Data structures and algorithms

1 assignment – Binary Search Trees, Hash Tables

Author: Marek Čederle

Instructor: MSc. Mirwais Ahmadzai

(Monday 11:00)

Theory

Introduction

Data structures are essential to computer science and computing, and selecting the best data structure can have a big impact on the program's effectiveness and speed. Implementation of BST (Binary search trees) like AVL tree are frequently used for finding and sorting tasks. However, Chaining and Open Addressing (linear probing) Hash Tables are well known for their quick search, insert, and delete actions.

The application of the insert, search, and delete functions on each of the previously mentioned data structures will be thoroughly explained in this documentation, along with information on their time requirements, benefits, and drawbacks. You will have a thorough understanding of these data structures by the conclusion of this documentation and be able to select the one that is best for your particular use case.

BST (Binary Search Tree)

BST is a binary tree data structure that sticks to the property that each node's left and right subtrees only contain nodes with values that are higher or less than their own value, respectively.

This characteristic of BSTs makes them effective for applications where searching is a common procedure and the data is frequently changing because we can search, insert, and delete nodes in O(log n) time complexity on average.

To implement the insert function, we first compare the new node's value with the root node's value. If it is less than the root node, we move to the left subtree and repeat the comparison. If it is greater than the root node, we move to the right subtree and repeat the comparison until we reach a leaf node with no child. We then insert the new node in that position.

To search for a node in a BST, we start from the root node and compare the value with the search key. If it is equal, we return the node, and if it is less than the search key, we move to the left subtree; otherwise, we move to the right subtree and continue the comparison until we find the node or reach a leaf node.

To delete a node, we first search for the node to be deleted and handle the deletion based on the number of children it has. If the node has no children, we delete it directly; if it has one child, we replace the node with its child; and if it has two children, we find its inorder successor and replace the node with it.

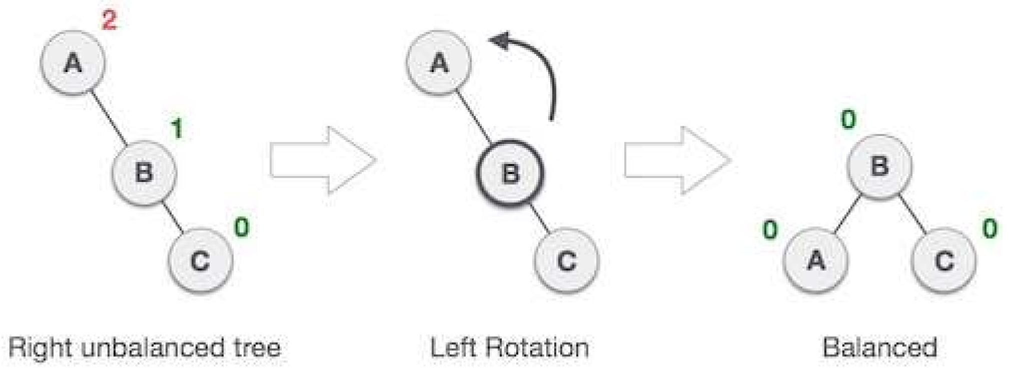
However, the time complexity of the BST operations heavily depends on the BST's shape. In the worst case, when the BST is skewed, the time complexity becomes O(n), which is the same as linear search. To overcome this limitation, AVL trees were introduced, which will be described next.

AVL

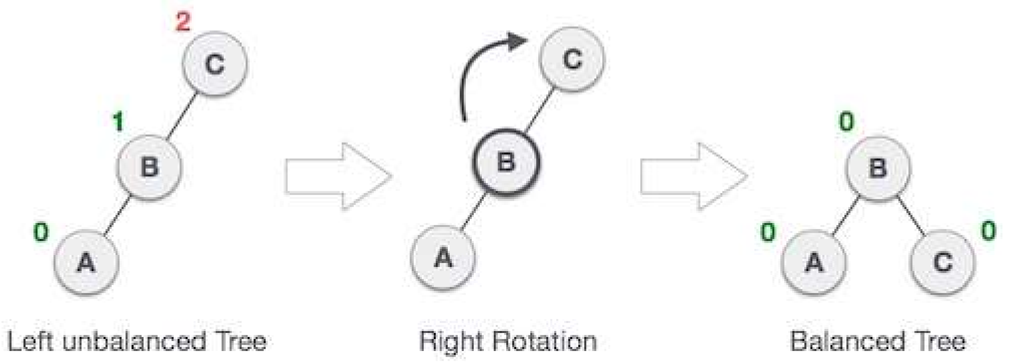
AVL tree is a self-balancing binary search tree that was introduced to overcome the worst-case time complexity of BSTs. In AVL trees, the height difference between the left and right subtrees of any node (also called the balance factor) is always kept less than or equal to 1.

To maintain this balance, AVL trees perform rotations on the nodes when the balance factor is violated. There are four types of rotations: left rotation, right rotation, left-right rotation, and right-left rotation. These rotations help in balancing the tree and maintaining the height difference.

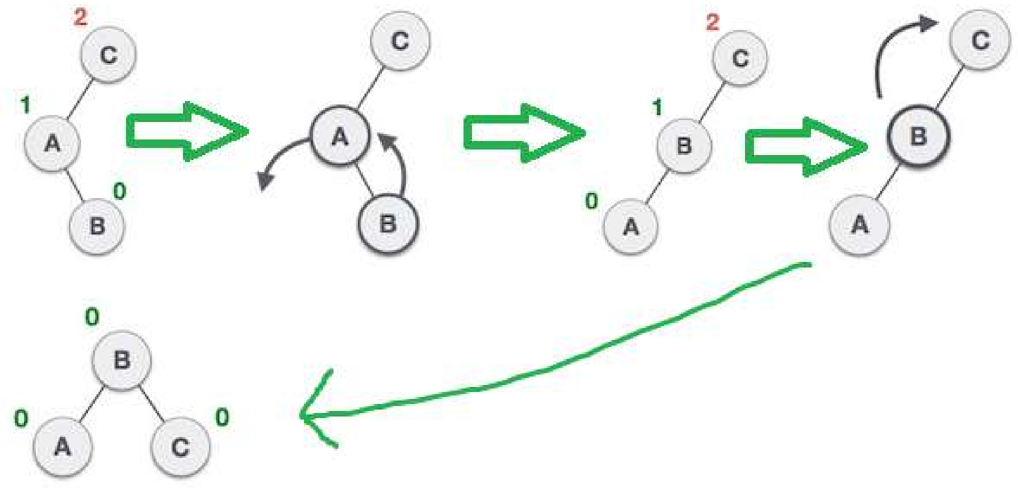
Left rotation:



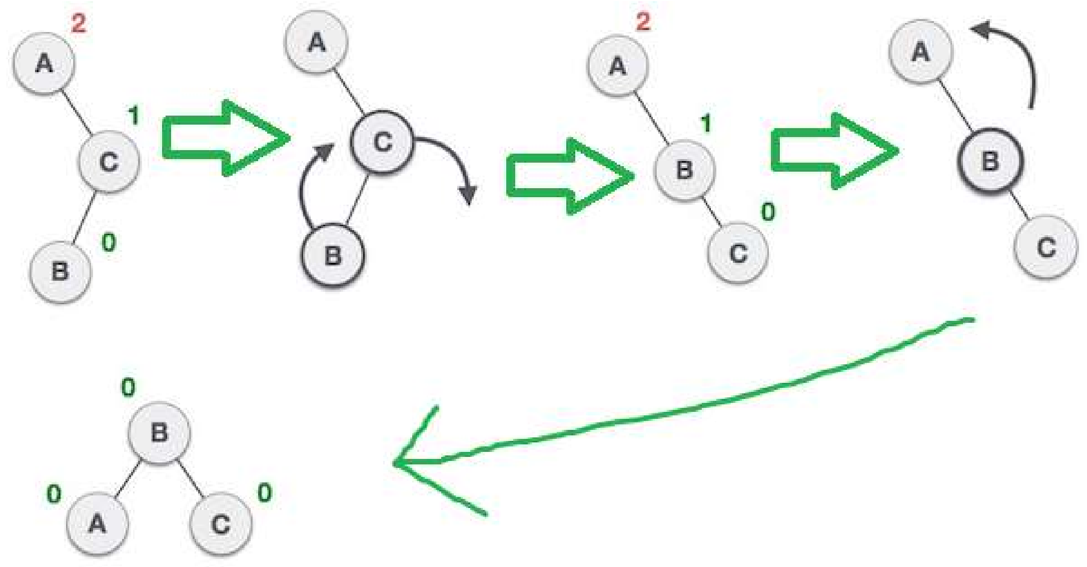
Right rotation:



Left-right rotation:



Right-Left rotation:



The insert, search, and delete operations in AVL trees have the same time complexity as that of BSTs, i.e., O(log n) on average. However, in the worst-case scenario, the time complexity of AVL trees is guaranteed to be O(log n), which is much better than the O(n) worst-case time complexity of BSTs.

The insert operation in AVL trees is similar to BSTs, with an additional step of checking the balance factor and performing rotations if necessary. The search and delete operations are also the same as that of BSTs.

AVL trees are widely used in applications where frequent insertions and deletions are required, and maintaining the balanced tree is crucial for efficient search and retrieval operations. However, the self-balancing property of AVL trees comes at the cost of additional memory overhead and complexity in the implementation compared to BSTs.

Hash Tables

Hash Tables are another popular data structure used for efficient searching, insertion, and deletion of elements. They store data in an array format, where each element or value has a unique key or hash function associated with it.

The hash function takes the key as input and returns the index in the array where the value is stored. Hash functions are designed to have a uniform distribution of values and to minimize the number of collisions (when two or more keys map to the same index) to achieve better performance.

The search, insert, and delete operations in hash tables have an average time complexity of O(1) and a worst-case time complexity of O(n) when there are many collisions. However, the time complexity can be significantly reduced by selecting the appropriate hash function, maintaining a load factor less than 1, and using a collision resolution technique that minimizes the number of collisions.

There are two common techniques for handling collisions in hash tables: Chaining and Open Addressing which we will describe next.

Chaining resolution of collisions

Each element in the hash table points to a linked list of elements that have the same hash value. When a collision occurs, the new element is inserted at the tail of the linked list because of checking duplicate keys. Chaining provides a simple way of handling collisions and is easy to implement.

To implement the Chaining technique, we first need to define a hash function that maps each key to a unique index in the hash table. Once we have the index, we can insert the element at the tail of the linked list pointed to by that index.

The search operation in Chaining also involves computing the hash function to get the index of the linked list, and then traversing the linked list to find the element with the matching key.

The delete operation requires finding the element to be deleted in the linked list, and then removing it from the linked list. To find the element, we perform search operation to get the index of the linked list and traverse the linked list until we find the element with the matching key and then we delete the node.

The time complexity of Chaining depends on the number of collisions and the length of the linked lists. In the average case, assuming the hash function distributes the keys uniformly, the time complexity of search, insert, and delete operations is O(1) plus the time required to traverse the linked list, which is proportional to the length of the list.

In the worst-case scenario, when all the keys map to the same index, the length of the linked list can be n, where n is the number of elements in the hash table. In this case, the time complexity of the search, insert, and delete operations is O(n).

Open addressing (Linear probing)

To implement Linear Probing, we need to define a hash function that maps each key to a unique index in the hash table. Once we have the index, we check if the slot is empty. If it is empty, we insert the new element into that slot. If it is not empty, we move to the next index and check again until we find an empty slot. This process is called probing.

The delete operation in Linear Probing requires finding the element to be deleted and marking the slot as deleted. To find the element, we perform search operation to get the index of the slot where the element should be located. We check if the element is in that slot. If it is not, we sequentially probe the next slots until we find the element or an empty slot. Once we find the element, we mark the slot as deleted to indicate that it is available for insertion in future operations. We remove the zombie nodes (marked as “deleted”) when we resize our table.

It's important to note that marking a slot as deleted is different from setting the slot to NULL. Marking a slot as deleted means that it is available for insertion, but we still need to continue probing for other elements that may be located after the deleted element.

The time complexity of Linear Probing depends on the number of collisions and the size of the hash table. In the average case, assuming the hash function distributes the keys uniformly, the time complexity of search, insert, and delete operations is O(1) plus the time required to probe the next slots. In the worst-case scenario, when the hash table is full, the time complexity of search, insert, and delete operations is O(n), where n is the size of the hash table.

My implementation

To implement these data structures in my project I these programs:

- IDE (integrated development environment) – VSCode (Visual Studio Code) version 1.76.2
- Compiler – g++.exe version 8.3.0
- Memory checker – valgrind.exe version 3.14.0 (my code is without memory leaks)
- Graphical representation of measured data - Google Sheets
- Documentation – Microsoft Word

In the next section I will describe only the crucial functions and classes. Other ones will be somewhat explained in the code or not if they are just basic functions.

BST – AVL

Node

This snippet of code defines the class for a binary search tree node using the AVL algorithm. The following public member variables belong to the "Node" class:

- "value": an integer value that represents the node's key
- "height" is an integer number that represents the node's height in the tree. "data" is a string value that indicates the node's related data.
- 'left' indicates a pointer to the left child node.
- "right": a pointing device to the appropriate child node

The class contains a constructor that accepts two inputs: a string of data and an integer called "value" for initializing the member variables. The constructor additionally initializes the left and right pointers to null and sets the node's "height" to 1.

The class additionally features a destructor that use the "delete" keyword to recursively remove the left and right child nodes. This is important for preventing memory leaks in the tree when nodes are removed or the entire tree is destroyed.

```cpp
class Node
{
public:
    int value;
    int height;
    string data;
    Node *left;
    Node *right;

    Node(int value, string data) /*constructor*/
    {
        this->value = value;
        this->data = data;
        this->height = 1;
        this->left = nullptr;
        this->right = nullptr;
    }
    ~Node() /*destructor*/
    {
        delete left;
        delete right;
    }
};
```

Right rotation

This code snippet demonstrates a function called "rotateRight" that rotates a binary search tree node to the right for a specified node. A pointer to the node that has to be rotated is passed as an input to the function, which then returns a pointer to the new root node.

Two new node pointers, "left" and "leftRight," are first created by the function. The input node's left child is indicated by "left," while the right child is indicated by "leftRight."

The links between the nodes are then updated by the program. The input node is made to be the "right" child of "left," and the "left" child of the input node is set to be "leftRight." By doing this, the subtree rooted at the input node is effectively rotated to the right.

Lastly, the code uses a helper function called "updateHeight" to update the "height" of the rotated nodes. A new subtree root is returned that contains the modified "left" and "node" nodes.

```
// function to rotate the tree to the right
Node *rotateRight(Node *node)
{
    Node *left = node->left;
    Node *leftRight = left->right;

    left->right = node;
    node->left = leftRight;

    node->height = updateHeight(node);
    left->height = updateHeight(left);

    return left;
}
```

Left rotation

Similar in concept to right rotation, but essentially mirrored, is left rotation.

Insert

This code adds a new node with a given value and data to a binary search tree. If the tree is empty, it creates a new node with the given value and data as the root node. Otherwise, it traverses the tree to find the appropriate location for the new node and inserts it there.

After insertion, the function checks the balance factor of the current node and performs necessary rotations to balance the tree. Finally, it returns a pointer to the new root node of the tree after the insertion and rotations.

```
// function to insert new node into the tree
Node *insert(Node *node, int value, string data)
{
    if (node == nullptr)
    {
        return new Node(value, data);
    }

    if (value < node->value)
    {
        node->left = insert(node->left, value, data);
    }
    else if (value > node->value)
    {
        node->right = insert(node->right, value, data);
    }
    else /*ignoring duplicates*/
    {
        return node;
    }

    /*balancing using rotations*/

    node->height = updateHeight(node);

    int balance = balanceFactor(node);

    if (balance > 1 && value < node->left->value)
    {
        return rotateRight(node);
    }

    if (balance < -1 && value > node->right->value)
    {
        return rotateLeft(node);
    }

    if (balance > 1 && value > node->left->value)
    {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }

    if (balance < -1 && value < node->right->value)
    {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}
```

Delete

This code deletes a node with a given value from a binary search tree. If the node is not found, it returns the original tree. If the node to be deleted has either no children or only one child, the node is simply deleted and its child, if any, takes its place. If the node to be deleted has two children, the node's successor (the node with the smallest value in its right subtree) is found and its value and data are copied into the node to be deleted. Then, the successor node is deleted.

After deletion, the function checks the balance factor of the current node and performs necessary rotations to balance the tree. Finally, it returns a pointer to the new root node of the tree after the deletion and rotations.

```
    // function to delete node from the tree
    Node *deleteNode(Node *node, int value)
    {
        if (node == nullptr)
        {
            return node;
        }

        if (value < node->value)
        {
            node->left = deleteNode(node->left, value);
        }
        else if (value > node->value)
        {
            node->right = deleteNode(node->right, value);
        }
        else
        {
            if (node->left == nullptr || node->right == nullptr)
            {
                Node *temp = node->left ? node->left : node->right; /*if
left not null then left, else right*/

                if (temp == nullptr)
                {
                    temp = node;
                    node = nullptr;
                }
                else
                    *node = *temp;
                delete temp;
            }
            else
            {
                Node *temp = minValueNode(node->right);

                node->value = temp->value;
                node->data = temp->data;

                node->right = deleteNode(node->right, temp->value);
            }
        }

        /*balancing using rotations*/

        if (node == nullptr)
            return node;

        node->height = updateHeight(node);

        int balance = balanceFactor(node);

        if (balance > 1 && balanceFactor(node->left) >= 0)
        {
            return rotateRight(node);
        }

        if (balance > 1 && balanceFactor(node->left) < 0)
        {
            node->left = rotateLeft(node->left);
            return rotateRight(node);
        }

        if (balance < -1 && balanceFactor(node->right) <= 0)
        {
            return rotateLeft(node);
        }

        if (balance < -1 && balanceFactor(node->right) > 0)
        {
            node->right = rotateRight(node->right);
            return rotateLeft(node);
        }

        return node;
    }
```

Search

This function is used to search for a node in a binary search tree based on its value. The function takes two arguments: a pointer to the root node of the tree, and the value to search for.

If the root node is null or its value is equal to the search value, the function returns the root node. Otherwise, if the search value is greater than the value of the root node, the function recursively searches the right subtree of the root node. If the search value is less than the value of the root node, the function recursively searches the left subtree of the root node.

The function returns a pointer to the node with the search value, or null if the node is not found in the tree.

```
    // function to search for a node in the tree
    Node *search(Node *node, int value)
    {
        if (node == nullptr || node->value == value)
        {
            return node;
        }

        if (node->value < value)
        {
            return search(node->right, value);
        }

        return search(node->left, value);
    }
```

Hash Table – Chaining

Node

NodeHashChain is a struct used to represent a node in a hash table that uses chaining to handle collisions. It contains three members:

- “Data” is an integer that represents the data associated with the node.
- “Key” is a string that represents the key associated with the node. The key is used to determine the position of the node in the hash table.
- “Next” is a pointer to the next node in the linked list that represents the chain at the index corresponding to the hash of the key. If there are multiple nodes with the same hash value (when collision occurs), they are stored in a linked list that starts at that index.

```
struct NodeHashChain
{
public:
    int data;
    string key;
    NodeHashChain *next;
    NodeHashChain(int data, string key) /*constructor*/
    {
        this->data = data;
        this->key = key;
        this->next = nullptr;
    }
};
```

Hash

The function uses the djb2 hash function to calculate the hash value of the input key. The hash value is calculated by iterating over the characters of the key and performing bitwise left shifts and additions on an initial hash value of 5381. Finally, the hash value is computed by performing a modulo operation with the size of the hash table.

```
    // function to get the hash value of the key
    // djb2 hash function for string keys
    int hash(string key)
    {
        int hashValue = 5381;
        for (size_t i = 0; i < key.length(); i++)
        {
            hashValue = ((hashValue << 5) + hashValue) + key[i];
        }
        return hashValue % size;
    }
```

Insert

This code defines a function called "insert" that inserts a key-value pair into a hash table using chaining method. First, the function calculates the hash value of the key using a hash function. If the corresponding index in the hash table is empty, a new "NodeHashChain" object is created with the provided key-value pair and assigned to the index. If not, the function traverses the linked list at that index to find the last node, and then appends a new "NodeHashChain" object to it with the provided key-value pair.

The function also updates the count of elements in the hash table and checks the load factor. If the load factor exceeds a threshold of 0.7, the function resizes the hash table by doubling its capacity. If the hash table is already in the process of being resized, the function simply returns without performing any additional resizing.

```
    // function to insert the key and data into the hash table using
chaining method
    void insert(string key, int data)
    {

        int hashValue = hash(key);

        // if the hash value is empty, insert the key and data
        if (table[hashValue] == nullptr)
        {
            table[hashValue] = new NodeHashChain(data, key);
        }
        else
        {
            // if the hash value is not empty, insert the key and data at
the end of the linked list
            NodeHashChain *temp = table[hashValue];
            while (temp->next != nullptr)
            {
                temp = temp->next;
            }
            temp->next = new NodeHashChain(data, key);
        }

        if (resizing)
        {
            return;
        }
        count++;
        if (loadFactor() > 0.7)
        {
            resize(size * 2);
        }
    }
```

Resize

This is a function to resize the hash table by increasing its capacity when the load factor (the ratio of the number of stored elements to the size of the hash table) exceeds 0.7. It creates a new hash table with the specified capacity, rehashes the data in the old hash table, and then deletes the old hash table. The resizing flag is used to prevent the insertion of new elements during the resizing process, as this could cause errors.

```
    // function to resize the hash table
    void resize(int capacity)
    {
        resizing = true;
        // backup data and keys
        vector<NodeHashChain *> temp = table;
        int tempSize = size;
        // resize the hash table
        table = vector<NodeHashChain *>(capacity);
        size = capacity;

        // rehash the data
        for (int i = 0; i < tempSize; i++)
        {
            NodeHashChain *temp2 = temp[i];
            while (temp2 != nullptr)
            {
                insert(temp2->key, temp2->data);
                temp2 = temp2->next;
            }
        }

        // delete the old hash table
        for (int i = 0; i < tempSize; i++)
        {
            NodeHashChain *temp2 = temp[i];
            while (temp2 != nullptr)
            {
                NodeHashChain *temp3 = temp2;
                temp2 = temp2->next;
                delete temp3;
            }
        }

        resizing = false;
    }
```

Search

This function searches for a given key in the hash table. It first computes the hash value of the key using the hash() function, and then searches for the key in the linked list at that hash value index. If the key is found, it returns the node containing both the key and data. If the key is not found, it returns nullptr. The function also keeps track of the number of times it successfully finds a key using the variable "found" and the number of times it fails to find a key using the variable "notFound".

```
    // function to search the key in the hash table
    NodeHashChain *search(string key)
    {
        int hashValue = hash(key);
        NodeHashChain *temp = table[hashValue];
        while (temp != nullptr)
        {
            if (temp->key == key)
            {
                found++;
                return temp;
            }
            temp = temp->next;
        }
        notFound++;
        return nullptr;
    }
```

Delete

The function first computes the hash value of the key using the hash function. It then traverses the linked list at the computed hash value to find the node with the given key. If the key is found, the function removes the node from the linked list and adjusts the count of elements in the hash table.

If the load factor of the hash table after deleting the key becomes less than 0.25, the function calls the resize function to resize the hash table to half of its current size.

```
    // function to delete the key from the hash table using chaining method
    void deleteKey(string key)
    {
        int hashValue = hash(key);
        NodeHashChain *temp = table[hashValue];
        NodeHashChain *prev = nullptr;
        while (temp != nullptr)
        {
            if (temp->key == key)
            {
                if (prev == nullptr)
                {
                    table[hashValue] = temp->next;
                }
                else
                {
                    prev->next = temp->next;
                }
                count--;
                delete temp;
                if (loadFactor() < 0.25)
                {
                    resize(size / 2);
                }
                return;
            }
            prev = temp;
            temp = temp->next;
        }
    }
```

Hash Table – Open addressing (linear probing)

Node

The NodeHashOpen struct has three member variables:

- “Data” which stores an integer value associated with the key
- “Key” which is a string that serves as the key to access the associated data
- A constructor that takes in the data and key values and initializes the corresponding member variables.

```
struct NodeHashOpen
{
public:
    int data;
    string key;

    NodeHashOpen(int data, string key) /*constructor*/
    {
        this->data = data;
        this->key = key;
    }
};
```

Hash

This is a hash function called "Jenkins' hash function" used for linear probing in hash tables. It takes a string key as input and returns an integer hash value. It calculates the hash value by iterating through each character in the key, adding its ASCII value to the hash, and performing bitwise operations to mix up the bits in the hash value. Finally, it returns the hash value modulo the size of the hash table to ensure it falls within the range of indices for the table.

```
    // Jenskin's hash function for linear probing
    int hash(string key)
    {
        int hash = 0;
        for (size_t i = 0; i < key.length(); i++)
        {
            hash += key[i];
            hash += (hash << 10);
            hash ^= (hash >> 6);
        }
        hash += (hash << 3);
        hash ^= (hash >> 11);
        hash += (hash << 15);
        return abs(hash % size);
    }
```

Insert

This function inserts a new node into the hash table using linear probing. If the index is empty, the new node is inserted at that index. If the index already contains a deleted node, the new node replaces the deleted node. If the index is not empty, the function uses linear probing to find the next available index and inserts the new node there. The function also checks if the load factor is greater than 0.75 and resizes the hash table accordingly.

```
    // insert a new node into the hash table
    void insert(string key, int data)
    {
        int index = hash(key);
        if (table[index] == nullptr)
        {
            table[index] = new NodeHashOpen(data, key);
        }
        else if (table[index]->key == "deleted")
        {
            table[index]->key = key;
            table[index]->data = data;
        }
        else
        {
            int i = 1;
            while (table[index] != nullptr)
            {
                index = (index + i) % size;
                i++;
            }
            table[index] = new NodeHashOpen(data, key);
        }

        if (resizing)
        {
            return;
        }
        count++;
        if (loadFactor() > 0.75)
        {
            resize(size * 2);
        }
    }
```

Resize

This function resizes the hash table by creating a new table with a new size and rehashing all of the data from the old table into the new table. The old table is then deleted. If any nodes were deleted in the old table, they are not reinserted into the new table. This function also sets the resizing flag to true during the resizing process to prevent other functions from modifying the hash table.

```
    // resize the hash table
    void resize(int newSize)
    {
        resizing = true;
        vector<NodeHashOpen *> oldTable = table;
        int oldSize = size;
        size = newSize;
        table = vector<NodeHashOpen *>(size);

        for (int i = 0; i < oldSize; i++)
        {
            if (oldTable[i] != nullptr && oldTable[i]->key != "deleted")
            {
                insert(oldTable[i]->key, oldTable[i]->data);
            }
        }

        // delete old table
        for (int i = 0; i < oldSize; i++)
        {
            if (oldTable[i] != nullptr)
            {
                delete oldTable[i];
            }
        }

        resizing = false;
    }
```

Search

This code searches for a string key in a hash table using open addressing for handling collisions. It returns a pointer to the corresponding object if found, or null if not found. It probes through the table by incrementing the index with a variable until it either finds the key or reaches an empty slot.

```
    // search for a node in the hash table
    NodeHashOpen *search(string key)
    {
        int index = hash(key);
        int i = 1;
        while (table[index] != nullptr)
        {
            if (table[index]->key == key)
            {
                found++;
                return table[index];
            }
            index = (index + i) % size;
            i++;
        }
        notFound++;
        return nullptr;
    }
```

Delete

This code deletes a key-value pair from a hash table that uses open addressing for handling collisions. It first searches for the key and, if found, sets its value to "deleted" and decrements the count of pairs. It then checks if the load factor is less than 0.25 and resizes the table if needed. If the key is not found, it prints a message to the console.

```
// delete a node from the hash table
void deleteKey(string key)
{
    int index = hash(key);
    int i = 1;

    if (search(key) != nullptr)
    {
        while (table[index]->key != key)
        {
            index = (index + i) % size;
            i++;
        }
        table[index]->key = "deleted";
        table[index]->data = 0;
        count--;
        if (loadFactor() < 0.25)
        {
            resize(size / 2);
        }
    }
    else
    {
        cout << "key not found:" << key << endl;
    }
}
```

Dataset generator

This program generates dataset to file “dataset_1M.txt”with unique keys and unique values which are randomly sorted. Integers consist of numbers between 1 and 1000000, strings consist of 5 characters from “aaaaa” to “zzzzz”. In AVL, integers are used as keys and strings are used as data. In hash tables it is reversed. The testing program then draws input data from this file.

Testing

The testing code includes a class called Test, which I used to test the implementation of three data structures: AVL tree, chaining hash table, and open addressing hash table with linear probing. It has several member functions within the Test class for inserting, searching, and deleting data from these data structures. Each of these member functions take a file pointer f and the number of nodes to insert/search/delete as input.

To test the implementation of these data structures, I created an input file that contains a list of nodes to insert, search, or delete. This input file was formatted according to the specifications of each member function. For example, "insertAVL" expect input in the form of pairs of integers and strings.

Once I had created the input file, I could run the various member functions of Test with this input file and the desired number of nodes to insert/search/delete. The member functions executed the corresponding operations on the data structures and returned the time it took to perform these operations in nanoseconds.

I could then use these execution times to compare the performance of the different data structures for various operations (insertion, searching, and deletion) and different numbers of nodes. This comparison helped me determine which data structure was the most efficient for my particular use case.

It is important to note that the specific tests I ran and the results I obtained depended on the input files I created and the hardware on which I ran my tests.

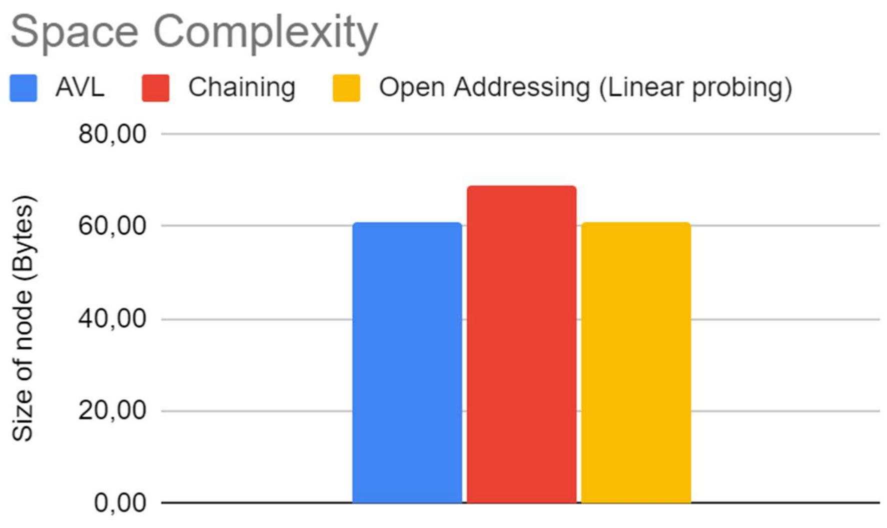
Results of the tests are shown in Comparison section.

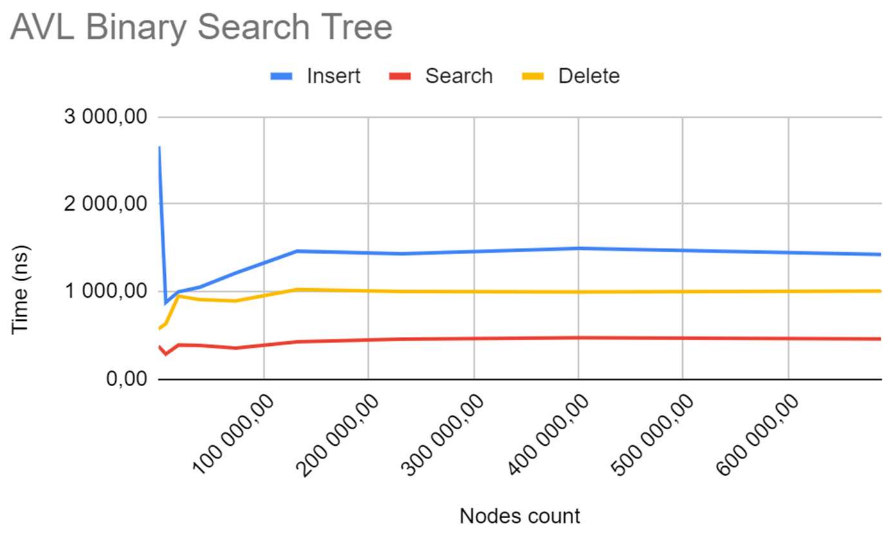
Comparison

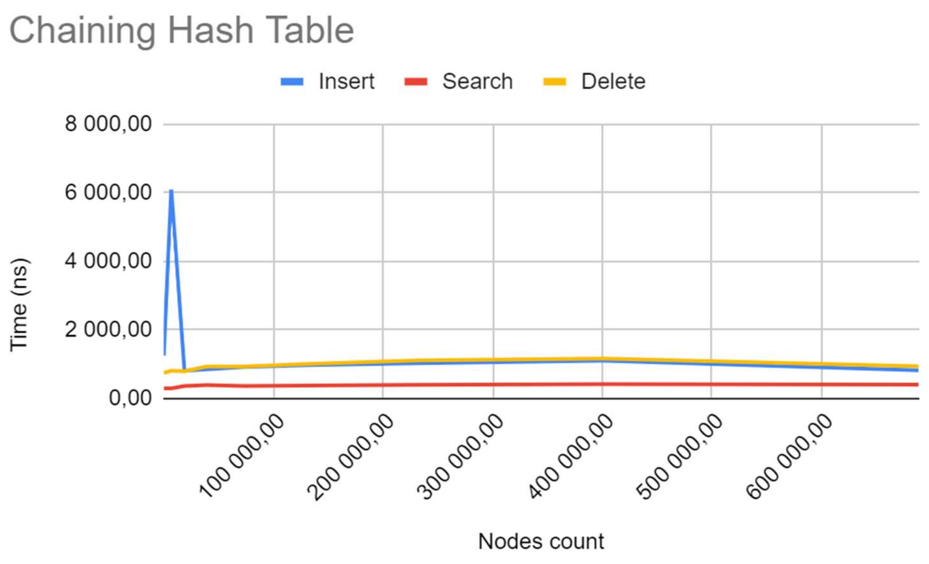
Here are graphs and table comparing time and space complexity of the data structures:

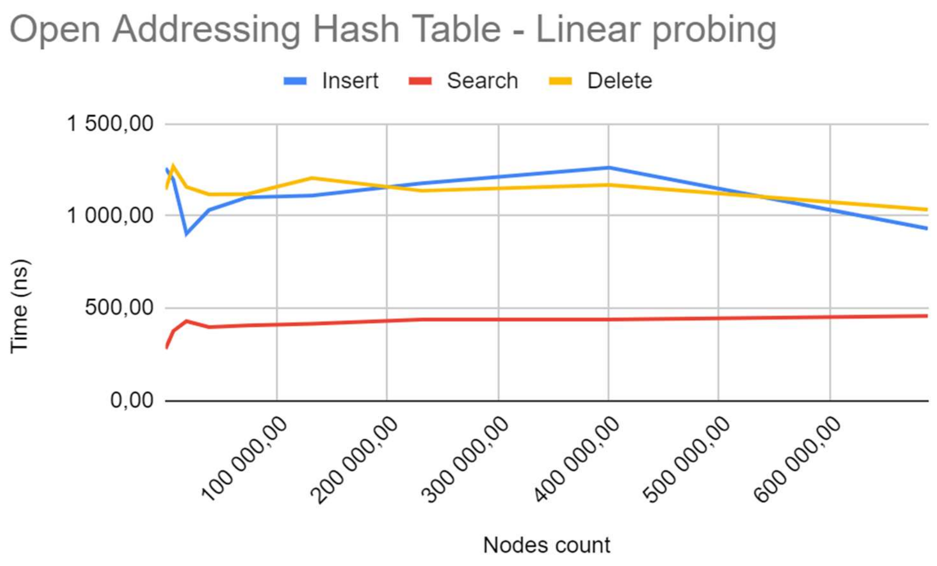
Spikes on the chaining are probably caused because of resizing the hash table. As you can see from the graphs AVL has overall the slowest insert and search operation but fast delete operation. Chaining hash table and open addressing hash table with linear probing have lower time complexity compared to AVL tree for these operations. However, as the number of nodes increases, the space required by the hash tables also increases significantly.

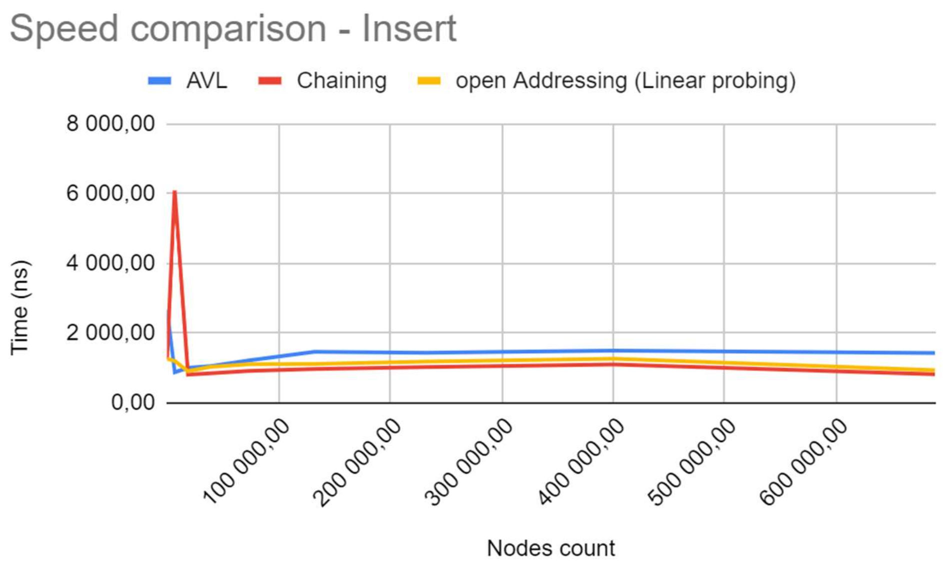
Overall, the performance of each data structure depends on the specific use case and the number of nodes to be stored. Therefore, it is important to carefully consider the trade-offs between time complexity and space requirements before selecting a data structure for a particular use case.

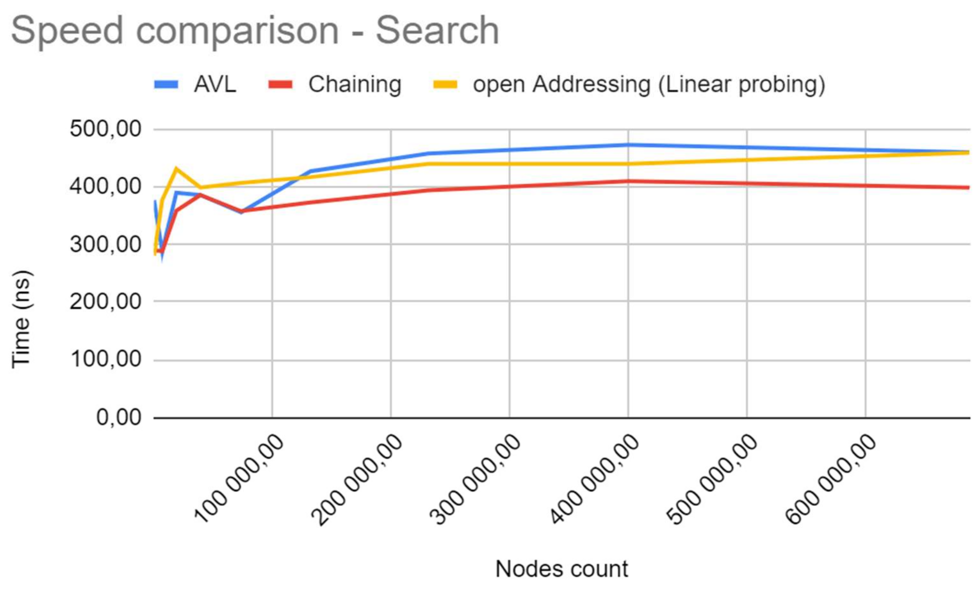


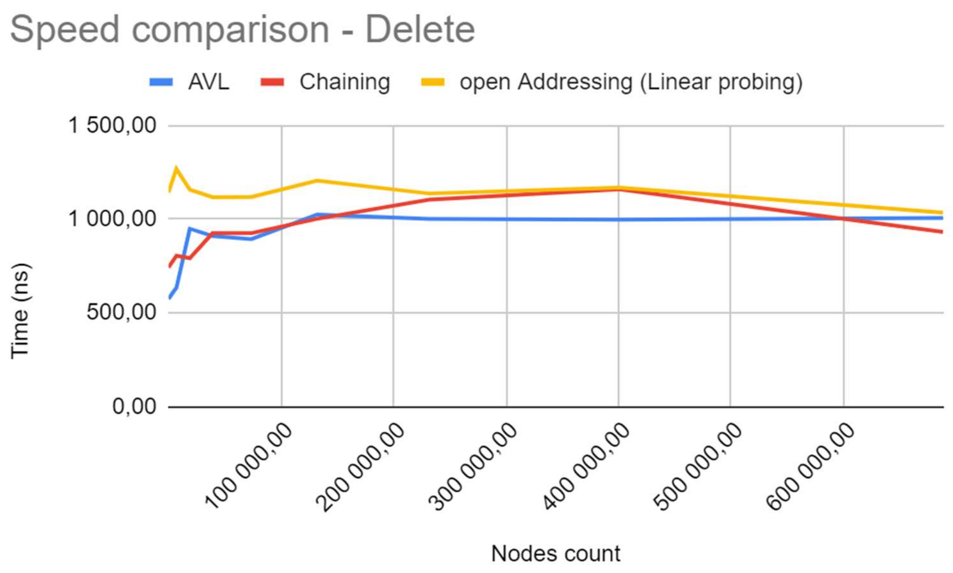


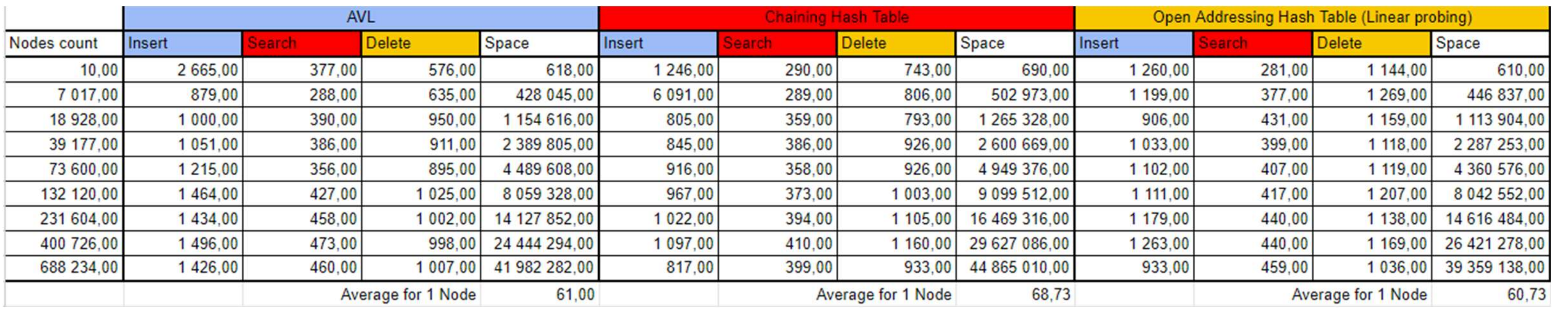

Chaining Hash Table
Insert
Search
Delete
8 000,00
6 000,00
4 000,00
2 000,00
0,00
Time (ns)
100 000,00
200 000,00
300 000,00
400 000,00
500 000,00
600 000,00
Nodes count


Open Addressing Hash Table - Linear probing
Insert
Search
Delete
1 500,00
1 000,00
500,00
0,00
Time (ns)
100 000,00
200 000,00
300 000,00
400 000,00
500 000,00
600 000,00
Nodes count


Speed comparison - Insert
AVL
Chaining
open Addressing (Linear probing)
8 000,00
6 000,00
4 000,00
2 000,00
0,00
Time (ns)
100 000,00
200 000,00
300 000,00
400 000,00
500 000,00
600 000,00
Nodes count







	AVL				Chaining Hash Table				Open Addressing Hash Table (Linear probing)			
Nodes count	Insert	Search	Delete	Space	Insert	Search	Delete	Space	Insert	Search	Delete	Space
10,00	2 665,00	377,00	576,00	618,00	1 246,00	290,00	743,00	690,00	1 260,00	281,00	1 144,00	610,00
7 017,00	879,00	288,00	635,00	428 045,00	6 091,00	289,00	806,00	502 973,00	1 199,00	377,00	1 269,00	446 837,00
18 928,00	1 000,00	390,00	950,00	1 154 616,00	805,00	359,00	793,00	1 265 328,00	906,00	431,00	1 159,00	1 113 904,00
39 177,00	1 051,00	386,00	911,00	2 389 805,00	845,00	386,00	926,00	2 600 669,00	1 033,00	399,00	1 118,00	2 287 253,00
73 600,00	1 215,00	356,00	895,00	4 489 608,00	916,00	358,00	926,00	4 949 376,00	1 102,00	407,00	1 119,00	4 360 576,00
132 120,00	1 464,00	427,00	1 025,00	8 059 328,00	967,00	373,00	1 003,00	9 099 512,00	1 111,00	417,00	1 207,00	8 042 552,00
231 604,00	1 434,00	458,00	1 002,00	14 127 852,00	1 022,00	394,00	1 105,00	16 469 316,00	1 179,00	440,00	1 138,00	14 616 484,00
400 726,00	1 496,00	473,00	998,00	24 444 294,00	1 097,00	410,00	1 160,00	29 627 086,00	1 263,00	440,00	1 169,00	26 421 278,00
688 234,00	1 426,00	460,00	1 007,00	41 982 282,00	817,00	399,00	933,00	44 865 010,00	933,00	459,00	1 036,00	39 359 138,00
			Average for 1 Node	61,00			Average for 1 Node	68,73			Average for 1 Node	60,73

Conclusion

In conclusion, my project on data structures and algorithms has allowed us to explore and compare the efficiency of several important data structures for organizing and accessing data: binary search trees, hash tables with chaining, and hash tables with open addressing and linear probing. Through my implementation and testing of these data structures, I have gained valuable insights into their relative strengths and weaknesses in terms of insertion, searching, and deletion times, as well as memory usage.

In particular, I found that binary search trees can, in the best case, provide excellent search times, but, in the worst case, can become highly unbalanced and inefficient, demanding careful balancing methods to maintain performance. Hash tables with chaining, on the other hand, can handle large amounts of data and are simple to apply, but their performance may suffer if hash collisions happen frequently. Last but not least, hash tables with open addressing and linear probing provide a trade-off between performance and memory usage as well as the capability to effectively manage collisions.

Generally, the type of data structure chosen relies on the particular use case and the properties of the stored data. We can decide which structure to utilize for the best performance by carefully examining our data and taking into account the tradeoffs of each structure. I learned significant skills for future programming projects in the implementation and testing of these crucial data structures that I used in this project.

References

https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm

https://www.programiz.com/dsa/hash-table

https://en.wikipedia.org/wiki/Hash_table

https://en.wikipedia.org/wiki/AVL_tree