CSPB 2270 Exam 2 Study Guide

BTree

Overview

A B-tree is a self-balancing tree data structure that maintains sorted data and is designed to efficiently handle large amounts of data and support fast insertions, deletions, and searches. It consists of a root node, internal nodes with multiple child pointers, and leaf nodes that store the actual data. The key feature of a B-tree is its ability to keep the tree balanced by ensuring that all leaf nodes are at the same level, which optimizes search operations. B-trees are commonly used in database systems and file systems where they provide efficient access and management of data in blocks or pages, reducing the number of disk reads and writes, and thereby improving overall performance.

Terminology

A B-tree is a self-balancing tree data structure that uses several key terms to describe its properties and operations. The B-tree is characterized by its 'order,' denoted as 'm,' which specifies the maximum number of keys a node can hold. Nodes in a B-tree can be of various types, such as the 'root node' representing the topmost node, 'internal nodes' having keys and corresponding child pointers, and 'leaf nodes' containing the actual data. The B-tree maintains the sorted order of keys within each node, ensuring that keys from left to right are in non-decreasing order. To handle insertions and deletions efficiently, the B-tree employs 'node splitting' to create two nodes when a node is full and 'node merging' when nodes become underflowed. The B-tree guarantees balance, where all leaf nodes have the same depth, and provides essential functions like search, insertion, deletion, and traversal to manage and manipulate data efficiently. With its balanced structure and logarithmic time complexity for search and modification operations, the B-tree is widely used in various applications involving large datasets, such as databases and file systems.

Structure

A B-tree is a self-balancing tree data structure that maintains a sorted collection of keys and efficiently organizes large amounts of data. The B-tree consists of nodes, including the root node, internal nodes, and leaf nodes. Each node has a fixed capacity and holds a certain number of keys, arranged in non-decreasing order from left to right. Internal nodes have corresponding child pointers, which direct to their child subtrees. All leaf nodes, which store the actual data, are at the same level, ensuring a balanced tree structure. When a node becomes full upon inserting a new key, it splits into two nodes, and the middle key moves up to the parent node. Conversely, when a node becomes empty upon deleting a key, it may merge with its adjacent sibling node. B-trees guarantee that all leaf nodes have the same depth, resulting in efficient search, insertion, and deletion operations. These properties make B-trees well-suited for applications like database systems and file systems, where they provide rapid data access and management, particularly when dealing with large datasets.

The following is a list of functions that are typically required for building a B-tree:

- **Deletion** - A function to remove a key from the B-tree, ensuring that the tree remains balanced and adheres to the B-tree rules. This function is usually recursive as it involves traversing the B-tree recursively to find the key's location and then handle node merging and underflow cases.
- **Initialization** - A function to create and initialize an empty B-tree.
- **Insertion** - A function to insert a new key into the B-tree while maintaining the B-tree properties, such as balanced nodes and sorted keys. This function is often recursive as it requires traversing the B-tree recursively to find the appropriate position for insertion and handle node splitting if necessary.
- **Merge Nodes** - A function to merge two nodes into one when they become underflowed (have less than $(m/2)$ keys) after deletion. This function can be recursive if the underflow propagates up the tree. The function may recursively call itself to merge parent nodes until the tree is balanced again.
- **Minimum & Maximum Key Retrieval** - Functions to find the minimum and maximum keys present in the B-tree.
- **Printing** - Functions to print the B-tree in a visually understandable format, helping with debugging and visualization. Printing functions can be recursive as they involve traversing the tree to display its structure and keys.
- **Search** - A function to search for a specific key in the B-tree, returning either a pointer to the node containing the key or indicating that the key is not present. This function is typically recursive as it involves traversing the B-tree recursively by calling itself on child nodes until the desired key is found or deemed not present.

- **Split Node** - A function to split a node into two when it becomes full (has $(m-1)$ keys) during insertion. This function can be recursive if the parent node also needs to split due to overflow. The function may recursively call itself to handle the cascading split of parent nodes.
- **Successor & Predecessor** - Functions to find the successor (the smallest key greater than a given key) and predecessor (the largest key smaller than a given key) of a given key. These functions may be recursive, especially if the current node does not have the requested key, and they need to traverse the tree recursively.
- **Traversal** - Functions to traverse the B-tree in various orders, such as in-order, pre-order, and post-order, for processing and displaying the keys. Traversal functions are usually implemented using recursion, as they involve visiting nodes and calling themselves on child nodes.

Efficiency

B-trees offer an efficient runtime complexity for various operations due to their balanced structure. The average and worst-case time complexities for search, insertion, and deletion operations in a B-tree are all $\mathcal{O}(\log(n))$, where n is the number of keys in the B-tree. This logarithmic complexity arises from the B-tree's ability to maintain a balanced height, ensuring that the number of keys per node remains within a constant range. As a result, B-trees are well-suited for managing large datasets, as their logarithmic time complexities enable fast data retrieval and manipulation even with substantial amounts of data. The B-tree's balanced nature makes it a popular data structure in database systems, file systems, and other applications that require efficient storage and retrieval of sorted data.

Operation	Best Case $\Omega(n)$	Average Case $\Theta(n)$	Worst Case $\mathcal{O}(n)$
Deletion	$\Omega(1)$	$\Theta(\log(n))$	$\mathcal{O}(\log(n))$
Insertion	$\Omega(1)$	$\Theta(\log(n))$	$\mathcal{O}(\log(n))$
Search	$\Omega(1)$	$\Theta(\log(n))$	$\mathcal{O}(\log(n))$
Traversal	$\Omega(n)$	$\Theta(n)$	$\mathcal{O}(n)$

Deletion

Deletion in a B-tree is the process of removing a key from the B-tree while maintaining its balanced and sorted properties. When deleting a key, the function first searches for the key in the B-tree. If the key is found in an internal node, it is replaced by its in-order predecessor or successor from the left or right subtree, respectively. If the key is found in a leaf node, it can be directly removed. After deletion, the B-tree might experience underflow in nodes, where a node has fewer keys than the minimum required. To restore balance, the function may perform node merging or redistribution with adjacent sibling nodes. Deletion in a B-tree ensures that the tree remains balanced and efficient in managing data.

Deletion Algorithm

Below is an example of the deletion algorithm.

```
procedure delete_key(B-tree node, key_to_delete)
    // Search for the key to delete
    position <- find_position_in_node(node, key_to_delete)

    if node is a leaf node
        // Key is found in a leaf node, directly remove it
        remove_key_from_node(node, position)

    else
        // Key is found in an internal node
        predecessor <- find_predecessor(node, position)
        predecessor_key <- predecessor's rightmost key
        replace key_to_delete with predecessor_key

        // Recursively delete the predecessor_key from the child node
        delete_key(predecessor, predecessor_key)

    end if

    // Check if underflow occurs in the node
    if node is underflowed
        sibling <- find_sibling(node)
        if sibling can spare a key
```

```
            redistribute_keys(node, sibling)
        else
            merge_nodes(node, sibling)
            delete_key(node's parent, key that caused the merge)
        end if
    end if
end procedure
```

Initialization

The initialization algorithm for a B-tree involves creating and setting up the data structure for an empty B-tree. It typically includes allocating memory for the root node, initializing its properties (e.g., setting the number of keys to 0 and marking it as a leaf node), and initializing any other necessary variables or data structures used to manage the B-tree. This algorithm is essential to ensure the B-tree starts with a valid state and can be ready to store and manage keys efficiently when further operations are performed.

Initialization Algorithm

Below is an example of the initialization algorithm.

```
procedure initialize_B_tree(order)
    root <- create_new_node()  // Create a new node
    root.is_leaf <- true       // Mark the root node as a leaf node
    root.num_keys <- 0         // Set the number of keys in the root node to 0
    root.order <- order        // Set the order of the B-tree

    return root                // Return the initialized root node
end procedure
```

Insertion

The insertion algorithm for a B-tree involves inserting a new key into the B-tree while preserving its balanced and sorted properties. The algorithm starts by searching for the appropriate leaf node where the key should be inserted. If the leaf node has space for the new key, it is simply added in a sorted manner. However, if the node is full, it is split into two nodes, and the middle key is moved up to the parent node. The process may cascade up the tree, potentially causing further splits until a node with available space is found or creating a new root node if the current root was split. The insertion algorithm ensures that the B-tree remains balanced and efficiently manages data even with frequent insertions.

Insertion Algorithm

Below is an example of the insertion algorithm.

```
procedure insert_key(B-tree node, key_to_insert)
    if node is a leaf node
        // Insert the key into the current leaf node
        position <- find_position_in_node(node, key_to_insert)
        insert_key_at_position(node, key_to_insert, position)

        // Check if the node is full and handle splitting if necessary
        if node has (order - 1) keys
            split_node(node)
        end if
    else
        // Traverse to the appropriate child node
        child_node <- find_child_node(node, key_to_insert)
        insert_key(child_node, key_to_insert)

        // Check if the child node split and update parent node if necessary
        if child_node has (order - 1) keys
            split_node(child_node)
        end if
```

```
    end if
end procedure
```

Merge Nodes

The Merge Nodes algorithm in a B-tree is employed when a node becomes underflowed during a deletion operation, meaning it has fewer keys than the minimum required. When a node underflows, it may merge with its adjacent sibling node, effectively reducing the tree's height and maintaining the B-tree's balanced property. The Merge Nodes algorithm redistributes the keys and child pointers from the sibling node into the underflowed node, creating a single merged node. The key in the parent node, which was in between the two merged nodes, is removed, and the Merge Nodes algorithm may recursively propagate up the tree if the parent node also becomes underflowed. By merging nodes, this algorithm helps to prevent further underflow and maintain the B-tree's efficient data management and search capabilities.

Merge Nodes Algorithm

Below is an example of the merge nodes algorithm.

```
procedure merge_nodes(B-tree parent, int left_index)
    left_child <- parent.children[left_index]
    right_child <- parent.children[left_index + 1]

    // Move the key from the parent into the left_child node
    move_key_from_parent(parent, left_index)

    // Move all keys and child pointers from the right_child into the left_child
    move_keys_and_pointers(right_child, left_child)

    // Remove the right_child node from the parent and free its memory
    remove_child_from_parent(parent, left_index + 1)
    free_memory(right_child)

    // Update the parent's key count and child pointers
    parent.num_keys <- parent.num_keys - 1
    update_parent_pointers(parent, left_index + 1)

    // If the parent becomes underflowed, recursively handle further merging
    if parent is underflowed
        merge_nodes(parent's parent, position of parent in its parent)
    end if
end procedure
```

Minimum & Maximum Key Retrieval

The Minimum & Maximum Key Retrieval algorithm in a B-tree is employed to find the smallest and largest keys present in the B-tree, respectively. To retrieve the minimum key, the algorithm traverses the leftmost path of the tree until it reaches a leaf node, where the smallest key resides. Similarly, to find the maximum key, the algorithm traverses the rightmost path of the tree until it reaches a leaf node, where the largest key exists. This algorithm does not require recursion and efficiently locates the minimum and maximum keys, providing quick access to the boundary values stored in the B-tree.

Minimum & Maximum Key Retrieval Algorithm

Below is an example of the minimum & maximum key retrieval algorithm.

```
procedure find_minimum_key(B-tree node)
    while node is not a leaf node
        node <- node.children[0]  // Traverse to the leftmost child
    end while

    return node.keys[0]          // Return the smallest key in the leaf node
end procedure
```

```
procedure find_maximum_key(B-tree node)
    while node is not a leaf node
        node <- node.children[node.num_keys]  // Traverse to the rightmost child
    end while

    return node.keys[node.num_keys - 1]      // Return the largest key in the leaf node
end procedure
```

Printing

The Printing algorithm for a B-tree involves traversing the entire tree and displaying its structure and keys in a visually understandable format. Various traversal methods, such as in-order, pre-order, or post-order, can be used to achieve this. During the traversal, the algorithm visits each node, displaying its keys and child pointers. Proper indentation is used to represent the hierarchy of nodes in the B-tree. The Printing algorithm helps with debugging, visualization, and understanding the organization of the B-tree, making it a valuable tool for analyzing the structure and contents of the tree.

Printing Algorithm

Below is an example of the printing algorithm.

```
procedure print_B_tree(B-tree node, indentation)
    if node is not null
        // Print keys and child pointers in the node
        for i <- 0 to node.num_keys - 1
            print indentation, node.keys[i]

        // Print the last child pointer
        print indentation, node.keys[node.num_keys]

        // Recursive call for each child node
        for i <- 0 to node.num_keys
            print_B_tree(node.children[i], indentation + "    ")
    end if
end procedure
```

Search

The Search algorithm for a B-tree involves finding a specific key within the tree efficiently. The algorithm starts from the root node and compares the target key with the keys in the current node. If the key is found, the algorithm returns a pointer to the node containing the key. If the key is not in the current node, the algorithm narrows down the search to the appropriate child subtree by comparing the key with the node's keys and follows the corresponding child pointer. This process continues until the key is found in a leaf node or determined to be absent. The Search algorithm's balanced nature and logarithmic time complexity enable rapid retrieval of keys in large datasets, making B-trees ideal for database systems and file systems.

Search Algorithm

Below is an example of the search algorithm.

```
procedure search_key(B-tree node, key_to_find)
    if node is not null
        position <- find_position_in_node(node, key_to_find)

        if position < node.num_keys and node.keys[position] = key_to_find
            // Key found in the current node
            return node
        end if

        if node is a leaf node
            // Key not found
            return NULL
```

```
        else
            // Recursively search in the appropriate child subtree
            return search_key(node.children[position], key_to_find)
        end if
    end if
end procedure
```

Split Node

The Split Node algorithm in a B-tree is employed when a node becomes full (contains $(m-1)$ keys) during an insertion operation. To maintain the B-tree's balanced structure and sorted keys, the Split Node algorithm divides the full node into two separate nodes. The middle key is moved up to the parent node, creating a space for the new key to be inserted. The child pointers of the split node are appropriately adjusted, and the new key is inserted into the parent node. The Split Node algorithm may trigger further splits up the tree if the parent node also becomes full, ensuring that the B-tree remains balanced and efficient in managing data.

Split Node Algorithm

Below is an example of the split node algorithm.

```
procedure split_node(B-tree parent, int child_index)
    full_node <- parent.children[child_index]  // Get the full node to split
    new_node <- create_new_node()             // Create a new node for splitting
    middle_key <- full_node.keys[(m-1)/2]      // Find the middle key to move up

    // Move keys and child pointers to the new node
    move_keys_and_pointers(full_node, new_node, (m-1)/2 + 1, m-1)

    // Adjust the number of keys in the nodes
    full_node.num_keys <- (m-1)/2
    new_node.num_keys <- (m-1)/2

    // Update the parent node with the middle key and new child pointer
    insert_key_in_node(parent, child_index, middle_key)
    insert_child_in_node(parent, child_index + 1, new_node)
end procedure
```

Successor & Predecessor

The Successor & Predecessor algorithm in a B-tree is used to find the key that comes immediately after or before a given key in sorted order within the B-tree. To find the successor of a key, the algorithm starts by searching for the key in the B-tree. If the key is present in an internal node, the successor is the leftmost key in its right subtree. If the key is found in a leaf node, the successor is the next key in the same node or the leftmost key in the right sibling node if the current node has no more keys. Similarly, to find the predecessor of a key, the algorithm follows a similar approach but looks for the rightmost key in the left subtree if the key is found in an internal node, or the previous key in the same node or the rightmost key in the left sibling node if the current node has no more keys. The Successor & Predecessor algorithm efficiently navigates the B-tree to identify the next or previous key for a given key, making it useful for range queries and data traversal in sorted order.

Successor & Predecessor Algorithm

Below is an example of the successor and predecessor algorithm.

```
procedure find_successor(B-tree node, key_to_find)
    position <- find_position_in_node(node, key_to_find)

    if node is a leaf node
        // Key not found in the current node
        if position < node.num_keys
            // Successor is the next key in the same node
            return node.keys[position + 1]
```

```
        else
            // Successor is the leftmost key in the right sibling node
            return find_leftmost_key_in_right_sibling(node.parent, position + 1)
        end if
    else
        // Recursively search in the right subtree
        return find_leftmost_key(node.children[position + 1])
    end if
end procedure

procedure find_leftmost_key(B-tree node)
    while node is not a leaf node
        // Continue traversing to the leftmost child
        node <- node.children[0]
    end while

    return node.keys[0]  // Return the leftmost key in the leaf node
end procedure

procedure find_leftmost_key_in_right_sibling(B-tree parent, int right_sibling_index)
    right_sibling <- parent.children[right_sibling_index]

    // Traverse to the leftmost key in the right sibling node
    return find_leftmost_key(right_sibling)
end procedure

procedure find_predecessor(B-tree node, key_to_find)
    position <- find_position_in_node(node, key_to_find)

    if node is a leaf node
        // Key not found in the current node
        if position > 0
            // Predecessor is the previous key in the same node
            return node.keys[position - 1]
        else
            // Predecessor is the rightmost key in the left sibling node
            return find_rightmost_key_in_left_sibling(node.parent, position)
        end if
    else
        // Recursively search in the left subtree
        return find_rightmost_key(node.children[position])
    end if
end procedure

procedure find_rightmost_key(B-tree node)
    while node is not a leaf node
        // Continue traversing to the rightmost child
        node <- node.children[node.num_keys]
    end while

    return node.keys[node.num_keys - 1]  // Return the rightmost key in the leaf node
end procedure

procedure find_rightmost_key_in_left_sibling(B-tree parent, int left_sibling_index)
    left_sibling <- parent.children[left_sibling_index]

    // Traverse to the rightmost key in the left sibling node
    return find_rightmost_key(left_sibling)
end procedure
```

Traversal

The Traversal algorithm in a B-tree is used to visit and process all the keys in the tree in a specific order. Various traversal methods, such as in-order, pre-order, and post-order, allow accessing the keys in different sequences. In an in-order traversal, the algorithm visits the keys in ascending order, starting from the smallest key. In a pre-order traversal, the algorithm visits the current node's key before traversing its children. In a post-order traversal, the algorithm visits the children nodes before the current node's key. Traversal is often implemented recursively, efficiently navigating the B-tree's hierarchical structure and providing a valuable mechanism for processing and displaying the keys in the desired order.

Traversal Algorithm

Below is an example of the traversal algorithm.

```
procedure in_order_traversal(B-tree node)
    if node is not null
        // Traverse left subtree
        in_order_traversal(node.children[0])

        // Process current node's keys
        for i <- 0 to node.num_keys - 1
            process_key(node.keys[i])

        // Traverse right subtree
        in_order_traversal(node.children[node.num_keys])

    end if
end procedure
```

Priority Queue

Overview

A priority queue is an abstract data type that allows for efficient insertion and retrieval of elements based on their priority. In a priority queue, each element is associated with a priority value, and the elements are arranged in a way that ensures the element with the highest (or lowest, depending on the implementation) priority is always at the front of the queue. Priority queues are useful in scenarios where elements need to be processed based on their importance, urgency, or other priority criteria. Common operations supported by a priority queue include inserting elements, extracting the element with the highest priority, and peeking at the highest-priority element without removing it. Priority queues can be implemented using various data structures, such as binary heaps, binomial heaps, Fibonacci heaps, or self-balancing binary search trees, each offering different trade-offs in terms of time complexity for different operations. Priority queues find applications in various domains, including task scheduling, network routing, Huffman coding, and Dijkstra's algorithm for finding the shortest path in a graph.

Heaps

A heap is a specialized binary tree-based data structure that satisfies the heap property, which determines the order of elements within the tree. In a binary heap, each node has a key, and the key of each node is either greater than or equal to (in a max heap) or less than or equal to (in a min heap) the keys of its children. This property ensures that the root node always contains the highest (max heap) or lowest (min heap) key in the heap, making it easy to access and remove the element with the highest or lowest priority in constant time. Binary heaps are usually implemented as arrays, where the element at index i has its left child at index $2i + 1$ and its right child at index $2i + 2$. Binary heaps are commonly used to implement priority queues, which efficiently manage tasks or elements based on their priority, and they find applications in various algorithms like Dijkstra's algorithm, heap sort, and Huffman coding. The balanced nature and logarithmic time complexity of heap operations make them a powerful tool for efficient data management and processing.

Structure

Priority queues can be constructed in a number of different ways. Typically, a priority queue requires some sort of container to hold the data types of the queue. This container can be a vector, array, list, or something similar. Beyond this container, priority queues typically require at least three types of operations. These operations usually include, inserting, removing, and peeking (looking at the element that is of highest or lowest priority). Priority queues also require other operations such as percolating up or down, depending on whether or not the heap is a min or max heap. In the context of the assignment for this course, the structure of the priority queue consists a structure and a class. We first look at the structure **pq**:

- **heapPriority** - This is a vector of float values that represent a nodes priority in the queue.
- **heapString** - This is a vector of string objects that are associated with a specific node that exists in the priority queue.
- **percolateUp** - This is a function that percolates elements up a priority queue to be inserted in the correct index of the queue. Since this priority queue incorporates a max heap, percolate up is needed instead of percolate down.

Along with the previous structure **pq**, there is a class that encapsulates the implementation of the priority queue. The class **PriorityQueue** consists of the following member functions:

- **InitPriorityQueue** - This function initializes a **pq** structure and sets the values inside the structure to a default value.
- **Insert** - This function inserts a node into a priority queue at its correct location within the queue.
- **Peek** - This function will retrieve the node with the highest element in a priority queue and return it without modifying the queue itself.
- **Remove** - This function removes a node with the highest priority in a queue.

Efficiency

The runtime complexity of a binary heap, a type of heap data structure, is efficient for essential operations. Insertion and deletion of elements in the heap have a worst-case time complexity of $\mathcal{O}(\log(n))$ and a best-case time complexity of $\mathcal{O}(1)$ when dealing with the root element. The average time complexity for these operations is also $\mathcal{O}(\log(n))$, considering the balanced nature of binary heaps. Peeking at the minimum or maximum element can be done in constant time ($\mathcal{O}(1)$) as it requires accessing the root directly. Building a binary heap from an unordered array has a time complexity of $\mathcal{O}(n)$ in both average and worst cases. The logarithmic and linear time complexities of heap operations make it an effective data structure for managing priorities and finding extremum values efficiently. The runtime complexities of the common operations in a heap can be seen below:

Operation	Best Case $\Omega(n)$	Average Case $\Theta(n)$	Worst Case $\mathcal{O}(n)$
Deletion	$\Omega(1)$	$\Theta(\log(n))$	$\mathcal{O}(\log(n))$
Heapify-ing	$\Omega(n)$	$\Theta(n)$	$\mathcal{O}(n)$
Insertion	$\Omega(1)$	$\Theta(\log(n))$	$\mathcal{O}(\log(n))$
Peeking	$\Omega(1)$	$\Theta(1)$	$\mathcal{O}(1)$

Heapifying is the process of converting an array (or a portion of it) into a binary heap, satisfying the heap property. In the context of binary heaps, heapify involves 'bubbling down' or 'sifting down' elements starting from the non-leaf nodes to their correct positions in the heap. This process is essential for maintaining the binary heap's balanced and ordered structure, ensuring that the root node holds the maximum (in a max heap) or minimum (in a min heap) value, and that the heap property holds true for all the elements. Heapify is typically used to build a binary heap from an unordered array or to restore the heap property after a deletion operation.

PQ Structure

As mentioned above, the **pq** structure is a structure that is used in tandem with the **PriorityQueue** class. The definition of this structure can be seen below.

PQ Structure

Below is the definition of the **pq** structure:

```
struct pq {
    vector<string> heapString;
    vector<float> heapPriority;

    void percolateUp(int index) {
        while (index > 0) {
            int parentIndex = (index - 1) / 2;
            if (this->heapPriority.at(index) <= this->heapPriority.at(parentIndex)) {
                return;
            }
            else {
                swap(this->heapString.at(index), this->heapString.at(parentIndex));
                swap(this->heapPriority.at(index), this->heapPriority.at(parentIndex));
                index = parentIndex;
            }
        }
    }
};
```

InitPriorityQueue

The function 'InitPriorityQueue' is a member function of the 'PriorityQueue' class that returns a shared pointer to a dynamically allocated 'pq' object, representing a priority queue. Within the function, a new 'pq' object is created and its data members, 'heapString' and 'heapPriority', are initialized to be empty by calling their 'clear()' methods. The 'pq' object is then wrapped in a shared pointer before being returned from the function. This function is responsible for initializing an empty priority queue, allowing it to be used for storing elements with associated priorities later in the program. The definition of this function can be seen below.

InitPriorityQueue Algorithm

Below is the definition of the 'InitPriorityQueue' function:

```
shared_ptr<pq> PriorityQueue::InitPriorityQueue(){
    shared_ptr<pq> queue(new pq);
    queue->heapString.clear();
    queue->heapPriority.clear();
    return queue;
}
```

Insert

The function 'Insert' is a member function of the 'PriorityQueue' class that allows inserting a new element with its associated priority into an existing priority queue represented by the shared pointer 'queue'. The function takes the 'text' (a string representing the element) and 'priority' (a float value representing its priority) as inputs. The element and its priority are added to the back of their respective vectors 'heapString' and 'heapPriority'. After insertion, the function calculates the index of the last element in the 'heapString' vector and performs the "percolate up" operation, which ensures that the newly inserted element moves to its correct position in the heap to maintain the heap property. The 'percolateUp' function is likely implemented to handle the upward movement of the element in the heap. This function enables efficient insertion and maintenance of the priority queue's ordered structure, ensuring that higher-priority elements are positioned closer to the root of the heap. The definition of this function can be seen below.

Insert Algorithm

Below is the definition of the 'Insert' function:

```
void PriorityQueue::Insert(shared_ptr<pq> queue, string text, float priority){
    queue->heapString.push_back(text);
    queue->heapPriority.push_back(priority);
    int currentIndex = queue->heapString.size() - 1;
    queue->percolateUp(currentIndex);
}
```

Peek

The function 'Peek' is a member function of the 'PriorityQueue' class that allows accessing the element with the highest priority in the priority queue represented by the shared pointer 'queue' without removing it. The function begins by creating a string variable 'headVal', which will hold the value of the element at the top of the priority queue. It then checks if the queue is empty, in which case it sets 'headVal' to an empty string. If the queue is not empty, the function retrieves the element at the top of the queue (root of the binary heap) by accessing the first element in the 'heapString' vector, which represents the elements in the binary heap. The value of the top element is then assigned to 'headVal'. The function finally returns 'headVal', which represents the element with the highest priority in the priority queue without removing it. This allows users to peek at the highest-priority element without altering the queue's contents. The definition of this function can be seen below.

Peek Algorithm

Below is the definition of the 'Peek' function:

```
string PriorityQueue::Peek(shared_ptr<pq> queue){
    string headVal;
    if (queue->heapString.size() == 0) {
        headVal = "";
    }
    else {
        headVal = queue->heapString.at(0);
    }
    return headVal;
}
```

Remove

The function 'Remove' is a member function of the 'PriorityQueue' class that removes and returns the element with the highest priority from the priority queue represented by the shared pointer 'queue'. The function begins by creating a default string 'returnVal', which is used to store the element that will be removed and returned. It first checks if the priority queue is empty, in which case it sets 'returnVal' to an empty string. If the queue is not empty, it searches for the element with the highest priority by iterating through the 'heapPriority' vector. Once the element with the highest priority is found, its corresponding string value is stored in 'returnVal', and both the priority and text values are removed from their respective vectors using the 'erase' method. The function then returns 'returnVal', which represents the element that was removed from the priority queue with the highest priority. The definition of this function can be seen below.

Remove Algorithm

Below is the definition of the 'Remove' function:

```
string PriorityQueue::Remove(shared_ptr<pq> queue){
    string returnVal;
    if (queue->heapString.size() == 0) {
        returnVal = "";
    }
    else {
        float maxVal = 0;
        int index = 0;
        for (int i = 0; i < queue->heapString.size(); i++) {
            if (maxVal <= queue->heapPriority.at(i)) {
                maxVal = queue->heapPriority.at(i);
                index = i;
            }
        }
        returnVal = queue->heapString.at(index);
        queue->heapString.erase(queue->heapString.begin() + index);
        queue->heapPriority.erase(queue->heapPriority.begin() + index);
    }
    return returnVal;
}
```

Hash Tables

Overview

A hash table is a data structure that organizes and stores data in a way that allows for efficient retrieval and storage operations. It uses a technique called hashing to map keys to corresponding values in an array. The purpose of a hash table is to provide fast access to data elements, making it particularly useful for tasks such as searching, inserting, and deleting items in constant time complexity on average. By using a hash function to calculate the index where data is stored, hash tables can achieve rapid data retrieval, making them invaluable for various applications, such as database management, caching, and implementing associative arrays or key-value pairs.

Chaining

Chaining in hash tables is a collision resolution technique where multiple elements with different keys hash to the same index or bucket. Instead of overwriting the existing value, chaining allows these elements to be stored together in a linked list or another data structure at the same bucket. Each bucket acts as a small container, holding multiple key-value pairs. When a collision occurs, the new element is appended to the existing chain in the bucket. During retrieval or deletion, the hash table traverses the chain, locating the specific key-value pair based on the key. Chaining is a simple and effective method to handle collisions, ensuring that hash tables can efficiently store and retrieve data, even when multiple keys map to the same location in the array.

Linear Probing

Linear probing in hash tables is a collision resolution technique where elements with different keys that hash to the same index are placed in the next available (unoccupied) position in the table, effectively 'probing' forward until an empty slot is found. When a collision occurs, the algorithm checks the next position in the table and repeats the process until it finds an empty location. This method is also known as open addressing because it involves exploring alternative slots within the array to resolve collisions. Linear probing ensures that all elements are eventually stored in the primary array, without using additional data structures like linked lists. However, it may suffer from clustering, where consecutive collisions lead to more collisions, potentially degrading performance. Despite this limitation, linear probing remains a widely used technique due to its simplicity and cache-friendliness, enabling hash tables to maintain $\mathcal{O}(1)$ average case complexity for basic operations when load factors are kept low.

Quadratic Probing

Quadratic probing in hash tables is a collision resolution technique that addresses collisions by systematically probing the table using quadratic increments until an empty slot is found. When a collision occurs and the initial hashed position is occupied, the algorithm checks positions at increasingly distant intervals, following a quadratic sequence (i.e., 1, 4, 9, 16, and so on), to determine the next potential position. This method aims to disperse the elements more evenly across the table, reducing the clustering effect observed in linear probing. Quadratic probing avoids the primary clustering issue but may still encounter secondary clustering, which can affect its performance in high-load scenarios. Nevertheless, it remains a valuable alternative to linear probing for collision resolution, as it maintains $\mathcal{O}(1)$ average case complexity for basic operations when the load factor is kept relatively low, while being more cache-efficient compared to some other open addressing methods.

Double Hashing

Double hashing is a collision resolution technique used in hash tables to address key collisions by probing alternative positions based on a secondary hash function. When a collision occurs and the primary hash function places an element at an occupied index, double hashing uses a secondary hash function to calculate an increment value that determines the next probing position. The secondary hash function ensures that the increment value is non-zero and relatively prime to the table size, enabling the algorithm to explore different positions and avoid clustering. Double hashing provides good distribution of elements in the table, reducing the likelihood of collisions and achieving efficient data retrieval. It is a popular choice for collision resolution, as it maintains the $\mathcal{O}(1)$ average case complexity for basic operations, such as insertion, deletion, and retrieval, even under high load factors, making it a robust and reliable approach for hash table implementations.

Efficiency

The runtime complexity of a hash table is generally considered to be $\mathcal{O}(1)$ on average for basic operations such as insertion, deletion, and retrieval. This constant time complexity arises from the efficient mapping of keys to their corresponding values using a hash function. In the ideal scenario, each key hashes to a unique index, ensuring direct access to the desired element. However, in certain cases, hash collisions may occur when multiple keys hash to the same index, leading to a slight increase in access time. To address collisions, hash tables use techniques like chaining or open addressing, which may, in rare instances, result in worst-case scenarios with a time complexity of $\mathcal{O}(n)$. Overall, though, the average case remains constant time,

making hash tables an invaluable data structure for a wide range of applications where rapid data retrieval and storage are crucial.

Operation	Best Case $\Omega(n)$	Average Case $\Theta(n)$	Worst Case $\mathcal{O}(n)$
Inserting	$\Omega(1)$	$\Theta(1)$	$\mathcal{O}(1)$
Inserting (Collisions)	$\Omega(n)$	$\Theta(n)$	$\mathcal{O}(n)$
Removal	$\Omega(1)$	$\Theta(1)$	$\mathcal{O}(1)$
Searching	$\Omega(1)$	$\Theta(1)$	$\mathcal{O}(1)$

In general, the runtime complexity of a hash table tends to be $\mathcal{O}(1)$. This is what makes it so efficient in the operations that are used with them. In the case where a collision occurs (insertion where a bucket is already occupied) the runtime complexity of the algorithm then tends to become $\mathcal{O}(n)$.

Structure

To implement a hash table, several key functions, data structures, and classes are required. The fundamental components include a hash function, an array (or a dynamic array like a list or vector) to store the data, and a mechanism to handle collisions. The hash function is responsible for converting the keys into array indices, ensuring uniform distribution and efficient retrieval. The array serves as the main storage container for the key-value pairs, and its size typically depends on the expected number of elements and the desired load factor. To handle collisions, a collision resolution technique is necessary, such as chaining (using linked lists or other data structures to store multiple elements at the same index) or open addressing (finding alternative positions in the array for collided keys). Additionally, it is beneficial to encapsulate the hash table functionalities into a class, providing a clean interface for insertion, deletion, and retrieval operations, as well as methods for resizing the array when needed to maintain efficiency. By combining these components, developers can create a robust and efficient hash table implementation suitable for various real-world applications.

In order for a hash table to function correctly it requires a couple of custom structures and classes. First, we examine the custom structures that are built for use in this hash table. The two custom structures that were created for this assignment are **Hash Node** and **Hash Table**.

- **Hash Node** - This custom structure represents the contents of an individual bucket in a hash table. This structure has the following data members:
  - **deleted** - Boolean value that represents if a node has been removed from a hash table.
  - **hashcode** - Unsigned integer value that represents the hash code that is computed with a hash function.
  - **key** - String value that represents a key associated with a value.
  - **value** - String value that represents the value that is associated with a key.
- **Hash Table** - This custom structure holds the hashed data. This structure has the following data members:
  - **bucket_func** - A function that returns an unsigned integer value that represents where a node should be placed.
  - **capacity** - Unsigned integer value that represents the number of addressable buckets in a hash table.
  - **hash_func** - A function that returns an unsigned integer value that represents a hash code.
  - **size** - Unsigned integer value that represents the number of actual entries in a hash table.
  - **table** - A dynamic array of of hash node pointers.

Along with the custom data structures that have been defined previously, there are two functions (**DJB2()** and **ModuloBucketFunc()**) and the class **Hash** that are used to implement the custom structures. First, we examine the two custom functions before we examine the **Hash** class:

- **DJB2()** - This function hashes a string value and assigns an unsigned integer value in the 32-bit integer value range.
- **ModuloBucketFunc()** - This function puts keys into a specified bucket index.

The above functions provide a base for how we calculate hash codes and where we determine where the codes will be stored in the hash table. The **Hash** class provides an implementation of all these individual structures and functions to create a functioning hash table. This class' member functions are now examined:

- **Contains** - Determines if a non-deleted node is present in a hash table.
- **GetVal()** - Returns the value associated with a key in a hash table.
- **Hash()** - A constructor.
- **~Hash()** - A de-constructor.
- **InitNode()** - Creates and initializes a 'hash_node' structure that will occupy a bucket in a hash table.

- **InitTable()** - Creates and initializes a 'hash_table' structure and returns a pointer to it.
- **Load()** - Returns a load factor that describes how 'full' a hash table may be at a given time.
- **PrintTable()** - Prints the contents that are present in a given hash table.
- **Remove()** - Marks a node as deleted and removes a node from a hash table if the given node is present in a hash table.
- **Resize()** - Resizes a hash table to have a new specified capacity.
- **SetKVP()** - Creates a mapping between a given key and a value pair in a hash table.

These member functions are responsible for encapsulating the implementation of the structures that were created previously. We now will examine each of these algorithms one by one to obtain a better understanding of how these functions operate.

DJB2

The DJB2 algorithm is a simple and widely used hash function designed to efficiently convert a given input string (key) into a corresponding unsigned integer hash value. It initializes the hash to 5381 and iterates through each character of the input string. For each character, it performs bitwise left shift (<<) on the current hash by 5 positions, then adds the original hash value to the result, and finally adds the ASCII value of the current character to the hash. This process continues for all characters in the input string, effectively combining their contributions to create the final hash value. The algorithm's simplicity and effectiveness in distributing hash values make it popular for various applications where a fast and reasonably distributed hash function is required, such as in hash tables, caching, and hashing-based data structures.

DJB2 Algorithm

The definition of the 'DJB2' algorithm can be seen below:

```
unsigned int DJB2(std::string key){
    unsigned int hash = 5381;
    for (size_t i=0; i < key.length(); i++) {
        char c = key[i];
        hash = ((hash << 5) + hash) + c;
    }
    return hash;
}
```

ModuloBucketFunc

The ModuloBucketFunc algorithm is responsible for calculating the index of a bucket in a hash table. Whether this algorithm is used for inserting or searching, it determines the index of a specific hash code. This bucket index is found by taking the specific hash code of a key and calculating the modulus of the hash code with the capacity of the table.

ModuloBucketFunc Algorithm

The definition of the 'ModuloBucketFunc' algorithm can be seen below:

```
unsigned int ModuloBucketFunc(unsigned int hashcode, unsigned int cap){
    unsigned int b = hashcode % cap;
    return b;
}
```

Contains

The function above begins by calculating the hash code and bucket index of the given key that is being searched for in the hash table. We then create an integer value that is designated to keep track of the number of buckets that have been 'probed'. We then begin traversing the hash table until the number of buckets that have been probed is equal to that of the tables size. In each iteration, we first check to see if they is present in the current bucket that we are examining, and it it is we return true. If the key is not present in the current bucket we Increment the bucket index and buckets probed variable and continue on to the next bucket. This process will repeat until either we find the bucket that contains the key we are searching for or until the number of buckets that we have probed is equal to that of the hash tables size. If the key is not found in the hash table then we return false.

Contains Algorithm

The definition of the 'Contains' algorithm can be seen below:

```
bool Hash::Contains(shared_ptr<hash_table> tbl, std::string key){
    unsigned int hash = DJB2(key);
    unsigned int bucket_idx = tbl->bucket_func(hash, tbl->capacity);
    unsigned int buckets_probed = 0;
    // Traverse the table until buckets probed reaches the size of the table
    while (buckets_probed < tbl->size) {
        if (tbl->table->at(bucket_idx)->key == key) {
            return true;
        }
        bucket_idx = (bucket_idx + 1) % tbl->size;
        buckets_probed++;
    }
    return false;
}
```

GetVal

The GetVal algorithm functions in a similar way as the Contains algorithm. Similar to that of the Contains algorithm, we begin by calculating the hash code and bucket index of the key. We then create a dummy variable that will keep track of the number of buckets that we have probed while traversing the hash table. Once we begin traversing the hash table, we check if the current bucket that we are examining contains the key that we are searching for. If the current bucket does contain the key that we are looking for, then we return that keys value and stop traversing the hash table. If the current bucket does not contain the key that we are searching for, then we increment both the bucket index and the buckets probed value to move on to the next bucket. If the key is not found in the hash table then we return an empty string.

GetVal Algorithm

The definition of the 'GetVal' algorithm can be seen below:

```
std::string Hash::GetVal(shared_ptr<hash_table> tbl, std::string key){
    unsigned int hash = DJB2(key);
    unsigned int bucket_idx = tbl->bucket_func(hash, tbl->capacity);
    unsigned int buckets_probed = 0;
    while (buckets_probed < tbl->size) {
        if (tbl->table->at(bucket_idx)->key == key) {
            return tbl->table->at(bucket_idx)->value;
        }
        bucket_idx = (bucket_idx + 1) % tbl->size;
        buckets_probed++;
    }
    return "";
}
```

InitNode

The InitNode algorithm is responsible for creating a hash node object that is assigned to default values. We first create a new hash node object called 'ret' and then we assign the data members of the object to the input parameters that are fed in the algorithm. We also flag the boolean value 'deleted' to be false and once these values have been assigned we return the node as the algorithms output.

InitNode Algorithm

The definition of the 'InitNode' algorithm can be seen below:

```
shared_ptr<hash_node> Hash::InitNode(std::string key, unsigned int hashcode, std::string val){
    shared_ptr<hash_node> ret(new hash_node);
    ret->deleted = false;
```

```
    ret->key = key;
    ret->hashcode = hashcode;
    ret->value = val;
    return ret;
}
```

InitTable

The InitTable algorithm is doing the same as the InitNode algorithm but instead of working a hash node we are working with a hash table. In this algorithm, we assign the default values of a hash table to the input parameters that are fed to this algorithm. We also initialize the functions 'hash_func' and 'bucket_func' to that of **DJB2** and **ModuloBucketFunc** respectively. All of the pointers inside the 'table' data member are initially set to null and the 'size' and 'occupied' members are initially set to zero. Once these data members have been assigned we return the table as the output of the algorithm.

InitTable Algorithm

The definition of the 'InitTable' algorithm can be seen below:

```
shared_ptr<hash_table> Hash::InitTable(unsigned int cap){
    shared_ptr<hash_table> ret(new hash_table);
    ret->capacity = cap;
    ret->size = 0;
    ret->occupied = 0;
    ret->table = shared_ptr<htable>(new htable(cap));
    for (int i = 0; i < ret->table->size(); i++) {
        ret->table->at(i) = nullptr;
    }
    ret->hash_func = &DJB2;
    ret->bucket_func = &ModuloBucketFunc;
    return ret;
}
```

Load

The Load algorithm is calculating how full a hash table is. This is done by calculating the ratio of the size to the capacity of the hash table in question. In order to calculate this, we have to statically cast the 'size' and 'capacity' data members to floats so that the ratio of the two can be calculated. Once these variables have been statically casted the ratio, the algorithm returns the ratio.

Load Algorithm

The definition of the 'Load' algorithm can be seen below:

```
float Hash::Load(shared_ptr<hash_table> tbl){
    float size = static_cast<float>(tbl->size);
    float cap = static_cast<float>(tbl->capacity);
    float load = size / cap;
    return load;
}
```

PrintTable

The PrintTable algorithm is a function designed to display the contents and statistics of a given hash table. It takes a shared pointer to the hash table as input and outputs details such as the capacity, current size, number of occupied slots, and the load factor (ratio of occupied slots to capacity). The function then checks if the hash table's capacity is less than 130; if so, it proceeds to print each element in the table. For each index, it displays either '<empty>' if the slot is empty, '<deleted>' if the slot was marked as deleted, or the key-value pair stored at that index. If the capacity is greater than or equal to 130, the function prints a message indicating that the hash table is too big to be printed out. This algorithm is useful for developers to inspect and understand the state of the hash table, aiding in debugging and performance analysis.

PrintTable Algorithm

The definition of the 'PrintTable' algorithm can be seen below:

```
void Hash::PrintTable(shared_ptr<hash_table> tbl){
    cout << "Hashtable:" << endl;
    cout << "  capacity: " << tbl->capacity << endl;
    cout << "  size:     " << tbl->size << endl;
    cout << "  occupied: " << tbl->occupied << endl;
    cout << "  load:     " << Load(tbl) << endl;
    if (tbl->capacity < 130) {
        for (unsigned int i=0; i < tbl->capacity; i++) {
            cout << "[" << i << "]    ";
            if (!tbl->table->at(i)) {
                cout << "<empty>" << endl;
            } else if (tbl->table->at(i)->deleted) {
                cout << "<deleted>" << endl;
            } else {
                cout << "\"" << tbl->table->at(i)->key << "\" = \""
                << tbl->table->at(i)->value << "\"" << endl;
            }
        }
    } else {
        cout << "<hashtable too big to print out>" << endl;
    }
}
```

Remove

The Remove algorithm is a function designed to remove a key-value pair from the given hash table. It calculates the hash code and the bucket index for the input 'key' using the DJB2 hash function and the provided bucket function. The function then traverses the table, probing the buckets to find the key to be removed. If the key is found in a non-null bucket, it marks the bucket as deleted and removes the node from the table, decreasing the table's size by one, before returning true to indicate a successful removal. If the key is not found after probing all buckets, the function returns false to signify that the key was not present in the hash table. This algorithm efficiently handles collisions and ensures the removal operation has a time complexity of $\mathcal{O}(1)$ on average, making it suitable for hash table implementations requiring a delete operation.

Remove Algorithm

The definition of the 'Remove' algorithm can be seen below:

```
bool Hash::Remove(shared_ptr<hash_table> tbl, std::string key){
    unsigned int hash = DJB2(key);
    unsigned int bucket_idx = tbl->bucket_func(hash, tbl->capacity);
    unsigned int buckets_probed = 0;
    while (buckets_probed < tbl->size) {
        if (tbl->table->at(bucket_idx) != nullptr && tbl->table->at(bucket_idx)->key == key) {
            tbl->table->at(bucket_idx) == nullptr;
            tbl->table->at(bucket_idx)->deleted = true;
            tbl->size--;
            return true;
        }
        bucket_idx = (bucket_idx + 1) % tbl->size;
        buckets_probed++;
    }
    return false;
}
```

Resize

The Resize algorithm is a function designed to resize the given hash table to a new specified capacity. It first creates a new hash table with the desired capacity using the "InitTable" function. The size and occupied data members of the old table are then copied to the new one to maintain consistency. Next, it iterates through the nodes of the old table and rehashes them to calculate their new bucket index in the resized table. The nodes are then copied to the corresponding positions in the new table. Finally, the old table is updated to point to the new table, effectively resizing and rehashing the hash table. This algorithm efficiently adjusts the capacity of the hash table, enabling it to handle changes in the number of elements while maintaining an average time complexity of $\mathcal{O}(n)$ for resizing, where n is the number of elements in the hash table.

Resize Algorithm

The definition of the 'Resize' algorithm can be seen below:

```
void Hash::Resize(shared_ptr<hash_table>& tbl, unsigned int new_capacity){
    shared_ptr<hash_table> new_table = InitTable(new_capacity);
    new_table->size = tbl->size;
    new_table->occupied = tbl->occupied;
    for (int i = 0; i < tbl->table->size(); i++) {
        if (tbl->table->at(i) != nullptr) {
            unsigned int hash = DJB2(tbl->table->at(i)->key);
            unsigned int bucket_idx = new_table->bucket_func(hash, new_table->capacity);
            new_table->table->at(bucket_idx) = tbl->table->at(i);
        }
    }
    tbl = new_table;
}
```

SetKVP

The SetKVP algorithm is a function designed to insert or update a key-value pair (KVP) in the given hash table. It first calculates the hash code and bucket index for the input key using the DJB2 hash function and the provided bucket function. Then, it creates a new hash node containing the key-value pair to be inserted or updated. The algorithm proceeds to probe the buckets in the table until all possible buckets have been checked. If an empty bucket is found, the new node is inserted at that index, and the table's occupied and size data members are updated accordingly. If a non-empty bucket with a matching key is found, the value is updated, and the function returns true to indicate a successful update. If all buckets are probed, and no suitable empty bucket or matching key is found, the function returns false, indicating that the insertion or update operation was not successful. This algorithm ensures efficient and collision-handling insertions and updates in the hash table, with an average time complexity of $\mathcal{O}(1)$ and a worst-case time complexity of $\mathcal{O}(n)$, where n is the number of elements in the hash table.

SetKVP Algorithm

The definition of the 'SetKVP' algorithm can be seen below:

```
bool Hash::SetKVP(shared_ptr<hash_table> tbl, std::string key, std::string value){
    unsigned int hash = DJB2(key);
    unsigned int bucket_idx = tbl->bucket_func(hash, tbl->capacity);
    unsigned int buckets_probed = 0;
    shared_ptr<hash_node> node = InitNode(key, hash, value);
    while (buckets_probed < tbl->table->size()) {
        if (tbl->table->at(bucket_idx) == nullptr) {
            tbl->table->at(bucket_idx) = node;
            tbl->occupied++;
            tbl->size++;
            return true;
        }
            else if (tbl->table->at(bucket_idx)->key == key) {
            tbl->table->at(bucket_idx)->value = value;
            return true;
        }
```

```
        bucket_idx = (bucket_idx + 1) % tbl->table->size();
        buckets_probed++;
    }
    return false;
}
```

Huffman Tables

Overview

Huffman encoding is a data compression technique that efficiently encodes data by assigning variable-length codes to characters based on their frequency of occurrence in the input. It is a form of entropy encoding that uses shorter codes for more frequent characters and longer codes for less frequent characters, resulting in overall compression of the data. The process begins by constructing a Huffman tree, a binary tree where each leaf node represents a character, and internal nodes are formed by combining the two least frequent characters. Traversing the tree from the root to each character yields its corresponding Huffman code. Huffman encoding is widely used in file compression and data transmission applications to reduce the size of data and optimize storage and transmission efficiency.

Efficiency

The runtime complexity of Huffman encoding depends on the number of unique characters in the input data and their frequencies. Constructing the Huffman tree involves building a binary heap, where each insertion and deletion has a time complexity of $\mathcal{O}(\log(n))$, where n is the number of characters. Since there are $n-1$ internal nodes in the tree, the overall time complexity for constructing the Huffman tree is $\mathcal{O}(n\log(n))$. After constructing the tree, generating the Huffman codes for each character requires traversing the tree, which takes $\mathcal{O}(\log n)$ time for each character, resulting in an additional $\mathcal{O}(n\log(n))$ complexity. Therefore, the overall runtime complexity of Huffman encoding is $\mathcal{O}(n\log(n))$, making it an efficient data compression technique for processing large datasets with varying character frequencies. It is important to note that the actual performance of Huffman encoding can vary based on the distribution of character frequencies in the input data. A summary of the runtime complexity of common operations in huffman coding can be seen below.

Operation	Best Case $\Omega(n)$	Average Case $\Theta(n)$	Worst Case $\mathcal{O}(n)$
Code Generation	$\Omega(n\log(n))$	$\Theta(n\log(n))$	$\mathcal{O}(n\log(n))$
Tree Construction	$\Omega(n\log(n))$	$\Theta(n\log(n))$	$\mathcal{O}(n\log(n))$

Structure

The structure of Huffman encoding and decoding involves two main steps: building the Huffman tree and generating the Huffman codes. In the encoding process, a frequency table is created to count the occurrences of each character in the input data. A priority queue, often implemented using a binary heap, is used to construct the Huffman tree, where each leaf node represents a character, and internal nodes are formed by combining the two least frequent characters. Traversing the tree from the root to each character yields its corresponding Huffman code. The generated Huffman codes are then used to encode the input data, replacing characters with their respective variable-length codes. In the decoding process, the encoded data is processed bit by bit, traversing the Huffman tree from the root until a leaf node is reached, which corresponds to a decoded character. The process continues until all encoded bits are processed, reconstructing the original data. Huffman encoding and decoding form a simple and efficient data compression technique, preserving data integrity while reducing its size for storage and transmission.

This implementation of the huffman encoding for this assignment consists of a custom structure and a custom class. The custom structure, **freq_info** consists of some custom data members. The declaration of this structure can be seen below:

- **count** - This is an integer value that represents the frequency of a given character in a string.
- **is_leaf** - This is a boolean value that represents if a node is a leaf in the tree or not.
- **left** - This is a smart pointer that represents the left child of a current node.
- **right** - This is a smart pointer that represents the right child of a current node.

With the above structure, we have a class that implements this structure. The declaration of the **Huffman** class can be seen below:

- **BuildEncodingTable** - 'BuildEncodingTable' creates a lookup table for an encoding process.
- **BuildTree** - 'BuildTree' builds a huffman tree based off of a given priority queue.
- **CombineNodes** - 'CombineNodes' combines to nodes into one node in a tree.
- **CreateLeaf** - 'CreateLeaf' creates a node and initializes it as a leaf node in a tree.
- **Decode** - 'Decode' returns a decoded string for a given input.
- **Encode** - 'Encode' returns an encoded string with a given lookup table.
- **IncrementLookup** - 'IncrementLookup' increments a lookup table by one integer value.
- **LoadQueue** - 'LoadQueue' turns key / value pairs into a lookup table.
- **TableHelper** - 'TableHelper' is a custom function that helps build an encoding table.

BuildEncodingTable

The function 'Huffman::BuildEncodingTable' is a member function of the 'Huffman' class responsible for constructing an encoding table, represented by a map, that associates characters with their corresponding Huffman codes. The function takes a shared pointer 'root' to a 'freq_info' object, which likely represents the root of the Huffman tree. Within the function, an empty map 'lookup' is created to store the character-to-code mappings. The function then calls the 'TableHelper' function, which recursively traverses the Huffman tree to build the encoding table by assigning Huffman codes to each character. The Huffman codes are represented as strings of 0s and 1s, where left edges in the tree correspond to 0, and right edges correspond to 1. The 'TableHelper' function updates the 'lookup' map as it traverses the tree. Finally, the function returns the completed encoding table 'lookup', containing the character-to-code mappings for the Huffman encoding. This table is essential for efficiently encoding the input data using Huffman coding, where frequent characters are assigned shorter codes, and infrequent characters are assigned longer codes, leading to data compression. Below is the definition of this algorithm.

BuildEncodingTable Algorithm

Below is the definition of the 'BuildEncodingTable' algorithm:

```
map<char, string> Huffman::BuildEncodingTable(shared_ptr<freq_info> root) {
    map<char, string> lookup;
    TableHelper(root, "", lookup);
    return lookup;
}
```

BuildTree

The function 'Huffman::BuildTree' is a member function of the 'Huffman' class responsible for constructing a Huffman tree from a given priority queue of frequency information represented by the 'tree_queue' data structure. The function follows the process of building a Huffman tree, where nodes with the lowest frequency are combined to form internal nodes until a single root node is created. Within the function, a 'while' loop is used to iterate through the priority queue 'q' until it contains only one element, which represents the root of the Huffman tree. During each iteration, the function extracts the two nodes with the lowest frequencies (probabilities) from the front of the priority queue. It then creates a new internal node, 'tempNode', with the sum of the frequencies of the left and right nodes, assigning the left and right nodes as its children. The 'tempNode' is then pushed back into the priority queue. This process continues until the priority queue is reduced to a single element, representing the root of the Huffman tree. Finally, the function returns a shared pointer to the root node of the Huffman tree. The constructed Huffman tree is essential for encoding and decoding data using Huffman coding, where the tree's structure determines the variable-length codes assigned to characters based on their frequencies. Below is the definition of this algorithm.

BuildTree Algorithm

Below is the definition of the 'BuildTree' algorithm:

```
shared_ptr<freq_info> Huffman::BuildTree(tree_queue& q) {
    while(q.size() > 1) {
        shared_ptr<freq_info> left = q.top();
        q.pop();
        shared_ptr<freq_info> right = q.top();
        q.pop();
        shared_ptr<freq_info> tempNode(new freq_info);
        tempNode->count = left->count + right->count;
        tempNode->left = left;
        tempNode->right = right;
        q.push(tempNode);
    }
    return q.top();
}
```

CombineNodes

The function 'Huffman::CombineNodes' is a member function of the 'Huffman' class responsible for creating a new internal node in a Huffman tree by combining two given leaf nodes. The function takes two shared pointers 'left' and 'right', which represent the leaf nodes to be combined. Within the function, a new 'freq_info' object is created, representing the internal

node that will have 'left' and 'right' as its children. The 'is_leaf' flag is set to 'false' to indicate that the node is not a leaf. The 'count' data member of the new node is calculated by summing the 'count' values of the 'left' and 'right' nodes, representing the combined frequencies of the characters they represent. The 'left' and 'right' nodes are then assigned as children of the new node. Finally, the function returns a shared pointer to the newly created internal node, which can be used to construct a Huffman tree during the process of building a Huffman encoding scheme. This function is a fundamental step in constructing a Huffman tree, where the frequencies of leaf nodes are combined to form internal nodes with appropriate frequencies, allowing efficient data compression and decompression using Huffman coding. Below is the definition of this algorithm.

CombineNodes Algorithm

Below is the definition of the 'CombineNodes' algorithm:

```
shared_ptr<freq_info> Huffman::CombineNodes(shared_ptr<freq_info> left,
                                            shared_ptr<freq_info> right) {
    shared_ptr<freq_info> ret(new freq_info);
    ret->is_leaf = false;
    ret->count = left->count + right->count;
    ret->left = left;
    ret->right = right;
    return ret;
}
```

CreateLeaf

The function 'Huffman::CreateLeaf' is a member function of the 'Huffman' class responsible for creating a leaf node in a Huffman tree that represents a character and its frequency of occurrence. The function takes two parameters: 'symbol', which represents the character to be encoded, and 'count', which indicates the frequency of that character in the input data. Within the function, a new 'freq_info' object is created to represent the leaf node. The character and frequency values are assigned to the 'symbol' and 'count' data members of the new node, respectively. The 'left' and 'right' pointers are set to 'nullptr' since leaf nodes have no children. The 'is_leaf' flag is set to 'true' to indicate that the node is a leaf. Finally, the function returns a shared pointer to the newly created leaf node. This function is essential in the process of constructing a Huffman tree, where characters and their frequencies are represented as leaf nodes and combined to form internal nodes during the tree-building process. The resulting Huffman tree is used for efficient data compression and decompression using Huffman coding. Below is the definition of this algorithm.

CreateLeaf Algorithm

Below is the definition of the 'CreateLeaf' algorithm:

```
shared_ptr<freq_info> Huffman::CreateLeaf(char symbol, int count) {
    shared_ptr<freq_info> ret(new freq_info);
    ret->symbol = symbol;
    ret->count = count;
    ret->left = nullptr;
    ret->right = nullptr;
    ret->is_leaf = true;
    return ret;
}
```

Decode

The function 'Huffman::Decode' is a member function of the 'Huffman' class responsible for decoding an encoded string using a given Huffman tree represented by the shared pointer 'root'. The function takes the 'root' node of the Huffman tree and the 'input' string, which represents the encoded data. Within the function, a new string 'decode' is created to store the decoded result. A node 'cur' is also created to keep track of the current position in the Huffman tree, initially set to the 'root'. The function then iterates through each character of the 'input' string, and based on the characters "." and "^" in the 'input', it moves either to the left or right child of the current node, traversing the Huffman tree. When reaching a leaf node, the function adds the corresponding character to the 'decode' string, indicating a successful decoding of a character. The 'cur' node is then reset to the 'root' to continue decoding the remaining characters. Finally, the function returns the 'decode' string, representing the original decoded data. This function is essential for efficiently decoding Huffman-encoded

data, as it navigates the Huffman tree based on the encoded input to reconstruct the original information. Below is the definition of this algorithm.

Decode Algorithm

Below is the definition of the 'Decode' algorithm:

```
string Huffman::Decode(shared_ptr<freq_info> root, string input) {
    string decode;
    shared_ptr<freq_info> cur = root;
    for (char c : input) {
        if (c == '.') {
            cur = cur->left;
        }
        else if (c == '^') {
            cur = cur->right;
        }
        if (cur->is_leaf) {
            decode += cur->symbol;
            cur = root;
        }
    }
    return decode;
}
```

Encode

The function 'Huffman::Encode' is a member function of the 'Huffman' class responsible for encoding an input string using a pre-built encoding table represented by the 'enc_table' map. The function takes two parameters: 'enc_table', which maps characters to their corresponding Huffman codes, and 'input', which represents the original data to be encoded. Within the function, an empty string 'code' is created to store the encoded result. The function then iterates through each character 'c' in the 'input' string and appends the corresponding Huffman code retrieved from the 'enc_table' to the 'code' string. By continuously appending Huffman codes for each character in the input string, the function constructs the encoded representation of the original data. Finally, the function returns the 'code' string, which represents the Huffman-encoded version of the input data. This function is crucial for efficiently encoding data using Huffman coding, where characters are replaced with variable-length codes based on their frequencies, resulting in data compression. Below is the definition of this algorithm.

Encode Algorithm

Below is the definition of the 'Encode' algorithm:

```
string Huffman::Encode(map<char, string> enc_table, string input) {
    string code;
    for (char c : input) {
        code += enc_table[c];
    }
    return code;
}
```

IncrementLookup

The function 'Huffman::IncrementLookup' is a member function of the 'Huffman' class responsible for updating a frequency lookup table represented by the 'lookup' map. The function takes two parameters: 'lookup', which is a map that associates characters with their frequencies, and 'symbol', which represents the character to be incremented in the lookup table. Within the function, the frequency count of the given 'symbol' is increased by one using the '++' operator, effectively updating its occurrence count in the lookup table. This function is typically used during the process of constructing a frequency table for characters in the input data. By updating the frequency lookup table with the occurrences of each character, it enables subsequent steps of building a Huffman tree and generating Huffman codes based on character frequencies, facilitating efficient data compression and decompression using Huffman coding. Below is the definition of this algorithm.

IncrementLookup Algorithm

Below is the definition of the 'IncrementLookup' algorithm:

```
void Huffman::IncrementLookup(map<char, int>& lookup, char symbol) {
    lookup[symbol]++;
}
```

LoadQueue

The function 'Huffman::LoadQueue' is a member function of the 'Huffman' class responsible for constructing a priority queue ('tree_queue') of frequency information based on a given lookup table ('lookup') containing characters and their corresponding frequencies. The function takes two parameters: 'lookup', which is a map that associates characters with their frequencies, and 'q', which represents the priority queue. Within the function, a 'for' loop is used to iterate through each element in the lookup table. For each character-frequency pair in the 'lookup', a new node ('shared_ptr<freq_info> node') is created to represent a leaf node in the Huffman tree. The character and frequency from the lookup table are assigned to the 'symbol' and 'count' data members of the new node, respectively, and the 'is_leaf' flag is set to 'true' to indicate that it is a leaf node. The new node is then pushed into the priority queue 'q'. This function is a crucial step in constructing the Huffman tree, where characters and their frequencies are represented as leaf nodes and combined to form internal nodes. The priority queue is essential for efficient data compression using Huffman coding, as it ensures that nodes with the lowest frequencies are combined first during tree construction. Below is the definition of this algorithm.

LoadQueue Algorithm

Below is the definition of the 'LoadQueue' algorithm:

```
void Huffman::LoadQueue(const map<char, int>& lookup, tree_queue& q) {
    for (const auto& pair : lookup) {
        shared_ptr<freq_info> node(new freq_info);
        node->symbol = pair.first;
        node->count = pair.second;
        node->is_leaf = true;
        q.push(node);
    }
}
```

TableHelper

The function 'Huffman::TableHelper' is a member function of the 'Huffman' class responsible for recursively constructing an encoding table ('lookup') used in Huffman coding. The function takes three parameters: 'node', which represents the current node in the Huffman tree, 'code', which represents the binary code constructed during the traversal, and 'lookup', which is the encoding table. Within the function, it checks if the current node is a leaf node by examining the 'is_leaf' flag. If the node is a leaf, it means it represents a character in the Huffman tree, and its corresponding binary code 'code' is added to the 'lookup' table, associating the character with its Huffman code. If the node is an internal node (not a leaf), the function continues recursively to traverse its left and right subtrees. When traversing left, it appends the character '.' to the current 'code', and when traversing right, it appends '^'. This process continues until all leaf nodes are reached and their corresponding Huffman codes are added to the 'lookup' table. The resulting 'lookup' table contains character-to-code mappings, allowing for efficient data encoding using Huffman coding. Below is the definition of this algorithm.

TableHelper Algorithm

Below is the definition of the 'TableHelper' algorithm:

```
void Huffman::TableHelper(shared_ptr<freq_info> node, string code, map<char, string>& lookup) {
    if (node->is_leaf) {
        lookup[node->symbol] = code;
    }
    else {
        if (node->left) {
            TableHelper(node->left, code + ".", lookup);
        }
```

```
        if (node->right) {
            TableHelper(node->right, code + "^", lookup);
        }
    }
}
```