Product Requirements Document

“Self-balancing binary search tree”

Maria Apitonian, Mariam Mkrtichyan

Version 3.1
Last modified: 14 December 2021
Number of pages: 7

Table of Contents

1. Introduction

1.1. Document Identifier

The following document is a project requirement document (pdr) for a project on balancing binary search trees. The document outlines the specifications and features of the project, as well as various requirements for ensuring completion of the project's goals. The working members on the project are Maria Apitonian and Mariam Mkrtichyan.

1.2. Scope

The introduction section provides a brief overview of the project's main goals and features, including a definition of relevant terms (1.3), references used for further information on the concepts used in the project (1.4), and an overview of what the project entails (1.5).

The requirements section goes over various hardware, software, and other requirements. This includes functional requirements that help determine if the project is complete (2.1), platform requirements that describe the required hardware (2.2), implementation requirements that go over required software elements (2.3), performance requirements such as time complexity (2.4), verification requirements that will be evaluated throughout the testing process (2.5), and documentation requirements that relate to user documentation (2.6).

Finally, the project management section goes over the managerial requirements and the process of completing the project. The section covers areas such as possible risks (3.1), schedule of tasks (3.2), and criteria that define the project as complete (3.3).

1.3. Definitions of terms and acronyms

Tree - A hierarchical data structure with a set of linked nodes. Trees have a starting value (root) and subtrees of children with linked nodes to their parents. Each parent node has a greater key than the child node below.

Subtree - For branching trees, each node is the root of its own tree. A smaller tree where the root node is itself the child of some other node is called a subtree.

Binary Search Tree (BST) - In computer science, a binary search tree is a data structure with two subtrees. The left subtree consists of nodes with lesser keys, while the right subtree consists of nodes with greater keys.

Degree - The degree of a node is the number of children it has.

Root - The starting value of a tree, which has no parent values.

Leaf - A terminal node of the binary search tree. Any node that has no children is considered a leaf node. In other words, a leaf is a node with degree 0.

Height/Level - The height or level of a node in a tree is equal to its distance from a leaf node. The height of the tree is the height of its root node.

Depth - The depth of a node in a binary tree is the total number of edges from the root node to the target node.(same as Height of the node)

Ancestor/Predecessor - A node A is considered the ancestor or predecessor of a node B if it is higher level than B and connected to it.

Descendant/Successor - A node A is considered the descendant or successor of a node B if it is lower level than B and connected to it.

Sibling - A node A is considered the sibling of another node B iff A and B are on the same level. Sibling nodes are not connected.

Tree traversal **-** is the process of visiting each node in the tree exactly once. Visiting each node in a graph should be done in a systematic manner (inorder, postorder, preorder).

BST path - is a collection of nodes from the root to any leaf of the tree.

1.4. References

Anderson-Freed, S, E. Horowitz and S. Sahni (2007). *Fundamentals of Data Structures in C*. Computer Science Press.

Kataria, A. (2018, June 11). Traversal technique for Binary Tree. Retrieved from www.includehelp.com/data-structure-tutorial/traversal-technique-for-binary-tree.aspx.

Srivastava, A. K. (2019). *A Practical Approach to Data Structure and Algorithm with Programming in C* (pp 337-392). Retrieved from EBSCO Publishing.

SPARTA, T. E. G., Gerasch, T. E., Sparta, Center, N. A. S. A. A. R., & Metrics, O. M. V. A. (1988, May 1).*An insertion algorithm for a minimal internal path length binary search tree*. Communications of the ACM. Retrieved December 3, 2021, from https://dlnext.acm.org/doi/10.1145/42411.42421**.**

1.5. Overview

The project involves building a self-balancing binary search tree using sorting methods.

The main goal of a self-balancing binary search tree is to automatically minimize its height (which is the number of levels in the search tree) upon insertions and deletions. This way, the structure can self-balance and avoid reaching heights that are larger than they need to be.

The project will involve building a structure for binary trees from scratch. Insertions and deletions will be done with a rebalancing feature at the end. There will be functions for ensuring the self-balanced feature, finding the minimum and maximum nodes, calculating the tree height, and more.

Hence, the primary required features of the project is to create a self-balancing binary search tree that can deal with insertions and deletions while maintaining minimal time complexity.

2. Requirements

2.1. Functional Requirements

- Insertion and deletion functions that insert or remove elements from the binary search tree, calling the self-balancing function at the end. An extra feature is added to the insert function to ensure uniqueness of values (quits function with a message if you attempt to insert an existing value), and to the delete function to ensure consideration of all cases (quits function with a message if the user attempts to delete values that do not exist).
  - Complexity: O(height)=O($\log_2(n)$)
- A search function that is able to locate elements in large BSTs at a much lower time complexity than for an array and will return the path of the searched element. There is also a boolean version of this function with constant time complexity which simply returns a statement on whether or not the value exists in the tree.
  - Maximum complexity: O($\log_2(n)$) in the case where the whole height is travelled
- A self-balancing feature, where the BST balances itself to maintain minimum height upon insertions and deletions. Includes left and right single and double rotation functions, a function that checks if a left or right rotation is required (rebalance).
  - Rebalance helper: Complexity: constant
  - Right single rotation: Complexity: maximum O($\log_2(n)$)
  - Right double rotation: Complexity: maximum O($\log_2(n)$)
  - Left single rotation: Complexity: maximum O($\log_2(n)$)
  - Left double rotation:Complexity: maximum O($\log_2(n)$)
- Create function that allocates memory to a tree/node and passes the characteristics listed in its structure
  - Complexity: constant, only memory allocation
- Minimum(tree) and Maximum(tree) functions that find the minimum and maximum keys
  - Complexity: O($\log_2(n)$), whole height travelled to reach leafs

- Height, Isempty functions that determine the height of the tree and whether or not it's empty
    - Complexity: $O(\log_2(n))$ for Height, constant for Isempty
- A find parent function that returns the parent node for any element in the tree
    - Complexity: maximum $O(\log_2(n))$ in case the element is a leaf
- A tree traversal function that goes over every node in the tree in a systematic manner. In this project, inorder traversal will be used
    - Complexity: O(n)

2.2. Platform Requirements

- 16 GB RAM
- Intel Core i7-8565U 1.99 GHz CPU clock or equivalent
- 20 GB free HDD space
- Supported OS:
    - Ubuntu 20.04 (64 bit)
    - MacOS 12.01 (64 bit)

2.3. Implementation Requirements

Since the binary tree structure will be implemented through C structs, libraries for trees will not be used. Some C libraries such as <stdio.h>, <stdlib.h> and <assert.h> will be used for functions like printf, assertions and more.

The programming language for the implementation is C.

2.4. Performance Requirements

The time complexity of the search function of the balanced BST should be at most O(log(N)), and it should be demonstrably lower than that of an unbalanced BST or an array. The time complexity of other functions executed on the BST should be at most linear, i.e. O(N). The exact numbers of the time taken will vary based on the hardware used.

2.5. Verification Requirements

The initial test plan of the code is to use it to run and sort a database of student IDs, then search certain values within this tree. The speed of the searching process will be compared to that of an array to prove the efficiency of the built binary search tree.

For the testing stage of the project, some sort of database of students will be required, which will be made into a balanced search tree. Insertions and deletions will be tested to make sure that the tree is able to self-balance. Subsequently, search speeds will be tracked for various elements to determine some sort of average speed, which will then be compared for verification of the program's efficiency.

2.6. Documentation Requirements

Comments should be included in the code to improve readability by clarifying the logic and structure of the code. The README file would be attached to the code in Github, and guidance on how to use the program. The final results will be presented to the user in an html file on github.

3. Project Management

3.1. Dependencies, Assumptions, Risks

Not applicable

3.2. Schedule and Effort Estimations

Rough schedule of tasks

First week:

- ➢ Creating a struct for a binary search tree and a node.
- ➢ Writing a create function for the binary tree.
- ➢ Writing an optimized search function.

Second week :

- ➢ Write minimum(tree) and maximum(tree) functions.
- ➢ Write codes for the insertion and deletion functions.
- ➢ Start the balancing function.

Third week:

- ➢ Complete the function for the self-balancing feature.
- ➢ Revisit the optimized search function.
- ➢ Test the self-balancing feature.

3.3. Acceptance Criteria

The maximum time complexity for any function within the code is O(N).

The maximum complexity for the search function is $O(\log_2(N))$

The structure is efficient in searches relative to a non balanced tree or other covered structures like arrays and linked lists.

The structure can handle large amounts of data.