# 2024 NCKU Program Design I HW10

> 請不要嘗試攻擊 Judge 系統，否則此堂課將不予以通過
>
> Don't attack judge system otherwise you will fail this course.
>
> 一旦發現作業抄襲或是請人代寫之情形，學期作業成績將 "全部" 以 0 分計算
>
> One instances of severe plagiarism, hiring someone to write assignments, or similar activities are detected, the semester's assignment scores will be calculated as 0 point across the board.

### Before you start

Make sure you can login the server by your personal account.

### submission

#### Judge (50 points)

- Server IP: `140.116.246.48` , Port: `2024`

```
> ssh 學號@140.116.246.48 -p 2024
```

- Create an directory with name `HW10` in your home directory.
  - You can use the "pwd" command to confirm your current directory.
  - The "mkdir [name]" command can create a directory with the name [name]
- In `HW10` directory, you need to create a file with name `pA.c` .
- You can directly use the command `hw10` to check whether the result of the question is correct. (The command hasn't been set yet, and also the submit I/O format hasn't been determined. Please wait for furthur announcements.)

#### Moodle (50 points)

- The **ppt** which will be mentioned below.
- **Named as** `學號_姓名_hw10.pdf` .

## Deadline: 2025/1/13 23:59:59

## Prerequisite (先決要件)

> *The oldest and strongest emotion of mankind is fear, and the oldest and strongest kind of fear is fear of the unknown.*
> ~ H.P. Lovecraft, Supernatural Horror in Literature (1927)

Some brief introductions for you to have a big picture to this homework.

### IP routing and Prefix Matching

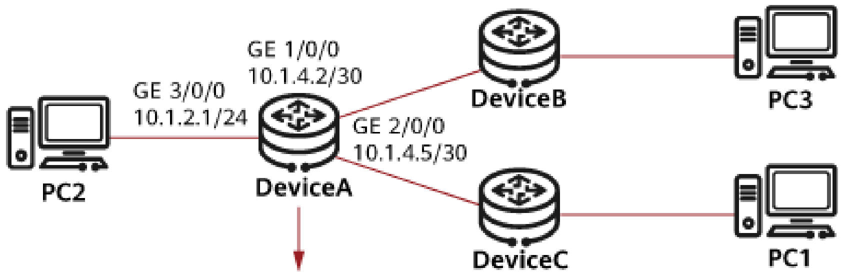
In this sub-section, we're going to talk about the basic of network routing, since the backgraound of this homework related to this topic.

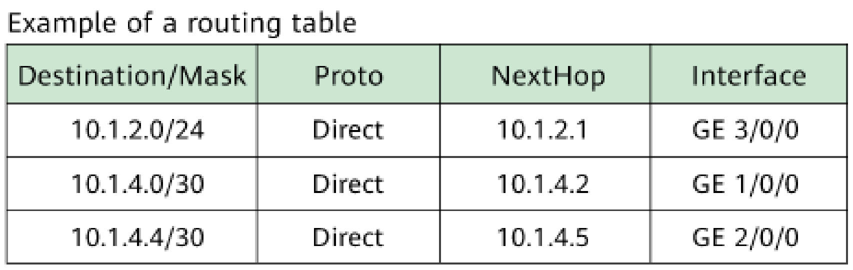
#### IP routing and Routing tables

> The "routing table" we taught in here **IS NOT** how the `routing_table.txt` is constructed.

IP routing is the process of forwarding data packets from a source to a destination across interconnected networks. It determines the optimal path based on the destination IP address contained in the packet's header. As a fundamental aspect of network communication, routing enables devices to exchange information efficiently over large and complex networks, such as the Internet.

The figure in the following is a simple IP routing graph through multiple routers (or switches).




Example of a routing table

| Destination/Mask | Proto | NextHop | Interface |
|---|---|---|---|
| 10.1.2.0/24 | Direct | 10.1.2.1 | GE 3/0/0 |
| 10.1.4.0/30 | Direct | 10.1.4.2 | GE 1/0/0 |
| 10.1.4.4/30 | Direct | 10.1.4.5 | GE 2/0/0 |

[source: Huawei - What is IP routing (https://info.support.huawei.com/info-finder/encyclopedia/en/IP+routing.html)]

However, this raises an important question—how does a router or switch identify a network packet and track its current forwarding state? This is precisely the role of a **routing table**.

Typically, routers and switches initialize their routing tables using a predefined file containing routing rules and entries. Once implemented and configured, these tables guide the devices in processing incoming packets.

As shown in the figure, the routing table consists of multiple entries, each representing a route associated with a specific network destination. Routers and switches use these entries to match the destination IP address of an incoming packet. Based on the matching result, the device decides whether to drop the packet or forward it to the next hop, ensuring proper delivery across the network.

### Prefix Matching and IP lookup

Let’s take a closer look at the mechanisms used in routers and switches.

As mentioned earlier, routers and switches rely on routing tables to match the IP address of incoming network packets and determine the next step for packet forwarding. But how exactly do these devices "match" the IP address of a network packet?

There are two primary methods for IP address matching: **exact matching** and **prefix matching**. Of these, **prefix matching** is the most widely used approach. The process of matching the IP of the network packet often referred to as **IP Lookup**.

**Exact matching** is fairly straightforward—the input IP address must match one of the target IP addresses in the routing table exactly.

In contrast, **prefix matching**, which is the focus of this homework, defines a match based on the **x** most significant bits (MSBs) of the input IP address. These bits must match exactly with one of the entries in the routing table.

Prefix matching plays a crucial role in IP routing by identifying the most suitable route for a given destination address. It evaluates the destination IP against routing table entries based on prefix lengths and subnet masks to ensure efficient and accurate packet forwarding.

Steps in Prefix Matching:

1. **Extract the Destination IP Address**: The IP address is extracted from the incoming packet's header.
2. **Apply Subnet Masks (For subnet mask, you can see the simple introduction through this link (https://hackmd.io/@rkhuncle/PD1TA-W14#Mask-))**: Each entry in the

routing table has a subnet mask associated with it, which is used to extract the network prefix.

3. **Compare Prefixes**: The destination IP address's prefix is compared against the prefixes in the routing table.
4. **Select the Longest Match (You won't face this task in this homework)**: The entry with the longest matching prefix is chosen, as it represents the most specific route.

- **Example**

Consider the following routing table as an example:

| Destination Prefix | Subnet Mask | Next Hop |
|---|---|---|
| 192.168.0.0 | 255.255.255.0 | Router A |
| 192.168.0.128 | 255.255.255.128 | Router B |
| 0.0.0.0 | 0.0.0.0 | Default Router |

For a packet destined for `192.168.0.140`:

- Match with `192.168.0.0/24`: Valid match.
- Match with `192.168.0.128/25`: More specific match.
- Match with `0.0.0.0/0`: Default match.

Result: The packet is forwarded to Router B due to the *longest* prefix match.

- **CIDR (Classless Interdomain Routing)**

In the midterm exam and previous homework assignments, you may have already encountered the IP address format, such as `192.168.0.1` and `10.108.1.2`.

However, you might be wondering why the routing IP format in the previous example specifies target IP addresses in a format like `192.168.0.0/24`.

In this format, the part before the `/`, such as `192.168.0.0`, follows the familiar IPv4 address format. The number after the `/`, in this case, `24`, represents the **length of the subnet mask to mask the IP address**—that is, the number of leading 1s in the mask.

This notation, written as `IP/prefix_subnet_length`, is officially called **Classless Inter-Domain Routing (CIDR)**. It is commonly used to define routing rules for matching entries in the routing table.

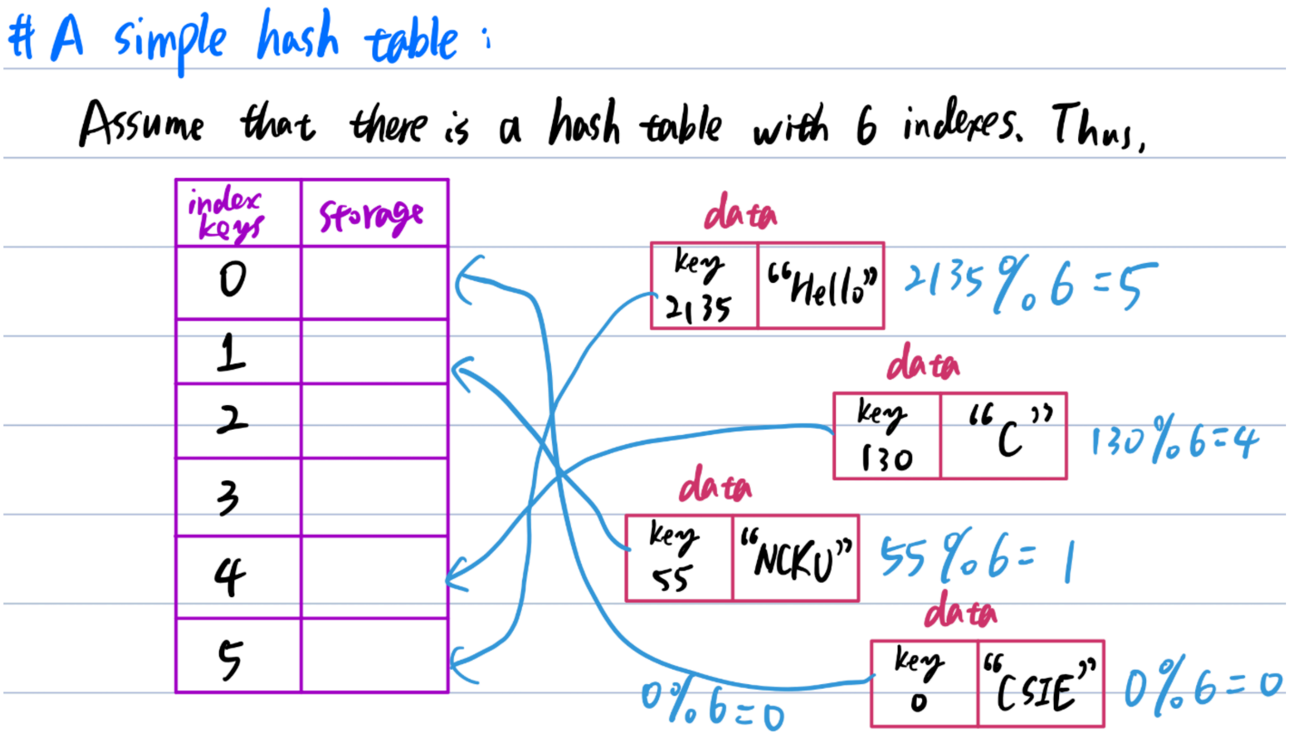
For example, the CIDR notation `192.168.0.0/24` can be interpreted as follows: The routing rule specifies a destination IP prefix of `192.168.0.0` with a subnet mask containing 24 leading 1s, equivalent to `255.255.255.0`.

### Hash Table

In the previous sections, we covered several key concepts, including IP routing, the use of routing tables, and the matching process employed by routers and switches. In this section, we will briefly discuss how routers store routing rules.

When dealing with large amounts of data, choosing an appropriate data structure is essential. Routers and switches employ various data structures to store routing rules, such as binary tries (https://opendatastructures.org/ods-java/13_1_BinaryTrie_digital_sea.html), binary prefix tries (https://en.wikipedia.org/wiki/Trie), B-trees (https://en.wikipedia.org/wiki/B-tree), B+ trees (https://www.geeksforgeeks.org/introduction-of-b-tree/), and hash tables (https://en.wikipedia.org/wiki/Hash_table). For this homework, you are required to implement a simple hash table, so we will provide a brief overview of its implementation.

A hash table is a data structure designed to store key-value pairs, offering fast updates and lookups with an average time complexity of `O(1)`. However, it may consume significant memory, proportional to the size of the table ( `O(n)` ), where `n` represents the number of entries. A hash table will be looked like the following figure:



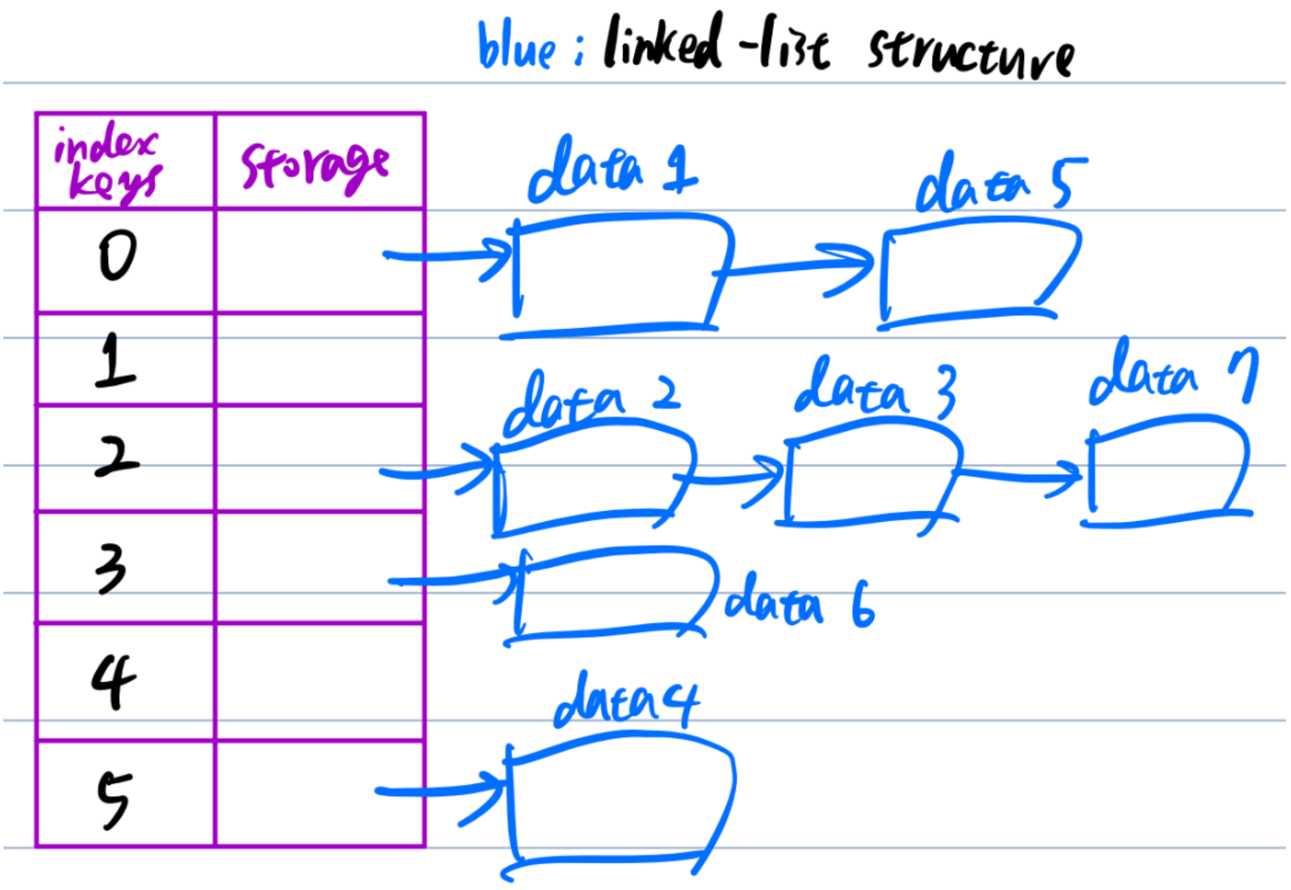
As shown in the figure, the length of a hash table is typically fixed, and inserting data relies on a modulo ( `%` ) operation. This operation calculates the index for storing a given key by dividing the key's value by the hash table size and using the remainder as the index. The keys can take various forms, such as strings, numerical values, or, in this homework, IP prefixes with specific lengths, as commonly used in router and switch architectures.

In practical implementations, hash tables often work in conjunction with hash functions to determine the storage location of input data (with the length of the hash table be carefully set). Popular hash functions include Multiplication Method (https://github.com/torvalds/linux/blob/master/tools/include/linux/hash.h), Bob Hash (https://burtleburtle.net/bob/hash/doobs.html), and Murmur Hash (https://github.com/aappleby/smhasher/tree/master). However, **for this homework**, you do not need to implement advanced hash functions, according to the problem descriptions. Instead, you can **use simpler techniques**, *such as* **bit masking** or **modulo operations**, as a hash function to store data in the hash table.

#### Collision Handling

While hash tables offer efficiency, they also introduce a potential issue known as **hash collision**—a situation where multiple keys map to the same index. As shown in the figure, collisions may lead to multiple data entries being stored in the same location.

To handle collisions, developers often use **linked-lists** to store multiple values at a single index. Linked-lists are commonly chosen because they are dynamic and do not require predefined sizes, unlike arrays. This approach ensures that hash tables remain flexible and efficient, even when collisions occur. The following figure is an example:



### Performance Evaluation

After a program or system is implemented, performance evaluation is crucial, as it helps determine how well your implementation performs compared to others. Typically, performance evaluation for large software systems or specific algorithms involves standard metrics, such as those used to evaluate machine learning algorithms or tools like perf.

In the following sections, I will introduce two methods for measuring the time performance of functions in your C code.

#### `<time.h>`

In user-level C programming, there is a library called `<time.h>` that provides a clock timer, allowing you to measure how long a specific function or part of your program takes to execute.

For example:

```
#include <stdio.h>
#include <time.h>

void example_function() {
    for (int i = 0; i < 1000000; i++);
}

int main() {
    clock_t start, end;
    double cpu_time_used;

    start = clock();
    example_function();
    end = clock();

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Execution time: %f seconds\n", cpu_time_used);

    return 0;
}
```

In the code example, it uses `clock_t` variable type and `clock()` function in `<time.h>` to measure the time consumption of the `example_function()` function.

#### `rtdsc` : Current CPU tick from Time Stamp Counter (TSC)

> This is the introduction to what `clock.c` wants you to do.

However, what if we want to measure our program more precisely, such as using CPU ticks?

On x86 systems, Intel provides an instruction called `rdtsc` that returns the current 64-bit value of the CPU cycle counter. This allows developers and engineers to create more accurate software counters or evaluate the performance of specific functions or techniques.

In this homework, you'll find a `.rar` file in the "Homework 10" directory on Moodle, named `hw10_2024-12-26.rar`. After unzipping it, you'll obtain a file called `clock.c`, which demonstrates the basic usage of the `rdtsc` instruction.

Take a function in the given `clock.c` as an example:

```
inline unsigned long long int rdtsc()//32-bit
{
        unsigned long long int x;
        asm   volatile ("rdtsc" : "=A" (x));
        return x;
}
```

In the example, as shown, you need to use `asm volatile` to communicate with the CPU and retrieve the current cycle count. The `asm volatile` tells the compiler to directly insert the assembly code into the program. The `volatile` keyword ensures that the compiler does not optimize or reorder the assembly code, which is essential for accurate timing.

On the other hand, `rdtsc` within the parentheses is the x86 instruction that reads the Time Stamp Counter into the `EDX:EAX` registers. The result is 64-bit, with the high 32 bits stored in `EDX` and the low 32 bits in `EAX`.

Lastly, `: "=A" (x)` is the output operand for the assembly instruction. The `=A` constraint tells the compiler to store the high 32-bit of Time Stamp Counter result that is stored in `EDX` register into the `unsigned long long int x` that was declared in the function.

## References

華為(Huawei): What Is IP Routing? (https://info.support.huawei.com/info-finder/encyclopedia/en/IP+routing.html)
鳥哥的 Linux 私房菜 伺服器架設篇 - 第二章、基礎網路概念 (https://linux.vbird.org/linux_server/centos6/0110network_basic.php)
Linux 核心的 hash table 實作 (https://hackmd.io/@sysprog/linux-hashtable)
使用rdtsc指令进行时钟周期级测量 (https://zhou-yuxin.github.io/articles/2018/%E4%BD%BF%E7%94%A8rdtsc%E6%8C%87%E4%BB%A4%E8%BF%9B%E8%A1%8C%E6

%97%B6%E9%92%9F%E5%91%A8%E6%9C%9F%E7%BA%A7%E6%B5%8B%E9%87%8F/index.html)
dudect/src/dudect.h (https://github.com/oreparaz/dudect/blob/master/src/dudect.h#L276)
Intel® Intrinsics Guide - rtdsc (https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.htm#text=rdtsc&ig_expand=4395,5273)

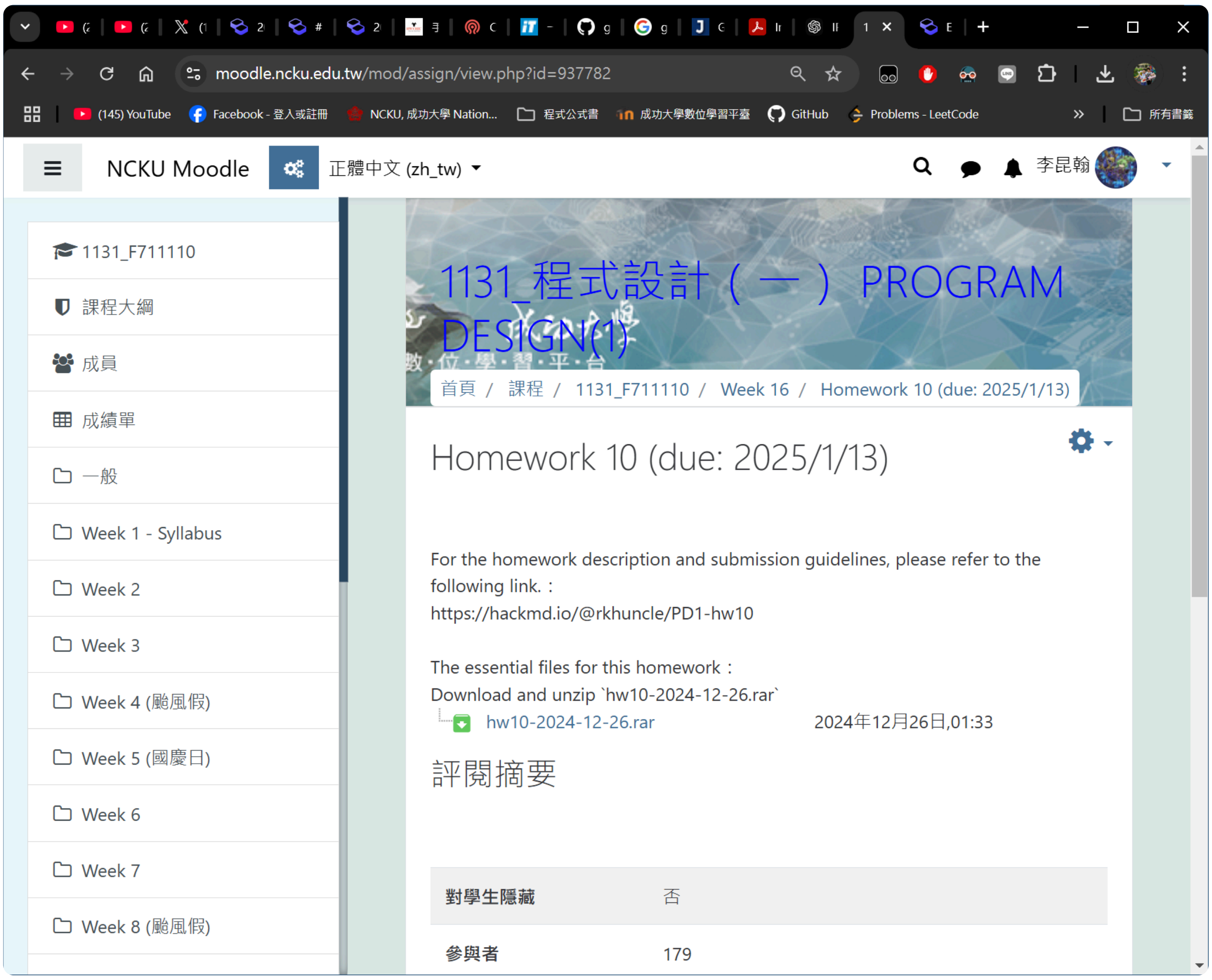
## Problem

In this homework, you are required to complete the following tasks to generate and test a simple router program:

1. **Prefix Grouping and Counting**:
   Using the provided input files, classify the IP prefixes based on their prefix lengths. Then, count the number of prefixes of the specific input file in each group.
2. **Hash Table Implementation and Simulation**:
   Based on the code developed in above task, design a hash table in a specified format to store the information from the given routing table. Implement and simulate the performance of the hash table by creating insertion, deletion, and search functions to operate as a simple router.

Both tasks will be supposed to be implemented in `pA.c`.

The following is the function that we supposed you to implement to help you finish these tasks, and the files mentioned in the following discriptions are given in the `.rar` file in the moodle.



Homework 10 (due: 2025/1/13)

For the homework description and submission guidelines, please refer to the following link. :
https://hackmd.io/@rkhuncle/PD1-hw10

The essential files for this homework :
Download and unzip `hw10-2024-12-26.rar`
hw10-2024-12-26.rar 2024年12月26日,01:33

評閱摘要

| | |
|---|---|
| 對學生隱藏 | 否 |
| 參與者 | 179 |

## Explanation of the files that you would get

After you download `hw10-2024-12-26.rar` and unzip it, you will have following files in the directory `hw10-2024-12-26` :

```
hw10-2024-12-26
 - clock.c
 - deleted_prefix.txt
 - hw10.doc: The original description of this homework.
 - inserted_prefix.txt
 - routing_table.txt
 - trace_file.txt
```

For some files, here are some simple descriptions for them.

- `clock.c` : This file contains sample code and usage instructions for the `rdtsc` instruction, which is used to measure the clock cycle consumption of your function implementations.
- `routing_table.txt` : The file that contains CIDRs (format: `32bit_IP/prefix_length` ) or 32-bit IP address for you to **initialize** your IP routing table **before** applying prefix insertion, deletion, and IP searching.

```
* Example of CIDR format:
1.0.0.0/8
2.0.0.0/8
3.0.0.0/8
4.0.0.0/8
4.1.18.0/24
4.2.62.0/24
...
```

- `inserted_prefix.txt` : The file that contains CIDRs (format: `32bit_IP/prefix_length` ) or 32-bit IP address for you to **insert** the specific IP in the IP routing table.
- `deleted_prefix.txt` : The file that contains CIDRs (format: `32bit_IP/prefix_length` ) or 32-bit IP address for you to **delete** the specific IP in the IP routing table.
- `trace_file.txt` : The file contains 32-bit IP addresses for you to search for a specific IP. Note that during the search, you **must** use **longest prefix matching** for the IP addresses.

> In this homework, in some given input files ( `routing_table.txt` , `inserted_prefix.txt` , `deleted_prefix.txt` ) in moodle, some of the prefix has no length means that
>
> - x.0.0.0 is of length 8
> - x.y.0.0 is of length 16
> - x.y.z.0 is of length 24
> - x.y.z.w is of length 32

## Format of the `struct` for storing IP prefixes (and IPs)

In this homework, you'll be given an amount of data in the form `a.b.c.d` , a 32-bit IP address, in multiple files. For *IP addresses*, you have to store them into an **unsigned variable**; for *prefix length*, you can use an **unsigned character**. Thus, for **all the prefixes**, you can use an **array** of

```
// You must use this structure to store the given IP prefixes that are in CIDR format
struct prefix {
    unsigned ip; // The full 32-bit IP addresses
    unsigned char len; // The prefix length (length of subnet mask)
    struct prefix* next; // singly linked-list pointer
};

struct prefix *p_routing[33]; // sample usage.
                              // you could use other structures to perform
                              // as similar to this.
...
```

to store all the prefixes. Assume prefix `0.0.0.0/0` does not exist.

### Task 1: Prefix Grouping and Counting

> You need to submit the result of this task to the judge.

In this task, you are required to achieve the following goals:

- [ ] Identify the input CIDRs or IP addresses from the given input files– `routing_table.txt` , `inserted_prefix.txt` , `deleted_prefix.txt` , respectively.
- [ ] For the prefixes in `routing_table.txt` , you need to divide the prefixes from them into groups based on their prefix length ( `unsigned char len` ).

Here are the functions you need to implement to complete this task.

#### Function 1 `input(...)`

- Write a function `input(...)` to read all the prefixes from the specific input files ( `routing_table.txt` , `inserted_prefix.txt` , `deleted_prefix.txt` ), saving them in the `struct prefix` structure (since there is amount of data in each file, you surely are saving them within linked-list) we mentioned before.
  - **Hint:** You can save them in different `struct prefix` linked-list based on their file name.
- For **`routing_table.txt`** , you are required to print out the **total number of prefixes** (i.e., how many **CIDRs or IPs** you read in this file) it contains after the calculation.

```
The total number of prefixes in the `routing_table.txt` is : %d.
```

#### Function 2 `group_len(...)`

- Write a function `group_len(...)` to divide the prefixes from **`routing_table.txt`** into groups **based on prefix length**.
  - **Hint:** In 32-bit IP address, the maximum prefix length would be 32.

#### Function 3 `length_distribution(...)`

- Write a function `length_distribution(...)` to compute the number of prefixes in **`routing_table.txt`** with prefix length `i` , for `i = 0 ~ 32` .
- For example:

```
    1.0.0.0/8
    2.0.0.0/8
    4.1.18.0/24
    4.2.62.0/24
    4.2.124.0/24
    4.2.124.0/22
    4.6.0.0/22
    4.6.0.0/16


    the number of prefixes with prefix length 8 = 2
    the number of prefixes with prefix length 16 = 1
    the number of prefixes with prefix length 22 = 2
    the number of prefixes with prefix length 24 = 3
```

### Submission for Task 1: I/O format for the judge

> The `hw10` command and the files aren't set up in the server. Please wait for furthur anouncement. 💦

#### input

```
$ ./pA directory_path/routing_table.txt directory_path/inserted_prefixes.txt \
directory_path/delete_prefixes.txt directory_path/trace_file.txt
```

#### output

1. Print out the **file name** that you read from `argc` , `argv` .
2. Print out the **total number of prefixes** in the input file.
3. Print out the **number of prefixes length** `i` for `i = 0 to 32` .

```
The file initializing the routing table: directory_path/routing_table.txt
The file for extra insertion into the routing table: directory_path/inserted_prefixes.t
The file to delete specific IPs from the routing table: directory_path/delete_prefixes.
The file for IP lookups: directory_path/trace_file.txt

The total number of prefixes in the routing_table.txt is : %d

The number of prefixes with prefix length 0 = %d
The number of prefixes with prefix length 1 = %d
The number of prefixes with prefix length 2 = %d
                           .
                           .
                           .
The number of prefixes with prefix length 32 = %d
```

**Example**

- [ ] **Input:**

```
$ ./pA /share/HW10/routing_table.txt  /share/HW10/inserted_prefixes.txt \
/share/HW10/delete_prefixes.txt /share/HW10/trace_file.txt
```

- [ ] **Output:**

TBA

### Task 2: Hash Table Implementation and Simulation

> You need to submit the simulation result of this task to moodle.

In this task, you are required to

- [ ] Build a hash table data structure to actually save the prefixes you read in Task 1.
- [ ] Implement the insertion and deletion function to the hash table you made.
- [ ] Implemenet a lookup function (search function) to visit the hash table, and find whether the given IP is in the hash table.
- [ ] Simulate how fast your insertion, deletion, and lookup function does during each operation for a given `struct prefix` node or IP address.

#### Function 4 `segment(...)`

- Continueing with the results from `group_len(...)` function in Task 1, for each **non-empty group of length** `i` , create a **hash table of size 256**, respectively, by using the **most significant 8 bits** of IP part of the prefixes.
- Using **singly linked-list** to chain the prefixes together in **each** of **256 buckets** of the hash table. In other words, the prefixes put into a bucket of the hash table must have the **same MSB 8 bits of their IP part**.
- For example:

```
4.1.18.0/24    00000100  00000001  00010010  *
4.2.62.0/24    00000100  00000010  00111110  *
8.2.3.0/24     00001000  00000010  00000011  *
10.2.124.0/24  00001010  00000010  01111100  *
```

```
| 00000000 |
| 00000001 |
|    :     |
| 00000100 | ---> | 4.1.18.0 |---> | 4.2.62.0 |
| 00000101 |
|    :     |
| 00001000 | ---> | 8.2.3.0 |
| 00001001 |
| 00001010 | ---> | 10.2.124.0 |
|    :     |
| 11111110 |
| 11111111 |
```

### Function 5 `prefix_insert(...)`

- Write a function `prefix_insert(...)` to insert a prefix from input file `inserted_prefix.txt` in a **one-by-one fashion in the increasing order** of the unsigned numbers of the prefixes (i.e., the `struct prefix` linked-list in each hash table must be in ascending order, based on `unsigned int IP`).
- A file named **inserted_prefixes** contains some prefixes that will be inserted after the length groups and hash tables are built from the file **routing_table**.

### Function 6 `prefix_delete(...)`

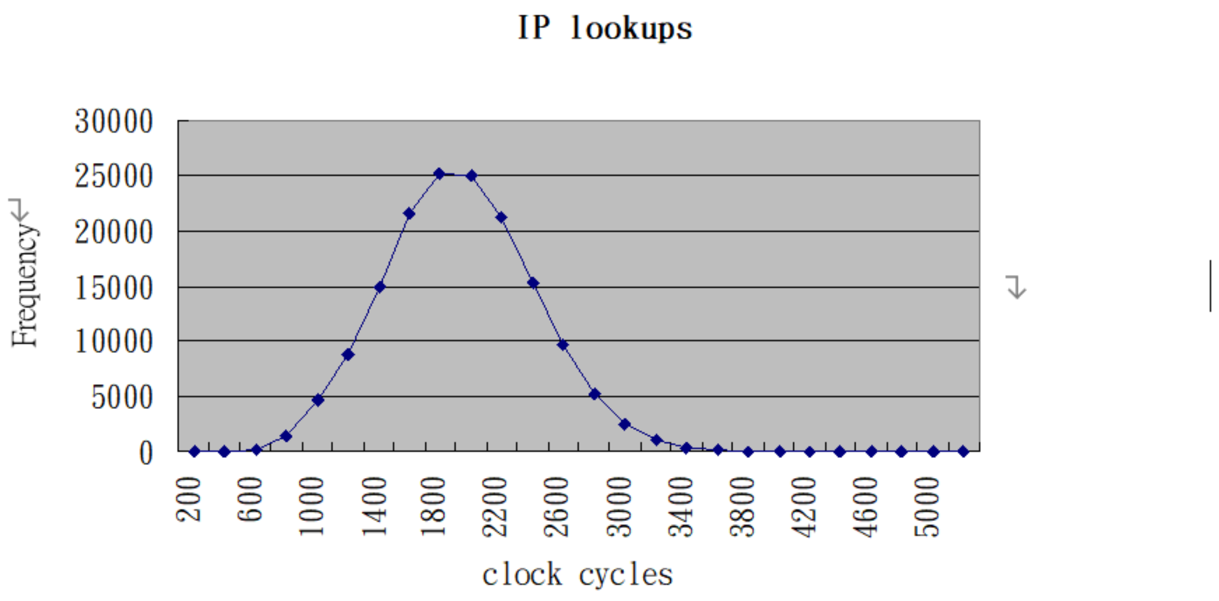
- Write a function `prefix_delete(...)` to delete a prefix that lists input file `deleted_prefix.txt` from the hash table structure.
- This function is expected to work after the function `prefix_insert(...)` is executed.

### Function 7 `search(...)`

- Write a function `search(...)` by giving an IP address from input file `trace_file.txt` to report **if the search is successful or failed**.
  - ▶ **Thoughts**
- This function is expected to work after the functions `prefix_insert(...)` and `prefix_delete(...)` are executed.

## Submission for Task 2: Prepare a PPT and submit it to moodle

- You have to **report** *average numbers* of clock cycles to perform **a search**, **an insertion**, **a deletion** and draw three figures as follows. How to measure the search/insertion/deletion in cycles is illustruated in `clock.c`.
  - ▶ **hint**



[The sample figure]

- **請同學製作一個簡報描述你的程式碼 (Function 4 ~ 7) 的思維與過程以及執行的結果 (insert/delete/search 三張圖片)。**
- **Please create a ppt to describe the thought process and steps behind your code (include Function 4 ~ 7), as well as the execution results (insert/delete/search three figures).**