent.color <- RED
                RB_Right_Rotate(ptr.parent.parent)
        else:
            uncle <- ptr.parent.parent.left
            if uncle != NULL and uncle.color == RED then:
                ptr.parent.color <- BLACK
                uncle.color <- BLACK
                ptr.parent.parent.color <- RED
                ptr <- ptr.parent.parent
            else:
                if ptr == ptr.parent.left then:
                    ptr <- ptr.parent
                    RB_Right_Rotate(ptr)
                ptr.parent.color <- BLACK
                ptr.parent.parent.color <- RED
```

```
                    RB_Left_Rotate(ptr.parent.parent)
        root.color <- BLACK
    end function
```

Basically, it calls right rotation if nodes conflict as left-left, it calls left rotation if they conflict as right-right, it calls first left then right rotation if they conflict as right-left, and it calls first right then left rotation if they conflict as left-right.

- `RB_left_rotate` switches placement of nodes. Main node's right (ptr->right) is called y. We make ptr y's left child. And ptr's parent points to y (Fig. 1.1). `RF_right_rotation` works similarly.

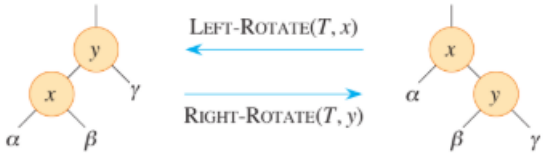


Figure 1.1: Rotations

Its time complexity is $O(logn)$ for every case.
It takes 55221 microseconds to insert all the data in the dataset into the tree.

1.2. Search Efficiency

1.2.1. Binary Search Tree - Search Efficiency

For this part, I implemented a random node generator function and a search function. `generate_random_node` takes a vector that includes all the nodes in the tree and returns a random index. The search function takes a tree and a node to search. It iterates through the tree and returns the address of the node.

The time complexity of the search depends on the shape of the tree. Because this is an unbalanced tree. In the best case, it is $O(1)$. In average case, it is $O(h)$ where h is the *height* of the tree. It is approximately $O(logn)$. In the worst case, it is $O(n)$.

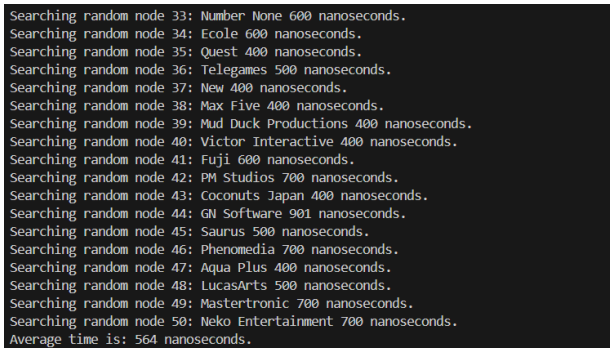


Figure 1.2: BST - random 50 nodes search

Searching in the binary search tree for 50 random nodes takes 564 nanoseconds on average (Fig. 1.2).

1.2.2. Red-Black Tree - Search Efficiency

For this part, I implemented a random node generator function and a search function also here.

The time complexity of the search is $O(1)$ for the best case in which the tree has just one node, $O(logn)$ for the average and worst case.

Searching in the red-black tree for 50 random nodes takes 802 nanoseconds on average (Fig. 1.3).

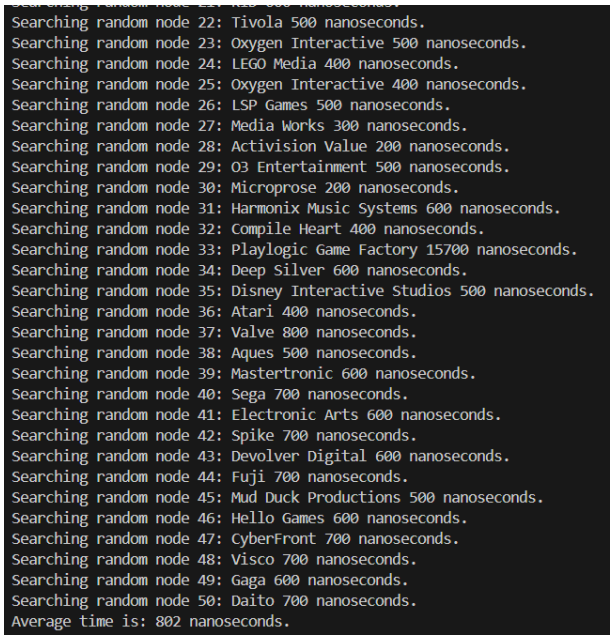


Figure 1.3: RBT - random 50 nodes search

1.2.3. Comparison

Tree Type	Best Case	Average Case	Worst Case
Red-Black Tree	$O(1)$	$O(\log n)$	$O(\log n)$
Unbalanced BST	$O(1)$	$O(\log n)$	$O(n)$

Table 1.1: Comparison of Search Complexity Between Red-Black Tree and Unbalanced BST

1. Red-Black Tree is more consistent. It maintains a balanced structure. Because it guarantees that the longest path from the root to any is no more than twice of the shortest path. So the height remains $O(logn)$.
2. RBT being self-balancing also causes it to complete searches more efficiently.
3. Binary Search Tree does not maintain a balanced structure. So it does not maintain a uniform height. If nodes are inserted in sorted order, the tree becomes a linked list.
4. In BST the height may grow to n. So making search, insertion, and deletion operations may be as slow as $O(n)$.
5. Unbalanced BSTs are easier to implement and they may be acceptable for small datasets.

1.3. Best-Selling Publishers at the End of Each Decade

1.3.1. Binary Search Tree - Best-Seller

- `generate_BST_tree_from_csv` function calls `find_best_seller` when a decade changes (e.g when the first 1991 year is read, or 2001, or 2011 and after the csv is file is read).
- It checks if the tree is empty. If it is not it creates a stack and pushes the root in it.
- Then, in a loop until stack becomes empty:
  - Loads the top element to a *current* node and pops the last element.
  - Checks if the `best_seller` array is empty. If it is empty, the values of the current element are put into it. If it is not empty, then pushes the values greater than the ones in array.
  - Pushes the children of the current node to stack (right child first, in order to create a preorder structure).
- After the loop ends, the `print_best_sellers` function is called by find_best_seller.
- `print_best_sellers` takes a year and an array as parameters. I provided the year using the global variable *currentDecade* (I defined), and provided the array using the array I created above.
- `print_best_sellers` prints the year and best sales (Fig. 1.4).

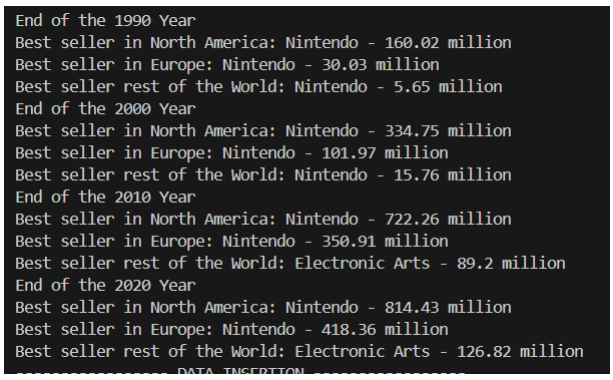


Figure 1.4: BST Best-Seller Output

1.3.2. Red-Black Tree - Best-Seller

I used the same functions also here. Its output looks like Fig. 1.5.

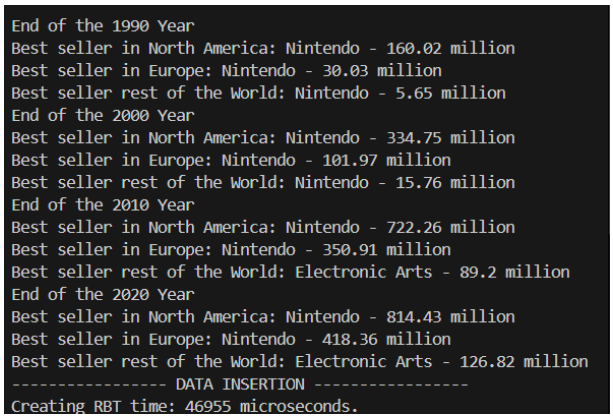
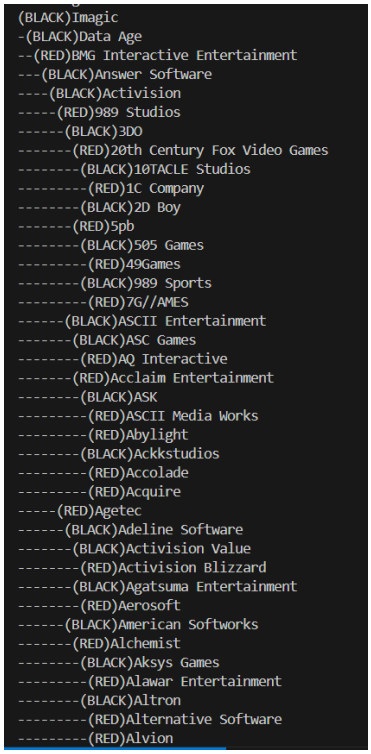


Figure 1.5: RBT Best-Seller Output

1.3.3. Preorder Traversal of RBT

I implemented the `preorder()` function to print the nodes of the tree preorder.

- It checks if the tree is empty. If it is empty then moves out of the function.
- Creates a stack of pairs. *pair* allows us to group related data (here it is node* and depth). Then, push the root and depth of root (0) to stack.
- In a loop until stack becomes empty:
  - Pop stack's top element's first element into *current* node, and second element into *depth*. Pop the last element of the stack.
  - Check the color of the current node. If it is 1 write "(RED)" inside *color* string, or if it is 0 write "(BLACK)" inside of color.
  - Then, it writes "-" *depth* times, color, current node's name on the screen (Fig. 1.6).
  - Push children of current into the stack (starting from the right to protect the preorder structure).



(a)

(b)

Figure 1.6

1.4. Final Tree Structure

How Balanced is the Binary Search Tree compared to the Red-Black Tree?

1. **Binary Search Tree (BST):**

   - Balance depends entirely on the order of insertions.
   - The tree may have a reasonably balanced structure if the dataset is randomly ordered. However, if the dataset is sorted or nearly sorted, it becomes highly unbalanced, in fact it becomes a linked list.
   - The height can grow to $O(n)$, where n is the size. And this results in poor performance for search, insert, and delete operations (also $O(n)$).

2. **Red-Black Tree (RBT):**

   - It can be called as a self-balancing binary search tree.
   - It guarantees the height to be always $O(logn)$, regardless of the order of the insertions or deletions. So it creates a consistent environment for search, insert, and delete operations.
   - Balance is maintained through color properties and tree rotations.

1.5. Write Your Recommendation

My tree structure recommendation for managing the dataset valley is *Red-Black Tree (RBT)*. Justification is:

1. Its performance is consistent. RBT ensures $O(logn)$ time complexity for tree operations. This is critical when dealing with large datasets. Because it provides efficiency and scalability.

2. The dataset order does not affect it. However, BST can be degraded by this property of datasets. Unlike BST, RBT's self-balancing property ensures predictable performance.

3. RBT's management of the dataset is better if frequent updates are required. Because it automatically maintains its balance.

4. RBTs can be used to implement *map* and *set*.

5. BST can be used when the dataset is small, the order of insertions is controlled (avoiding imbalance), or the implementation being simple is important.

1.6. Ordered Input Comparison

1.6.1. Binary Search Tree - Search

For this part, I implemented a code that reads the original csv file *VideoGames.csv* orders it by game names, and creates a new csv file *ordered_name.csv*. Then, I generated and searched 50 random nodes using the search algorithm I coded for part B. It takes 1224 nanoseconds to search 50 random nodes. It was 564 nanoseconds for the unordered dataset.

Actually, this result surprised me. Because I thought BST being imbalanced would cause more trouble. But, it still highlights the weakness of BST: its efficiency depends on the order of the input.

1.6.2. Red-Black Tree - Search

It takes 1424 nanoseconds to search 50 random nodes. It was 802 nanoseconds for the unordered dataset. So it can be observed that the red-black tree structure is consistent regardless of the order of the data.

1.6.3. Conslusion

For unordered datasets, BST may offer a faster search in some cases. However, it is not consistent, so the tree must be relatively balanced. For ordered datasets, RBT outperforms BST as RBT is self-balancing. Overall, RBT is more consistent, and this makes it more reliable.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 4 edition, 2022.