TCP2101: Algorithm Design and Analysis

Assignment 1 Report

Submitted to

Ts. Dr. Ng Kok Why

Prepared by

John Christian Gonzales Escobia

Submitted on

January 1, 2020

Revised and Resubmitted on

January 6, 2020

Abstract

This report examines the time complexities of two collision resolution methods: chaining and linear probing, and two data structures: an AVL binary search tree and a priority queue. The tasks performed by the four algorithms are insertion, search, enqueue, and dequeue. The datasets used are of sizes 100, 100000, and 500000 emails.

Table of Contents

Overview **2**
Environment 2
Pre-Execution Notes 2
Program Execution 2
Files Included 2
Menus and Search Data 3
Regular Expression 3
Data Sets 3

Data Generation **4**
Algorithm 4
Experimental Results 6

Program 1a **7**
Algorithm 7
Experimental Results 10

Program 1b **12**
Algorithm 12
Experimental Results 13

Program 2 **15**
Algorithm 15
Experimental Results 16

Program 3 **18**
Algorithm 18
Experimental Results 19

Comparison **21**
Insert Time versus Data Set Graph 21
Search Time for Searchable Data versus Data Set Graph 22
Search Time for Unsearchable Data versus Data Set Graph 23

Overview

Environment

Operating System: Ubuntu 20.04

Integrated Development Environment: Geany v1.36

Pre-Execution Notes

Running the data generation program will *overwrite the current datasets* (SET_A.txt, SET_B.txt and SET_C.txt). If this happens, the data in SearchData.cpp needs to be changed accordingly. For convenience, there is a copy for each dataset in the folder named "backup".

Program Execution

There are five programs in total. Each program runs by executing a main file with the program name which will call other source code files as follows:

No	Program	Main File	Files Included
1	Generate Data	GenerateData.cpp	Menu.cpp
2	Program 1a	Program_1a.cpp	HashTable_1a.cpp, LinkedList.cpp
3	Program 1b	Program_1b.cpp	HashTable_1b.cpp
4	Program 2	Program_2.cpp	BinarySearchTree.cpp
5	Program 3	Program_3.cpp	PriorityQueue.cpp

Table 1: Shows the main file and files included for each program.

Files Included

Apart from what is mentioned in Table 1, there are two files that are included in Program 1a, 1b, 2, and 3. These files are: Menu.cpp and SearchData.cpp.

Menus and Search Data

Menu.cpp implements the menus for each program and SearchData.cpp contains 30 searchable and 30 unsearchable emails.

Regular Expression

Emails generated are in the format of the following regular expression: [A-Za-z0-9]{5}\.[A-Za-z0-9]{5}@[A-Za-z]{5}\.(com|net|org)

Data Sets

- SET_A.txt - 100 emails
- SET_B.txt - 100000 emails
- SET_C.txt - 500000 emails

Data Generation

Algorithm

GenerateData.cpp

```
// A function that returns a random character from the passed string
START generate_random_char(s, x)
    RETURN a random character in s
START generate_random_char

START main()
    // Number of emails
    n = 0

    // Option determines combination of n and fileName
    option = 0

    // Text file name
    fileName = ""

    // Variable to write file
    ofstream writeFile

    alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
    alphaNum = alpha + "0123456789"
    domain[] = {"com","net","org"}

    // Show menu and get number of emails to generate
    menu(n, fileName, "gd", option)

    IF n == 0
        THEN end program

    email = ""
    counter = n
    srand(time(0))

    // Generate emails
    While counter > 0
        // Generate and append 10 random letters and numbers to email
        FOR i = 0 to 9
            IF i == 5
                THEN append '.' to email
            email += generate_random_char(alphaNum, alphaNum.LENGTH-1)
        NEXT i

        Append character '@' to email

        // Generate and append 5 random letters to email
        FOR i = 0 to 4
```

```
            email += generate_random_char(alpha, alpha.LENGTH-1)
        NEXT i

        Append character '.' to email

        // Append random domain to email
        email += domain[rand() % 3]

        Append a newline to email
    ENDWHILE

    // Create a text file fileName.txt and insert variable email into it
    writeFile = OPENWRITE(fileName)
    writeFile.WRITE(email)
    writeFile.CLOSE()

    // Exit message
    PRINT(n " emails have been generated and inserted to " fileName)

    RETURN 0
END main
```

Experimental Results

Generating SET A:

```
SELECT NUMBER OF EMAILS
[1] 100
[2] 100000
[3] 500000

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 1

100 emails have been generated and inserted to SET_A.txt
```

Generating SET B:

```
SELECT NUMBER OF EMAILS
[1] 100
[2] 100000
[3] 500000

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 2

100000 emails have been generated and inserted to SET_B.txt
```

Generating SET C:

```
SELECT NUMBER OF EMAILS
[1] 100
[2] 100000
[3] 500000

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 3

500000 emails have been generated and inserted to SET_C.txt
```

Program 1a

Algorithm

Program_1a.cpp

```
START insert(HashTable, FileName, Time, InsertTime)
    OPEN FileName.txt
    Initialize ReadFile = contents of FileName.txt
    Initialize Email = ""

    START infinite while-loop
        GETLINE from ReadFile and store to Email

        If ReadFile reaches end of file stop while-loop

        Start record time
        Insert Email to HashTable
        End record time
        Time = end record time - start record time
        InsertTime += Time

        NEXTLINE
    END while-loop

    CLOSE FileName.txt
END insert

START search(HashTable, FileName, Time, SearchableTime,
             UnsearchableTime)
    Initialize Email = ""

    // Record duration for searching data
    FOR i = 1 to 20
        IF FileName == "SET_A.txt"
            THEN Email = search_set_a[i]
        ELSEIF FileName == "SET_B.txt"
            THEN Email = search_set_b[i]
        ELSE
            THEN Email = search_set_c[i]

        Start record time
        Find Email in HashTable
        End record time
        Time = end record time - start record time

        IF Email is found in HashTable
            Add Time to SearchableTime
        ELSE
            Add Time to UnsearchableTime
    NEXT i
```

```
END search

START main()
    Initialize n = 0
    Initialize option = 0
    Initialize FileName = ""
    Initialize Time
    Initialize InsertTime = 0
    Initialize SearchableTime = 0
    Initialize UnsearchableTime = 0

    CALL PROCEDURE menu(n, FileName, "1a", option)

    IF n == 0 THEN end program

    Initialize HashTable HT(n*0.9)

    CALL PROCEDURE insert(HT, FileName, Time, InsertTime)
    CALL PROCEDURE search(HT, FileName, Time, SearchableTime,
                          UnsearchableTime)

    // Exit message
    PRINT(
        "CHAINING METHOD SUMMARY"
        "Dataset: " fileName
        "Total data: " n " emails"
        "Array size: " n*0.9
        "Insertion time: " InsertTime "s"
        "Average search time for searchable data: "
        SeachableTime/10 "s"
        "Average search time for unsearchable data: "
        UnsearchableTime/10 "s"
    )

    RETURN 0
END main
```

HashTable_1a.cpp

```
template <typename T>
START CLASS HashTable
    ATTRIBUTE vector< LinkedList<T> > table

    START hashFunction(HashItem)
        Initialize Num = 0

        Convert each character of HashItem to integer and add to Num

        RETURN Num % table.size()
    END hashFunction
```

```
    CONSTRUCTOR HashTable(Size) Resize HashTable according to Size

    DECONSTRUCTOR ~HashTable() Empties HashTable

    FUNCTION size() Returns size of HashTable

    START insert(NewItem)
        Call procedure hashFunction(NewItem) to get HashTable index
        Store HashTable index to Location
        Insert NewItem in the front of Location
    END insert

    START retrieve(Target)
        Call procedure hashFunction(Target) to get HashTable index
        Store HashTable index to Location
        IF Target is not found Location THEN RETURN false
        By default RETURN true
    END retrieve

    START ostream& operator<< (os, HashTable)
        FOR i = 0 to end of HashTable
            os << PRINT(i " = " HashTable[i])
        NEXT i
        RETURN os
    END ostream& operator<<
END CLASS HashTable
```

LinkedList.cpp

```
#ifndef LINKEDLIST_CPP
#define LINKEDLIST_CPP

template <typename T>
START STRUCT Node
    T info
    Node<T> *next
END STRUCT Node

template <typename T>
START CLASS LinkedList
    ATTRIBUTE Node<T> *start

    CONSTRUCTOR LinkedList() Point start to NULL

    DECONSTRUCTOR ~LinkedList() Calls makeEmpty()

    START insertFront(Element)
        Initialize Node<T> *newNode = new Node<T>
        Initialize newNode->info = Element
        Initialize newNode->next = start
        Initialize start = newNode
```

```
    END insertFront

    START find(T & target)
        Initialize found = false
        Initialize Node<T> *ptr = start

        WHILE ptr is not NULL and not found
            IF ptr->info == target THEN found = true
            ELSE ptr = ptr->next
        WHILE

        RETURN found
    END find

    isEmpty() returns start == NULL

    makeEmpty() Empties the linked lists

    START ostream& operator<<(os, list)
        Initialize Node<T> *ptr = list.start
        WHILE ptr is not NULL
            os << ptr->info << " "
            ptr = ptr->next
        ENDWHILE

        RETURN os
    END ostream& operator<<
END CLASS LinkedList
#endif
```

Experimental Results

Processing SET A:

```
SELECT DATASET
[1] SET A
[2] SET B
[3] SET C

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 1

CHAINING METHOD SUMMARY
Dataset: SET_A.txt
Total data: 100 emails
Array size: 90
Insertion time: 0.000206688s
Average search time for searchable data: 1.1435e-06s
Average search time for unsearchable data: 1.0454e-06s
```

Processing SET B:

```
SELECT DATASET
[1] SET A
[2] SET B
[3] SET C

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 2

CHAINING METHOD SUMMARY
Dataset: SET_B.txt
Total data: 100000 emails
Array size: 90000
Insertion time: 0.0435829s
Average search time for searchable data: 1.64077e-05s
Average search time for unsearchable data: 8.0898e-06s
```

Processing SET C:

```
SELECT DATASET
[1] SET A
[2] SET B
[3] SET C

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 3

CHAINING METHOD SUMMARY
Dataset: SET_C.txt
Total data: 500000 emails
Array size: 450000
Insertion time: 0.190516s
Average search time for searchable data: 8.22278e-05s
Average search time for unsearchable data: 1.70354e-05s
```

Program 1b

Algorithm

Program_1b.cpp

```
Similar to Program_1a.cpp. There are two differences:
   -  Different array sizes.
   -  The objects created are based on HashTable_1b.cpp and not
      HashTable_1a.cpp.
```

HashTable_1b.cpp

```
template<typename K>

START CLASS HashNode
    ATTRIBUTE K key
    CONSTRUCTOR HashNode(K key) this->key = key
END CLASS HashNose

template<typename K>

START CLASS HashTable
    ATTRIBUTE HashNode<K> **arr
    ATTRIBUTE size

    CONSTRUCTOR HashTable(n)
        Initialize size = n
        Initialize arr = new HashNode<K>*[size]
        Initialize all elements in arr as NULL
    END CONSTRUCTOR HashTable

    START hashFunction(key, fileName)
        Initialize num = 0
        Initialize n = 1

        IF fileName is "SET_B.txt" OR "SET_C.txt" THEN n = 987654

        Convert each character of key to integer
        Multiply each integer by n and add to num

        RETURN num % size
    END hashFunction

    START insertNode(K key:byVal, fileName:byVal)
        Initialize HashNode<K> *temp = new HashNode<K>(key)
        Initialize hashIndex = hashFunction(key, fileName)
        WHILE arr[hashIndex] is not NULL
            {Increment hashIndex by 1}
```

```
            IF hashIndex > size-1 THEN hashIndex = 0
        ENDWHILE

        arr[hashIndex] = temp
    END insertNode

    START find(key, fileName)
        Initialize hashIndex = hashFunction(key, fileName)

        WHILE arr[hashIndex] is not NULL
            IF hashIndex > size-1 THEN hashIndex = 0
            IF arr[hashIndex]->key == key THEN RETURN true
            {Increment hashIndex by 1}
        ENDWHILE

        RETURN false
    END find

    sizeofMap() Returns size

    isEmpty() Returns size == 0

    START display()
        FOR i = 0 to size
            IF arr[i] != NULL && arr[i]->key != ""
                THEN PRINT("Index: " i " key = " arr[i]->key")
        NEXT i
    END display
END CLASS HashTable
```

Experimental Results

Processing SET A:

```
SELECT DATASET
[1] SET A
[2] SET B
[3] SET C

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 1

LINEAR PROBING METHOD SUMMARY
Dataset: SET_A.txt
Total data: 100 emails
Array size: 151
Insertion time: 0.000185324s
Average search time for searchable data: 1.5158e-06s
Average search time for unsearchable data: 1.5446e-06s
```

Processing SET B:

```
SELECT DATASET
[1] SET A
[2] SET B
[3] SET C

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 2

LINEAR PROBING METHOD SUMMARY
Dataset: SET_B.txt
Total data: 100000 emails
Array size: 150001
Insertion time: 0.0933015s
Average search time for searchable data: 1.41476e-05s
Average search time for unsearchable data: 9.9655e-06s
```

Processing SET C:

```
SELECT DATASET
[1] SET A
[2] SET B
[3] SET C

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 3

LINEAR PROBING METHOD SUMMARY
Dataset: SET_C.txt
Total data: 500000 emails
Array size: 758099
Insertion time: 1.40762s
Average search time for searchable data: 7.13164e-05s
Average search time for unsearchable data: 4.91828e-05s
```

Program 2

Algorithm

Program_2.cpp

```
Program_2.cpp is similar to Program_1a.cpp. There are three differences:
  ● There are no array sizes as it is based on the data structure of a
    vector and thus the size is dynamically allocated.
  ● The emails are sorted in an ascending order before the insert
    function is performed.
  ● The objects created are based on BinarySearchTree.cpp and not
    HashTable_1a.cpp.
```

BinarySearchTree.cpp

```
START STRUCT TNode
    ATTRIBUTES string data, TNode* left, TNode* right
END STRUCT TNode

START newNode(data)
    Initialize TNode* node = new TNode()
    Initialize node->data = data
    Initialize node->left = NULL
    Initialize node->right = NULL
    RETURN node
END newNode

START sortedArrayToBST(arr, start, end)
    IF start > end THEN return NULL
    // Get the middle element and make it root
    Initialize mid = (start + end)/2
    Initialize *root = newNode(arr[mid])

    // Recursively construct left subtree of the root
    Initialize root->left = sortedArrayToBST(arr, start, mid - 1)

    // Recursively construct right subtree of the root
    Initialize root->right = sortedArrayToBST(arr, mid + 1, end)

    RETURN root
END sortedArrayToBST

START searchBST(TNode* node, key, check)
    IF key is not found until AVL leaf THEN RETURN check = false
    IF key is found THEN RETURN check = true
    IF key is greater than current node THEN searchBST(node->right, key)
    ELSE THEN searchBST(node->left, key)
END searchBST
```

Experimental Results

Processing SET A:

```
SELECT DATASET
[1] SET A
[2] SET B
[3] SET C

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 1

AVL TREE SUMMARY
Dataset: SET_A.txt
Total data: 100 emails
Insertion time: 0.000199419s
Average search time for searchable data: 1.672e-06s
Average search time for unsearchable data: 1.9842e-06s
```

Processing SET B:

```
SELECT DATASET
[1] SET A
[2] SET B
[3] SET C

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 2

AVL TREE SUMMARY
Dataset: SET_B.txt
Total data: 100000 emails
Insertion time: 0.0810339s
Average search time for searchable data: 1.9049e-06s
Average search time for unsearchable data: 2.2048e-06s
```

Processing SET C:

```
SELECT DATASET
[1] SET A
[2] SET B
[3] SET C

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 3

AVL TREE SUMMARY
Dataset: SET_C.txt
Total data: 500000 emails
Insertion time: 0.543496s
Average search time for searchable data: 2.7316e-06s
Average search time for unsearchable data: 2.4974e-06s
```

Program 3

Algorithm

Program_3.cpp

```
Program_3.cpp is similar to Program_1a.cpp. There are three differences:
   ●  The naming convention of the "insert" function in Program_1a.cpp
      is changed to "enqueue" in Program_3.cpp.
   ●  The objects created are based on PriorityQueue.cpp and not
      HashTable_1a.cpp.
   ●  Program_3.cpp has a "dequeue" function with the pseudocode below.

START dequeue(PriorityQueue, PQsize, Time, DequeueTime)
    FOR i = 0 to PQsize*0.1
        Start record time
        Dequeue largest element in the PriorityQueue
        End record time
        Time = end record time - start record time
        DequeueTime += Time
    NEXT i
END dequeue
```

PriorityQueue.cpp

```
template <typename T>
START CLASS PriorityQueue
    ATTRIBUTEvector<T> A

    START heapify_enqueue(index)
        IF index == 0 THEN stop heapify_enqueue

        Initialize parent_index = (index-1)/2

        IF A[index] > A[parent_index]
            THEN swap(A[index], A[parent_index])

        RECURSE heapify_enqueue(parent_index)
    END heapify_enqueue

    START heapify_dequeue(index)
        Initialize max = 0
        Initialize left = 2*index+1
        Initialize right = 2*index+2
        Initialize size = A.size()

        IF left < size && A[left] > A[max] THEN max = left
        ELSE max = index
```

```
        IF right < size && A[right] > A[max] THEN max = right

        IF max != index THEN
            swap (A[index], A[max])
            heapify_dequeue(max)
    END heapify_dequeue

    START enqueue(element)
        A.push_back(element)
        heapify_enqueue(A.size()-1)
    END enqueue

    START dequeue()
        Initialize T removed_element = A[0]
        Initialize A[0] = A[A.size()-1]
        Remove last element of A
        Call heapify_dequeue(0)
        RETURN removed_element
    END dequeue

    size() RETURN A.size()
END CLASS PriorityQueue
```

Experimental Results

Processing SET A:

```
SELECT DATASET
[1] SET A
[2] SET B
[3] SET C

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 1

PRIORITY QUEUE SUMMARY
Dataset: SET_A.txt
Total data: 100 emails
Enqueue time: 0.00019452s
Dequeue time: 1.5932e-05s
Average search time for searchable data: 3.7398e-06s
Average search time for unsearchable data: 7.8039e-06s
```

Processing SET B:

```
SELECT DATASET
[1] SET A
[2] SET B
[3] SET C

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 2

PRIORITY QUEUE SUMMARY
Dataset: SET_B.txt
Total data: 100000 emails
Enqueue time: 0.0533765s
Dequeue time: 0.00669162s
Average search time for searchable data: 0.000798543s
Average search time for unsearchable data: 0.00191623s
```

Processing SET C:

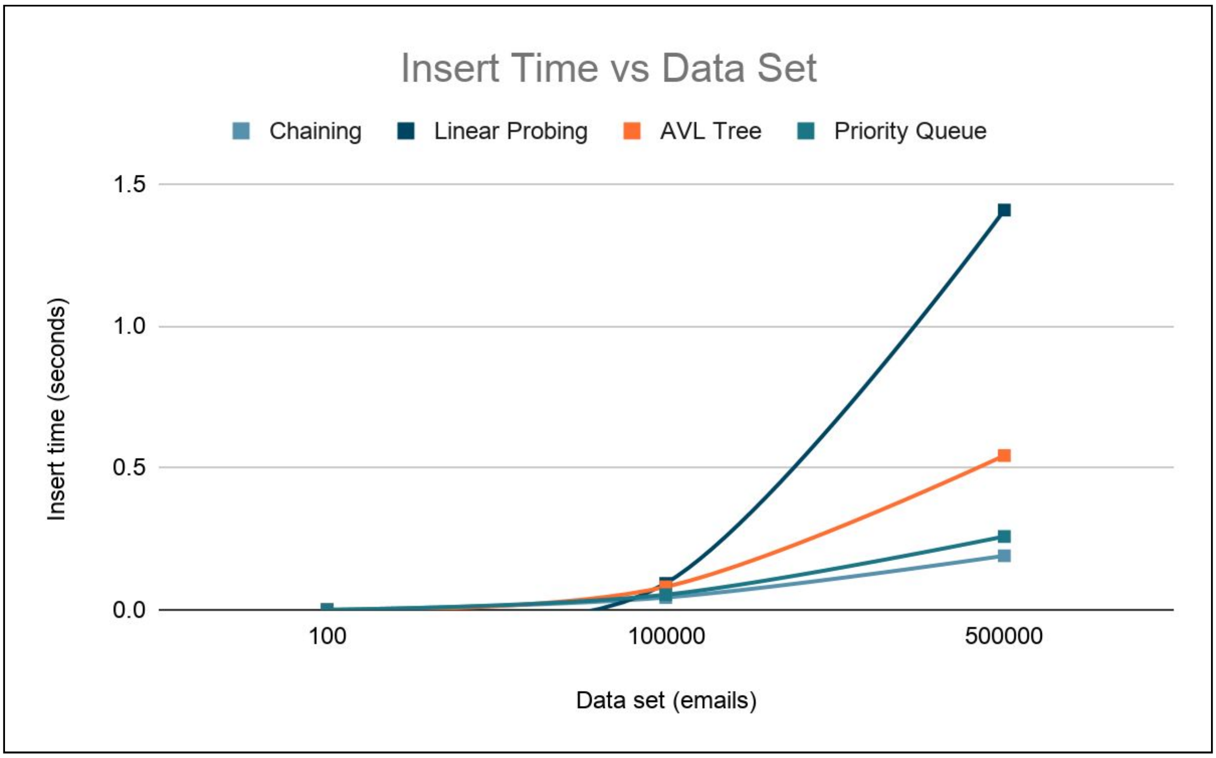
```
SELECT DATASET
[1] SET A
[2] SET B
[3] SET C

Enter 1, 2 or 3 to proceed.Other keys will exit the program.
>> 3

PRIORITY QUEUE SUMMARY
Dataset: SET_C.txt
Total data: 500000 emails
Enqueue time: 0.258386s
Dequeue time: 0.0362049s
Average search time for searchable data: 0.00485732s
Average search time for unsearchable data: 0.00966543s
```

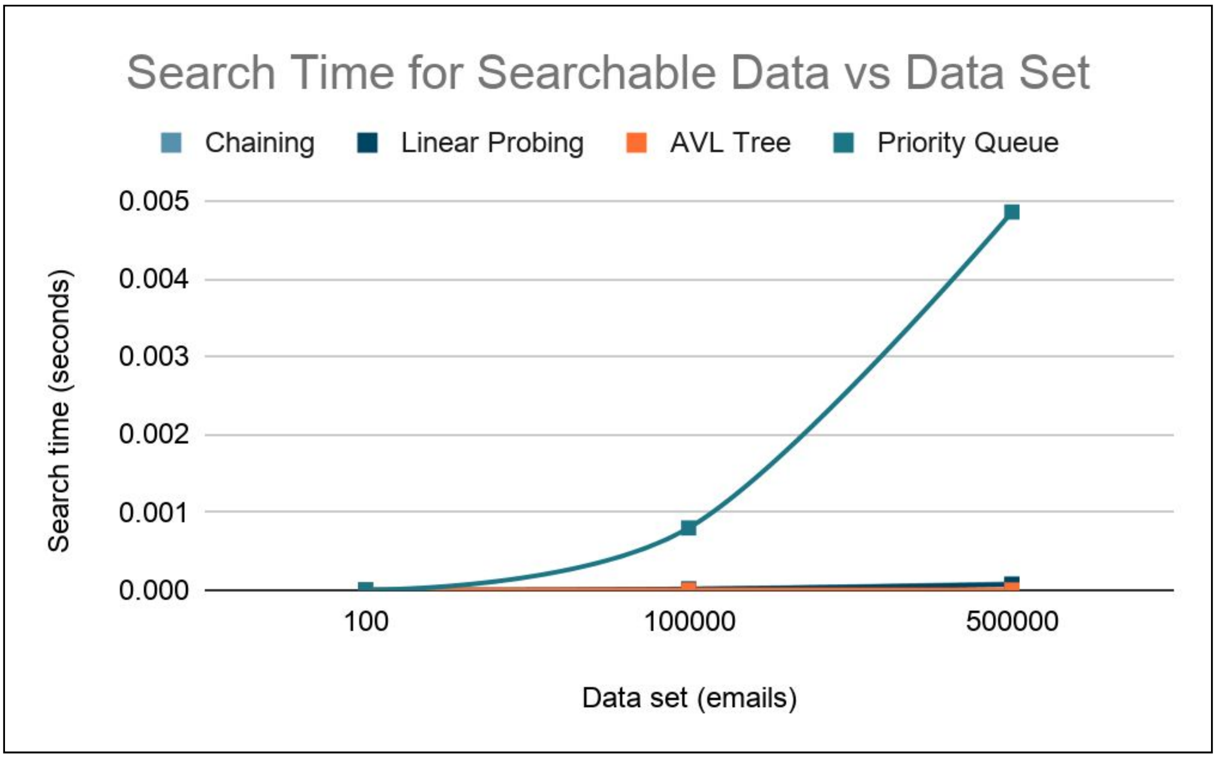
Comparison

Insert Time versus Data Set Graph

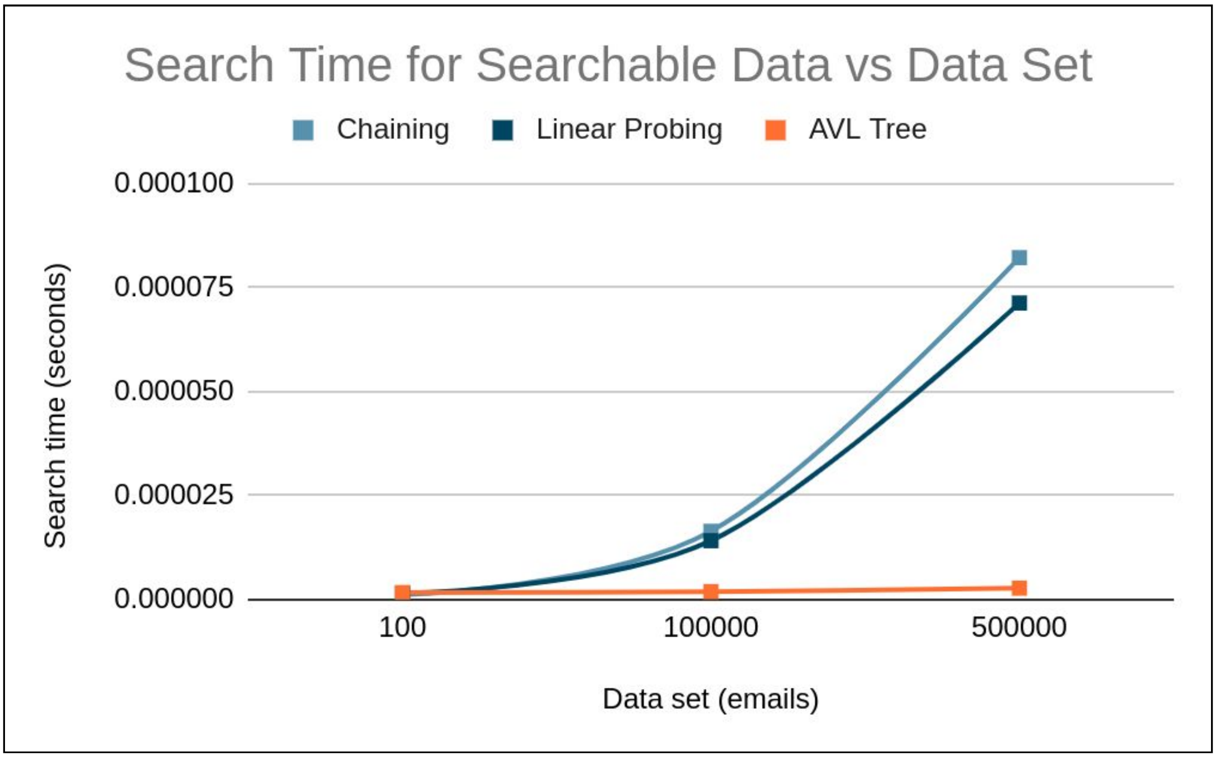


Search Time for Searchable Data versus Data Set Graph

Priority Queue Included:

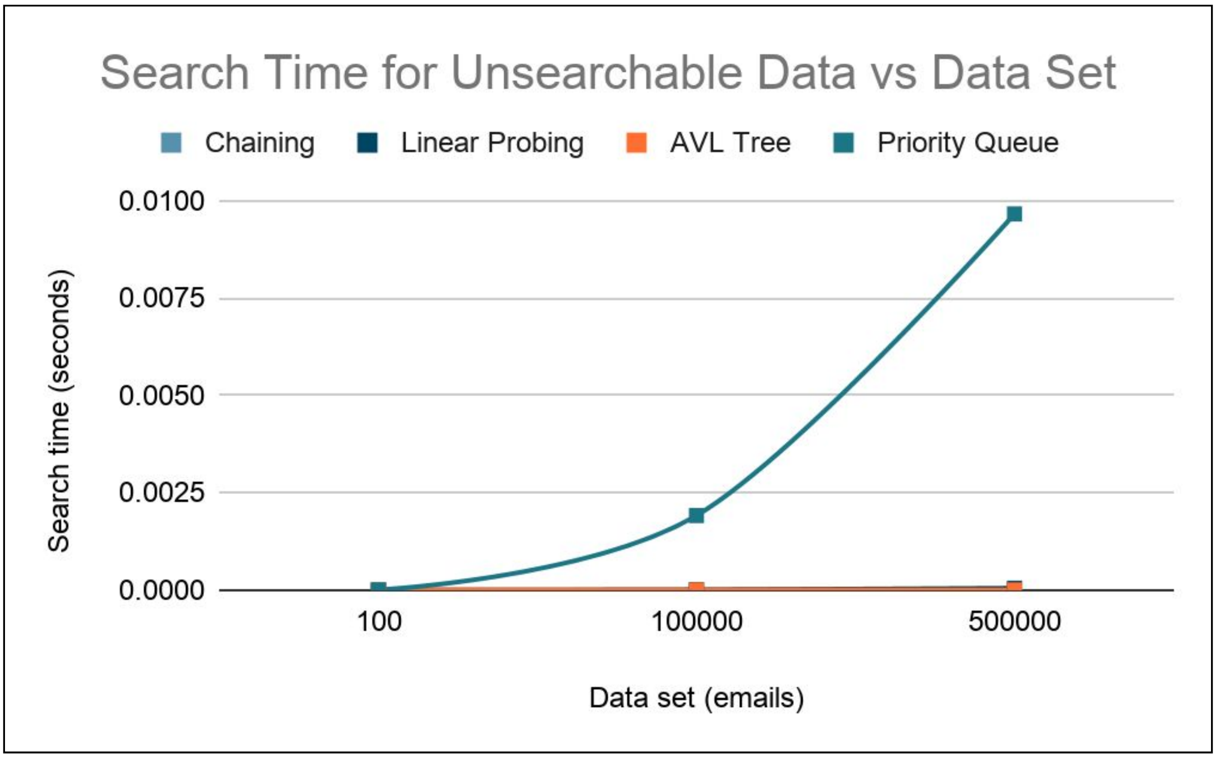


Priority Queue Not Included:



Search Time for Unsearchable Data versus Data Set Graph

Priority Queue Included:



Priority Queue Not Included: