‘DNS LOOKUP SYSTEM’
DATA STRUCTURES AND ALGORITHM
PROJECT REPORT

By
GROUP
Avadhut Gurjar
Mohsin Patel
Shraddha Pandhe
Contents
1. Introduction ........................................................................................................................................... 3
2. DNS Recursive Query Mechanism: ........................................................................................................ 5
   2.1. Client Side ......................................................................................................................................... 5
   2.2. DNS Server Side ............................................................................................................................... 6
   2.3. Root DNS Server Side ....................................................................................................................... 7
3. Search Algorithms: ................................................................................................................................. 8
   3.1. Binary Search using arrays .............................................................................................................. 8
      3.1.1. Data Members: ......................................................................................................................... 8
      3.1.2. Search Function ....................................................................................................................... 8
      3.1.3. Insertion Function .................................................................................................................... 9
      3.1.4. Binary Search using arrays Complexity .................................................................................... 9
   3.2. Linear Search using Linked List .................................................................................................... 10
      3.2.1. Data Members .......................................................................................................................... 10
      3.2.2. Search Function ....................................................................................................................... 10
      3.2.3. Insertion Function .................................................................................................................... 10
      3.2.4. Linear Search using Linked List Complexity ............................................................................ 10
   3.3. Hash Tables ................................................................................................................................... 11
      Chained Hash Tables .......................................................................................................................... 11
      Open-addressed hash tables (Linear Probing) .................................................................................... 11
      3.3.1. Data Members .......................................................................................................................... 11
      3.3.2. Search Function ....................................................................................................................... 11
      3.3.3. Insertion Function .................................................................................................................... 12
      3.3.4. Linear Probing Hash Table Search Complexity .......................................................................... 12
   3.4. Binary Search Tree Algorithm ...................................................................................................... 13
      3.4.1. Data Members .......................................................................................................................... 13
      3.4.2. Search for a place .................................................................................................................... 13
      3.4.3. Inserting Domain names in BST .............................................................................................. 14
      3.4.4. Binary Search Tree Complexity ............................................................................................... 15
4. Future Work .......................................................................................................................................... 16
5. References ........................................................................................................................................... 16
1. Introduction

Whenever a host requests for a web page, it sends a query to the DNS server, which resolves the hostname of the requested page, and returns the IP address of the server hosting that page. In our project, we have implemented this DNS lookup with 2 levels.

When a host requests a page (www.google.com), DNS server (dns.isp) looks up for the entry in its database using a search algorithm. If found, it returns the IP address of the server. If the entry is not found, it queries the second level of DNS server (dns.com). This server has the entries of all the servers. Now, dns.com will return the IP to the ISP's dns server, which caches this new entry in its database and forwards it further to the host. Following diagram explains this architecture.

*Figure 1. DNS Lookup System*
In this project, we have implemented the DNS lookup using various search algorithms like:

1. **Binary Search**
2. **Linked List**
3. **Hash Map**
4. **Binary Search Tree**

The number of entries that we are caching are of the order of $10^6$.

Examples are as follows:

dns.com  
dns.edu  
www.google.com  
www.rutgers.edu  
dns.isp
2. DNS Recursive Query Mechanism:

This mechanism is a classic example of Client-Server socket programming. The server listens for a connection. When a connection is established by a client, the client can send data. In this module, the client sends the message, which contains the Hostname. Server replies back with the IP address of the designated host. Finally the connection is ended and the server waits for another connection. The two programs run on different machines, in same subnet.

Java Socket Programming is used for communicating between DNS and root servers with client. There are two types of java program that needs to be installed on the machine. DNS and root servers execute server program while client machine executes client program. DNS Server also runs client program when it forwards the query to the Root DNS. The server program listens for a connection which is established by a client.

2.1. Client Side

- An object of class Socket is created to connect to the DNS server. It takes DNS server’s IP address and port number as its arguments. Port number 2004 is used for client-DNS server communication.

        requestSocket = new Socket("172.31.101.242", 2004);

- Socket’s output and input streams are instantiated and flushed to clear any random data if present.
- User’s input (request - Hostname) is placed in socket’s output stream and is sent to the DNS server.

        sendMessageToDNSServer(HostName);

- DNS Server’s response (IP Address) is taken in the socket’s input stream and is displayed to the user.

        message = (String)in.readObject();
        IPAddress=message;

- When user wants to terminate the program, it sends ‘Bye’ message to DNS server and terminates the program after receiving ‘Bye’ message from the server.
2.2. DNS Server Side

- DNS Server waits for a connection request from the client.

  ```java
  connection = providerSocket.accept();
  ```

- DNS Server receives a request (Hostname) from the client. Server first looks for the entry in its own database. This search takes place using any of the search algorithms mentioned.

  ```java
  message = (String) in.readObject();
  System.out.println("client>" + message);
  hostIP = search.Search(message);
  ```

- If the key (Hostname) is not found in the database, query is forwarded to the Root DNS Server. Now, this DNS Server acts as a Client (Requester) for Root DNS Server.

  ```java
  if(hostIP.equals("Not Found"))
  {
      hostIP = requester.run(message); // Query
      // forward
      write(message, hostIP); // save to local cache
  }
  ```

- An object of class Socket is created to connect to the Root server. It takes Root server’s IP address and port number as its arguments. Port number 2005 is used for DNS server-Root server communication.

  ```java
  requestSocket = new Socket("172.31.99.171", 2005);
  ```

- Socket’s output and input streams are instantiated and flushed to clear any random data if present.
- DNS server sends domain name as a request to the Root server and receives corresponding IP address.
- It then adds that key-value entry in its database and also returns the same to the client machine by sending it through 2004 socket port.
- The program remains open until user wants to terminate. After each request is processed, DNS and Root servers wait for next request.
- When user wants to terminate the program, it sends ‘Bye’ message to DNS server. DNS server in turn sends same message to Root server and then DNS server closes both socket connections and the programs are terminated respectively.
2.3. Root DNS Server Side

- An object of class Socket is created to connect to the DNS server. It takes DNS server’s IP address and port number as its arguments. Port number 2005 is used for DNS server-Root server communication.

  ```java
  providerSocket = new ServerSocket(2005, 10);
  connection = providerSocket.accept();
  ```

- Socket’s output and input streams are instantiated and flushed to clear any random data if present.
- Root server waits for DNS server request. As soon as it receives the request, it uses the given domain name to call various search algorithms to fetch corresponding IP address value.

  ```java
  hostIP = search.Search(message);
  ```

- When Root server has the requested record it returns the IP address to DNS server by sending reply message on socket’s output stream.

  ```java
  sendMessageToDNSServer(hostIP);
  ```

- The program remains open until user wants to terminate. After each request is processed, DNS and Root servers wait for next request.
- When user wants to terminate the program, it sends ‘Bye’ message to DNS server. DNS server in turn sends same message to Root server. After receiving ‘Bye’ from Root server, DNS server closes both socket connections and the programs are terminated respectively.
3. Search Algorithms:

3.1. Binary Search using arrays

3.1.1. Data Members:

```java
private Val[] vals;        // symbol table values
private Key[] keys;        // symbol table keys
private int N = 0;         // number of elements
```

3.1.2. Search Function

```java
// binary search in interval [left, right]
private int bsearch(Key key) {
    int left = 0, right = N-1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        int cmp = key.compareTo(keys[mid]);
        if (cmp < 0) right = mid - 1;
        else if (cmp > 0) left = mid + 1;
        else return mid;
    }
    return -1;
}
```

```java
// return value associated with given key, or null if no such key
public Val get(Key key) {
    int i = bsearch(key);
    if (i == -1) return null;
    return vals[i];
}
```
3.1.3. Insertion Function

// add key-value pair
public void put(Key key, Val val) {
    if (N >= vals.length) resize(2*N);
    // duplicate
    if (contains(key)) {
        int i = bsearch(key);
        vals[i] = val;
        return;
    }
    // shift key-value pairs one position to right, and add new key-value pair
    int i = N;
    while ( (i > 0) && (key.compareTo(keys[i-1]) < 0) ) {
        keys[i] = keys[i-1];
        vals[i] = vals[i-1];
        i--;
    }
    vals[i] = val;
    keys[i] = key;
    N++;
}

3.1.4. Binary Search using arrays Complexity

- O(log n) time expected search, O(n) worst case
- O(log n) time expected insertion
- O(n) space
3.2. Linear Search using Linked List

3.2.1. Data Members

```java
private int N;       // number of key-value pairs
private Node first;  // the linked list of key-value pairs

// a helper linked list data type
private class Node {
    Key key;
    Value value;
    Node next;
}
```

3.2.2. Search Function

// return the value associated with the key, or null if the key is not present
```java
public Value get(Key key) {
    for (Node x = first; x != null; x = x.next) {
        if (key.equals(x.key)) return x.value;
    }
    return null;  // not found
}
```

3.2.3. Insertion Function

// add a key-value pair, replacing old key-value pair if key is already present
```java
public void put(Key key, Value value) {
    for (Node x = first; x != null; x = x.next)
        if (key.equals(x.key)) { x.value = value; return; }
    first = new Node(key, value, first);
    N++;
}
```

3.2.4. Linear Search using Linked List Complexity

- O(n) time expected search
- O(n) time expected insertion
- O(n) space
3.3. Hash Tables

Hash tables support one of the most efficient types of searching: hashing. Fundamentally, a hash table consists of an array in which data is accessed via a special index called a key. The primary idea behind a hash table is to establish a mapping between the set of all possible keys and positions in the array using a hash function. A hash function accepts a key and returns its hash coding, or hash value. Keys vary in type, but hash codes are always integers.

Since both computing a hash value and indexing into an array can be performed in constant time, the beauty of hashing is that we can use it to perform constant time searches. When a hash function can guarantee that no two keys will generate the same hash coding, the resulting hash table is said to be directly addressed. This is ideal, but direct addressing is rarely possible in practice. Typically, the number of entries in a hash table is small relative to the universe of possible keys. Consequently, most hash functions map some keys to the same position in the table. When two keys map to the same position, they collide. A good hash function minimizes collisions, but we must still be prepared to deal with them.

There are two broad types of implementation of Hash Tables:

**Chained Hash Tables**

Hash tables that store data in buckets. Each bucket is a linked list that can grow as large as necessary to accommodate collisions.

**Open-addressed hash tables (Linear Probing)**

Hash tables that store data in the table itself instead of in buckets. Collisions are resolved using various methods of probing the table. We have used Linear Probing in our implementation.

### 3.3.1. Data Members

```java
private int key;
private int value;
```

### 3.3.2. Search Function

```java
public int get(int key) {
    int hash = (key % TABLE_SIZE); // Generating hash key
    while (table[hash] != null && table[hash].getKey() != key)
        hash = (hash + 1) % TABLE_SIZE; // If key not found at the expected position, check next possible position
    if (table[hash] == null)
        return -1;
    else
        return table[hash].getValue(); // return the value if key found
}
```
### 3.3.3. Insertion Function

```java
public void put(int key, int value) {
    int hash = (key % TABLE_SIZE); // Generate Hash Key
    while (table[hash] != null && table[hash].getKey() != key) { // If that position is
        hash = (hash + 1) % TABLE_SIZE; //already filled,
        //compute next
        //position. Continue
        //until empty slot is
        //found
    }
    table[hash] = new HashEntry(key, value);
}
```

### 3.3.4. Linear Probing Hash Table Search Complexity

- O(1) time expected search
- O(1) time expected insertion
- O(1) space
3.4. Binary Search Tree Algorithm

3.4.1. Data Members

```java
private Node root; // root of BST
```

Node class members are as follows:

```java
private Key key; // sorted by key
private Value val; // associated data
private Node left, right; // left and right subtrees
private int N; // number of nodes in subtree
```

Adding a value to BST can be divided into two stages:

- Search for a place to put a new key-value pair by looking at keys;
- Insert the new key-value pair to this place.

Let us see these stages in more detail.

3.4.2. Search for a place

![Binary Search Tree](image)

*Figure 2. Binary Search Tree*

At this stage an algorithm should follow binary search tree property. If a new key value is less than the current node's key value, go to the left subtree, else go to the right subtree. Following this simple rule, the algorithm reaches a node, which has no left or right subtree. By the moment a place for insertion is found, we can say for sure, that a new key-value pair has no duplicate in the tree. Initially, a new node has no children, so it is a leaf. Let us see it at the picture. Gray circles indicate possible places for a new node. Domain names are keys whereas IP addresses are corresponding values.
Search Function

```java
public Value get(Key key) {
    return get(root, key);
}
```

```java
private Value get(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else return x.val;
}
```

3.4.3. Inserting Domain names in BST

![Binary Search Tree Realization for Insertion](image)

Now, let's go down to algorithm itself. Here and in almost every operation on BST recursion is utilized. Starting from the root,

1. Check, whether ‘first character’ value of key in current node and a new value are equal. If so, duplicate is found.
2. In this case repeat step 1 for second character in the key. However, if duplicate is not found, go to step 3.
3. If a new value is less than the node's value, there are two cases possible:
   - If a current node has no left child, place for insertion has been found;
   - Otherwise, handle the left child with the same algorithm.
4. If a new value is greater than the node's value, again two possible cases are as follows:
   - If a current node has no right child, place for insertion has been found;
   - Otherwise, handle the right child with the same algorithm.

Just before code snippets, let us have a look on the example, demonstrating a case of insertion in the binary search tree.
DNS LOOKUP SYSTEM

For example, in BST shown above, let’s insert the key-value pair

| passport.net | 65.54.179.226 |

into the BST. Now according to the algorithm, look for the first character values in key fields and select left/right subtree accordingly for recursive traversal. Figure 4 indicates the possible place for inserting ‘passport.net’ key.

![Binary Search Tree Diagram]

Figure 4. Inserting key = ‘passport.net’ in Binary Search Tree

Insertion Function

```java
public void put(Key key, Value val) {
    if (val == null) { delete(key); return; }
    root = put(root, key, val);
    assert check();
}
```

```java
private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val, 1);
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = put(x.left, key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else x.val = val;
    x.N = 1 + size(x.left) + size(x.right);
    return x;
}
```

3.4.4. Binary Search Tree Complexity

- O(1.39 log n) time expected search, O(n) worst case
- O(1.39 log n) time expected insertion, O(n) worst case
- O(n) space
4. Future Work

- Number of client machines can be increased and many clients can simultaneously query IP addresses from the servers.
- Two tier system can be enhanced to multiple tiers to develop a huge system to handle multiple clients.
- We can assign different servers for each domain and further increase the scalability of the system.

5. References