Slovak University of Technology in Bratislava

Faculty of Informatics and Information Technologies

SEARCH IN DYNAMIC SETS

Data Structures and Algorithms- Assignment 1

Abstract

Implementation and comparison of 4 different data structures in terms of effectiveness of insert, delete and search operation in different situations.

David Truhlar

xtruhlar@stuba.sk

ID: 120897

Assignment:

There are a number of algorithms designed to efficiently search for elements in dynamic sets: binary search trees, multiple approaches to balancing them, hash tables, and multiple approaches to resolving collisions. Different algorithms are suitable for different situations according to the nature of the processed data, distribution of values, performed operations, etc. In this task, your task is to implement and compare these approaches.

Your task in this assignment is to implement and then compare 4 implementations of data structures in terms of the effectiveness of **insert**, **delete** and **search** operations in different situations:

- (3 points) Implementation of a binary search tree (BVS) with any balancing algorithm, e.g. AVL, Red-Black Trees, (2,3) Trees, (2,3,4) Trees, Splay Trees
- (3 points) Second implementation of BVS with a different balancing algorithm than in the previous point.
- (3 points) Implementation of a hash table with collision resolution of your choice. The hash table size adjustment must also be implemented.
- (3 points) Second implementation of the hash table with collision resolution in a different way than in the previous point. The hash table size adjustment must also be implemented.

You can obtain a total of 12 points for the implementations themselves according to the points above. You should upload each implementation in one separate source file (if you want to upload all four, you upload them in four files). **It is not allowed to download foreign source code! You must implement at least two of the above implementations for a successful upload**. Verify correctness by testing - comparison with other implementations.

In the technical documentation, your task is to document all implemented data structures and provide detailed testing scenarios, based on which you have found out in which situations which of these implementations are more effective**. It is also required to submit a program that is used to test** and measure the effectiveness of these implementations as a single source file (it contains the main function). **Without a test program, and thus without a successful comparison of at least two implementations, the solution will be evaluated with 0 points**. You can get a maximum of 8 points for documentation, testing and achieved results (identifying suitable test scenarios). The quality of testing and processing of results in the documentation is mainly evaluated. Tables and graphs with results that compare the performance (speed) of individual solutions should be included as well. As the results depend not only on the implementation of the solution, but also on the test scenarios (sequence of insert, delete and search operations), it is important to try and document various different test scenarios.

You can obtain a total of 20 points at most. The minimum requirement is 8 points.

The solution of the assignment must be uploaded in AIS by the set deadline (delays are allowed only in very serious cases, such as sickness, the decision whether the solution is accepted after deadline is decided by the instructor). One zip archive should be uploaded, which contains individual source code files with implementations, a test file and one documentation file in **pdf** format.

Search in dynamic sets

1. PART 1: Implementation of AVL Trees and Splay Trees on Binary Search Tree

1.1 Binary Search Tree

A Binary Search Tree is a data structure that enables the efficient maintenance of a sorted list of numbers. It is called a binary tree because each node can have a maximum of two children. A Binary Search Tree is a search tree because it can be used to search for the presence of a number in logarithmic time.

The properties that distinguish a Binary Search Tree from a regular binary tree are:

1. All nodes of the left subtree are less than the root node.
2. All nodes of the right subtree are greater than the root node.
3. Both subtrees of each node are also Binary Search Trees, i.e. they possess the above two properties.

There are three basic operations that can be performed on a Binary Search Tree:

1. **Search**: This operation searches for a particular element in the Binary Search Tree and returns the node if it is present, else it returns null.
2. **Insert**: This operation inserts a new element into the Binary Search Tree in its appropriate position according to the Binary Search Tree properties.
3. **Delete**: This operation removes a specified node from the Binary Search Tree while maintaining the Binary Search Tree properties.

My implementation of Binary Search Tree (BST)

In this section, we will discuss the implementation of a Binary Search Tree (BST) in C programming language. The implementation includes a structure with an integer value, string, and two pointers that represent a node in the linked-list. The pointers left and right connect the root to its children.

```
typedef struct Node {
    int number;
    char* string;
    struct Node *left;
    struct Node *right;
}NODE;
```

The BST is implemented in C programming language using a structure called NODE. The NODE structure consists of four components: an integer number, a character string, a pointer to the left child, and a pointer to the right child. The integer number represents the value stored in the node, while the character string represents any additional data associated with the node. The left and right pointers represent the connections between nodes in the BST.

Search

The search operation in the BST enables searching for a particular key in the BST in logarithmic time. The search function takes two arguments: the root node and the key value to be searched. The function returns the character string associated with the node containing the key if it is found, or NULL otherwise.

```
char* search(NODE* root, int key) {
        while (root != NULL && root->number != key) {
            if (key < root->number) {
                root = root->left;
            } else {
                root = root->right;
            }
        }
        printf("%s of %d key", (*root).string, (*root).number);
        return (*root).string;
    }
```

The search function works by traversing the BST from the root node, comparing the key with the value stored in each node. If the key is less than the value of the current node, the search function moves to the left child. If the key is greater than the value of the current node, the search function moves to the right child. This process is repeated until the key is found or the search reaches the end of the BST.

If the search function finds the key, it prints the associated character string and returns it. If the search function does not find the key, it returns NULL.

Insert

In this section, we will discuss the implementation of the insert operation in a Binary Search Tree (BST) in C programming language. The insert operation enables inserting a new node with a given number and string into the BST. We will discuss the implementation of the insert function and the logic behind it.

The insert operation in the BST enables inserting a new node with a given number and string into the BST. The insert function takes three arguments: a double pointer to the root node, the number to be inserted, and the string associated with the number. The function creates a new node with the given number and string, and then inserts it into the BST in the appropriate position.

The insert function works by first creating a new node with the given number and string using the malloc function. The left and right pointers of the new node are set to NULL. If the root of the BST is NULL, the new node becomes the root of the BST. Otherwise, the function uses a temporary node to traverse the BST until it finds the appropriate position for the new node.

The function uses a while loop to traverse the BST. If the number in the new node is less than the number in the current node, the function tries to find a position on the left of the current node. If the number in the new node is greater than the number in the current node, the function tries to find a position on the right of the current node. If the number in the new node is already in the BST, the function simply returns.

When the appropriate position for the new node is found, the left or right pointer of the current node is set to the new node, depending on whether the new node should be on the left or right of the current node.

```
void insert(NODE**root, int num, char* name) {
    NODE* new = (NODE *)malloc(sizeof(NODE));
    new->number = num;
    new->string = name;
    new->left = NULL;
    new->right = NULL;

    // if the linked-list is empty, create root
    if ((*root) == NULL) {
        (*root) = new;
    } else {
        // while
        NODE* actual = (*root);
        while (1) {
            // is num is lower than actual root
            if (num < actual->number) {
                if (actual->left == NULL) {
                    actual->left = new;
                    break;
                } else {
                    actual = actual->left;
                }
                // if num is bigger than actual root
            } else if (num > actual->number){
                if (actual->right == NULL) {
                    actual->right = new;
                    break;
                } else {
                    actual = actual->right;
                }
                // case num already is in list
            } else {
                return;
            }
        }
    }
}
```

In summary, the implementation of the insert operation in a Binary Search Tree in C programming language involves creating a new node with the given number and string, and then inserting it into the BST in the appropriate position. The insert function uses a while loop to traverse the BST and find the appropriate position for the new node. When the appropriate position is found, the new node is inserted into the BST by setting the left or right pointer of the current node to the new node.

Delete

In the delete function, we start by initializing two temporary nodes, **parent** and **actual**, to NULL and **root**, respectively. We then use a while loop to traverse the tree until we find the node with the key we want to delete or reach the end of the tree. At each iteration, we update the **parent** and **actual** nodes based on whether the key we want to delete is smaller or greater than the current node's number.

If we reach the end of the tree without finding the node with the key we want to delete, we return the root as the key was not found.

```
NODE* delete(NODE* root, int key) {
    NODE* parent = NULL;
    NODE* actual = root;

    while (actual != NULL && actual->number != key) {
        parent = actual;
        if (key < actual->number) {
            actual = actual->left;
        } else {
            actual = actual->right;
        }
    }

    if (actual == NULL) {
        return root; // key not found
    }
```

Now we need to take care of three possible cases that may occur when deleting a node from a binary search tree:

Case 1:

The node we want to delete has no children. In this case, we simply need to check if the node we want to delete is the root of the tree or not. If it is the root, we set the root to NULL. Otherwise, we check if the node is the left or right child of its parent and set the corresponding child pointer to NULL. Finally, we free the memory of the node we just deleted.

```
    if (actual->left == NULL && actual->right == NULL) {
        if (parent == NULL) {
            root = NULL; // current is root
        } else if (actual == parent->left) {
            parent->left = NULL;
        } else {
            parent->right = NULL;
        }
        free(actual);
    }
```

Case2:
The node we want to delete has one child. In this case, we need to find the child of the node we want to delete and set it as the child of the node's parent. If the node we want to delete is the root of the tree, we set the child as the new root. If the node is a child of its parent, we set the corresponding child pointer to the child node. We then free the memory of the node we just deleted.

```
    else if (actual->left == NULL || actual->right == NULL) {
        NODE* child;
        if (actual->left == NULL) {
            child = actual->right;
        } else {
            child = actual->left;
        }
        if (parent == NULL) {
            root = child; // current is root
        } else if (actual == parent->left) {
            parent->left = child;
        } else {
            parent->right = child;
        }
        free(actual);
    }
```

Case3:
The node we want to delete has two children. In this case, we need to find the node with the smallest number in the right subtree of the node we want to delete. This node is called the successor of the node we want to delete. We then replace the number of the node we want to delete with the number of its successor. We then delete the successor node using either Case 1 or Case 2.

```
    else {
        NODE* successor = actual->right;
        parent = actual;
        while (successor->left != NULL) {
            parent = successor;
            successor = successor->left;
        }
        actual->number = successor->number;
        if (parent == actual) {
            parent->right = successor->right;
        } else {
            parent->left = successor->right;
        }
        free(successor);
    }
    return root;
}
```

If the successor is a child of the node we want to delete, we set the successor's right child as the new child of the node's parent. If the successor is not a child of the node we want to delete, we need to update its parent's left child pointer to point to the successor's right child.

After updating the tree to remove the node we want to delete, we free the memory of the node and return the root of the updated tree.

Testing

The program contains several functions that implement the binary search tree data structure. The testing function, **main()**, was designed to demonstrate the functionality of these functions.

```
int main() {
    int space = 0;
    // create empty linked-list
    NODE *root = NULL;
    insert(&root, 7, "David");
    insert(&root, 8, "Peter");
    insert(&root, 4, "Julia");
    insert(&root, 2, "Patricia");
    insert(&root, 1, "Monika");
    insert(&root, 3, "Ondrej");
    insert(&root, 6, "Juraj");
    insert(&root, 5, "Roman");
    insert(&root, 9, "Kamil");
    printf("--------------------------\nTree after Inserts:\n\n");
    print(root, space);
    delete(root, 4);
    printf("--------------------------\nTree after 4 is deleted:\n\n");
    print(root, space);
    printf("--------------------------\nSearching in BVS with key 1:\n\n");
    search(root, 1);

    return 0;
}
```

The testing demonstrates the functionality of the binary search tree implementation, including insertion, deletion, and searching. It also shows how the **print()** function can be used to visualize the structure of the tree.

```
Tree after Inserts:          Tree after 4 is deleted:    Searching with key 1:
            9<                           9<              Monika of 1 key
      8<                           8<
7<                           7<
            6<                           6<
                  5<
      4<                           5<
                  3<                           3<
            2<                           2<
                  1<                           1<
```

1.2 Implementation of AVL tree

To implement AVL Tree, we have used a struct called AVLNode. This struct contains the and a string associated with that key. Additionally, it contains pointers to the left and right child nodes, as well as a height field that keeps track of the vertical position of the node in the tree.

```c
// struct of AVL Node
typedef struct AVLNode {
    long long key;
    char* string;
    struct AVLNode *left;
    struct AVLNode *right;
    int height;
} AVLNode;
```

In comparison to a standard binary search tree, the AVL Tree requires additional functions to perform rotations to the left or right, calculate balance factors, and implement rebalancing. Moreover, we need a function to return the node with the minimal value in the right subtree of a given node, which is utilized in the delete function. Lastly, a function is implemented to find the maximum value of two integers, which is used in height calculations.

By utilizing these additional functions, we can ensure that the AVL Tree remains balanced, leading to efficient insertion, deletion, and searching of nodes in the tree.

Functions rotate_left and rotate_right

Functions, **rotate_left** and **rotate_right**, are used to perform left and right rotations, respectively, on a given AVL node. These rotations are crucial in maintaining the balance of the tree, as they shift the tree's nodes around to ensure that the height of the left and right subtrees of a given node do not differ by more than 1.

```c
AVLNode *rotate_left(AVLNode *root) {
    AVLNode *new_root = root->right;
    AVLNode *subtree = new_root->left;
    new_root->left = root;
    root->right = subtree;

    // actualisation of height
    root->height = 1 + max(height(root->left), height(root->right));
    new_root->height = 1 + max(height(new_root->left), height(new_root->right));

    return new_root;
}

AVLNode *rotate_right(AVLNode *root) {
    AVLNode *new_root = root->left;
    AVLNode *subtree = new_root->right;
    new_root->right = root;
    root->left = subtree;

    // actualisation of height
    root->height = 1 + max(height(root->left), height(root->right));
    new_root->height = 1 + max(height(new_root->left), height(new_root->right));

    return new_root;
}
```

Balance factor

The **balance_factor** function is used to calculate the balance factor of a given AVL node. The balance factor is defined as the difference between the height of the left and right subtrees of the node. If the

balance factor of a node is greater than 1 or less than -1, it is considered unbalanced and requires rebalancing.

```
int balance_factor(AVLNode *root) {
    if (root == NULL) {
        return 0;
    }

    return height(root->left) - height(root->right);
}
```

Node with minimal value and Maximal value of two integers

The min_value_node function returns the node with the smallest value in a given subtree, which is useful in certain operations such as deletion. And the max function is a simple utility function that returns the maximum of two given integers.

```
AVLNode *min_value_node(AVLNode *n)
{
    AVLNode *current = n;
    while (current->left != NULL) {
        current = current->left;
    }

    return current;
}


int max(int a, int b) {
    return (a > b) ? a : b;
}
```

Create new Node

The **create_AVLNode()** function creates a new AVLNode with the specified key and string value. This function is used to initialize a new AVLNode whenever one needs to be added to the AVL tree. The function allocates memory for a new AVLNode structure using **malloc()**, sets the key and string values, initializes the left and right child pointers to **NULL**, and sets the height of the new node to 1.

```
AVLNode *create_AVLNode(int key, char* string) {
    AVLNode *new_AVLNode = (AVLNode *) malloc(sizeof(AVLNode));

    new_AVLNode->key = key;
    new_AVLNode->string = string;
    new_AVLNode->left = NULL;
    new_AVLNode->right = NULL;
    new_AVLNode->height = 1; // height of new node is 1

    return new_AVLNode;
}
```

Insert

The AVLinsert function is responsible for inserting a new node into the AVL tree. The function takes three parameters: the root node, an integer key, and a string.

```
AVLNode * AVLinsert(AVLNode * node, int key, char* string) {
    if (node == NULL) {
        return create_AVLNode(key, string);
    }

    if (key < node->key) {
        node->left = AVLinsert(node->left, key, string);
    }

    else if (key > node->key) {
        node->right = AVLinsert(node->right, key, string);
    }

    else {
        return node;
    }
        …
```

If the tree is empty, the new node is set as the root. If the root already exists in the tree, we compare the value of the key attribute and move to the left or right based on this comparison. If the node with the same key already exists in the tree, the insertion process is terminated.

After inserting the node, the function updates the height of the node and checks the balance factor of the tree using the balance_factor subfunction.

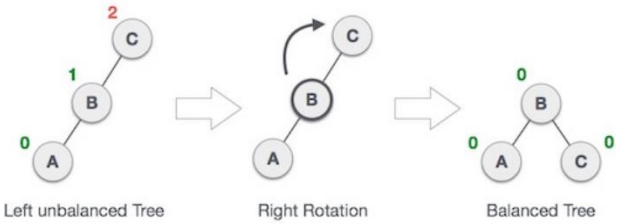
```
    node->height = 1 + max(height(node->left), height(node->right));
    int balance = balance_factor(node);

    if (balance > 1 && key < node->left->key) {  // rotate rigth
        return rotate_right(node);
    }
    if (balance < -1 && key > node->right->key) { // rotate left
        return rotate_left(node);
    }
    if (balance > 1 && key > node->left->key) { // LR rotation
        node->left = rotate_left(node->left);
        return rotate_right(node);
    }
    if (balance < -1 && key < node->right->key) { // RL rotation
        node->right = rotate_right(node->right);
        return rotate_left(node);
    }
    return node;
}
```

If the balance factor of the tree is greater than 1 and the key of the new node is less than the key of the left child of the root, the AVL tree is right-rotated. If the balance factor of the tree is less than -1 and the key of the new node is greater than the key of the right child of the root, the AVL tree is left-rotated.

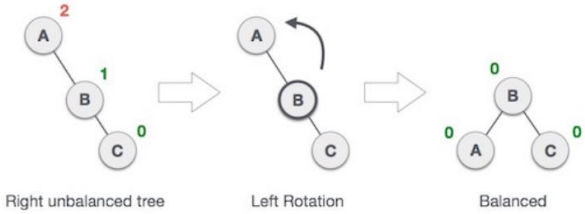
If the balance factor of the tree is greater than 1 and the key of the new node is greater than the key of the left child of the root, the AVL tree performs a left-right rotation. If the balance factor of the tree is less than -1 and the key of the new node is less than the key of the right child of the root, the AVL tree performs a right-left rotation.

There are four possible situations we need to take care of.

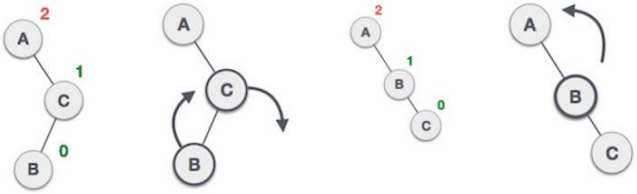
Right rotation



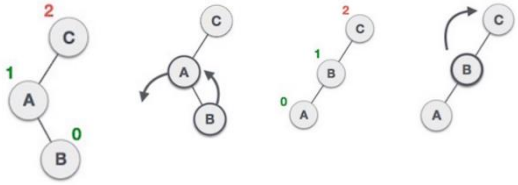
Left rotation



Right rotation and then left rotation



Left rotation and then right rotation



Search

The AVL search function is used to find a node with a given key in the AVL tree. The function takes the root node of the tree and the key to be searched for as arguments. The search algorithm used is similar to that of a binary search tree. Starting from the root node, the function moves left or right in the tree based on whether the key to be searched is less than or greater than the key of the current node. This process is repeated until either the node with the desired key is found or a leaf node is reached with no matching key.

If the node with the key is found, the function returns a pointer to that node. If the key is not found, the function returns NULL.

```
AVLNode * AVLsearch(AVLNode *root, int key) {
    AVLNode* actual = root;
    while (actual != NULL && actual->key != key) {
        if (key < actual->key) {
            actual = actual->left;
        } else {
            actual = actual->right;
        }
    }
    root = actual;
    free(actual);
    return root;

}
```

Delete

When deleting a node from an AVL tree, we must consider all possible cases that can occur. The deletion process is similar to that of a binary search tree.

```
AVLNode *AVLdelete(AVLNode *root, int key) {
    // Step1. Delete as in BVS
    if (root == NULL) {
        return root;
    }

    if (key < root->key) {
        root->left = AVLdelete(root->left, key);
    } else if (key > root->key) {
        root->right = AVLdelete(root->right, key);
    } else {
```

First, we check if the root exists. If it doesn't, we return the root. After that, we search for the node with the key we want to delete. Once we find the node, we must determine if it has 0, 1, or 2 children so we don't lose any pointers in our structure.

Case 1: No Child - If the node has no children, we simply delete it and set the pointer to NULL.

Case 2: One Child - If the node has only one child, we replace the node with its child by updating the pointers accordingly.

```
if ((root->left == NULL) || (root->right == NULL)) {
     // this returns node which is not NULL
    AVLNode *temp = root->left ? root->left : root->right;

    // Case : No child
    if (temp == NULL) {
        temp = root;
        root = NULL;
    } else {
        // Case : 1 child
        *root = *temp;
    }

    free(temp);
```

Case 3: Two Children - If the node has two children, we replace it with the next highest node in the right subtree. We recursively delete the next highest node, which will either have no children or one child.

```
    } else {
        // Case : 2 children
        AVLNode *temp = min_value_node(root->right);

        root->key = temp->key;

        root->right = AVLdelete(root->right, temp->key);
    }

}

if (root == NULL) {
    return root;
}
```

After deleting the node, we need to check if the tree is still balanced and perform any necessary rotations to ensure it remains balanced.

```
    // Actualisation of height
    root->height = 1 + max(height(root->left), height(root->right));

    int balance = balance_factor(root);

    // Left Left Case
    if (balance > 1 && balance_factor(root->left) >= 0) {
        return rotate_right(root);
    }

    // Left Right Case
    if (balance > 1 && balance_factor(root->left) < 0) {
        root->left = rotate_left(root->left);
        return rotate_right(root);
    }

    // Right Right Case
    if (balance < -1 && balance_factor(root->right) <= 0) {
        return rotate_left(root);
    }

    // Right Left Case
    if (balance < -1 && balance_factor(root->right) > 0) {
        root->right = rotate_right(root->right);
        return rotate_left(root);
    }

    return root;
}
```

The following is a comparison between a balanced and unbalanced binary tree after the deletion of node with key 4 (same sample inserted):

```
      Tree after 4 is deleted      |  Tree after 4 is deleted
         without balancing         |       with balancing
                                   |
                  9<               |                  9<
            8<                     |            8<
      7<                           |      7<
                  6<               |            6<
            5<                     | 5<
                        3<         |            3<
                  2<               |      2<
                        1<         |            1<
```

1.3 Implementation of Splay tree

The Splay Tree is a self-balancing binary search tree that can achieve a favorable amortized time complexity. In this project, we implement the Splay Tree data structure in C language.

Splay Tree Node

The Splay Tree Node is defined as a struct named "SPLAYNode". It contains atributes: "key", "left", and "right" and "string" . The "key" field stores the value of the node, and the "left" and "right" fields point to the left and right child of the node respectively. Also "string" should represent any data that can be stored in the node.

Rotation Functions

The Splay Tree uses two types of rotations, "rightRotate" and "leftRotate". Unlike other self-balancing trees, such as AVL Tree or Red-Black Tree, the Splay Tree does not require complex rotations to maintain its balance. The rightRotate and leftRotate functions simply perform a rotation around a node.

```
SPLAYNode *rightRotate(SPLAYNode *x) {
    SPLAYNode *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

SPLAYNode *leftRotate(SPLAYNode *x) {
    SPLAYNode *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}
```

The rightRotate function rotates the node "x" to the right, with its left child "y". The right child of "y" becomes the new left child of "x". The function returns the new root of the subtree.

The leftRotate function rotates the node "x" to the left, with its right child "y". The left child of "y" becomes the new right child of "x". The function returns the new root of the subtree.

Splay Function

The "splay" function is the core operation of the Splay Tree. It "splays" the node with the given key to the root of the tree. Splaying means moving the accessed node to the root of the tree by performing rotations.

The splay function takes two arguments: the root of the subtree and the key of the node to be splayed. It returns the new root of the subtree after the splaying operation.

```
SPLAYNode *splay(SPLAYNode *root, long key) {
    if (root == NULL || root->key == key) {
        return root;
    }

    if (root->key > key) {
        if (root->left == NULL) {
            return root;
        }

        if (root->left->key > key) {
            root->left->left = splay(root->left->left, key);
            root = rightRotate(root);
        } else if (root->left->key < key) {
            root->left->right = splay(root->left->right, key);
            if (root->left->right != NULL) {
                root->left = leftRotate(root->left);
            }
        }

        if (root->left == NULL) {
            return root;
        } else {
            return rightRotate(root);
        }
    } else {
        if (root->right == NULL) {
            return root;
        }

        if (root->right->key > key) {
            root->right->left = splay(root->right->left, key);
            if (root->right->left != NULL) {
                root->right = rightRotate(root->right);
            }
        } else if (root->right->key < key) {
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }

        if (root->right == NULL) {
            return root;
        } else {
            return leftRotate(root);
        }
    }
}
```

The splay function first checks if the root is NULL or if the root's key is equal to the given key. If either condition is true, it returns the root.

Otherwise, the function compares the given key with the root's key. If the key is less than the root's key, it recursively splays the left subtree. If the key is greater than the root's key, it recursively splays the right subtree.

Once the desired node is found, the splay operation begins. The function performs a series of rotations to move the accessed node to the root of the tree. The specific rotations performed depend on the location of the accessed node relative to its parent and grandparent.

Insert

The splayInsert function is responsible for inserting a new node into the Splay tree. The function takes three parameters: the root node, a long integer key, and a string.

Splay tree is a self-adjusting binary search tree. In this implementation, the SplayInsert function adds a new node to the Splay tree by first finding the key in the tree and then moving the node with the key to the root. If the key does not exist in the tree, the function creates a new node and splays the tree to move the new node to the root.

If the tree is empty, the new node is set as the root. If the root already exists in the tree, the function splays the tree to move the node with the key to the root. If the node with the same key already exists in the tree, the insertion process is terminated.

After inserting the new node, the function updates the left and right pointers of the new node and returns the new root of the Splay tree.

```
SPLAYNode *splayInsert(SPLAYNode *root, long key, char* string) {
    if (root == NULL) {
        return createNode(key, string);
    }
    root = splay(root, key);
    if (root->key == key) {
        return root;
    }

    SPLAYNode *SplayNode = createNode(key, string);
    if (root->key > key) {
        SplayNode->right = root;
        SplayNode->left = root->left;
        root->left = NULL;
    } else {
        SplayNode->left = root;
        SplayNode->right = root->right;
        root->right = NULL;
    }
    return SplayNode;
}
```

Note that the splayInsert function does not check for balance factors or perform any rotations like in other self-adjusting binary search trees such as AVL trees or Red-Black trees. Instead, the splay function is used to optimize search operations by bringing the most recently accessed node to the root of the tree.

Search

The splaySearch function is used to search for a node in the Splay tree using its key. The function takes two parameters: the root node of the tree and the key to search for.

SPLAYNode *splaySearch(SPLAYNode *root, long key)

The function traverses the tree starting from the root node and compares the key of each node with the search key. If the search key is less than the key of the current node, the function moves to the left child of the node. Otherwise, it moves to the right child of the node.

```
SPLAYNode *splaySearch(SPLAYNode *root, long key) {
    while (root != NULL && root->key != key) {
        if (key < root->key) {
            root = root->left;
        } else {
            root = root->right;
        }
    }
    return root;
}
```

If the search key is found in the tree, the function returns the node with that key. Otherwise, it returns NULL. Note that this function does not perform any splaying operation. It simply returns the node with the search key, if it exists.

Delete

The splayDelete function is responsible for removing a node with a given key from a splay tree. The function takes two parameters: the root node and the key of the node to be deleted.

If the root is NULL, the function simply returns NULL. Otherwise, the function recursively calls itself on either the left or right subtree, depending on whether the key is less than or greater than the root node's key.

When the key matches the root node's key, the function checks whether the root has one or two children. If the root has one child, the function replaces the root with that child. If the root has two children, the function finds the minimum value node in the right subtree, replaces the root's key with that value, and then recursively deletes that minimum value node from the right subtree.

```
SPLAYNode *splayDelete(SPLAYNode *root, long key) {
    if (root == NULL) {
        return root;
    }
SPLAYNode *temp;
if (root->key < key) {
    root->right = splayDelete(root->right, key);
} else if (root->key > key) {
    root->left = splayDelete(root->left, key);
} else {
    if (root->left == NULL) {
        temp = root->right;
        free(root);
        return temp;
    } else if (root->right == NULL) {
        temp = root->left;
        free(root);
        return temp;
    }
    temp = min_value_nodeS(root->right);
    root->key = temp->key;
    root->right = splayDelete(root->right, temp->key);
}
    return root;
}
```

The function then returns the root of the tree.

Visual comparison (same sample)

Insert

Delete (node with key 4)

1.4 Testing

1.4.1 Methodology

The effectiveness of each implemented algorithm was tested by simulating a scenario where each tree was filled with nodes starting at 25,000, 50,000, 75,000... up to 1,000,000 nodes. Each test was repeated 10 times, where nodes were inserted, searched, and deleted. The average time in nanoseconds was recorded for each of the functions, and then divided by 10 and the number of nodes to obtain the time for one insert, one search, and one delete.

FOR EXAMPLE:

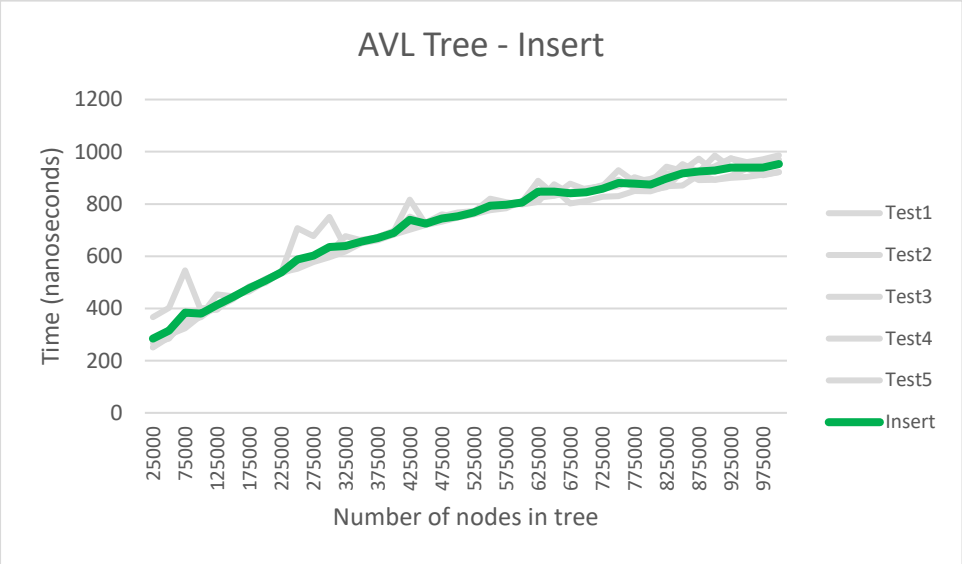
INSERT
75 000 NODES --> REPEAT 10X, EACH TIME AVERAGE += ACTUAL-TIME --> AVERAGE /= 10 --> AVERAGE /= 75 000 = 1 INSERT (NANOSECONDS)

To increase the accuracy of the results, this process was repeated five times for each function, and the average number from these five tests was calculated. The results were stored in a spreadsheet, and graphs were generated to illustrate the performance of each tree.

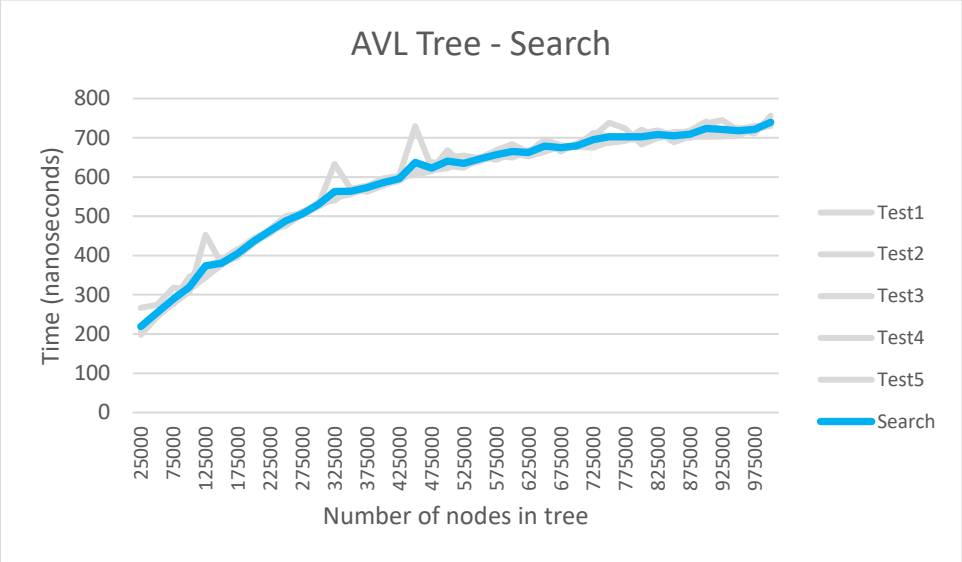
Results AVL Tree

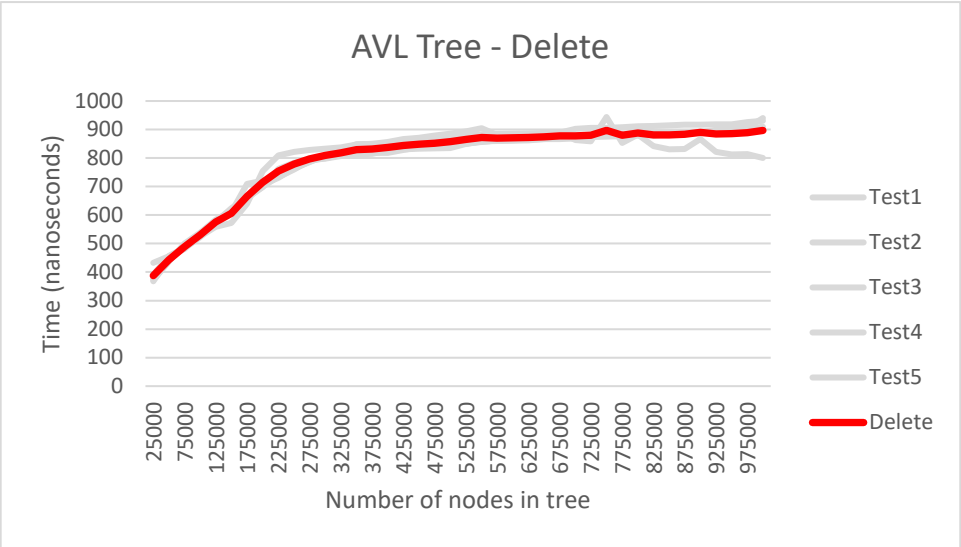
In this section, the results of the tests conducted on the AVL Tree are presented. The first graph shows the range of 5 x 10 x 25,000 to 5 x 10 x 1,000,000 inserts, while the second and third graphs show the same range for search and delete operations, respectively.

Graph 1.



Graph 2.



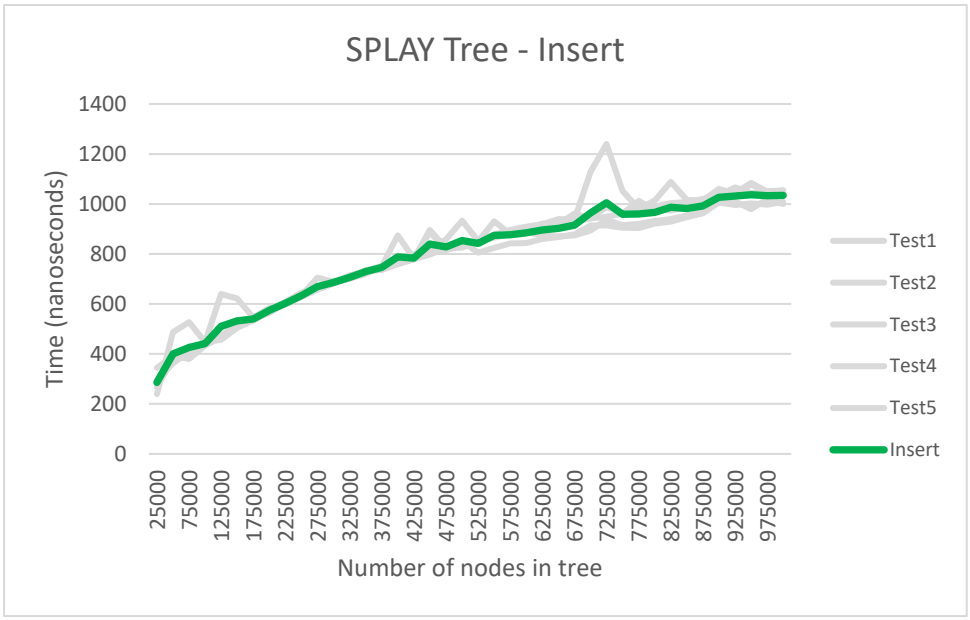


Graph 3.

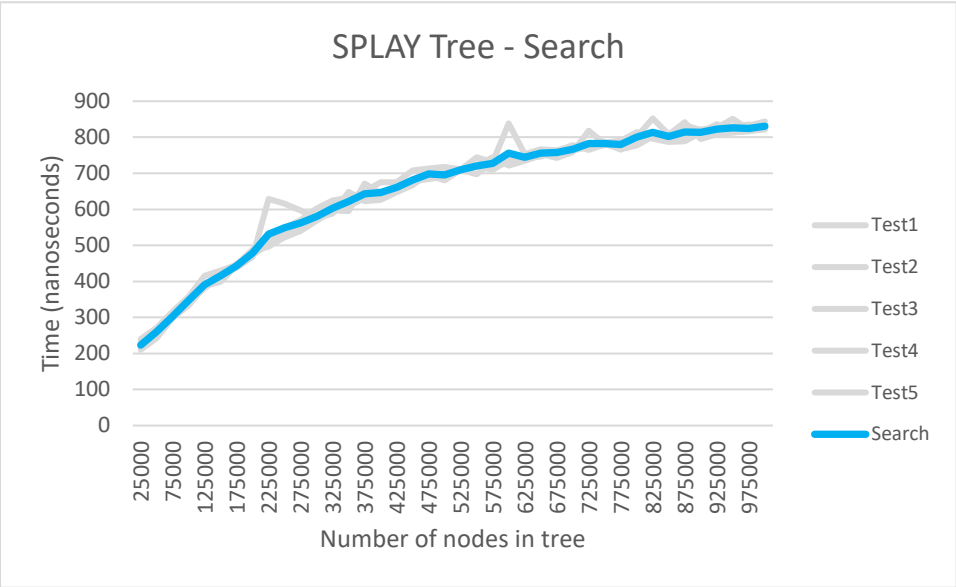
The results were obtained using the methodology described in section 1.4.1, with the average time for each operation (insert, search, delete) calculated and divided by the number of nodes to obtain the time for a single operation. The testing was repeated five times, and the results were averaged and recorded in an Excel sheet, from which the graphs were generated.

Results SPLAY Tree

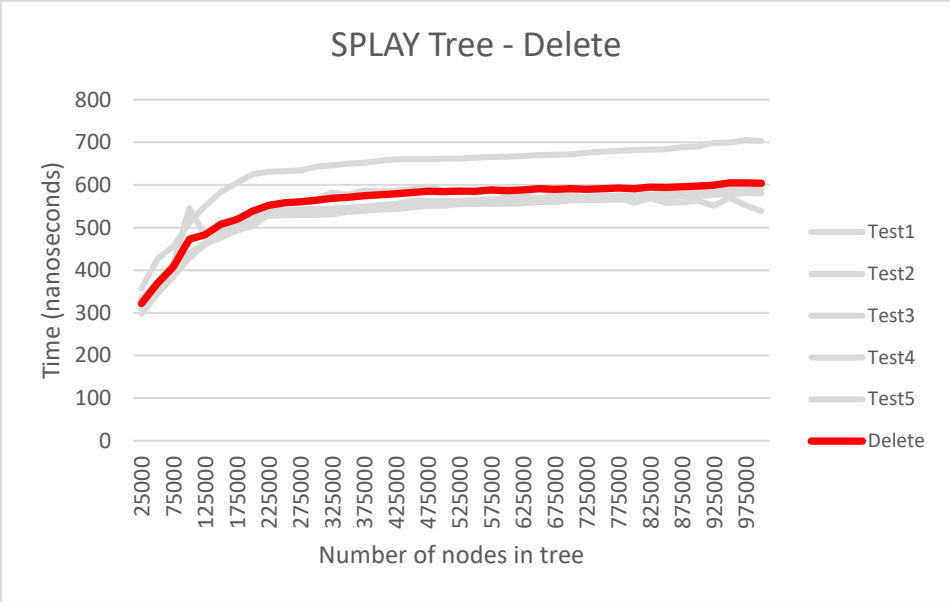
In this section, the results of the tests conducted on the SPLAY Tree are presented. The first graph shows inserts, while the second and third graphs show the same range for search and delete operations, respectively. Ranges and methodology stays same as in the previous section.



Graph 4.



Graph 5.

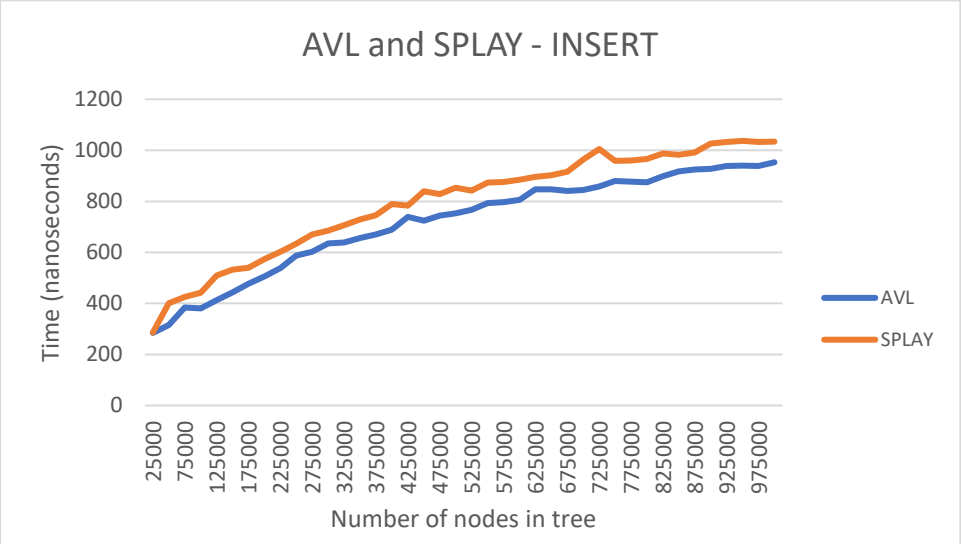


Graph 6.

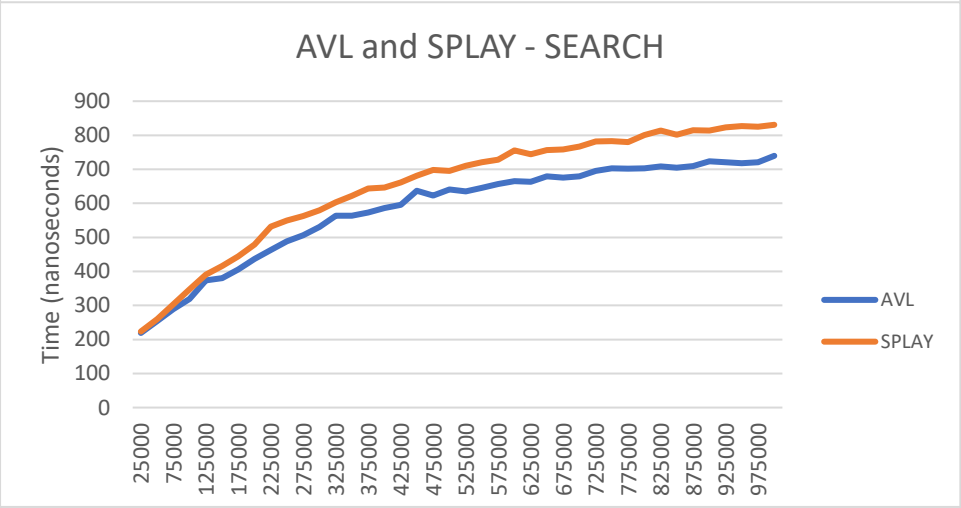
The results were also obtained using the methodology described in section 2.1 just like AVL Tree results.

Results side by side

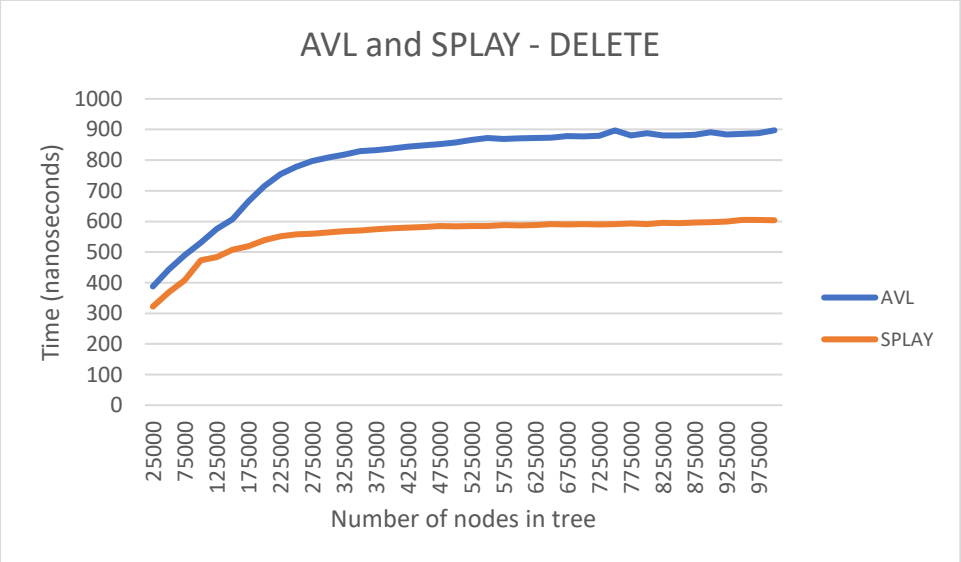
In this section, we present the results of our tests on the Splay Tree, as well as a comparison with the AVL Tree. Our methodology, as detailed in section 1.4.1, involved conducting multiple repetitions of each operation.



Graph 7.



Graph 8.



Graph 9.

Conclusion

The results from the experiments conducted on AVL and Splay trees show that the insertion performance is almost identical in both tree structures, with the AVL tree being slightly faster. According to theory, both algorithms should exhibit logarithmic time complexity, which means that as the number of nodes increases, the efficiency of the algorithm should also improve. In our tests, inserting one million nodes took 953.02 nanoseconds in AVL tree and 1033.78 nanoseconds in Splay tree.

The performance of the search operation was found to be very similar for both Splay and AVL trees. Both curves represented logarithmic trends, and there was almost no difference observed in small datasets between the two structures. However, as the number of nodes in the tree increased, performance differences became more significant. For example, in a tree with just 25,000 nodes, the difference between each structure was only 4 nanoseconds, with Splay taking 223.31 nanoseconds and AVL taking 219 nanoseconds. In contrast, in a tree with one million nodes, the search operation took 830.60 nanoseconds for Splay and 739.55 nanoseconds for AVL. This suggests that AVL tree is slightly faster than Splay tree, similar to the results obtained for the insert function.

The largest difference between Splay and AVL trees was observed in the delete operation, as demonstrated in the corresponding graph. Once again, both curves represented logarithmic trends, but in this operation, the Splay tree performed significantly better than the AVL tree. As with the insert and search operations, the difference between the two structures was relatively small for small datasets. However, as the dataset size increased, the difference in performance between the two structures became more noticeable.

In conclusion, both algorithms have their own advantages and disadvantages. It is challenging to determine which one is better since the choice of algorithm depends on the dataset and the operation required. However, considering the complexity of balancing, AVL tree may be preferred over Splay tree.

2. PART 2: Implementation of Hash table with chaining and hash table with

2.1 Hash table

A Hashing Table is a data structure used to store and retrieve data in constant time. It is also known as a Hash Table, and it works by mapping a set of keys to their corresponding values using a hash function. The hash function takes in a key and returns an index to a slot in the table, where the value associated with the key can be found.

The basic structure of a Hashing Table consists of an array of slots, where each slot can hold a key-value pair. The keys are usually unique, and the values can be any type of data. The hash function is responsible for mapping the keys to the slots in the table, and there are different way to solve collisions in the keys.

The key properties that distinguish a Hashing Table from other data structures are:

1. The hash function maps keys to unique indices in the table.
2. The table can resize, and each slot can hold a single key-value pair or more when chaining is used.
3. The table uses different methods to handle collisions.

There are three basic operations that can be performed on a Hashing Table:

1. Insert: This operation inserts a new key-value pair into the Hashing Table in its appropriate position according to the hash function. If a key already exists in the table, the value associated with that key is updated.
2. Search: This operation searches for a particular key in the Hashing Table and returns the corresponding value if it is present, else it returns null.
3. Delete: This operation removes a specified key-value pair from the Hashing Table while maintaining the Hashing Table properties. If the key is not found, the operation has no effect.

2.2 Hashing table with chaining

In this section, we will discuss the implementation of a hashing table with chaining in our project.

To begin with, we define a node which is used to store data in the table. This node has three attributes - a numerical value representing the data, a key used to hash the node and place it in the table, and a pointer to the next node, in case two or more nodes are placed in the same index.

Here is the structure of the node:

```
typedef struct HashWithChaining {
    char* key;
    int value;
    struct HashWithChaining* next;
} HashWithChaining;
```

Additionally, we define the structure of the hash table with chaining, which has three attributes - the size of the table, the number of elements in the table, and an array of pointers to the nodes in the table.

```
typedef struct HashChain {
    int size;
    int count;
    HashWithChaining** array;
```

After defining the structures, the next important step is to define a hash function. The hash function used in this implementation is selected because it has a low probability of producing collisions.

```
// Pôvodná hash funckia, ktorá bola veľmi pomalá a mala množstvo kolízií
/*
int hash_f(char* key, int table_len){
    int hash = 0;
    for (int i = 0; i < strlen(key); i++){
        hash += (int)key[i] + ((int)key[i]) * (i + 1);
    }
    return hash %= table_len;
}
 */
```

I used this simple hash function which add ASCII value of each symbol in string and then multiply it with (index + 1), however this function was very slow and there were too much collisions. That is the reason I used this murmur3 hash function which is specialized on strings.

```
uint32_t hash_function_chain(const void* key, size_t len) {
    int leng = 25;
    const uint8_t* data = (const uint8_t*)key;
    const uint32_t c1 = 0xcc9e2d51;
    const uint32_t c2 = 0x1b873593;
    const uint32_t r1 = 15;
    const uint32_t r2 = 13;
    const uint32_t m = 5;
    const uint32_t n = 0xe6546b64;
    uint32_t hash = 0x1b873593;
    uint32_t k;

    for (size_t i = 0; i < leng; i += 4) {
        k = *(uint32_t*)(data + i);
        k *= c1;
        k = ROTL32(k, r1);
        k *= c2;

        hash ^= k;
        hash = ROTL32(hash, r2);
        hash = hash * m + n;
    }

    const uint8_t* tail = (const uint8_t*)(data + (leng & ~3));
    k = 0;
    switch (len & 3) {
        case 3: k ^= tail[2] << 16;
        case 2: k ^= tail[1] << 8;
        case 1: k ^= tail[0];
            k *= c1;
            k = ROTL32(k, r1);
            k *= c2;
            hash ^= k;
```

```
    }

    hash ^= leng;
    hash ^= hash >> 16;
    hash *= 0x85ebca6b;
    hash ^= hash >> 13;
    hash *= 0xc2b2ae35;
    hash ^= hash >> 16;

    // mod by the table size to get a value within the table range
    return hash % len;
}
```

In order to insert nodes into the hashing table with chaining, we need to first create the table and decide on its starting size. To accomplish this, we have created a separate function that will facilitate the process and make it easier to read the code.

Here is the implementation of the function to create a table with chaining:

```
HashChain* create_table_chain(int size) {
    HashChain* table = (HashChain*) malloc(sizeof(HashChain));
    table->size = size;
    table->count = 0;
    table->array = (HashWithChaining**) calloc(size,
sizeof(HashWithChaining*));
    return table;
}
```

The **create_table_chain** function takes an integer argument which represents the size of the table to be created. It allocates memory for the table using the **malloc** function and initializes its size and count attributes. The **array** attribute of the table is then allocated memory using the **calloc** function to initialize all elements to NULL.

Once the table is created, we can insert nodes into it. To insert a new node, we must first create the node using a separate function.

```
HashWithChaining* create_node_chain(char* key, int value) {

    HashWithChaining* node = (HashWithChaining*)malloc(sizeof(HashWithChaining));
    node->key = key;
    node->value = value;
    node->next = NULL;
    return node;
}
```

The **create_node_chain** function takes two arguments - a string representing the key and an integer representing the value to be stored in the node. It allocates memory for the node using the **malloc** function and initializes its key, value, and next attributes.

By using these two functions, we can now easily insert nodes into the hash table with chaining.

Insert

The insertChain() function is responsible for inserting a new node with a given key-value pair into the hash table using chaining. The function first calculates the index for the given key using the hash function and retrieves the current node at the index. It also keeps track of the previous node to update its next pointer.

```
void insertChain(HashChain* table, char* key, int value) {
    long index = hash_function_chain(key, table->size);
    HashWithChaining* current = table->array[index];
    HashWithChaining* prev = NULL;
```

If the linked-list at the index is not empty, the function traverses the linked-list to find the matching key or the end of the linked-list. If a node with a matching key is found, the function updates the value of the existing node and returns. Otherwise, it adds the new node to the end of the linked-list at the calculated index and updates the count of nodes in the table.

```
while (current != NULL) {
    if (strcmp(current->key, key) == 0) {
        current->value = value;
        return;
    }
    prev = current;
    current = current->next;
}

HashWithChaining* node = create_node_chain(key, value);

if (prev == NULL) {
    table->array[index] = node;
} else {
    prev->next = node;
}

table->count++;
```

After inserting the new node, the function checks if the load factor has exceeded the defined threshold.

```
#define LOAD_FACTOR 1.5
#define SHRINK_FACTOR 0.45
```

If so, the function resizes the table by creating a new table with a larger size and rehashing all nodes into the new table.

```
    double load_factor = (double) table->count / table->size;

    if (load_factor >= LOAD_FACTOR) {
        HashChain* new_table = create_table_chain(table->size * 2);

        for (int i = 0; i < table->size; i++) {
            HashWithChaining* current = table->array[i];
            while (current != NULL) {
               insertChain(new_table, current->key, current->value);
                current = current->next;
            }
        }
```

After all nodes are placed into new table we update original table with the new one and deallocate memory which was allocated for the new table.

```
        *table = *new_table;
        free(new_table);
    }
}
```

Search

The searchChain function searches for a node with a given key in the hash table with chaining and returns the key of the found node or "No" if the node is not found. First, we have to calculate the index of the given node using the hash function and get the current node at the calculated index.

```
char* searchChain(HashChain* table, char* key) {
    int index = hash_function_chain(key, table->size);
    HashWithChaining *current = table->array[index];
    while (current != NULL) {
        if (strcmp(current->key, key) == 0) {
            return current->key;
        }
        current = current->next;
    }

    return "No";
}
```

Then we traverse the linked-list to find the matching key or the end of the linked-list. If we find the matching key, we return the key of the current node.

Delete

The deleteChain function is responsible for removing a node with a given key from the hash table with chaining. First, we calculate the index of the node using the hash function and get the current node at the calculated index. We keep track of the previous node to update its next pointer.

```
void deleteChain(HashChain* table, char* key) {
    int index = hash_function_chain(key, table->size);
    HashWithChaining *current = table->array[index];
    HashWithChaining *prev = NULL;
```

Then we traverse the linked-list to find the matching key or the end of the linked-list. If we find the matching key, we remove the node by updating the pointers of the previous and the next nodes. Then we deallocate memory for the removed node and update the count of nodes in the table.

```
while (current != NULL) {
    if (strcmp(current->key, key) == 0) {
        if (prev == NULL) {
            table->array[index] = current->next;
        } else {
            prev->next = current->next;
        }
        free(current);
        table->count--;
```

If the load factor of the table falls below the defined threshold and the size of the table is greater than 1, we create a new table with a smaller size and rehash all the remaining nodes into it. Then we update the original table with the new one and deallocate memory which was allocated for the new table.

```
      double load_factor = (double) table->count / table->size;

      if (load_factor <= SHRINK_FACTOR && table->size > 1) {
         HashChain *new_table = create_table_chain(table->size / 2);

         for (int i = 0; i < table->size; i++) {
             HashWithChaining *current = table->array[i];
             while (current != NULL) {
               insertChain(new_table, current->key, current->value);
                 HashWithChaining *temp = current;
                 current = current->next;
                 free(temp);
             }
         }
         *table = *new_table;
         free(new_table);
     }
     return;
        }
 prev = current;
 current = current->next;
    }
}
```

2.3 Hashing table with linear probing

When using linear probing to handle collisions in a Hashing Table, the table checks the next slot in the array if the slot associated with a hashed key is already occupied. If the next slot is also occupied, it checks the next one and so on, until it finds an empty slot. This means that all keys that collide with each other are stored sequentially in the array. When searching for a key, the table checks the hashed index and continues to search sequentially until it finds the key or an empty slot. This method has the advantage of being simple and efficient in terms of memory usage, but it can suffer from clustering, where groups of keys that hash to nearby indices cause long sequences of occupied slots, slowing down operations.

I used different struct for this implementation, because there is no need for linked-list structures.

```
typedef struct HashTable {
    char** keys;
    int* values;
    int size;
    int count;
} HashTable;
```

The implementation uses the Murmur3 hash function for hashing keys, which is also used in the implementation of hash table with chaining. Additionally, two constants LOAD_FACTOR_LP and SHRINK_FACTOR_LP are defined to control the load factor and shrink factor of the hash table.

```
#define LOAD_FACTOR_LP 0.90
#define SHRINK_FACTOR_LP 0.25
```

This function initializes the hash table by allocating memory for the HashTable struct, the keys array and the values array. The function returns a pointer to the HashTable struct.

```
// Function to initialize the hash table
HashTable* hash_table_init(int size) {
    HashTable* ht = malloc(sizeof(HashTable));
    ht->size = size;
    ht->keys = calloc(size, sizeof(char*));
    ht->values = calloc(size, sizeof(int));
    ht->count = 0;
    return ht;
}
```

This function deletes the hash table by freeing memory for the keys array, values array and the HashTable struct.

```
void hash_table_delete_lp(HashTable* ht) {
    for (int i = 0; i < ht->size; i++) {
        if (ht->keys[i]) {
            free(ht->keys[i]);
        }
    }
    free(ht->keys);
    free(ht->values);
    free(ht);
}
```

This function resizes the hash table to a new size. It creates a new keys array and a new values array with the new size, and rehashes all the elements from the old arrays into the new arrays using the linear probing technique. The old arrays are then freed, and the new arrays are assigned to the HashTable struct.

```
void hash_table_resize_lp(HashTable* ht, int new_size) {
    char** new_keys = calloc(new_size, sizeof(char*));
    int* new_values = calloc(new_size, sizeof(int));
    for (int i = 0; i < ht->size; i++) {
        if (ht->keys[i]) {
            int index = hash_function_lp(ht->keys[i], new_size);
            while (new_keys[index]) {
                index = (index + 1) % new_size;
            }
            new_keys[index] = ht->keys[i];
            new_values[index] = ht->values[i];
        }
    }
    free(ht->keys);
    free(ht->values);
    ht->keys = new_keys;
    ht->values = new_values;
    ht->size = new_size;
}
```

Insert

The hash_table_insert function is used to insert a key-value pair into the hash table. The function takes in a pointer to a HashTable struct, a char* key and an integer value as parameters.

```
void hash_table_insert(HashTable* ht, char* key, int value) {
    if ((float)ht->count / ht->size >= LOAD_FACTOR_LP) {
```

```
        hash_table_resize_lp(ht, ht->size * 2);
    }
    int index = hash_function_lp(key, ht->size);
    while (ht->keys[index]) {
        if (strcmp(ht->keys[index], key) == 0) {
            ht->values[index] = value;
            return;
        }
        index = (index + 1) % ht->size;
    }
    ht->keys[index] = strdup(key);
    ht->values[index] = value;
    ht->count++;
}
```

The implementation of the hash_table_insert function starts by checking the current load factor of the hash table. If the load factor is greater than or equal to the predefined LOAD_FACTOR_LP, the hash table is resized using the hash_table_resize_lp function.

The function then hashes the key using the hash_function_lp and determines the index to insert the key-value pair. If the key already exists in the hash table, the function updates the value of the existing key-value pair and returns.

If the index is already occupied by another key-value pair, the function uses linear probing to find the next available index. The key-value pair is then inserted into the next available index, and the count of the hash table is incremented.

Search

The hash_table_search function is used to search for a key in the hash table and return the corresponding value. The function takes in a pointer to a HashTable struct and a char* key as parameters.

```
int hash_table_search(HashTable* ht, char* key) {
    int index = hash_function_lp(key, ht->size);
    while (ht->keys[index]) {
        if (strcmp(ht->keys[index], key) == 0) {
            return ht->values[index];
        }
        index = (index + 1) % ht->size;
    }
    return 0; // default value if key not found
}
```

The implementation of the hash_table_search function starts by hashing the key using the hash_function_lp and determining the index of the key in the hash table. The function then uses linear probing to search for the key in the hash table.

If the key is found, the function returns the corresponding value. If the key is not found in the hash table, the function returns a default value of 0.

Delete

The function first computes the hash index for the given key using the linear probing hash function. It then iterates through the table at that index until it finds the key or an empty slot.

If the key is found, the function frees the memory allocated for the key and sets the key and value at the index to **NULL** and **0**, respectively. It also decrements the count of key-value pairs in the hash table.

```
// Function to remove a key-value pair from the hash table
void hash_table_remove(HashTable* ht, char* key) {
    int index = hash_function_lp(key, ht->size);
    while (ht->keys[index]) {
        if (strcmp(ht->keys[index], key) == 0) {
            free(ht->keys[index]);
            ht->keys[index] = NULL;
            ht->values[index] = 0;
            ht->count--;
// rehash remaining items in the cluster
            int next_index = (index + 1) % ht->size;
            while (ht->keys[next_index]) {
                char* temp_key = ht->keys[next_index];
                int temp_value = ht->values[next_index];
                ht->keys[next_index] = NULL;
                ht->values[next_index] = 0;
                ht->count--;
                hash_table_insert(ht, temp_key, temp_value);
                next_index = (next_index + 1) % ht->size;
            }
            if ((float)ht->count / ht->size <= SHRINK_FACTOR_LP && ht->size
> 1) {
                hash_table_resize_lp(ht, ht->size / 2);
            }
            return;
        }
        index = (index + 1) % ht->size;
    }
}
```

The function then rehashes the remaining items in the cluster by iterating through the next slots and inserting them into the hash table again.

If the load factor of the hash table becomes less than or equal to the shrink factor (**SHRINK_FACTOR_LP**) after removing the key-value pair, and the size of the hash table is greater than 1, the function resizes the hash table by half using the **hash_table_resize_lp** function.

2.4 Testing

2.4.1 Methodology

To test the effectiveness of the hash table with chaining algorithm, a new table was first defined using the create_table_chain function with a size of 13.

The testing involved simulating a scenario where the hash table was filled with nodes, starting at 10,000, 20,000, 35,000 and so on, up to 1,000,000 nodes. For each test, nodes were inserted, searched, and deleted. The test was repeated 10 times to obtain the average time in nanoseconds for each function. The time for one insert, one search, and one delete was then calculated by dividing the average time by 10 and the number of nodes in the test.

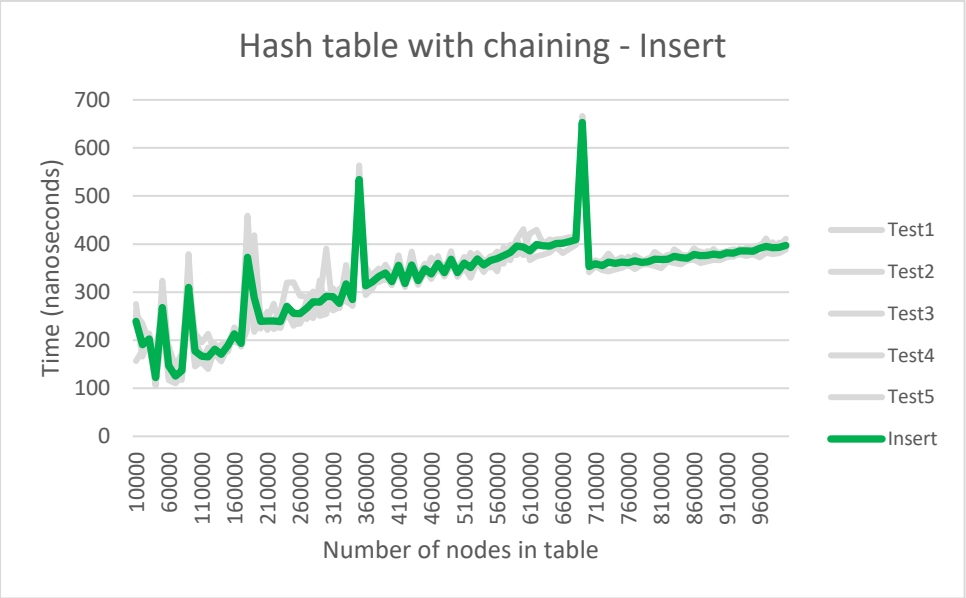
FOR EXAMPLE:

INSERT
70 000 NODES --> REPEAT 10X, EACH TIME AVERAGE += ACTUAL-TIME --> AVERAGE /= 10 --> AVERAGE /= 70 000 = 1 INSERT (NANOSECONDS)

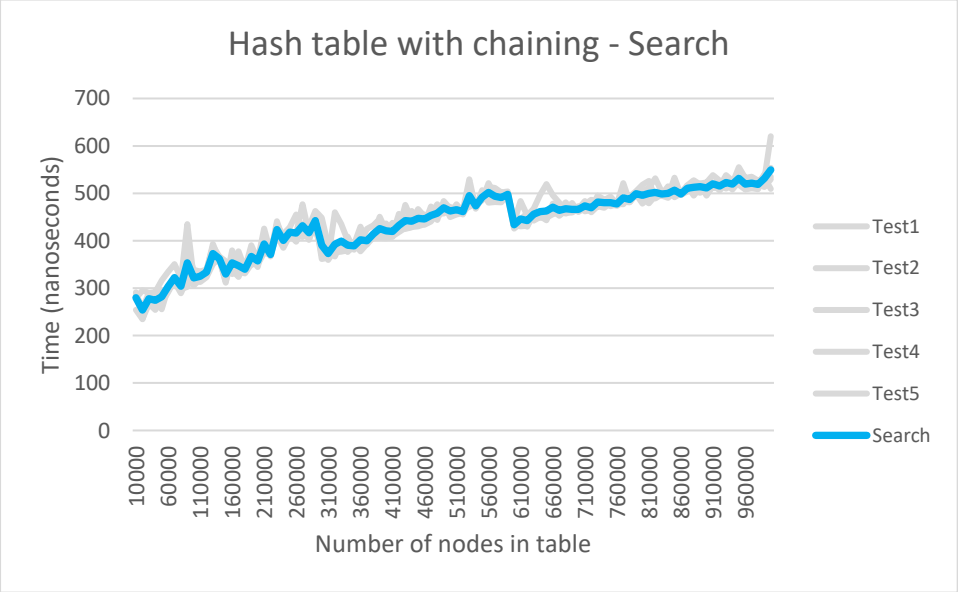
To increase the accuracy of the results, the entire testing procedure was repeated five times for each function. The average number from these five tests was calculated and recorded. The results were then stored in a spreadsheet, and graphs were generated to illustrate the performance of each hash table.

2.4.2 Results Hash table with chaining

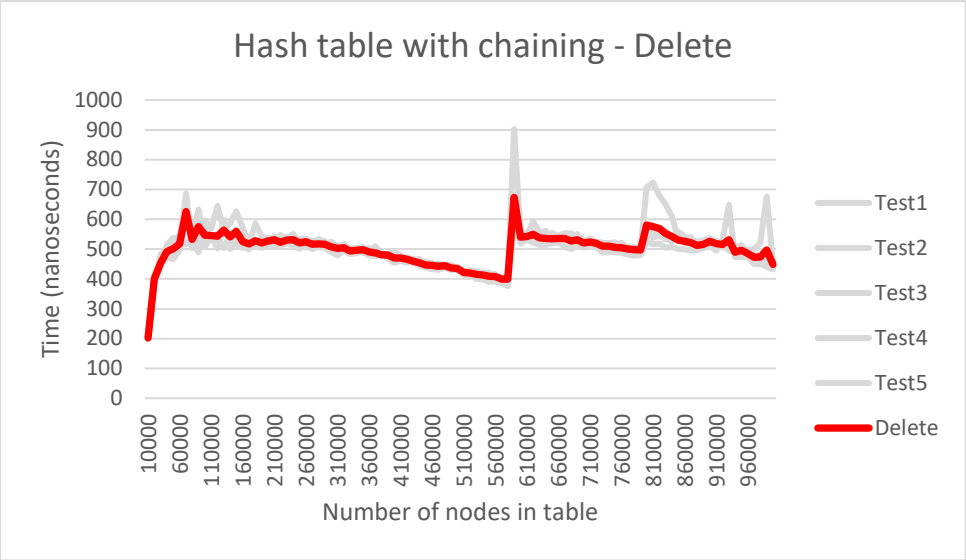
In this section, the results of the tests conducted on the hash table with chaining are presented. The first graph shows the range of 5 x 10 x 10,000 to 5 x 10 x 1,000,000 inserts, while the second and third graphs show the same range for search and delete operations, respectively.



Graph 10.



Graph 11.

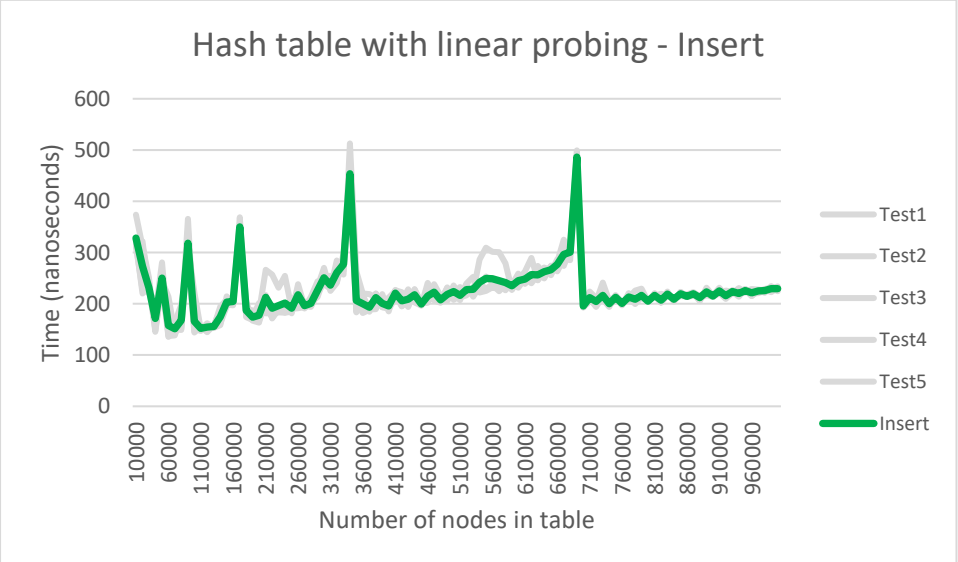


Graph 12.

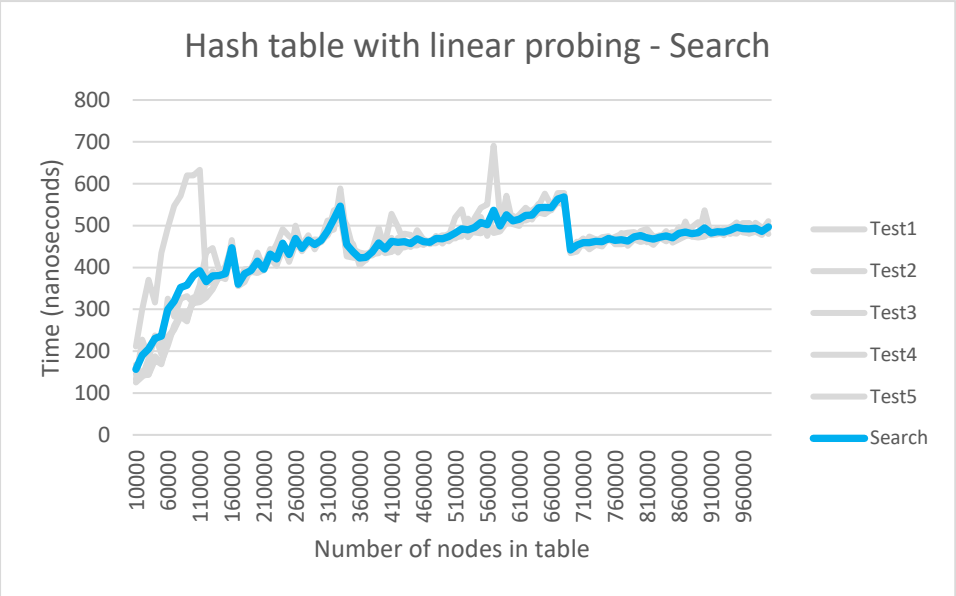
The results were obtained using the methodology described in section 2.4.1, with the average time for each operation (insert, search, delete) calculated and divided by the number of nodes to obtain the time for a single operation. The testing was repeated five times, and the results were averaged and recorded in an Excel sheet, from which the graphs were generated.

2.4.3 Results Hash table with linear probing

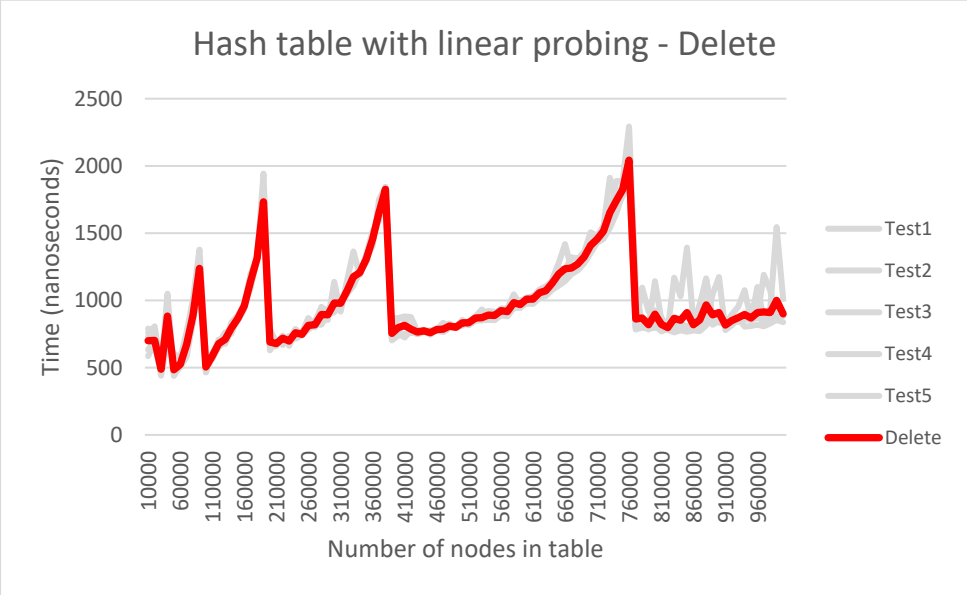
In this section, the results of the tests conducted on the hash table with linear probing are presented. The fourth first shows inserts, while the second and third graphs show the same range for search and delete operations, respectively. Ranges and methodology stays same as in the previous section.



Graph 13.



Graph 14.

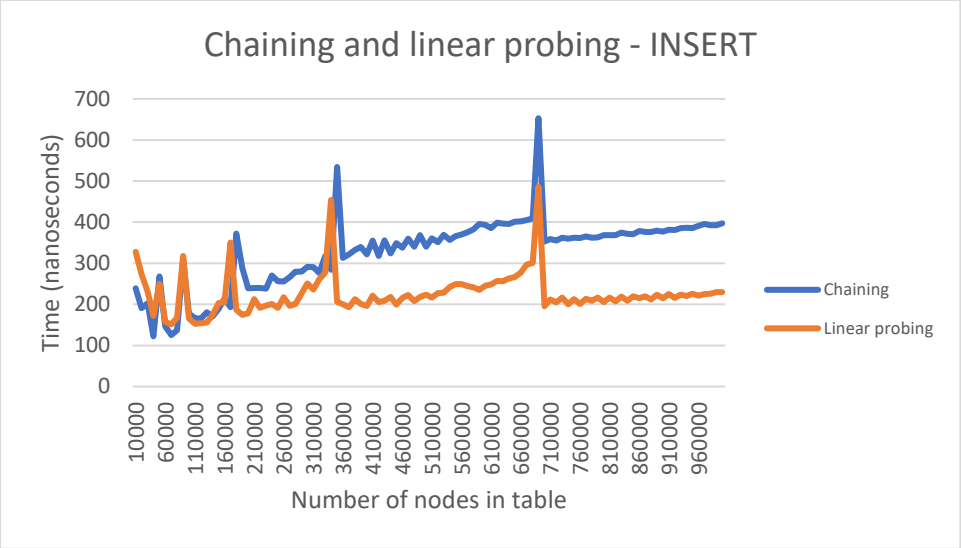


Graph 15.

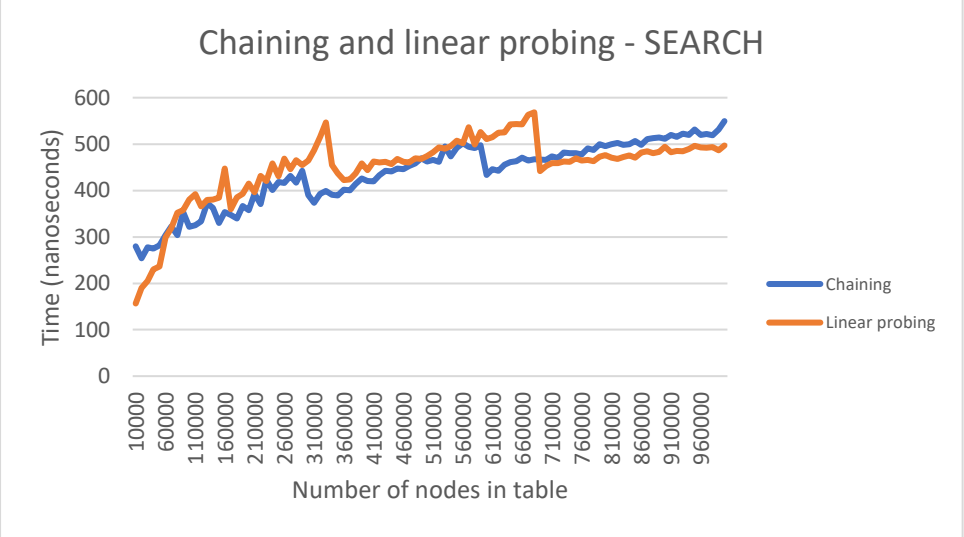
The results were also obtained using the methodology described in section 2.4.1 just like hash table with chaining results.

Results side by side

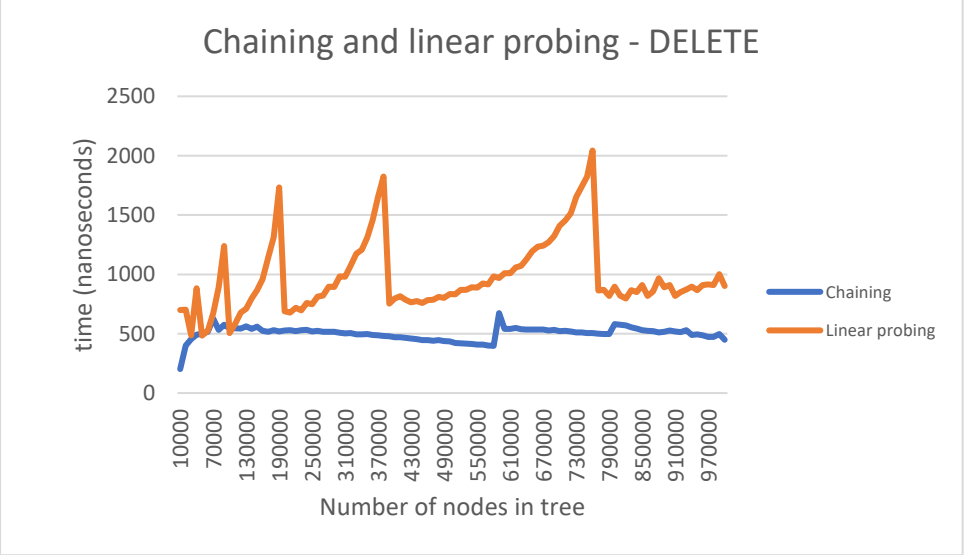
In this section, I present the results of my tests on the hash table with chaining, as well as a comparison with the hast table with linear probing. My methodology, as detailed in section 2.4.1, involved conducting multiple repetitions of each operation.



Graph 16.



Graph 17.



Graph 18.

Conclusion

Experimental results indicate that hash tables with chaining and linear probing exhibit similar insertion performance, with chaining being slightly faster. Both algorithms theoretically should have logarithmic time complexity, meaning their efficiency should improve with an increasing number of nodes. Resizing when all nodes have to be inserted again can cause spikes in the graph. Our tests indicate that inserting one million nodes took 229.35 nanoseconds in a hash table with chaining and 397.36 nanoseconds in a hash table with linear probing.

Regarding search operation, both hash tables show almost identical performance. The hash table with linear probing is more efficient for small datasets, then in the middle-sized datasets is chaining faster, but with large dataset linear probing again outperforms chaining. For example, searching for 50,000 nodes in chaining takes an average of 281.96 nanoseconds, while in linear probing, it takes 236.02 nanoseconds. And also, when the dataset contains one million nodes, linear probing takes 496.89 nanoseconds, whereas chaining takes 549.49 nanoseconds.

In contrast to insertion and search operations, the delete operation shows completely different results. Chaining is faster than linear probing from start to end, and both structures require resizing when the shrink factor is used, which is represented with spikes in both graphs. Deleting one million nodes took 900.74 nanoseconds for linear probing and 448.54 nanoseconds for chaining, which is just half the time linear probing requires.

In conclusion, both algorithms have their advantages and disadvantages, and the choice of algorithm depends on the dataset and the required operation. However, based on the experimental results, chaining appears to be more effective for the delete operation, whereas linear probing performs better for search operation in larger datasets.

Comparison of all four data structures

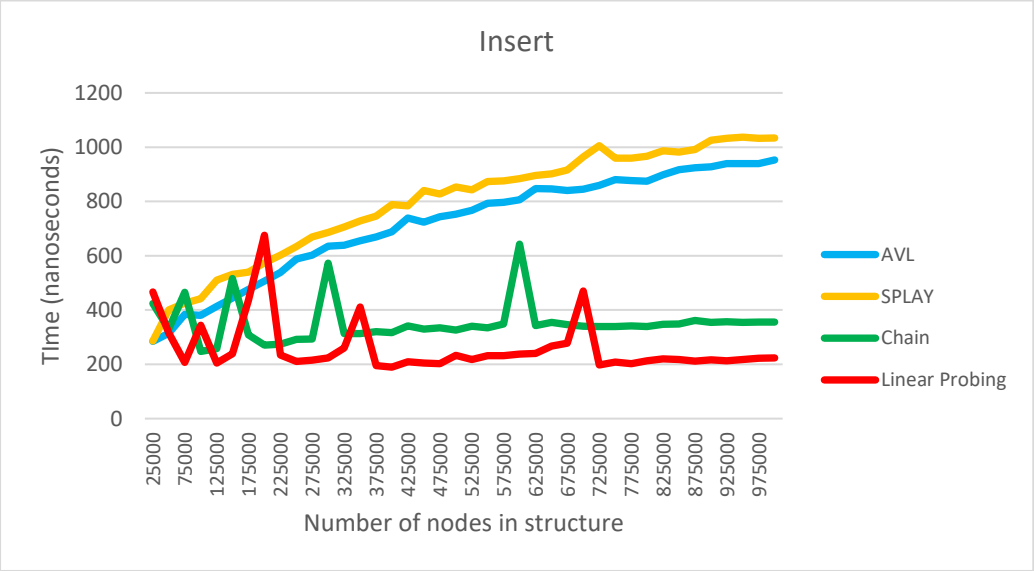
This section presents the final comparison of four data structures that have been implemented and tested. Specifically, this comparison evaluates the effectiveness of tables and trees in managing and retrieving data. The methodology adopted for the comparison is discussed in detail below.

Methodology

To facilitate an accurate comparison between tables and trees, the testing methodology was modified for tables. Instead of increasing the number of nodes in tables by 10,000 as previously done, the number of nodes was increased by 25,000. This modification allows for a more precise comparison between the two structures in terms of their ability to manage large amounts of data.

Comparison of function Insert

This section provides a graphical representation of the performance of different data structures in terms of insertion time for varying sizes of structures. The data was obtained through tests carried out on AVL Trees, Hash Tables with linear probing, and Binary Search Trees with splay balancing.



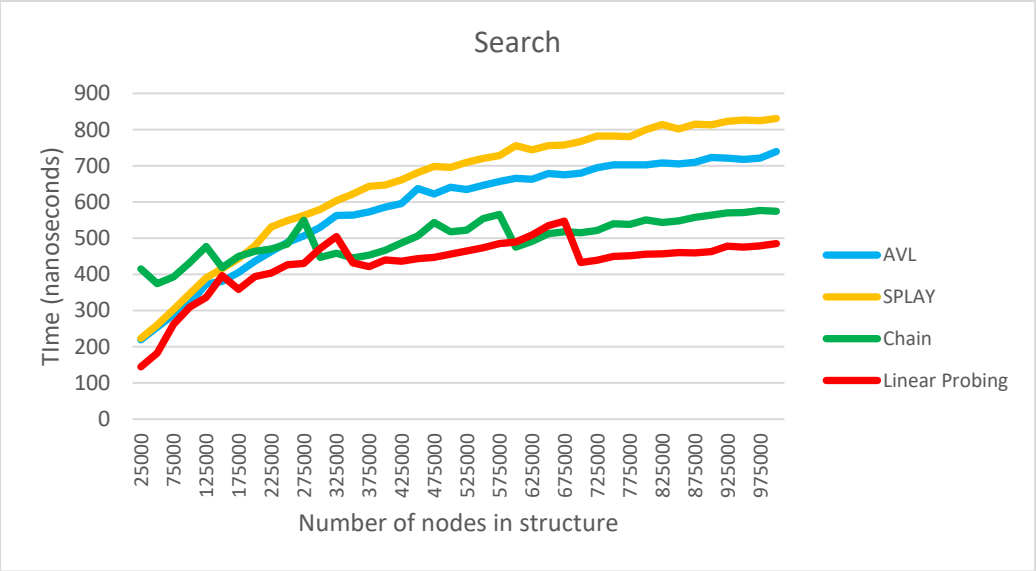
Graph 19.

The graphs display the time taken to insert one item in structures with different sizes, represented by the number of nodes. For instance, inserting an item in an AVL Tree with 500,000 nodes took 753.44 nanoseconds, while inserting an item in a Hash Table with linear probing of the same size took only 232.86 nanoseconds. The results indicate that the Hash Table with linear probing is the fastest structure for almost every structure size. Although there is no significant difference in smaller structures, the difference in performance becomes more pronounced as the size of the structures increases.

It is worth noting that both tables outperformed the trees, and Binary Search Tree with splay balancing was found to be the slowest in this test. It is important to note that the performance difference between one tree compared to another is negligible, but it becomes more apparent with an increasing structure size.

Comparison of function Search

This section provides an analysis of the search operation in different data structures. Although there are similarities between the search and insert operations, there are some notable differences. Similar to insert, linear probing provides the fastest search performance among the tested structures. However, the peaks in the graph look different for search operations, with smaller peaks in all four structures following the same logarithmic pattern. In contrast, the peaks in the insert graph were higher and more evident when the table was resized.

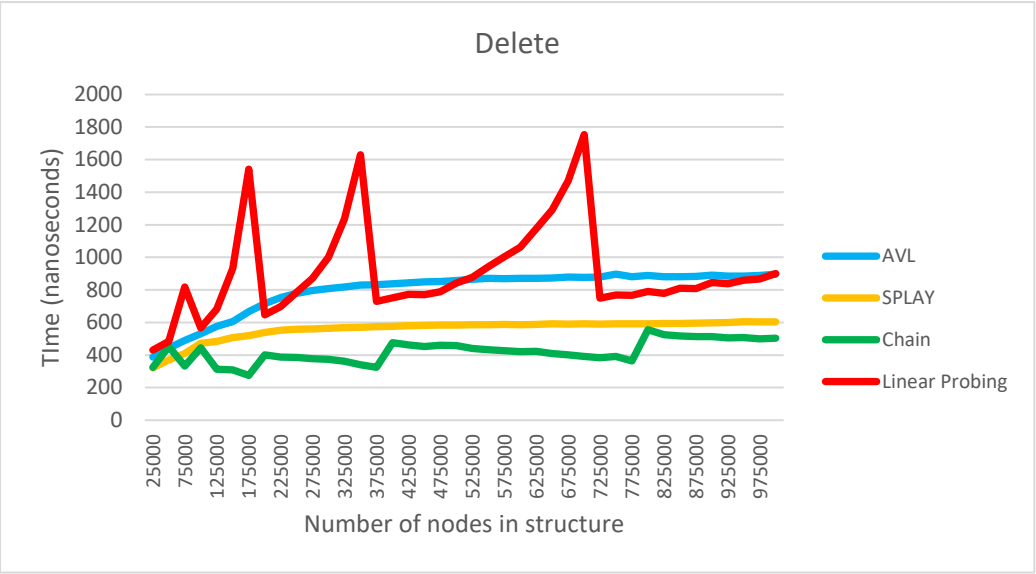


Graph 20.

For smaller datasets, all four structures exhibit comparable efficiency in search operations. However, as the size of the dataset increases, performance differences become more apparent. As with insert, linear probing remains the fastest structure, while AVL tree is the slowest. For instance, in a table with 900,000 nodes, one search operation took 462.749 nanoseconds, while in an AVL tree with 900,000 nodes, it took 813.32 nanoseconds.

Comparison of function Delete

This section provides an analysis of the delete operation in different data structures. Upon initial observation, it is evident that delete operation differs from both insert and search operations. Surprisingly, linear probing, which exhibited the fastest performance in both insert and search operations, was the slowest in delete operation. All four structures exhibit a similar pattern, with linear probing peaking significantly higher than the average of all four structures each time the table is resized. In contrast, both trees and hash tables with chaining exhibit logarithmic curves, while linear probing exhibits logarithmic curves with high peaks.



Graph 21.

In this operation, the fastest structure was found to be hash table with chaining. As with other operations, there were no significant differences in the time taken for each delete operation in smaller datasets. However, as the size of the dataset increased, performance differences became more apparent. For example, in a structure with 1,000,000 nodes, one delete operation took 503 nanoseconds for chaining, 603.80 nanoseconds for Splay, and roughly 900 nanoseconds for AVL tree and linear probing.

Conclusion

In conclusion, it is difficult to determine a single data structure that is best for all situations, datasets, and operations. Each structure has its advantages and disadvantages, and the optimal choice depends on specific use cases.

Based on the results of the experiments, it appears that the most suitable solution for most cases would be hash table with linear probing. However, if a tree structure is preferred, the optimal choice would be AVL Tree, which demonstrated faster performance than Splay in both insert and search operations.