Inter-Planetary File System

Group Members:
- Shayan Memon (22i-0773)
- Ahmad Ryan (22i-0781)
- Zubair Adnan (22i-0789)

Semester Final Project

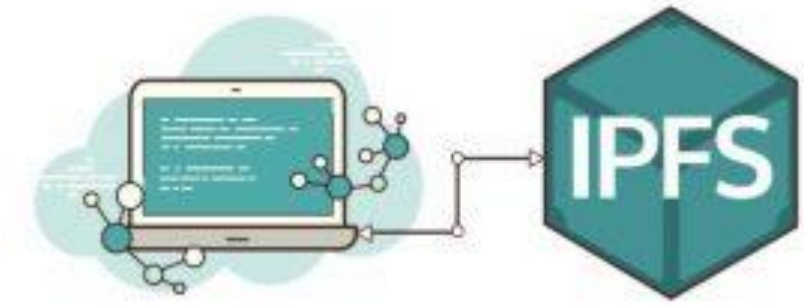
—

Data Structures (CS2001)

—

CS-3B – FAST NUCES Islamabad

Contents



Introduction

IPFS is kind of a filesharing and file storage system, which efficiently stores the files into different machines based on their hashes and lets the users to efficiently retrieve their data. Our implementation of IPFS is discussed further in this document.

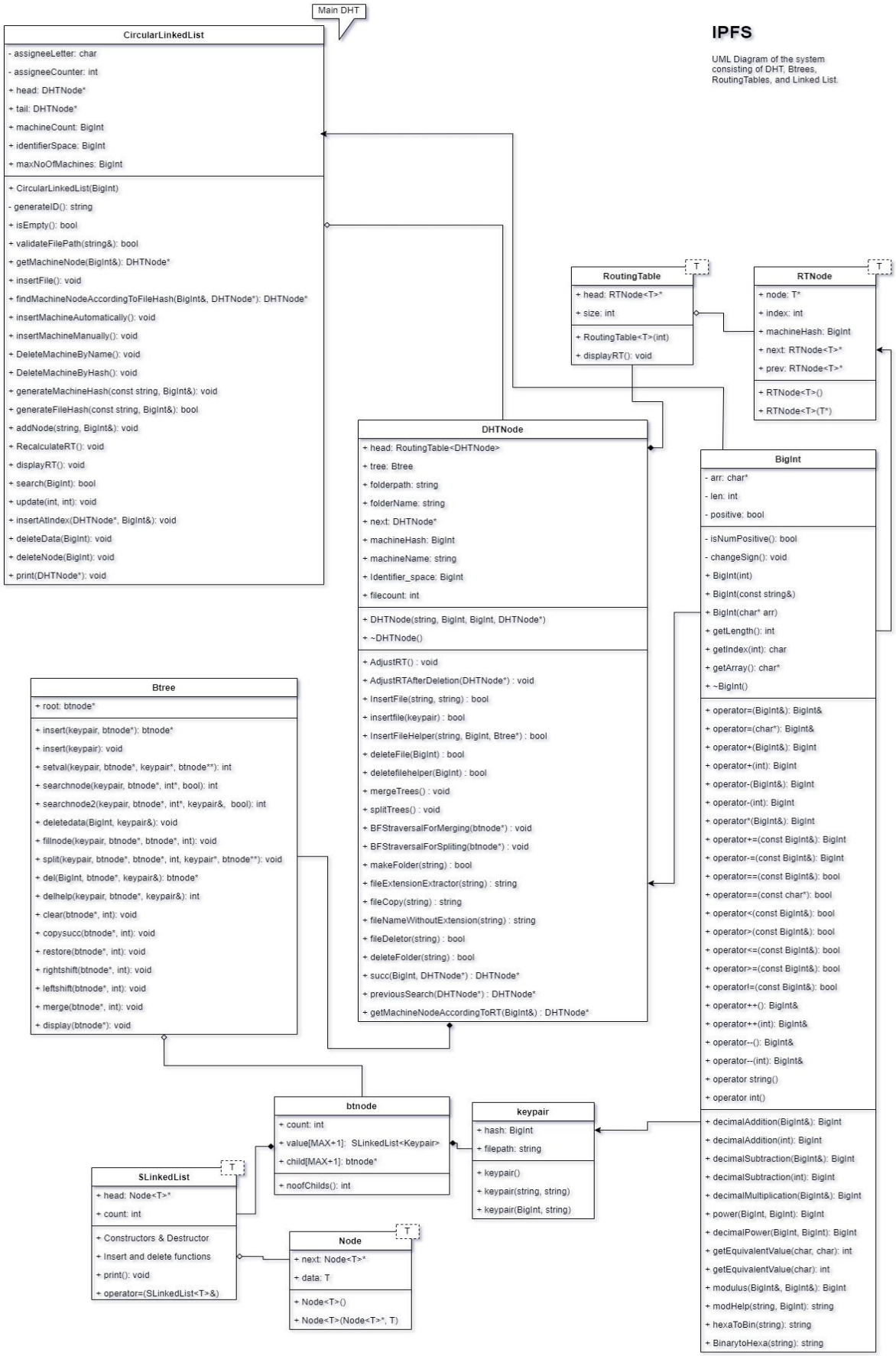
Data Structures used

- **Circular Singly Linked List:** for the connection of the Machines.
- **Doubly Linked List:** for the implementation of Routing Tables.
- **Btree:** for the storage of files in a machine.
- **Singly Linked List:** for the chaining of files with same hashes.

Also, character array was used to make another class of BigInt, which was used for storing the hashes and performing operations on it. It was made in a way to store and perform operations on hexadecimal values. It was made since there was no support for numbers more than 64bit, and hexadecimals.

Classes Information

It is shown through the UML class diagram shown below:


Main DHT
IPFS
UML Diagram of the system consisting of DHT, Btrees, RoutingTables, and Linked List.
CircularLinkedList
- assigneeLetter: char
- assigneeCounter: int
+ head: DHTNode*
+ tail: DHTNode*
+ machineCount: BigInt
+ identifierSpace: BigInt
+ maxNoOfMachines: BigInt
+ CircularLinkedList(BigInt)
- generateID(): string
+ isEmpty(): bool
+ validateFilePath(string&): bool
+ getMachineNode(BigInt&): DHTNode*
+ insertFile(): void
+ findMachineNodeAccordingToFileHash(BigInt&, DHTNode*): DHTNode*
+ insertMachineAutomatically(): void
+ insertMachineManually(): void
+ DeleteMachineByName(): void
+ DeleteMachineByHash(): void
+ generateMachineHash(const string, BigInt&): void
+ generateFileHash(const string, BigInt&): bool
+ addNode(string, BigInt&): void
+ RecalculateRT(): void
+ displayRT(): void
+ search(BigInt): bool
+ update(int, int): void
+ insertAtIndex(DHTNode*, BigInt&): void
+ deleteData(BigInt): void
+ deleteNode(BigInt): void
+ print(DHTNode*): void
RoutingTable
T
+ head: RTNode<T>*
+ size: int
+ RoutingTable<T>(int)
+ displayRT(): void
RTNode
T
+ node: T*
+ index: int
+ machineHash: BigInt
+ next: RTNode<T>*
+ prev: RTNode<T>*
+ RTNode<T>()
+ RTNode<T>(T*)
DHTNode
+ head: RoutingTable<DHTNode>
+ tree: Btree
+ folderpath: string
+ folderName: string
+ next: DHTNode*
+ machineHash: BigInt
+ machineName: string
+ Identifier_space: BigInt
+ filecount: int
+ DHTNode(string, BigInt, BigInt, DHTNode*)
+ ~DHTNode()
+ AdjustRT() : void
+ AdjustRTAfterDeletion(DHTNode*) : void
+ InsertFile(string, string) : bool
+ insertfile(keypair) : bool
+ InsertFileHelper(string, BigInt, Btree*) : bool
+ deleteFile(BigInt) : bool
+ deletefilehelper(BigInt) : bool
+ mergeTrees() : void
+ splitTrees() : void
+ BFStraversalForMerging(btnode*) : void
+ BFStraversalForSpliting(btnode*) : void
+ makeFolder(string) : bool
+ fileExtensionExtractor(string) : string
+ fileCopy(string) : string
+ fileNameWithoutExtension(string) : string
+ fileDeletor(string) : bool
+ deleteFolder(string) : bool
+ succ(BigInt, DHTNode*) : DHTNode*
+ previousSearch(DHTNode*) : DHTNode*
+ getMachineNodeAccordingToRT(BigInt&) : DHTNode*
BigInt
- arr: char*
- len: int
- positive: bool
- isNumPositive(): bool
- changeSign(): void
+ BigInt(int)
+ BigInt(const string&)
+ BigInt(char* arr)
+ getLength(): int
+ getIndex(int): char
+ getArray(): char*
+ ~BigInt()
+ operator=(BigInt&): BigInt&
+ operator=(char*): BigInt&
+ operator+(BigInt&): BigInt
+ operator+(int): BigInt
+ operator-(BigInt&): BigInt
+ operator-(int): BigInt
+ operator*(BigInt&): BigInt
+ operator+=(const BigInt&): BigInt
+ operator-=(const BigInt&): BigInt
+ operator==(const BigInt&): bool
+ operator==(const char*): bool
+ operator<(const BigInt&): bool
+ operator>(const BigInt&): bool
+ operator<=(const BigInt&): bool
+ operator>=(const BigInt&): bool
+ operatorI=(const BigInt&): bool
+ operator++(): BigInt&
+ operator++(int): BigInt&
+ operator--(): BigInt&
+ operator--(int): BigInt&
+ operator string()
+ operator int()
+ decimalAddition(BigInt&): BigInt
+ decimalAddition(int): BigInt
+ decimalSubtraction(BigInt&): BigInt
+ decimalSubtraction(int): BigInt
+ decimalMultiplication(BigInt&): BigInt
+ power(BigInt, BigInt): BigInt
+ decimalPower(BigInt, BigInt): BigInt
+ getEquivalentValue(char, char): int
+ getEquivalentValue(char): int
+ modulus(BigInt&, BigInt&): BigInt
+ modHelp(string, BigInt): string
+ hexaToBin(string): string
+ BinarytoHexa(string): string
Btree
+ root: btnode*
+ insert(keypair, btnode*): btnode*
+ insert(keypair): void
+ setval(keypair, btnode*, keypair*, btnode**): int
+ searchnode(keypair, btnode*, int*, bool): int
+ searchnode2(keypair, btnode*, int*, keypair&, bool): int
+ deletedata(BigInt, keypair&): void
+ fillnode(keypair, btnode*, btnode*, int): void
+ split(keypair, btnode*, btnode*, int, keypair*, btnode**): void
+ del(BigInt, btnode*, keypair&): btnode*
+ delhelp(keypair, btnode*, keypair&): int
+ clear(btnode*, int): void
+ copysucc(btnode*, int): void
+ restore(btnode*, int): void
+ rightshift(btnode*, int): void
+ leftshift(btnode*, int): void
+ merge(btnode*, int): void
+ display(btnode*): void
btnode
+ count: int
+ value[MAX+1]: SLinkedList<Keypair>
+ child[MAX+1]: btnode*
+ noofChilds(): int
keypair
+ hash: BigInt
+ filepath: string
+ keypair()
+ keypair(string, string)
+ keypair(BigInt, string)
SLinkedList
T
+ head: Node<T>*
+ count: int
+ Constructors & Destructor
+ Insert and delete functions
+ print(): void
+ operator=(SLinkedList<T>&)
Node
T
+ next: Node<T>*
+ data: T
+ Node<T>()
+ Node<T>(Node<T>*, T)

Algorithms:

The following algorithm has been employed to efficiently search and insert files on the system:
First from a given machine, if the hash of the file is equal to the hash of the machine, it is inserted in the same machine's btree. If the hash of the file is greater than the machine, then the whole system is divided into two parts, one from the first machine to the current, and other is the current to last machine. In this case, The file is inserted into a machine found in the second half of the system by efficient traversal using Routing Tables. Otherwise, it uses the same technique in the first half of the system.

Time Complexities:

For the above-mentioned algorithm, it is half of $\log_2 n$ which is **$O(\log_2 n)$.**
For the adjustment of Routing tables, it is **O(n)** since the circular linked list was restricted to be singly.

Testing and Error Handling

The program was vigorously tested for many known cases, and all the errors that appeared had been addressed.

Performance

The program can handle kilobytes of files with full efficiency and is trusted to handle all the complications that can appear.

Dependencies

The program uses a pre-built hashing algorithm called "SHA-1" whose code was taken from GitHub. It can hash either a string (machine name) or the content of a file and returns a string in 160-bit hexadecimal format, which is then stored in the system using BigInt class made by us.

Endnote

This document was a brief overview of the project. The project will be explained in detail in the demos by all the group members.

Thank you!