**InterPlanetary File System (IPFS)**

Table of Contents

[Program components 3](#_Toc153551218)

[Driver\_DHT 3](#_Toc153551219)

[Ring\_DHT 5](#_Toc153551220)

[Automatic generation: 10](#_Toc153551221)

[B-Tree: 11](#_Toc153551222)

# Program components

* Driver\_DHT.cpp
  + This is the main controller for our DHT. It contains menu printing functionality as well as object to Ring\_DHT class. Users can select multiple options to perform operations corresponding to options.
* Ring\_DHT.h
  + This header file contains the class of Node\_Machine that represents a single node in our circular linked list of machines.
  + This header also contains class of Ring\_DHT that in essence emulates our complete system having objects of Node\_Machine, SHA, and variables necessary for our system.
* Routing\_Table.h
  + This header contains Routing\_Table and Node\_Routing\_Table classes responsible for maintaining routing table of each Node\_machine object/node.
* SHA.h
  + The SHA functions takes the path of the file as string and returns the 40 character long string hash in the form of Hexadecimal and then it is converted into decimal using the Hex\_Dec function. The bit\_extractor function uses the required no of bits given the size of the identifier space in bits.
* Bigint.h
  + The BigInt classes is used to handle the Decimal values that the built-in primitive data types of C++ does not support providing functionality including addition, subtraction, multiplication, comparison operators, etc.
* Btree.h
  + The B-tree exhibits adaptability by allowing flexible selection of its degree. Nodes store key-value pairs, with numeric keys as unique identifiers and string values indicating corresponding file paths. Efficient handling of duplicate keys is achieved through linked lists. The B-tree ensures balance during key-value pair insertion and adapts during deletion, employing techniques like filling, borrowing, or merging keys.

# Driver\_DHT

Main functionality of this Driver\_DHT.cpp is to act as an interface between User and our other drivers. It asks the user to give identifier space i.e. Bit space, Number of machines, order of Btree, and options to manually enter machine ids or automatically generate them. Then creates an object of Ring\_DHT class and initializes important variables to class.

Then asks the user whether it would enter a string for which we would create a hash value or user would directly assign the machine id without the hash key. As machines are being added it shows the current doubly linked list of machines and creates their folders respectively.

After that it prints a menu for the user to select from multiple options to perform operations on the system.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

# Ring\_DHT

This class has objects from other classes and provides the functionality and functions to achieve those functionalities. The class have several functions such as:

* Add machine.
* Delete machine.
* Automatic assignment of machines.
* Add file btree.
* Delete file to btree.
* Set routing table.
* Routing (for searching machine using routing table).
* Print functionalities.
* Several helper functions.

A computer screen shot of a code

Description automatically generated

After the initialization in Driver\_DHT following are some functions performed.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer screen

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

## Automatic generation

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

# B-Tree:

* B-tree allows flexible selection of the tree's degree, adapting seamlessly to diverse scenarios and data sizes.
* Each B-tree node contains an array of key-value pairs, with numeric keys serving as unique identifiers and string values indicating corresponding file paths.
* By building linked lists duplicate keys are effectively controlled and numerous values associated with a single key can be stored and retrieved.
* The B-tree supports key-value pair insertion, maintaining balance by splitting nodes when necessary. Special attention is given to handling duplicates during insertion.
* Deletion handles scenarios such as removal from internal or leaf nodes. B-tree adapts with balancing techniques like filling, borrowing, or merging keys from neighboring nodes.
* B-tree facilitates key retrieval through a recursive search, efficiently navigating the tree's structure to locate the desired key or indicating its absence.
* Every exception is dealt with appropriately.

# SHA

## Definitions of Bit Strings and Integers

The following terminology related to bit strings and integers will be used:

* A hex digit is an element of the set {0, 1, ... , 9, A, ... , F}. A hex digit is the representation of a 4-bit string. Examples: 7 = 0111, A = 1010.
* A word equals a 32-bit string which may be represented as a sequence of 8 hex digits. To convert a word to 8 hex digits each 4-bit string is converted to its hex equivalent as described in (a) above. Example: 1010 0001 0000 0011 1111 1110 0010 0011 = A103FE23.
* An integer between 0 and 2^32 - 1 inclusive may be represented as a word. The least significant four bits of the integer are represented by the right-most hex digit of the word representation. Example: the integer 291 = 2^8+2^5+2^1+2^0=256+32+2+1 is represented by the hex word, 00000123.

If z is an integer, 0 <= z < 2^64, then z = (2^32)x + y where 0 <= x < 2^32 and 0 <= y < 2^32. Since x and y can be represented as words X and Y, respectively, z can be represented as the pair of words (X,Y).

* Block = 512-bit string.A block (e.g., B) may be represented as a sequence of 16 words.

### Operations on Words

The following logical operators will be applied to words:

* Bitwise logical word operations
* X AND Y = bitwise logical "and" of X and Y.
* X OR Y = bitwise logical "inclusive-or" of X and Y.
* X XOR Y = bitwise logical "exclusive-or" of X and Y.
* NOT X = bitwise logical "complement" of X.

Example:

01101100101110011101001001111011

XOR 01100101110000010110100110110111

------------------------------------------------

= 00001001011110001011101111001100

* The operation X + Y is defined as follows: words X and Y represent integers x and y, where 0 <= x < 2^32 and 0 <= y < 2^32. For positive integers n and m, let n mod m be the remainder upon dividing n by m. Compute z = (x + y) mod 2^32. Then 0 <= z < 2^32. Convert z to a word, Z, and define Z = X +Y.
* The circular left shift operation S^n(X), where X is a word and n is an integer with 0 <= n < 32, is defined by

S^n(X) = (X << n) OR (X >> 32-n).

In the above, X << n is obtained as follows: discard the left-most n bits of X and then pad the result with n zeroes on the right (the result will still be 32 bits). X >> n is obtained by discarding the right-most n bits of X and then padding the result with n zeroes on the left. Thus S^n(X) is equivalent to a circular shift of X by n positions to the left.

### Message Padding

SHA-1 is used to compute a message digest for a message or data file that is provided as input. The message or data file should be considered to be a bit string. The length of the message is the number of bits in the message (the empty message has length 0). If the number of bits in a message is a multiple of 8, for compactness we can represent the message in hex. The purpose of message padding is to make the total length of a padded message a multiple of 512. SHA-1 sequentially processes blocks of 512 bits when computing the message digest. The following specifies how this padding shall be performed. As a summary, a "1" followed by m "0"s followed by a 64-bit integer are appended to the end of the message to produce a padded message of length 512 \* n. The 64-bit integer is the length of the original message. The padded message is then processed by the SHA-1 as n 512-bit blocks.

Suppose a message has length l < 2^64. Before it is input to the SHA-1, the message is padded on the right as follows:

* "1" is appended. Example: if the original message is "01010000", this is padded to "010100001".
* "0"s are appended. The number of "0"s will depend on the original length of the message. The last 64 bits of the last 512-bit block are reserved for the length l of the original message.

Example: Suppose the original message is the bit string

01100001 01100010 01100011 01100100 01100101.

After step (a) this gives

01100001 01100010 01100011 01100100 01100101 1.

Since l = 40, the number of bits in the above is 41 and 407 "0"s are appended, making the total now 448. This gives (in hex)

61626364 65800000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000.

* Obtain the 2-word representation of l, the number of bits in the original message. If l < 2^32 then the first word is all zeroes. Append these two words to the padded message.

Example: Suppose the original message is as in (b). Then l = 40 (note that l is computed before any padding). The two-word representation of 40 is hex 00000000 00000028. Hence the final padded message is hex

61626364 65800000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000028.

The padded message will contain 16 \* n words for some n > 0. The padded message is regarded as a sequence of n blocks M(1) ,M(2), first characters (or bits) of the message.

### Functions and Constants Used

A sequence of logical functions f(0), f(1),..., f(79) is used in SHA-1. Each f(t), 0 <= t <= 79, operates on three 32-bit words B, C,D and produces a 32-bit word as output. f(t;B,C,D) is defined as follows: for words B, C, D,

f(t; B,C,D) = (B AND C) OR ((NOT B) AND D) ( 0 <= t <= 19)

f(t;B,C,D) = B XOR C XOR D (20 <= t <= 39)

f(t;B,C,D) = (B AND C) OR (B AND D) OR (C AND D) (40 <= t <= 59)

f(t;B,C,D) = B XOR C XOR D (60 <= t <= 79).

A sequence of constant words K(0), K(1), ... , K(79) is used in the SHA-1. In hex these are given by

K(t) = 5A827999 ( 0 <= t <= 19)

K(t) = 6ED9EBA1 (20 <= t <= 39)

K(t) = 8F1BBCDC (40 <= t <= 59)

K(t) = CA62C1D6 (60 <= t <= 79).

## Method

The message digest is computed using the message padded.

. The computation is described using two buffers, each

consisting of five 32-bit words, and a sequence of eighty 32-bit

words. The words of the first 5-word buffer are labeled A,B,C,D,E.

The words of the second 5-word buffer are labeled H0, H1, H2, H3, H4.

The words of the 80-word sequence are labeled W(0), W(1),..., W(79).

A single word buffer TEMP is also employed.

To generate the message digest, the 16-word blocks M(1), M(2),...,

M(n) are processed in order. The processing of

each M(i) involves 80 steps.

Before processing any blocks, the H's are initialized as follows: in

hex,

H0 = 67452301

H1 = EFCDAB89

H2 = 98BADCFE

H3 = 10325476

H4 = C3D2E1F0.

Now M(1), M(2), ... , M(n) are processed. To process M(i), we

proceed as follows:

a. Divide M(i) into 16 words W(0), W(1), ... , W(15), where W(0)

is the left-most word.

b. For t = 16 to 79 let

W(t) = S^1(W(t-3) XOR W(t-8) XOR W(t-14) XOR W(t-16)).

c. Let A = H0, B = H1, C = H2, D = H3, E = H4.

d. For t = 0 to 79 do

TEMP = S^5(A) + f(t;B,C,D) + E + W(t) + K(t);

E = D; D = C; C = S^30(B); B = A; A = TEMP;

e. Let H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4

+ E.

After processing M(n), the message digest is the 160-bit string

represented by the 5 words

H0 H1 H2 H3 H4.