Semester Project

Data Structures

“InterPlanetary File System”

Group Members:	Roll Numbers
Ahmed Bin Asim	**22I-0949**
Maryam Masood	**22I-1169**
Jawad Ahmad Khan	**22I-0791**

Table of Contents

Introduction and Methodology:

IPFS works like BitTorrent, making it simple to share and access content anytime, anywhere. While file sharing is a common use, IPFS can do more than that.

Our implementation of IPFS provides the user with a range of options and functionalities to choose from. In the beginning the user must choose the total number of machines they want to enter and the size of the identifier space the user wants to work with. That done the program then prompts the user to enter machines.

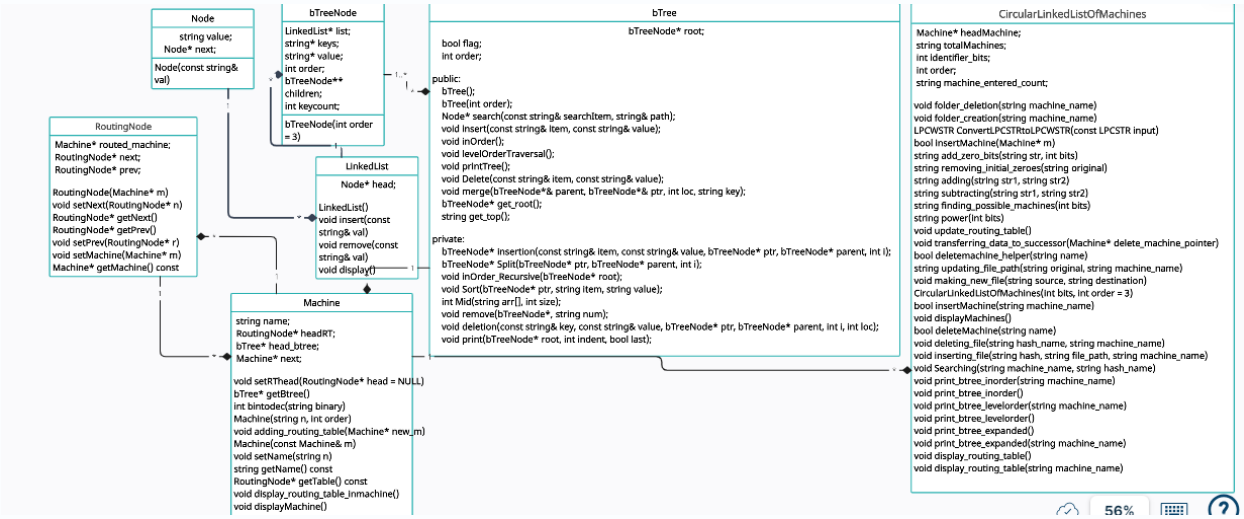
To enter a machine the user may choose one of two options either to enter the hexadecimal hash value for the machine or the name of the machine which is then run through our hashing algorithm and then inserted.

After this the program displays various options to the user, mainly:

1. Add a new machine
2. Delete an existing machine
3. Print the Routing Table for a machine
4. Delete files from a specific machine
5. Insert files from a specific machine
6. Searching files from a specific machine
7. View the B-tree of a machine
8. Viewing all Machines
9. Clear Screen
10. Exit Program

All in all, our IPFS implementation offers users flexibility in managing machines within a chosen identifier space. Users can easily add, delete machines, manipulate files, and explore routing tables, providing a comprehensive solution for decentralized content sharing.

UML



Hashing:

SHA-1, which stands for Secure Hash Algorithm 1, is a widely used cryptographic hashing function that produces a fixed-size (160-bit) hash value from input data of any size. Developed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) in 1993, SHA-1 was initially designed to provide a secure means of creating digital signatures and ensuring data integrity.

We were asked to make use of SHA 1 in our implementation of the Interplanetary File System. After installing the required CryptoPP library, we used the documentation of this library to put the algorithm to use.

We implemented a function to open a text file and read its content into locally declared variables which we then passed by reference to the following function:

~ **StringSource**(source, true, **new HashFilter**(sha1, **new HexEncoder**(new StringSink(hash))));

In this function, the StringSource class is used to supply the input data (source) to the pipeline.

The data is then processed through a HashFilter with the SHA-1 hash function (sha1). The resulting hash is then encoded in hexadecimal format using the HexEncoder. Finally, the hashed and encoded value is written to a string sink (StringSink) named hash. This sequence of operations allows for the generation of a SHA-1 hash in hexadecimal representation for the given input string.

After generation of the hash we mask the hexadecimal hash value to the number of digits we require as per the Identifier Space. This masked hash value is then used to sort data appropriately in B-trees.

Additionally, we implemented 2 converter functions for hexadecimal to binary conversion and vice versa to offer a variety in I/O operations for hash generation if the user so wishes.

Due to the masking and some drawbacks of the SHA-1 algorithm, hashes generated may be the same for different inputs although that happens extremely rarely.

B-Trees (MAX):

The B-tree extends the concept of a binary search tree by accommodating nodes with more than two children. In contrast to other self-balancing binary search trees, the B-tree proves particularly effective in storage systems that involve the frequent reading and writing of substantial data blocks, typical in scenarios like databases and file systems.

The B-tree serves as an indexing mechanism for data, ensuring fast access to the actual data stored on disks. This is especially crucial as retrieving values from a large database stored on a disk can be a time-consuming process.

Classes:

class Node {

```
public:
        string value;
        Node* next;
        Node(const string& val) ;
};
class LinkedList {
public:
        Node* head;
        LinkedList();
        void insert(const string& val);
};

class bTreeNode
{
public:
        LinkedList* list;
        string* keys;
        string* value;
        int order;
        bTreeNode** children;
        int keycount;
        bTreeNode(int order = 3);
```

```
};
class bTree
{
protected:
        bTreeNode* root;
        bool flag;
        int order;
public:
        bTree();
        bTree(int order);
        Node* search(const string& searchItem, string& path);
        void insert(const string& item, const string& value);
        void inOrder();
        void levelOrderTraversal();
        void printTree();
        void Delete(const string& item, const string& value);
        void merge(bTreeNode*& parent, bTreeNode*& ptr, int loc, string key);
        bTreeNode* get_root();
        string get_top();
private:
        bTreeNode* insertion(const string& item, const string& value, bTreeNode* ptr,
bTreeNode* parent, int i);
        bTreeNode* Split(bTreeNode* ptr, bTreeNode* parent, int i);
```

```
    void inOrder_Recursive(bTreeNode* root);

    void Sort(bTreeNode* ptr, string item, string value);

    int Mid(string arr[], int size);

    void remove(bTreeNode*, string num);

    void deletion(const string& key, const string& value, bTreeNode* ptr, bTreeNode* parent, int i, int loc);

    void print(bTreeNode* root, int indent, bool last);

};
```

Attributes:

Linked List* List : A pointer to a linked list used for managing duplicates associated with keys in a B-tree node.

String* keys: Description: A pointer to an array containing the keys present in the node.

String* value: A pointer to an array containing values associated with the keys in the node.

Int order: The order of the B-tree, indicating the maximum number of children each internal node can have.

bTreeNode** children: A pointer to an array of child nodes.

keycount: The number of keys currently present in the node.

bTreeNode* root: A pointer to the root node of the B-tree.

Bool flag: A boolean flag, the purpose of which is to check if root is updated during insertion.

Int order: The order of the B-tree, representing the maximum number of children each internal node can have.

Insertion:

In B-trees, the insertion of a new key follows a specific procedure. The key is always inserted at a leaf node, causing the B-tree to grow upwards. Similar to Binary Search Trees (BST), the process starts from the root and traverses down until a leaf node is reached. At the leaf node, the key is inserted. Unlike BSTs, B-trees have a predefined limit on the number of keys a node can contain, denoted as "order - 1". Therefore, upon inserting a key into a node, it is ensured that the node has additional space.

Algorithm:

The **void insert(const string& item, const string& value)** function takes the key and value (path) and passes them to the recursive function **bTreeNode* insertion(const string& item, const string& value, bTreeNode* ptr, bTreeNode* parent, int i).**

If the root node is empty, a new node is created, and the new item is inserted into the keys array. Otherwise, the B-tree is traversed recursively to find the correct place for the new item (key) insertion. After identifying the correct position, the current node and item to be inserted are sent to the **void Sort(bTreeNode* ptr, string item, string value)** function.

The `void Sort` function traverses the “ptr->keys” array to determine the correct index for the new item insertion. The array must always be in ascending order for the B-tree to maintain the Binary Search Tree property. After finding the correct position, it right-shifts all the keys after that index to make space for the new item and then inserts the new item.

Once the item is inserted, the key count of the current node is checked. If it is equal to the order of the B-tree, the node and its parent are passed to the **bTreeNode* Split(bTreeNode* ptr, bTreeNode* parent, int i)** function. This function begins by passing the current node to the **int Mid(string arr[], int size)** function, which returns the middle index of the keys array. If the size of the array is even, it returns `size/2 - 1`, making the tree left-biased.

After determining the middle index, two new nodes are created: LeftChild and RightChild. Keys less than the middle key are stored in LeftChild along with their children, while keys greater than the middle key are stored in RightChild with their corresponding children.

Next, the middle element is removed from the node and inserted into the parent node using the Sort function. The position of the middle element is found in the parent node, and the children of that parent node are right-shifted by one position after the middle key index to make space for the right child. Finally, the left and right children of the middle element are inserted into the parent node.

This process of splitting is recursively continued for the entire B-tree until the root is encountered.

Deletion:

Deleting a key from a B-tree is a complex process compared to insertion because it involves removing keys from any node, not just leaves. Adjustments to rearrange the node's children are necessary when deleting a key from an internal node. Similar to insertion, it's crucial to ensure that deletion operations do not violate the B-tree properties. This involves preventing a node from becoming too small during deletion, except for the root, which is allowed to have fewer than the minimum required number of keys (order-1). Backtracking may be necessary in a deletion approach, similar to how a simple insertion algorithm might backtrack when encountering a full node along the path to the insertion point.

Algorithm:

The **void Delete(const string& item, const string& value)** function takes the item (key) to be deleted and the value (path), passing them to the **void deletion(const string& key, const string& value, bTreeNode* ptr, bTreeNode* parent, int i, int loc)** function.

The process involves a recursive search for the key to be deleted. When the key is found, the function first checks if the node is a leaf node.

If the key to be deleted is in any of the internal nodes, it is swapped with its predecessor (maximum key of the left subtree). After swapping, the key is recursively searched again until it reaches the location of its previous predecessor.

Following the search, the key count of that leaf node is checked. If it is greater than the minimum allowable number of keys for that node (i.e., (order + 1) / 2 – 1), then the key is simply deleted using the **void remove(bTreeNode*, string num)** function. The remove() function identifies the location of the key to be deleted and left-shifts all keys after that location by one position. If 'keycount == (order + 1) / 2 – 1', then the maximum key of the sibling is borrowed if the left sibling contains more than the minimum keys. If the left sibling doesn't have more than the minimum keys, borrowing occurs from the right sibling if it contains more than the minimum number of keys. If both the left and right children do not have more than the minimum number of keys, the node is merged with the left child (along with the parent key) if it exists; otherwise, it merges with the right child and removes the key to be deleted. The process recursively checks if the node contains fewer than the minimum number of keys, passing it to the **void merge(bTreeNode*& parent, bTreeNode*& ptr, int loc, string key)** function until the root is reached.

Routing Table:

```
class RoutingNode
{
private:
        Machine* routed_machine;
        RoutingNode* next;
        RoutingNode* prev;
public:
        RoutingNode(Machine* m);
        void setNext(RoutingNode* n);
        RoutingNode* getNext() ;
```

```
        RoutingNode* getPrev() ;
        void setPrev(RoutingNode* r);
        void setMachine(Machine* m);
        Machine* getMachine() const;
};
```

The class Routing Node is a **doubly linked class which acts as the routing table**. It has a **pointer routed_machine which points to the machine it points to**. Other than that it has **next and prev** which points to the **predecessor and successor nodes** of the routing table.

Machine

```
class Machine
{
private:
        string name;
        RoutingNode* headRT;
        bTree* head_btree;
        Machine* next;
public:
        void setRThead(RoutingNode* head = NULL);
        bTree* getBtree() ;
        void setBtree(bTree* bt);
        int bintodec(string binary);
        Machine(string n, int order);
        void adding_routing_table(Machine* new_m);
        Machine(const Machine& m);
        void setName(string n) ;
        string getName() const ;
        RoutingNode* getTable() const ;
```

```
        Machine* getNext() ;
        void setNext(Machine* m) ;
        void display_routing_table_inmachine();
        void displayMachine();
};
```

The class Machine acts as a folder in the computer. It has a name as a unique identifier. No two machines can have the same name. It is a **circular Linked List** and thus contains the **next pointer** which points to the next machine in line. Since each machine has a Routing Table, the **headRt** acts as the head of the routing table. Since all data members are private, it contains **getters and setters** to use them. Each machine stores files in the form of a B-tree so a **pointer head_btree** is introduced as well.

Circular Linked List of Machines

```
class CircularLinkedListOfMachines
{
private:
        Machine* headMachine;
        string totalMachines;
        int identifier_bits;
        int order;
        string machine_entered_count;
        void folder_deletion(string machine_name);
        void folder_creation(string machine_name);
        LPCWSTR ConvertLPCSTRtoLPCWSTR(const LPCSTR input);
        bool insertMachine(Machine* m);
        string removing_initial_zeroes(string original);
        string adding(string str1, string str2);
        string subtracting(string str1, string str2);
        string finding_possible_machines(int bits);
```

```
        string power(int bits);
        string updating_file_path(string original, string machine_name);
        void making_new_file(string source, string destination, string temp_destination);
        void update_routing_table();
        bool deletemachine_helper(string name);
        void deleting_all(Machine* last_machine);
        void transferring_data_to_successor(Machine* delete_machine_pointer);
        void predecessor_transfer(Machine* new_machine, Machine* successor);
Public:
        static string add_zero_bits(string str, int bits)
         CircularLinkedListOfMachines(int bits, int order = 3);
         bool insertMachine(string machine_name);
         void displayMachines();
         bool deleteMachine(string name);
         void deleting_file(string hash_name, string machine_name);
         void Searching(string hash_name, string machine_name);
         void inserting_file(string hash, string file_path, string machine_name);
         void print_btree_inorder(string machine_name);
         void print_btree_inorder();
         void print_btree_levelorder(string machine_name);
         void print_btree_levelorder();
         void print_btree_expanded();
         void print_btree_expanded(string machine_name);
         void display_routing_table();
         void display_routing_table(string machine_name);
};
```

CircularLinkedListOfMachines is the place where the main functionality of the code is performed. It has the head node of the Machines **headMachine**, which points to the first/smallest node of the list. It keeps the count of the total machines that can be inserted by

using the given formulae **2^(identifier_bits) - 1**, where identifier bits are taken as an input. It also keeps track of the number of machines entered. **Total_machines and machine_count_entered** are used with data types **strings** to keep track of large numbers as well. The functions **ConvertLPCSTRtoLPCWSTR, folder_creation** and **folder_deletion** takes care of folders being added or deleted once a machine is added or deleted. The functions **add_zero_bits, removing_initial_zeroes, adding, subtracting, finding_possible_machines** and **power** are used for string manipulation. The functions **updating_file_path** and **making_new_file** are used for file creation. The **transferring_data_to_successor, deletemachine_helper** and **deleting_all** handle machine deletions. The **public methods** include various overloaded functions for **printing all machines**, **printing routing tables** of machine/machines, as well as 3 types of traversals: **In-Order**, **Level-Order** as well as an **expanded view**. Moreover, file addition and deletion, machine addition and deletion as well as methods for searching are also present.

Contributions

-> Ahmed bin Asim: Took the lead and was responsible for the initial structure of the code. Provided with the function prototypes so the entire group can work together remotely. Responsible for all converter functions and string manipulation functions used in CircularLinkedListOfMachines. Also was responsible for the main layout and design as well as the SHA 1 function.

-> Maryam Masood: Was responsible for B-Trees entirely so no issue arose regarding it as the entire code depended on it. Was provided with the function prototypes so each function would return exactly what was needed. Began the work earliest and was responsible for half of the testing of the program.

-> Jawad Khan: Was responsible for the inner workings of the classes. Was provided with the string manipulation functions and data members of the classes. Was also responsible for folder

and file handling. Also compiled the work making sure that all 3 group members classes and functions worked together with each other. Also responsible for testing of the program.