Data Structure Project1

Chinese-English Dictionary based on Binary Search Tree

吴嘉骜 21307130203

2023 年 10 月 31 日

Abstract

This report presents the implementation of an English-Chinese dictionary using advanced tree data structures: Red-Black trees and B-trees. We first introduce the significance and characteristics of these structures, setting the context for their application. In the Implementation section, we provide a description on structures and procedures involved in the manipulation of Red-Black trees and B-trees. We then move on to experiment and list the time cost of initializing, deletion and insertion operations from files, together with search operations. Additionally, we analysis the results and compare the two data structures. A user-friendly interface is offered to ensure that end-users can easily navigate and utilize the system to its full potential. Finally, we discuss the limitations encountered during implementation and propose possible directions for future enhancements and research.

Project Objective

The objective of this project is to delve deeper into the understanding of advanced tree data structures, particularly focusing on Red-Black trees and B-trees, by implementing a Chinese-English dictionary.

Experiment environment

Windows 11 VsCode Python 3.11.5 64-bit

1 Introduction

Data structures play a crucial role in computer science, significantly impacting the efficiency and performance of algorithms and applications. Among these, tree structures such as **Red-Black trees** and **B-trees** stand out due to their ability to maintain balanced states, ensuring optimized operations even in the worst-case scenarios. It is of importance to understand the theoretical foundations of these structures, as well as their practical applications, in order to harness their full potential.

1.1 Red-Black Trees

Red-Black trees are a type of self-balancing binary search tree, where every node contains an extra bit for denoting the color of the node, either red or black. This color coding is vital in ensuring a balanced structure, which in turn guarantees the tree's operations—such as search, insert, and delete—have a time

complexity of $O(logn)$. The Red-Black tree maintains its balance by following specific properties and performing color changes and rotations during insertions and deletions.

1.2 B-Trees

B-trees, on the other hand, are balanced tree data structures optimized for systems that read and write large blocks of data, such as databases and file systems. They are particularly well-suited for storage systems since they can efficiently handle large amounts of data while maintaining a balanced structure. B-trees ensure all leaves are at the same level, and each node contains a certain number of keys sorted in a particular order. This characteristic guarantees fast search, insert, and delete operations, which are all performed in $O(logn)$ time.

1.3 Project Task

In this project, we focus on implementing an English-Chinese dictionary using Red-Black trees and B-trees respectively. The tasks involve initializing both trees with specific input files, performing deletions and insertions, and ensuring correct search operations. Additionally, the project requires printing the tree in a preorder traversal to a text file, ensuring a clear and accurate representation of the tree‘s structure. It is renewcommanded to devise a user-friendly interface to allow end-users to interact with the dictionary.

2 Implementation

Most of the algorithms refer to *Introduction to Algorithms* [1], which are omitted here because they are too verbose and can be found in the textbook. We will describe the structure of the main two `Python` scripts, as well as some noteworthy details in programming.

2.1 Red-Black Trees

The implementation of Red-Black trees is in `rb_tree.py`.

Structure

First, we define an `RBNode` class to represent a node in the Red-Black tree. Its attributes and typical functions are as follows:

`RED` and `BLACK` constants represent the color of the node.
`__init__`: Initializes a new node with *key, value, color, left child, right child*, and*parent*.
`is_red`: Returns `True` if the node's color is red.
`is_black`: Returns `True` if the node's color is black.
`set_red`: Sets the node's color to red.
`set_black`: Sets the node's color to black.
`__str__`: Provides a string representation of the node.

Then we come to the `RedBlackTree` class, which represents a Red-Black tree. Its attributes and main functions are described as follows, where we omit the `self` parameter for brevity:
`__init__`: Initializes an empty Red-Black Tree with a sentinel `nil` node, which is black with arbitrary key and value, representing a NULL leaf node.

Search functions:
`search(x, key)`: Searches for a node with the given key in the tree rooted at `x`.
`minrb(x)`: Finds the minimum node in the tree rooted at `x`.
`maxrb(x)`: Finds the maximum node in the tree rooted at `x`.
`successor(x)`: Finds the successor of node `x`.
`predecessor(x)`: Finds the predecessor of node `x`.
`rangesearch(low,high)`: Search for words in the specified range [low, high].
`singlesearch(en)`: Search for an English word in the tree.

Rotations:
These two auxiliary functions are used to maintain the Red-Black Tree properties after insertions and deletions.
`_left_rotate(x)`: Performs a left rotation on the subtree rooted at `x`.
`_right_rotate(y)`: Performs a right rotation on the subtree rooted at `y`.

Insertion functions:
The main insertion function is `insertrb(z)`, and the auxiliary function `_insert_fixup(z)` is used after a node is inserted to fix up the possible violations of the Red-Black Tree properties.
`insertrb(z)`: Inserts a new node `z` into the tree.
`_insert_fixup(z)`: Fixes the Red-Black Tree properties after an insertion.
`insert_word(en,cn)`: Insert an English word and its Chinese translation in the Red-Black Tree.

Deletion functions:
The main deletion function is `deleterb(z)`, the auxiliary function `_transplant(u, v)` is for a node replacement, and `_delete_fixup(x)` are used after a node is deleted to fix up the properties.
`deleterb(z)`: Deletes the node `z` from the tree.
`_transplant(u,v)`: Replaces the subtree rooted at node `u` with the subtree rooted at node `v`.
`_delete_fixup(x)`: Fixes the Red-Black Tree properties after node `x` is actually deleted from the tree.
`delete_word(en)`: Delete an English word and its Chinese translation from the Red-Black Tree.

Printing and file operations:
`preorder_print`: Preorder prints the Red-Black Tree to a file or the console, with a specific format.
`initialize`: Initializes the Red-Black Tree with data from a file.
`batch_op`: Performs batch operations of insertion/deletion from text files on the Red-Black Tree.

Details

To ensure that there are no duplicate insertions, we examine the return values of the functions related to insertion. During each recursive search for the position of a key (i.e., the subtree where it should be placed), we check whether the value equals that of an existing node. If a match is found, a value of `False` is returned, which propagates up through the recursion stack. This mechanism ensures that a single downward search suffices to either insert the key or confirm that insertion is not feasible.
For deletion, avoiding the removal of non-existent nodes is somewhat more complex compared to insertion. Our approach entails an initial search to confirm the presence of the node. If the node is found, deletion proceeds; otherwise, it is aborted. While this procedure incurs an additional cost due to the extra search, it allows us to circumvent the complexities of deletion when the node is confirmed to be absent.

2.2 B-Trees

The implementation of B-trees is in `b_tree.py`.

Structure

First, we define a `BTNode` class to represent a node in the B-tree. Its attributes and typical functions are as follows:

`__init__`: Initializes a new node with *n, keys, values, isleaf,* and *c*.
`n`: Represents the number of keys currently stored in the node.
`keys`: A list of keys (English words) stored in the node, arranged in ascending order.
`values`: A list of values (Chinese translations) stored in the node, corresponding to the keys.
`isleaf`: A boolean value indicating whether the node is a leaf. The default value is True.
`c`: A list of pointers to child nodes.
`is_full`: Checks if the node is full based on the order of the B-tree and returns a boolean value.

Then we come to the `BTree` class, which represents a B-tree. Its attributes and main functions are described as follows, where we omit the `self` parameter for brevity:
`__init__`: Initializes a B-tree with a specified order, `t`, and creates an empty root node.

Search functions:
`search(key, x)`: Searches for a node with the given key in the tree rooted at node `x`.
`singlesearch(en)`: Searches for a given English word in the tree and returns the Chinese translation if found.
`rangesearch(low,high)`: Searches for words in a specified range `[low, high]` and returns a list of tuples containing the words and their meanings.

Insertion functions:
The main insertion function is `insertb(z)`, and `_insert_nonfull(x, k, v)` and `_split_child(x, i)` are auxiliary functions.

`insertb(key, value)`: Inserts a given key and its value into the B-tree.
`_insert_nonfull(x, key, value)`: Inserts a given key and its value into a non-full node.
`_split_child(x)`: Splits a full child node of a given node.
`_split_root`: Splits the root of the tree if it becomes full post-insertion.
`insert_word(en, cn)`: Inserts a given English word and its Chinese translation into the B-tree.

Deletion functions:
The main deletion function is `deleteb(z)`, the rest are auxiliary functions except `deleteword`.
`deleteb(key,x)`: Deletes a given key from the tree rooted at node `x`.
`_delete_internal_node(x,key)`: Deletes a given key from an internal node.
`_delete_predecessor(x)`: Deletes the predecessor of a given key.
`_delete_successor(x)`: Deletes the successor of a given key.
`_delete_merge(x,i,j)`: Merges a key and one of its two neighboring child nodes.
`_delete_sibling(x,i,j)`: Borrows a key from a sibling node.
`delete_word(en)`: Deletes a given English word from the B-tree.

Printing and file operations:
`preorder_print`: Preorder prints the B-tree to a file or the console.
`initialize`: Initializes the B-tree with data from a specified file.
`batch_op`: Performs batch operations of insertion/deletion from a specified text file on the B-tree.

Details

The strategies to avoid duplicate insertions and the deletion of non-existent nodes in B-tree (BT) are consistent with those employed in the Red-Black Tree (RBT). Just like in RBT, before an insertion or deletion operation, a search is performed to check the existence or absence of the node in question, thereby ensuring that only valid nodes are inserted or deleted from the tree, minimizing the chances of erroneous operations.
As the pseudocode for deletion is not available in the textbook, I struggle to implement it with efforts to ensure that the tree remains balanced after the operation. There are some details that need to be paid attention to, as I got confused for some moment during my own work:
First and foremost, we have to guarantee that the node on which a resursion is performed has at least t keys, which is 1 plus the minimum number of keys in a non-root node. When searching down the tree, we have to take care of the child node with underfull keys, and we have to merge or borrow keys from its sibling node if necessary.
For Case2a and Case2b, we need to find the predecessor or successor in the subtree, which is the leftmost or rightmost node of it. The point is that once we have chosen a target, we have to pass down in a fixed direction, and no longer consider again finding predecessors or successors. The resursion also requires that the node on which it is performed have at least t keys, so a merge or borrow operation may also be needed.
For Case2c and Case3b, we are encountered with merge or borrow operations, making indices out of

range a common problem. One special situation is that if parent node is the root, it could end up having no keys. In this case, we have to delete the root and set its child node as the new root, and change the resursion accordingly.

3 Results and Analysis

We test our codes both on single operations and batch operations, according to the requirements of the project.

1. Insert into trees the data in the file 1_initial.txt. The results are saved to rbt_1.txt and bt_1.txt.
2. Delete the data in the file 2_delete.txt. The results are saved to rbt_2.txt and bt_2.txt.
3. Add the data in the file 3_insert.txt. The results are saved to rbt_3.txt and bt_3.txt.
4. Query a word.
5. Query some words.

For the first three steps, the whole time cost results are saved to timecost.txt, and we only show the average time cost of each operation in the following table.

Table 1: Average time for different operations in RBT and BT (10^{-5}s)

	Initialization	**Deletion**	**Insertion**
RBT	5.437	9.954	9.623
BT	4.527	9.847	9.153

We plot the result in a bar chart, as shown in Figure 1.

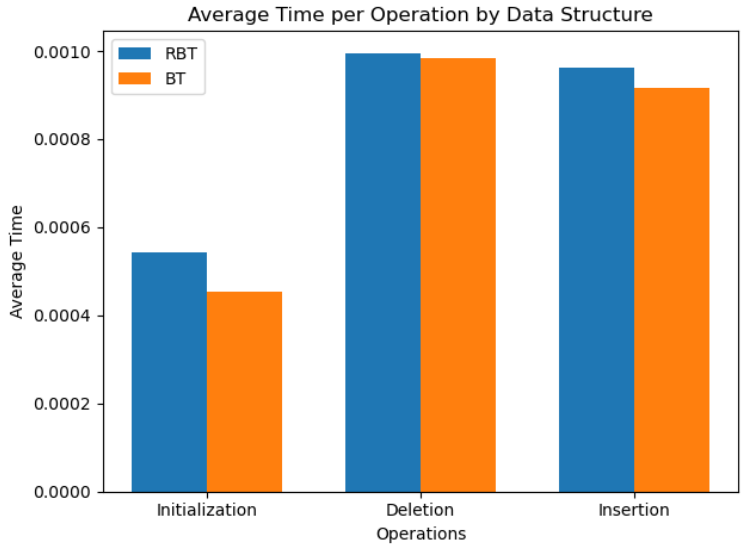

Figure 1: Average time for different operations in RBT and BT (10^{-5}s)

For the last two steps, the results are undoubtedly the same. We exhibit a query example here.

```
    # Single word query
    meaning = bt.singlesearch('cybernetic')
    print(f'{meaning}')

    # Output
    控制论的

    # Range query
    result = bt.rangesearch('cuculliform', 'culmination')
    for word, meaning in result:
        print(f'{word}: {meaning}')

    # Output
    cuculliform: 兜帽状的
    cucumiform: 黄瓜状的
    cuddle: 拥抱
    cuffy: 黑人
    culicid: 蚊科的
    culicine: 库蚊的
    cull: 精选
    cullet: 碎玻璃
    culminate: 达到顶点
    culmination: 顶点
```

Analysis:

From the time record, we can see that both trees present a very small time requirement for the three operations, which is a good indication of efficiency. The times are very close to each other, probably due to our limited data size.

However, the B-tree consistently shows lesser time across all operations, probably because the branching factor of the B-tree contributes to its efficiency. A larger branching factor can lead to a more shallow tree, potentially reducing the number of steps required to perform insertions, deletions, and searches.

Furthermore, the B-tree's structure can lead to faster disk access times when dealing with disk storage, even though in our case the operations were performed in memory.

In conclusion, while both data structures show strong performance and are well-suited for different use cases, the B-tree demonstrated a slight edge in terms of efficiency in our tests. However, if the difference in time is negligible, then the choice between RBT and BT might come down to other factors such as ease of implementation, memory usage, or specific use case requirements.

4 Interface and Usage

The interface picture is shown in Figure 2.

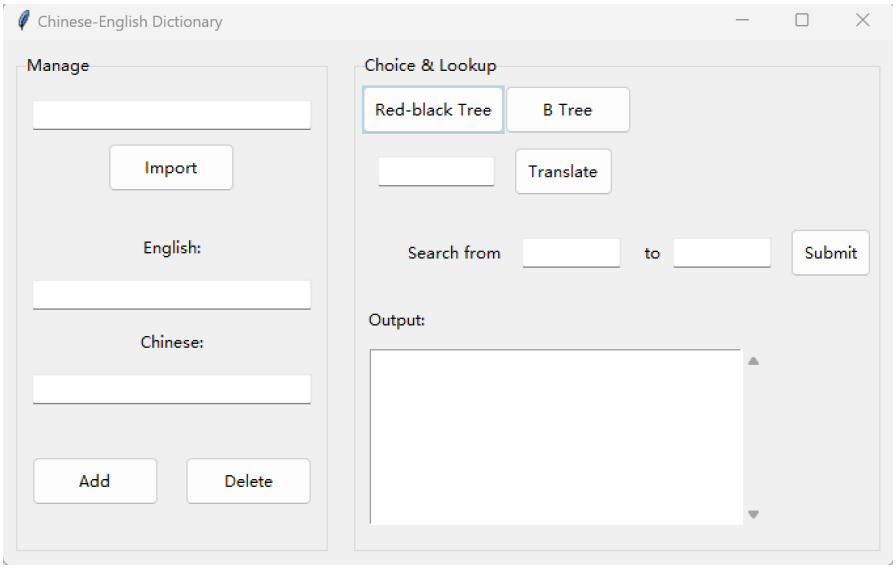

Figure 2: Interface

As is shown in the picture, the interface is quite simple and easy to use.
The left part is designed for a user to import a file to initialize the dictionary, or to add or delete a single word from the dictionary. For insertion, both English and Chinese words are needed; for deletion, only the English word is needed.
The right-top part is a button to choose which kind of tree to use, and the right-bottom part is for look-up. For a single word query, it gives the corresponding Chinese translation if the word is in the dictionary; for a range query, it gives a list of words and their meanings in the specified range.

5 Discussion

5.1 Limitations

The current implementation of the English-Chinese dictionary has several limitations that could be addressed in future work:
1. **Disk Utilization:** Despite the B-tree being a data structure that is well-suited for disk storage, in the current implementation, operations are carried out in memory, which does not leverage the potential for efficient disk access and storage. The inherent design of B-trees to minimize disk I/O operations has not been fully utilized, thus possibly affecting the performance when handling large datasets.
2. **User Interface Design:** The User Interface (UI) design is quite plain, which could affect the ease of use and the overall user experience.
3. **Code Reusability:** The code is not highly reusable. While it serves the purpose of implementing an

English-Chinese dictionary efficiently, adapting the code for other applications or data types may require significant modifications.

5.2 Future Work

Going forward, there are several improvements that can be considered to enhance the functionality and applicability of the project:
1. **Satellite Data Storage:** The application can be extended to handle satellite data storage, which often requires efficient indexing and retrieval mechanisms. The structure of B-trees could be utilized to build robust indexing systems to manage and query satellite data effectively.
2. **Utilizing B+ Trees:** Transitioning to a B+ tree structure could be explored for future implementations. B+ trees have some advantages over B-trees, including more efficient range querying and better utilization of disk space due to the linked leaves.
3. **Disk-Based Operations:** Modifying the implementation to perform disk-based operations could significantly improve the performance of the application, especially when dealing with large datasets. This would align with the inherent design of B-trees for disk storage and access.
4. **Enhanced User Interface:** Improving the User Interface to be more intuitive and user-friendly can enhance the usability of the application.
In conclusion, we have implemented main functions of Red-Black trees and B-trees successfully despite some struggles in the process. We applied these two data structures to build an English-Chinese dictionary, and we have also provided an interface for users to use the dictionary. The experiment result

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2022). *Introduction to Algorithms* (4th ed.). The MIT Press.