VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

REPORT PROJECT
OPTIMAL BINARY SEARCH TREE
GROUP 16

Lecture:
Nguyen Thanh Phuong
Instructor:
Nguyen Thanh Tinh
Nguyen Ngoc Thao
Member:
24127004 - Ngo Tien Binh
24127242 - Nguyen Hoang Phuc Thinh
24127244 - Pham Tan Nhat Thinh

Table of contents

Acknowledgements

This project, “Optimal Binary Search Tree”, was completed with the invaluable guidance of Teacher Nguyen Thanh Tinh, Teacher Nguyen Ngoc Thao. Their mentorship, encouragement, and unwavering support have been essential in helping us navigate challenges and achieve our goals. We are deeply grateful for their patience, expertise and dedicated assistance throughout this process.
The success of this project owes much to more than just our own efforts. Our classmates have played a significant role by offering constant support and motivation during our academic journey and the development of this project.
Throughout the project, we had the opportunity to gain more knowledge about the OBST algorithm, from theorem basis, complexity analysis, dynamic programming approach to comparison of Binary Search Tree ‘s variants such as Splay Tree, Treap and Tango Tree. Thereby, we not only completed the code, slide and presentation video but also improved our teamwork skills, programming and handling study report.
While we have strived to produce our best work, we recognize that there may still be areas for improvement. We sincerely thank all the professors, mentors, and peers who have contributed their time and insights to assist us in this endeavor.

I. Overview

1. Group and member information

NO	Student ID	Full name	Gmail	Task
1	24127004	Ngô Tiến Bình	ntbinh2419@clc.fitus.edu.vn	Programmer
2	24127242	Nguyễn Hoàng Phúc Thịnh	nhpthinh2416@clc.fitus.edu.vn	Report Writer
3	24127244	Phạm Tấn Nhật Thịnh	ptnthinh2410@clc.fitus.edu.vn	Project Manager

2. Job description

Programmer:

- Implement Optimal Binary Search Algorithm

Report Writer:

- Write full report

Product Manager:

- Testing
- Design Slide

3. Timeline

Week 7: Researching Algorithms
Week 8 - 9: Coding and Report Writing
Week 10: Slide Designing and Video Recording

II. Report Content

1. Introduction

In many applications, especially in database systems and information retrieval, efficient searching is of paramount importance. Binary Search Tree is a common data structure used in computer science for organizing and storing data in a sorted manner, which can maintain a dynamic set of items with efficient search, insert, and delete operations. The Optimal Binary Search Tree (OBST) was born to address the problem of bad performance due to cases where keys have different access frequencies by constructing a BST that minimizes the expected search cost based on the given frequency of each key.

2. Problem Definition

Given:

- A sorted sequence of keys $K = \{k_1, k_2, ..., k_n\}$
- Their corresponding each frequencies $F = \{f_1, f_2, ..., f_n\}$, where f_i is the number of searches of k_i.

The goal is to construct a BST such that its cost is the most minimal. The cost of a BST is defined as the sum of the frequencies of keys multiplied by their levels in the tree.

$$Cost = \sum_{i=1}^{n} f_i \times d(k_i)$$

[$d(k_i)$ represents the depth of key k_i in the tree]

The problem becomes one of selecting a root and recursively building optimal left and right subtrees to ensure that the sum of these weighted costs is minimized.

* Note: There are $\frac{(2n)!}{(n+1)!n!}$ for $n = sizeof(K)$ different Binary Search Trees can be created!

3. Methodology

3.1. Recursive Approach

To find the optimal cost of arranging nodes from i to j (called $optCost(i, j)$), take the sum of their frequencies ($fsum$), then add the smallest cost of picking any node r as the root. That cost is $optCost(i, r-1)$ (left side) + $optCost(r+1, j)$ (right side), testing all r from i to j.

Base cases:

- If $i > j$, cost is 0 (no nodes).
- If $i = j$, cost is just f_i (one node).

It's like trying every split and picking the cheapest, plus the total frequency!

```cpp
int optCost(vector<int> &freq, int i, int j) {

    // Base case: no elements in this subarray
    if (j < i)
      return 0;

    // Base case: one element in this subarray
    if (j == i)
        return freq[i];

    // Get sum of freq[i], freq[i+1], ... freq[j]
    int fsum = prefixSum[j] - ((!i) ? prefixSum[i - 1] : 0);

    // Initialize minimum value
    int min = INT_MAX;

    // One by one consider all elements
    // as root and recursively find cost
    // of the BST, compare the cost with
    // min and update min if needed
    for (int r = i; r <= j; ++r) {
        int cost = optCost(freq, i, r - 1) + optCost(freq, r + 1, j);
        if (cost < min)
            min = cost;
    }

    // Return minimum value
    return min + fsum;
}
```

3.1.1. Breakdown

Function Purpose:

$optCost(i, j)$ computes the minimum cost to construct an optimal binary search tree for the keys with indices from i to j, given their frequency values.

Base Cases:

- No Nodes ($i > j$):
  If there are no nodes in the current subarray, there is nothing to arrange, so the cost is 0.
- Single Node ($i = j$):
  When there's only one key, the optimal cost is simply its frequency. Return: f_i.

Computing the Frequency Sum ($fsum$):

For a subarray from index i to j, calculate the sum of frequencies using a precomputed $prefixSum$ array. This sum ($fsum$) represents the total cost added by accessing each key during every search in this subtree.

Recursive Case (When There Are Multiple Nodes):

- Loop Over Possible Roots:
  Iterate over each index r from i to j considering it as the potential root of the current subtree.
- Cost Calculation for Each Choice of Root:
  Left Subtree: $optCost(i, r-1)$ calculates the cost for the left segment of keys.
  Right Subtree: $optCost(r+1, j)$ calculates the cost for the right segment of keys.
  Combined Cost: For each potential root r, the total cost is:

$$cost = optCost(i, r-1) + optCost(r+1, j)$$

Adding the Frequency Sum:

Every access in this subtree also involves a search through all keys in the current segment, so you add $fsum$ to the combined cost.

Determine Minimum Cost:

As you iterate over each possible root, keep track of the minimum total cost found.

Return the Result:

After testing all possible roots for the subarray from i to j, return the minimum cost plus the frequency sum:

$$optCost(i,j) = min_{r\in[i,j]}(optCost(i,r-1) + optCost(r+1,j)) + fsum$$

3.1.2. Complexity Analysis

Each recursive call splits the problem into two smaller subproblems.
The recurrence relation follows:

$$T(n) = \sum_{r=1}^{n}(T(r-1) + T(n-r)) + O(n)$$

Since there are $O(n)$ choices for the root and each choice leads to two recursive calls, the time complexity is **exponential**: $O(2^n)$
This makes the recursive approach impractical for large n due to redundant calculations.

3.2. Dynamic Programming Approach

We're using a $dp[n][n]$ array to store subproblem solutions, aiming for $dp[0][n-1]$ as the final answer. The trick is filling it diagonally: start with $dp[i][i]$ (length 1), then $dp[i][i+1]$ (length 2), $dp[i][i+2]$ (length 3), and so on, like in Matrix Chain Multiplication. We loop over chain length l from 1 to n, set i from 0 to $n-1$, and calculate $j = i + l - 1$. For each $dp[i][j]$, we'd use smaller subproblems already

filled, say, splitting at some r between i and j. It builds up naturally, and $dp[0][n-1]$ pops out at the end. Simple, yet clever!

```
int optimalSearchTree(vector<int> &keys, vector<int> &freq) {

    int n = keys.size();

    // Create an auxiliary 2D matrix to store
     // results of subproblems
    vector<vector<int>> dp(n, vector<int>(n, 0));

    // dp[i][j] = Optimal cost of binary search tree
    // that can be formed from keys[i] to keys[j].
    // dp[0][n-1] will store the resultant dp

    // For a single key, dp is equal to frequency of the key
    for (int i = 0; i < n; i++) {
        dp[i][i] = freq[i];
    }

    // Now we need to consider chains of length 2, 3, ... .
    // l is chain length.
    for (int l = 2; l <= n; l++) {

        // i is row number in dp[][]
        for (int i = 0; i <= n - l; i++) {

            // Get column number j from row number i
               // and chain length l
            int j = i + l - 1;
            dp[i][j] = INT_MAX;
            int fsum = prefixSum[j] - ((!i) ? prefixSum[i - 1] : 0);

            // Try making all keys in interval keys[i..j] as root
            for (int r = i; r <= j; r++) {

                // c = dp when keys[r] becomes root of this subtree
                int c = ((r > i) ? dp[i][r - 1] : 0) +
                        ((r < j) ? dp[r + 1][j] : 0) + fsum;
                if (c < dp[i][j])
                    dp[i][j] = c;
            }
        }
    }

    return dp[0][n - 1];
}
```

3.2.1. Breakdown

- Setting Up the DP Table
  First, we need a place to keep track of our solutions, so we create a 2D table called dp. Picture it as a grid with n rows and n columns, where n is the number of keys we're working with. Each cell $dp[i][j]$ will tell us the minimum cost to build an optimal BST using the keys from index i to index j Our big goal? Fill out this table so that $dp[0][n-1]$ gives us the minimum cost for all the keys.
- Starting Simple: The Base Case
  Let's begin with the easiest case: when we're dealing with just one key. If the starting index i equals the ending index j, there's no tree to build, it's just that single key sitting there as the root. The cost here is simply how often that key gets searched, which we call $freq[i]$. So, for every i from 0 to $n-1$, we set:

```
dp[i][i] = freq[i];
```

- Growing the Solution: Tackling Bigger Chunks
  Now that we've got the single-key cases down, let's build up to bigger groups of keys. We'll do this by looking at subproblems of increasing size. Think of a variable l as the length of the key range we're considering, it starts at 2 (two keys) and goes all the way up to n (all the keys).

  For each length l, we pick a starting point i (from 0 up to $n-l$ so we don't run out of bounds), and figure out the ending point with:

```
int j = i + l - 1;
```

- Finding the Minimum Cost for Each Chunk
  For each $dp[i][j]$, we need to find the cheapest way to make a BST with the keys from i to j. Since it's a BST, one of those keys has to be the root, and the rest split into a left subtree and a right subtree. So, we'll try each key in that range as the root and see which one costs the least.

  Say we pick a key at index r (where r goes from i to j) as the root. Here's what happens:
  - Left Subtree: Covers keys from i to $r-1$. If r is i, there's nothing on the left, so the cost is 0. Otherwise, it's $dp[i][r-1]$, which we've already figured out.
  - Right Subtree: Covers keys from $r+1$ to j. If r is j, the right side's empty, so cost is 0. Otherwise, it's $dp[r+1][j]$.
  - Total Frequencies: On top of the subtree costs, every key in this BST (from i to j) gets searched, and their depths go up by 1 compared to their own subtrees because we added a root above them. So, we add the sum of all frequencies from i to j, written as sum($freq[i]$ to $freq[j]$).

Putting it together, the cost when r is the root is:

```
int c = ((r > i) ? dp[i][r - 1] : 0) +
        ((r < j) ? dp[r + 1][j] : 0) + fsum;
```

- Putting It All Together

  So, we start with our base cases, then loop over lengths l from 2 to. For each l, loop over i from 0 to $n - l$, set $j = i + l - 1$, and for each pair (i, j):

  1. Compute the sum of frequencies:
     $fsum = prefixSum[j] - prefixSum[i - 1]$.
  2. Try each r from i to j:
     - Left cost: 0 if $r == i$, else $dp[i][r - 1]$.
     - Right cost: 0 if $r == j$, else $dp[r + 1][j]$.
     - Total cost: Left + Right + $fsum$.

  Set $dp[i][j]$ to the minimum cost over all r.

  After all that, $dp[0][n - 1]$ holds the minimum cost for the whole set of keys.

- The Final Answer

  Once we've filled out the table, we just peek at $dp[0][n - 1]$. That's the smallest possible cost to build an optimal BST for all our keys, balancing their frequencies and depths perfectly.

3.2.2. Complexity Analysis

Outer Loop:

This runs $O(n)$ times because it covers a range proportional to the input size n.

Middle Loop:

For each iteration of the outer loop, this runs a number of times that depends on the subproblem size. On average, across all subproblem sizes, it iterates $O(n)$ times per outer loop iteration.

Inner Loop:

For each iteration of the middle loop, this runs a number of times proportional to the subproblem size, which can be as large as n. Thus, it contributes another $O(n)$ factor.

When we combine these:

The outer loop runs $O(n)$ times.

For each of those, the middle loop runs $O(n)$ times.

For each of those, the inner loop runs $O(n)$ times.

Multiplying these together gives:

$O(n) \times O(n) \times O(n) = O(n^3)$.

This cubic complexity comes from the fact that the three loops together explore a three-dimensional space of possibilities defined by the input size n.

3.2.3. Example of running algorithm

Find the optimal binary search tree for N = 3, with keys = {10, 20, 30} and freq = {34, 8, 50}

- Step 1: Initialize DP Table for Single Elements (i = j)
  DP[i][i] = freq[i];

DP	0	1	2
0	34	-	-
1	0	8	-
2	0	0	50

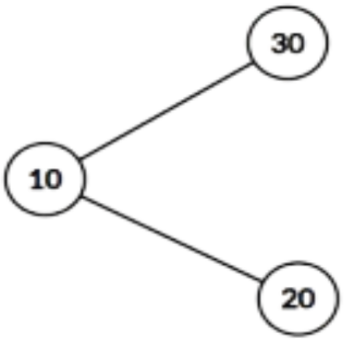
- Step 2: Compute cost for subarrays of length 2
  - Compute DP[0][1] (Subarray {10, 20})
    Root = 10: DP[0][1] = DP[1][1] + Fsum(0 - 1) = 8 + (34 + 8) = 50.
    Root = 20: DP[0][1] = DP[0][0] + Fsum(0 - 1) = 34 + (8 + 34) = 76.
    Choose 10 as the root because it gives the minimum cost: 50.
  - Compute DP[1][2] (Subarray {20, 30})
    Root = 20: DP[1][2] = DP[2][2] + Fsum(1 - 2) = 50 + (8 + 50) = 108
    Root = 30: DP[1][2] = DP[1][1] + Fsum(1 - 2) = 8 + (8 + 50) = 66
    Choose 30 as the root because it gives the minimum cost: 66.
    Update DP Table:

DP	0	1	2
0	34	50	-
1	0	8	66
2	0	0	50

- Step 3: Compute cost for subarrays of length 3
  - Compute DP[0][2] (Subarray {10, 30})
    Root = 10: DP[0][2] = DP[1][2] + Fsum(0 - 2) = 66 + (34 + 8 + 50) = 158.
    Root = 20: DP[0][2] = DP[0][0] + DP[2][2] + Fsum(0 - 2)
    = 34 + 50 + (34 + 8 + 50) = 176.
    Root = 30: DP[0][2] = DP[0][1] + Fsum(0 - 2) = 50 + (34 + 8 + 50) = 142.
    Choose 30 as the root because it gives the minimum cost: 142.
    Update DP Table:

DP	0	1	2
0	34	50	142
1	0	8	66
2	0	0	50

⇒ The optimal root of the entire tree is 30, leading to the lowest cost: 142.



(Tree's construction having the lowest cost)

3.2.4. Trace BST Tree

- To begin building OBST, we use a 2D array root[i][j] to record the optimal root for every key interval [i, j]. For base cases where the interval contains only one key, the root is the key itself. This is initialized as follows:

```
for (int i = 0; i < n; i++) {
     root[i][i] = i;
}
```

This step sets up the structure needed to apply dynamic programming in the next phase.

- The core of the OBST construction lies in minimizing the search cost. For each interval [i, j], we try every key r between i and j as the root. We compute the cost val for choosing r and compare it with the current minimum dp[i][j]. If val is smaller, we update dp[i][j] and record r as the optimal root:

```
for (int r = i; r <= j; r++) {
// c = dp when keys[r] becomes root of this subtree
     int c = ((r > i) ? dp[i][r - 1] : 0) +
             ((r < j) ? dp[r + 1][j] : 0) + fsum;
     if (c < dp[i][j]){
         // record the optimal root for this interval
         root[i][j] = r;
         dp[i][j] = c;
     }
```

- Once the root table is filled, we reconstruct the OBST using a recursive function. Starting from the full interval [0, n-1], we select the recorded optimal root r = root[i][j], create a new node, and recursively build its left and right subtrees:

```
Node* trace_tree(vector<vector<int>> root,
                 vector<int> keys, int i, int j) {
    if (i > j) return nullptr;
    int r = root[i][j];
    Node* new_node = new Node(keys[r]);
    new_node->left = trace_tree(root, keys, i, r - 1);
    new_node->right = trace_tree(root, keys, r + 1, j);
    return new_node;
}
```

This function yields the full binary search tree with the minimum expected search cost.

4. Conclusion

Optimal Binary Search Tree (OBST) is an effective data structure for reducing average search time. OBST outperforms standard binary search trees in terms of search efficiency because they strategically organize nodes based on their frequency of access. The dynamic programming technique used to build OBST takes advantage of their optimal substructure and overlapping subproblems, guaranteeing that the finished tree reduces the projected search cost.

While the creation procedure might be computationally demanding for big datasets, the performance advantages during retrieval operations frequently justify the original investment. OBSTs are extremely useful in applications such as compilers, database indexing, and memory hierarchies, where certain searches are more common than others.

5. Overview of Binary Search Tree Variants

Beyond the common variants covered in lectures, several other binary search tree (BST) variants optimize search, insertion, and deletion operations. Below is a brief overview of some notable BST variants:

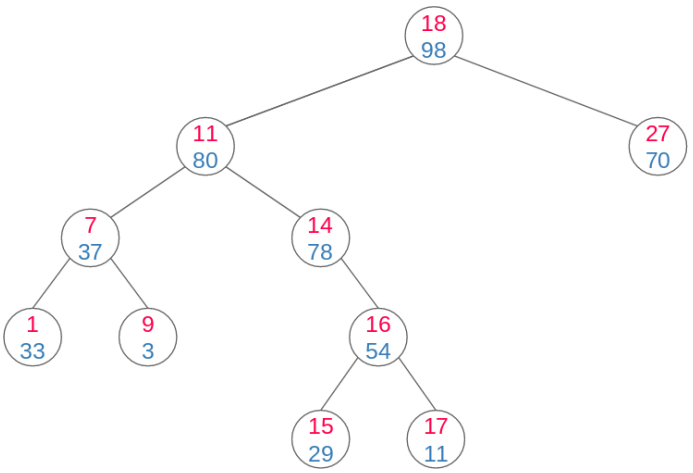
5.1. Splay tree

- A splay tree are the altered versions of BST that perform basic operations like inserting, searching and deleting nodes in the tree. Splay trees perform these basic operations in combination with splay operations. Splaying involves rearranging the tree to bring a certain element to the root node of the tree.
- Trees moves recently accessed nodes closer to the root for faster repeated access
- Uses three types of rotations : zig(single rotation), Zig - Zig(double same direction rotation), Zig - Zag (double opposite direction rotation)

- Performs operations in amortized $O(logn)$ time.
- Advantages:
  - Frequently accessed elements are moved closer to the root, making future access faster.
  - Splay trees do not require extra storage for balancing, so they use less memory.
- Disadvantages:
  - Performance may degrade with random access due to frequent rotations.
  - Does not guarantee O(log n) time per operation, only amortized efficiency.
  - Tree structure can become highly unbalanced after certain operations.

5.2. Treap

- Combines BST and heap properties by assigning random priority values to nodes.
- A treap is simply a binary tree with two keys, where one key satisfies the heap property, and the other key satisfies the search tree property.
- Performs operations in amortized $O(logn)$ time.
- Advantages:
  - Combines benefits of Binary Search Tree (BST) and Heap: maintains key order and randomized balance.
  - Simple structure, easy to implement without complex rebalancing.
- Disadvantages:
  - Requires an additional random priority for each node, which uses extra memory.
  - The random factor makes it hard to predict the exact structure of the tree after operations.



(An example of Treap's construction)

5.3. Tango Tree

- A Tango Tree is a self-adjusting binary search tree designed to optimize search queries based on previous query sequences. It operates using a Preferred Search Tree, which helps track the path of recent queries to adjust the tree structure accordingly.

- When a query is made, the tree records the optimal path (preferred path) to improve future accesses.
- Performs operations in amortized $O(loglogn)$ time.
- Advantages:
  - The tree maintains a balanced structure using rotations and height control, ensuring efficient insert, delete, and search operations.
  - It reorganizes itself automatically when data changes, making it ideal for applications with frequent updates and queries.
  - It can flexibly adjust to different situations, ensuring stable and efficient performance in dynamic environments.
- Disadvantages:
  - The continuous self-balancing process can consume additionally computational resources and memory, especially with frequent insertions and deletions

6. Practical Implementation

The Optimal Binary Search Tree is important in Design and Analysis of Algorithms (DAA) because it provides an effective approach to arrange search operations when search probabilities are known but not uniform. It is a famous example of dynamic programming, in which subproblems are addressed optimally in order to create an overall optimal solution. It aids in the resolution of real-world situations that require optimum searching.

- Compiler Design
  When you write code, a compiler turns it into something your computer can run. It's constantly looking up stuff like variable names in a "symbol table" OBST makes sure the ones you use all the time are quick to find, speeding up the whole process.
- Database Indexing
  Picture a giant database like one for an online store. Some products ("iPhone") get searched way more than others ("obscure gadget #36"). OBST organizes the index so those hot items are at the top, making queries zippy.
- Huffman Encoding
  Huffman encoding ZIP file, used in compression, is a cousin of OBST. It arranges data based on how often it's used to shrink file sizes efficiently. OBST does something similar for lookups.
- Spell Checkers and Auto-Completion
  When you're typing in a word processor or search bar, some words (like "the" or "cat") come up a lot more than others ("xylophone"). OBST helps the system suggest or check those common words super fast.

7. Self-evaluate

7.1. Overall Evaluation

In this project, we are responsible for installing the Optimal Binary Search Tree algorithm (OBST) and testing to ensure the correctness of the algorithm. In addition, I also contribute

to writing reports and creating presentation slides. The percent of our work is about 90%, because we have completed the programming and testing of algorithms on some datasets. However, during the implementation process, we encountered a number of challenges, especially when we understood the dynamic programming method used to build the optimal tree. To overcome this, we consulted more documents and thanks to the support from the group members to improve the understanding and complete the solution. We have actively participated in group discussions and supported members of the group with their duties. One of the challenges that we encountered in group work is the coordination among members when there are similar parts of work, but we believe we have solved this problem through delivery. Continuing and cooperating regularly. In terms of learning, we have gained valuable experience in dynamic programming, this is a new concept for us. This project also helps us improve the debugging skills and optimize algorithms. In the future, we think we can improve time management skills, because we have spent more time than expected for the algorithm installation. Overall, we believe that we have contributed positively to the success of the project and have learnt a lot throughout the process. We want to apply what we have learnt in future projects and continue to improve our problem solving skills.

7.2. Evaluation of Completion Rate

NO	Student ID	Full name	Task	Rate
1	24127004	Ngô Tiến Bình	Programmer	100%
2	24127242	Nguyễn Hoàng Phúc Thịnh	Report Writer	100%
3	24127244	Phạm Tấn Nhật Thịnh	Project Manager	100%

8. Reference

[1]https://www.geeksforgeeks.org/optimal-binary-search-tree-dp-24/, last access at 3:12PM in 31st March, 2025

[2]https://en.wikipedia.org/wiki/Optimal_binary_search_tree, last access at 2:07PM in 25th March, 2025

[3]https://www.rose-hulman.edu/class/cs/csse473/201540/MiscDocuments/OptimalBinarySearchTree.ppt, last access at 5:46PM in 2nd April, 2025

[4]https://blogs.asarkar.com/assets/docs/algorithms/Optimal%20BST%20-%20University%20of%20Craiova.pdf, last access at 1:39PM in 3rd April, 2025

[5]https://medium.com/carpanese/a-visual-introduction-to-treap-data-structure-part-1-6196d6cc12ee, last access at 05:11PM in 4th April, 2025

[6]https://herovired.com/learning-hub/blogs/optimal-binary-search-tree/, last access at 10:28PM in 4th April, 2025

[7]https://youtu.be/vLS-zRCHo-Y?si=zYEyriboi0HJ5PAM, last access at 03:25PM in 5th April, 2025