Analysis of Algorithms

BLG 335E

Project 3 Report

Hilal Kartal
kartal@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 18.12.2024

1. Implementation

1.1. Data Insertion

1.1.1. BINARY SEARCH TREE

1. Nodes are ordered on the tree by the keys($publisher\ name$).
2. At each line read, code stores the necessary values for $insertValue$ in a vector of string vectors called row_values
    - At row_values each subvector stores:

    0 : publisher name

    1 : NA sales

    2 : EU sales

    3 : Other sales
3. In $insertValue$, a new publisher is declared with the values from row_values subvectors. Also a new node is created with the new publisher and passed to BST_insert.
4. In BST_insert, corresponding place for the new node is found.
    - If publisher already exists sales values are updated with the new nodes
    - If it does not exist new node is inserted into tree.
5. Data insertion takes 62914 microseconds. Figure:1.1

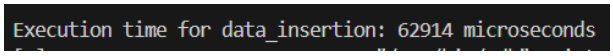

Figure 1.1: Time for BST data insertion

1.1.2. RED-BLACK TREE

1. Nodes are ordered on the tree by the keys($publisher\ name$).
2. At each line read, code stores the necessary values for $insertValue$ in a vector of string vectors called row_values
    - At row_values each subvector stores:

    0 : publisher name

    1 : NA sales

2 : EU sales

3 : Other sales

3. In $insertValue$, a new publisher is declared with the values from row_values subvectors. Also a new node is created with the new publisher and passed to RB_insert.

4. In RB_insert, corresponding place for the new node is found.

   - If publisher already exists sales values are updated with the new nodes
   - If it does not exist new node is inserted into tree.

5. At the end of RB_insert, RB_insert_fixup is called.

   - It fixes the the structure of the tree fit red-black properties:
     (a) Node Color: Each node is either red or black
     (b) Root Property: The root of the tree is always black.
     (c) Red Property: Red nodes cannot have red children (no two consecutive red nodes on any path).
     (d) Black Property: Every path from a node to its descendant null nodes (leaves) has the same number of black nodes.
     (e) Leaf Property: All leaves (NIL nodes) are black.

6. Data insertion takes 81413 microseconds. Figure:1.2

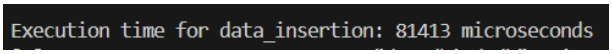

Figure 1.2: Time for BST data insertion

1.2. Search Efficiency

1.2.1. BINARY SEARCH TREE

To search 50 random publishers and take the average time:

1. A global vector $publishers$ is defined to store each unique publisher name.

2. Function $random\_search$ is defined

   - It takes tree and $publishers$ vectors size and does 50 random search operations with $find\_publishers$. Then calculates the average time. Figure: 1.3

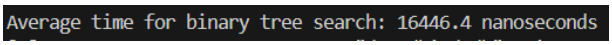

Figure 1.3: Time for average BST search

3. $find\_publishers$ is defined.

    - The function takes a publisher name as input and does binary search on the tree to find corresponding publisher. Figure: 1.4

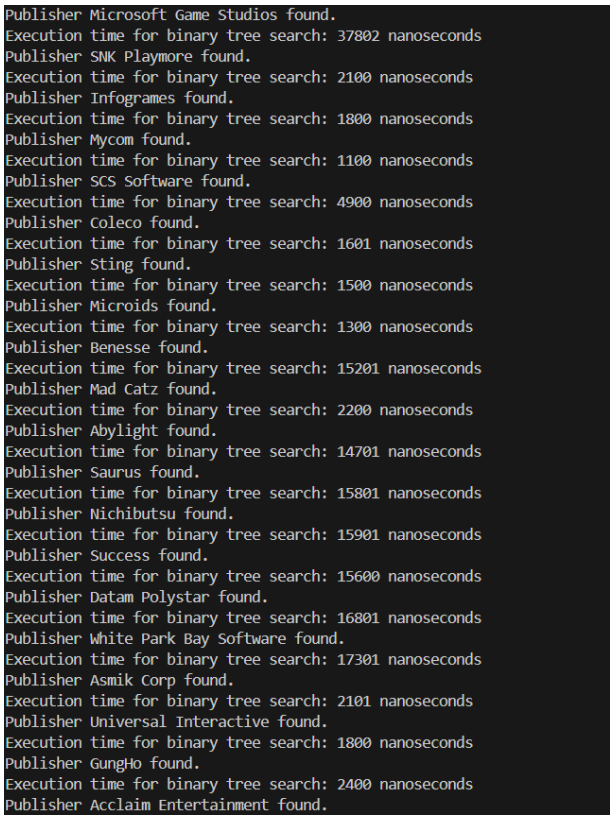

Figure 1.4: Time for BST search

1.2.2. RED-BLACK TREE

To search 50 random publishers and take the average time:

1. A global vector $publishers$ is defined to store each unique publisher name.
2. Function $random\_search$ is defined

    - It takes tree and $publishers$ vectors size and does 50 random search operations with $find\_publishers$. Then calculates the average time. Figure: 1.5

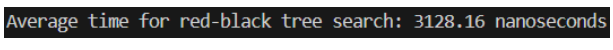

Figure 1.5: Time for average RBT search

3. $find\_publishers$ is defined.

- The function takes a publisher name as input and does binary search on the tree to find corresponding publisher. Figure: 1.6

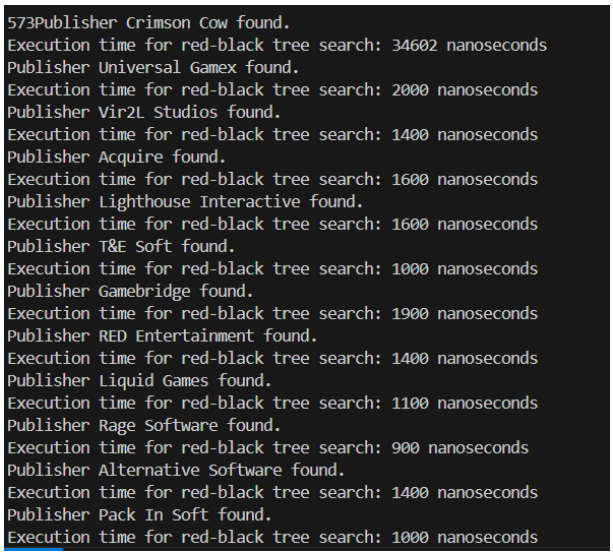

Figure 1.6: Time for RBT search

4. Each search is timed and added and divided by 50 to find the average time for binary search.

1.2.3. Comparison

From the Figures: 1.3 and 1.5, it can be seen that RBT is much more better for search as the tree is more balanced.

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

1.3.1. BINARY SEARCH TREE

1. Best sellers are printed while generating the tree from csv according to their decade
    - there are boolean checks for each decade to only do the print once. They are set to true when that decade is printed to ensure code does not enter the condition again.
2. In each condition $find\_best\_seller$ is called to find the best seller from the tree.
    - In $find\_best\_seller$, a default publisher with all sales equal to 0 is created and pushed to array.
    - then code goes over the tree with $preorder$ to find best sellers.
3. Actual finding operation is done in $preorder$
    - It recurs the tree and does the condition checks for each sale to update the $best\_seller$

4. Then $print\_best\_sellers$ is called to print best sellers from $best\_seller$ 1.7

```
End of the 1990 Year
Best seller in North America: Nintendo - 160.02 million
Best seller in Europe: Nintendo - 30.03 million
Best seller rest of the World: Nintendo - 5.65 million
End of the 2000 Year
Best seller in North America: Nintendo - 334.75 million
Best seller in Europe: Nintendo - 101.97 million
Best seller rest of the World: Nintendo - 15.76 million
End of the 2010 Year
Best seller in North America: Nintendo - 722.26 million
Best seller in Europe: Nintendo - 350.91 million
Best seller rest of the World: Electronic Arts - 89.2 million
End of the 2020 Year
Best seller in North America: Nintendo - 814.43 million
Best seller in Europe: Nintendo - 418.36 million
Best seller rest of the World: Electronic Arts - 126.82 million
```

Figure 1.7: Best Sellers from BST

1.3.2. RED-BLACK TREE

1. Best sellers are printed while generating the tree from csv according to their decade
    - there are boolean checks for each decade to only do the print once. They are set to true when that decade is printed to ensure code does not enter the condition again.
2. In each condition $find\_best\_seller$ is called to find the best seller from the tree.
    - In $find\_best\_seller$, a default publisher with all sales equal to 0 is created and pushed to array.
    - then code goes over the tree with $preorder\_for\_bestseller$ to find best sellers.
    - It recurs the tree and does the condition checks for each sale to update the $best\_seller$
3. Then $print\_best\_sellers$ is called to print best sellers from $best\_seller$ 1.8

```
End of the 1990 Year
Best seller in North America: Nintendo - 160.02 million
Best seller in Europe: Nintendo - 30.03 million
Best seller rest of the World: Nintendo - 5.65 million
End of the 2000 Year
Best seller in North America: Nintendo - 334.75 million
Best seller in Europe: Nintendo - 101.97 million
Best seller rest of the World: Nintendo - 15.76 million
End of the 2010 Year
Best seller in North America: Nintendo - 722.26 million
Best seller in Europe: Nintendo - 350.91 million
Best seller rest of the World: Electronic Arts - 89.2 million
End of the 2020 Year
Best seller in North America: Nintendo - 814.43 million
Best seller in Europe: Nintendo - 418.36 million
Best seller rest of the World: Electronic Arts - 126.82 million
```

Figure 1.8: Best Sellers from RBT

1.3.3. RBT - PREORDER

1. To print out the trees nodes in preorder. $preorder$ function is called.
2. In the $preorder$, a call to $actual\_preorder$ is made by giving the root and a empty string value for depht.
3. In $actual\_preorder$:
    - Preorder traversal is made
    - Also $tree\_deep\_stack$ is used to keep the depht.
4. Result of the preorder can be seen from 1.9

```
(BLACK)Imagic
-(BLACK)Data Age
--(RED)BMG Interactive Entertainment
---(BLACK)Answer Software
----(BLACK)Activision
-----(RED)989 Studios
------(BLACK)3DO
-------(RED)20th Century Fox Video Games
--------(BLACK)10TACLE Studios
---------(RED)1C Company
--------(BLACK)2D Boy
-------(RED)5pb
--------(BLACK)505 Games
---------(RED)49Games
--------(BLACK)989 Sports
---------(RED)7G//AMES
------(BLACK)ASCII Entertainment
-------(BLACK)ASC Games
--------(RED)AQ Interactive
-------(RED)Acclaim Entertainment
--------(BLACK)ASK
---------(RED)ASCII Media Works
---------(RED)Abylight
--------(BLACK)Ackkstudios
---------(RED)Accolade
---------(RED)Acquire
-----(RED)Agetec
------(BLACK)Adeline Software
```

Figure 1.9: Preorder of RBT

1.4. Final Tree Structure

1. RBT is more balanced than BST.
2. BST has no property that makes it balanced. Its structure is totally dependent on the order of input data. As can be seen from subsection: 1.6.2
3. However, RBT is kept balanced to fit the properties defined in list: 5
4. Heights for the trees can be compared as:

- BST: $O(n)$
- RBT: $O(log(n))$

1.5. Write Your Recommendation

- Based on the analysis of both trees, it can be seen that RBT is much better when dataset is large.
- Also, structure of the RBT is not dependent on input order, so it is safer to use with large data and expect a normal tree.
- In conclusion, RBT is better to use in this Data Valley.

1.6. Ordered Input Comparison

1.6.1. ORDERING OF CSV

1. Function $ordercsv$ is defined.
2. It reads from the csv and orders each rows values at the vector of string vectors row_values.
    - At row_values each subvector stores:

    0 : name

    1 : platform

    2 : year of release

    3 : publisher name

    4 : NA sales

    6 : EU sales

    7 : Other sales
3. The vector is then sorted using $sort()$
4. Values are read to file $ordered\_data.csv$

1.6.2. AVERAGE SEARCH TIMES FOR BST AND RB

```
Average time for binary tree search: 9538.9 nanoseconds
```

Figure 1.10: Ordered data search for BST

```
Average time for red-black tree search: 5634.26 nanoseconds
```

Figure 1.11: Ordered data search for RBT

From figures 1.10 and 1.11, we can see that data being ordered did not effect anything as trees are still being ordered by publisher names not game names

- For the BST, $O(n)$.
- For the RBT, $O(log(n))$.