Zarządzanie bazą zakresów adresów IPv4.

Raport z wykonania zadania dla kandydatów na programistę C/C++ do DSB Atende Software sp. z o.o.

Autor
Damian Krata

Warszawa 2021

1 Wstęp

Niniejszy raport powstał w odpowiedzi na zadanie rekrutacyjne przesłane dnia 04.01.2021. Realizacja zajęła tydzień. Poszczególne komponenty realizacji to:

- Przygotowanie do realizacji zadania (research literatury, opracowanie koncepcji struktury, przygotowanie projektu).
- Realizacja implementacji - poprawa koncepcji.
- Jednoczesne wykonywanie testów dla poszczególnych komponentów (funkcji, wyrażeń lambda, itp.).
- Wykonanie testów końcowych na dużych zbiorach danych.
- Dodanie plików CMake oraz Makefile.
- Wygenerowanie raportu z wykonania zadania.

2 Zadanie

1. Napisać program do zarządzania bazą zakresów adresów IPv4 umożliwiający maksymalnie szybkie lookupowanie zawartości bazy.
2. Opracować samodzielnie struktury i algorytmy.
3. Wskazać problemy implementacji, oraz jej potencjalne kierunki poprawy.

3 Opis interfejsu użytkownika

Interfejs użytkownika opiera się o standardowe wejście/wyjście i jest odzwierciedleniem metod działających podczas wykonywania programu. Do programu można przesłać następujące polecenia:

- `add p`

  gdzie p to adres IPv4 wraz z maską w formacie `a.b.c.d/maska_CIDR`.

- `del p`

  gdzie p to adres IPv4 wraz z maską w formacie `a.b.c.d/maska_CIDR`.

- `check a`

  gdzie a to adres IPv4 bez maski w formacie `a.b.c.d`. Wywołanie komendy zwraca werdykt tak/nie oraz dopasowany prefiks w postaci `a.b.c.d/maska_CIDR`.

- `bench x <lista adresów IPv4>`

  . wywołanie tej metody sprowadza się do wykonania operacji check x razy. W tym wypadku check nic nie zwraca.

- `dump`

  zwraca zawartość struktury w celach debugowych. Z uwagi na szybki wzrost wysokości drzewa oraz liczbę jego elementów, postanowiono wyświetlić jedynie wierzchołek wraz z wektorem jego bezpośrednich dzieci.

4 Struktura bazy

4.1 Wstęp

Początkowo rozważano zastosowanie zwykłego wektora podsieci. Zrezygnowano z tego pomysłu, ze względu na liczbę potencjalnych elementów wektora.

Następnie rozważano strukturę drzewiastą, w której rodzic posiadałby wektor swoich dzieci (podsieci). Postanowiono zrezygnować z tej koncepcji ze względu na dużą złożoność przy dodawaniu i usuwaniu elementów (z powodu posiadania std::vector, wymagane byłoby kopiowanie elementow a nie ich przenoszenie).

Analizując powyższe, postanowiono:

Baza zakresów adresów IPv4 powstała na podstawie struktury drzewiastej. Korzeniem drzewa jest adres `0.0.0.0/0`. Poszczególne węzły struktury mają następującą budowę:

```
struct Subnet {
uint32_t netAddress;
uint32_t mask;
uint32_t broadcastAddress;
bool isValid;
std::vector<std::unique_ptr<Subnet>> subnets;
Subnet();
Subnet(uint32_t address, uint32_t mask);
```

```
    ~Subnet();
};
```

Każdy węzeł posiada adres podsieci, maskę oraz adres rozgłoszeniowy. Podczas realizacji zadania, przyjęto założenie, że mimo tego, że adres podsieci oraz adres rozgłoszeniowy nie są nadawane hostom podsieci, to adresy te faktycznie występują i będą zwracane w przypadku gdy np. administrator wyśle do programu zapytanie `check 0.0.0.0`. W odróżnieniu od drzewa binarnego, w podanej strukturze każda podsieć nie ma wskazania na swoje lewe lub prawe dziecko, ale na wektor wskaźników do poszczególnych podsieci danej podsieci, ponieważ każda sieć może mieć więcej niż jedną podsieć.

Strukturę uzupełniono o wartość logiczną `isValid`, której działanie występuje w operacjach na bazie.

4.2 Operacje na strukturze

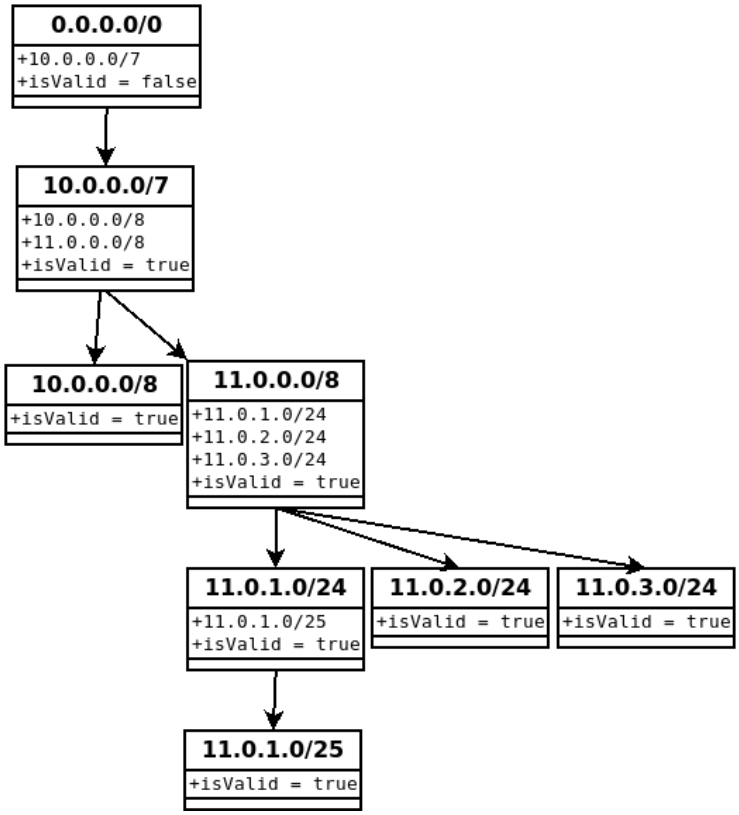
4.2.1 Dodawanie

Początkowy stan, struktury to jej brak. Podczas dodania pierwszego elementu, wykonywane są dwie czynności:

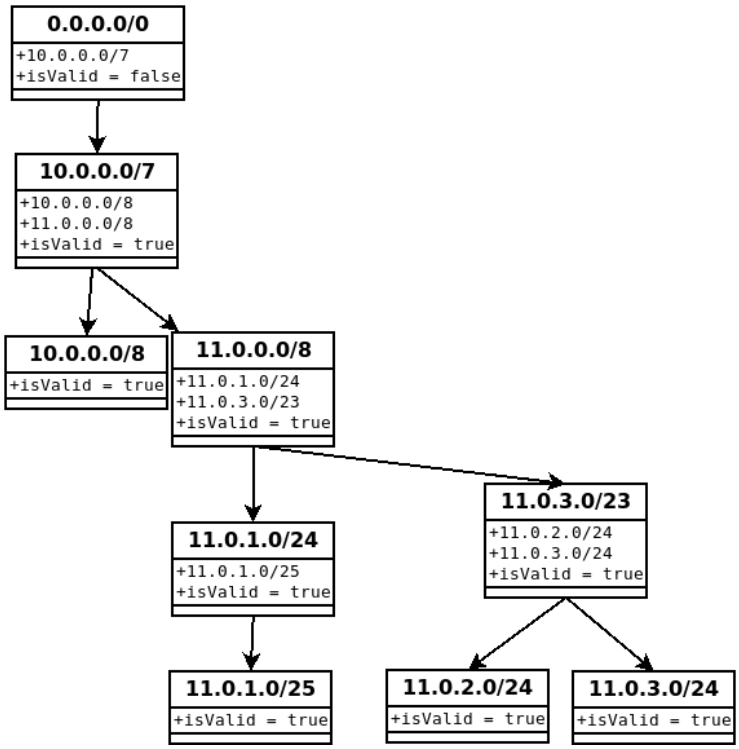
1. Dodanie węzła z siecią `0.0.0.0/0`.
2. Dodanie węzła z siecią wskazaną.

W przypadku dodanie na początku węzła `0.0.0.0/0`, wykona się punkt pierwszy oraz zmienna `isValid` zostanie ustawiona na true;

Dodawanie do struktury zawierającej już elementy, sprowadza się do znalezienia "nadsieci" dla danej podsieci. przeniesienia węzłów z "nadsieci" będących podsieciami danej podsieci oraz dodania tej podsieci do wektora "nadsieci". Dla zilustrowania logiki powstał diagram:


0.0.0.0/0
+10.0.0.0/7
+isValid = false
10.0.0.0/7
+10.0.0.0/8
+11.0.0.0/8
+isValid = true
10.0.0.0/8
+isValid = true
11.0.0.0/8
+11.0.1.0/24
+11.0.2.0/24
+11.0.3.0/24
+isValid = true
11.0.1.0/24
+11.0.1.0/25
+isValid = true
11.0.2.0/24
+isValid = true
11.0.3.0/24
+isValid = true
11.0.1.0/25
+isValid = true

Po dodaniu `11.0.3.0/23`:



Po dodaniu węzła, wykonywane jest także sortowanie elementów wektora należącego do "nadsieci". Sortowanie uwzględnia wartość `netAddress`.

4.2.2 Usuwanie

Usuwanie realizowane jest w podobny sposób. podsieci usuwanej sieci dodawane są w miejsce usuwanej sieci.

4.2.3 Sprawdzanie

Operacja check znajduje najbliższą podsieć oraz zwraca jej parametry.

4.2.4 Ranking

Funkcja bench wykonuje sprawdzenie check podaną liczbę razy.

5 Testowanie

W celu wykonania testów dla działającej aplikacji zrealizowano trzy scenariusze testowe.

1. Test wewnętrzny,
2. Test z pliku,
3. Test z STDIN.

Przed wykonaniem testu wewnętrznego i testu z pliku, zakomentowano w kodzie linijki odpowiedzialne za wyświetlanie komunikatów, w celu sprawdzenia szybkości wykonania operacji dodawania, usuwania i rankingu dla dużych zbiorów danych.

5.1 Test wewnętrzny

W programie zaimplementowano funkcję Test(), mającą za zadanie przeprowadzenie testu. Funkcja realizuje następujące rzeczy:

1. Tworzenie dwóch wektorów po 2 miliony par (adres, maska). Jeden dla dodawania elementów, drugi dla ich usuwania.
2. Uzupełnienie wektorów z generatora pseudolosowego.
3. Wykonanie serii testów (add, del, bench) dla kilku zestawów danych wejściowych.

Początkowo wykonano bench dla jednego elementu w drzewie. Następnie na zmianę dodawano i usuwano po 100 000 elementów (w innych wektorów przez co liczba węzłów w drzewie się zwiększa) i wykonywano bench. Uruchomienie funkcji Test(), to odkomentowanie jednej linijki w kodzie i zakomentowanie `inputManager.getCommand();` (w pliku `main.cpp`). Wyniki przedstawia tabela:

L.p.	operacja	liczba węzłów w drzewie	liczba sprawdzeń	czas [ms]
1	bench	1	10^8	**10976**
2	add/del			2493
3	bench	62208	10^8	**374367**
4	add/del			2825
5	bench	118255	10^8	**176372**
6	add/del			2767
7	bench	172136	10^8	**301533**
8	add/del			2824
9	bench	224556	10^8	**133495**
10	add/del			3036
11	bench	276071	10^8	**266533**
12	add/del			3135
13	bench	326440	10^8	**423639**
14	add/del			3098
15	bench	376024	10^8	**225095**
16	add/del			3423
17	bench	425197	10^8	**307608**
18	add/del			3311
19	bench	473541	10^8	**169249**
20	add/del			3359
21	bench	521424	10^8	**169859**

Wyniki działania dla tego testu umieszczone są w `wynikiTest.txt`.

W procesie przygotowania CMake oraz Makefile, dodano flagi optymalizacji. Pozwoliło to na uzyskanie lepszych rezultatów.

5.2 Test z pliku

Za pomocą skryptu w języku Python wygenerowano plik tekstowy `large-command-file-test` realizujący podobne założenia jak test wewnętrzny. Jedyna zmiana odnosi się do liczby dodanych i usuwanych węzłów. Oraz organizacji ich dodawania i usuwania. W teście wewnętrznym najpierw dodawano wszystkie węzły, a następnie usuwano wszystkie z innego zakresu. W tym teście wykonywane jest dodawanie i usuwanie naprzemiennie. Liczba wywołań add oraz del została zmniejszona do 1000 ponieważ zauważono, że obsługa stru-

mienia jest dużo wolniejsza, co generowało problemy. Otrzymano następujące wyniki:

L.p.	operacja	liczba węzłów w drzewie	liczba sprawdzeń	czas [ms]
1	bench	1	10^7	**972**
3	bench	825	10^7	**32780**
5	bench	1588	10^7	**11783**
7	bench	2323	10^7	**8867**
9	bench	3082	10^7	**9211**
11	bench	3786	10^7	**14137**
13	bench	4488	10^7	**8895**
15	bench	5199	10^7	**12393**
17	bench	5889	10^7	**22430**
19	bench	6584	10^7	**9832**
21	bench	7271	10^7	**40947**

5.3 Test z STDIN

Wyniki z testu otrzymującego dane ze standardowego wejścia znajdują się z pliku `wynikiSTDIN.txt`. Test polegał na wpisaniu w terminalu przykładowej sekwencji poleceń podanych w "Zadaniu dla kandydatów". Wyniki pokrywają się z tymi w zadaniu.

5.4 Optymalizacja

W procesie przygotowania CMake oraz Makefile, dodano flagi optymalizacji. Pozwoliło to na uzyskanie lepszych rezultatów.

5.4.1 Test wewnętrzny

Osiągnięto następujące wyniki:

L.p.	operacja	liczba węzłów w drzewie	liczba sprawdzeń	czas [ms]
1	bench	1	10^8	**247**
2	add/del			92
3	bench	62308	10^8	**4581**
4	add/del			108
5	bench	118677	10^8	**2101**
6	add/del			115
7	bench	172345	10^8	**2326**
8	add/del			122
9	bench	224772	10^8	**5348**
10	add/del			136
11	bench	276217	10^8	**8772**
12	add/del			149
13	bench	326945	10^8	**1736**
14	add/del			150
15	bench	376390	10^8	**2343**
16	add/del			163
17	bench	425583	10^8	**3350**
18	add/del			166
19	bench	474227	10^8	**6527**
20	add/del			169
21	bench	522214	10^8	**8484**

Optymalizacja pozwoliła na nawet 20 krotne przyspieszenie. Wyniki znajdują się w pliku `wynikiTest2.txt`.

5.4.2 Test z pliku

Stosując flagę optymalizacji otrzymano następujące rezultaty:

L.p.	operacja	liczba węzłów w drzewie	liczba sprawdzeń	czas [ms]
1	bench	1	10^7	**25**
3	bench	825	10^7	**414**
5	bench	1588	10^7	**186**
7	bench	2323	10^7	**177**
9	bench	3082	10^7	**170**
11	bench	3786	10^7	**201**
13	bench	4488	10^7	**151**
15	bench	5199	10^7	**241**
17	bench	5889	10^7	**331**
19	bench	6584	10^7	**197**
21	bench	7271	10^7	**735**

W podanym przykładzie przyspieszenie było nawet 50 krotne. Plik z wynikami tego testu to `wynikiPlik2.txt`.

6 Uruchamianie

Program został napisany w języku C++. Zastosowano w nim elementy standardów C++11/14/17. Początkowo powstał plik `CMakeLists.txt`, który automatycznie generuje plik Makefile, który następnie kompiluje, buduje i linkuje projekt. Było to moje pierwsze doświadczenie z plikami CMake oraz Makefile. W treści zadania znajduje się wymaganie, aby program kompilował się za pomocą GCC. Niestety podczas próby kompilacji za pomocą GCC otrzymałem komunikat `undefined reference std::cout...`. Okazuje się, że GCC automatycznie nie linkuje biblioteki standardowej. Pozostałem zatem przy g++.

6.1 CMake

W celu kompilacji projektu powstał plik CMakeList.txt. Wskazuje on ścieżkę do plików źródłowych, nagłówkowych oraz standard C++17. Dodatkowo umieszczona została w nim flaga optymalizacji `-O2`. Procedura kompilacji:

- cmake CMakeLists.txt
- make
- ./main (bądź przekierowanie strumienia z pliku i do pliku).

6.2 Makefile

w pliku Makefile powstały dyrektywy:

1. main - główna dyrektywa budująca projekt,
2. clean - czyszczenie poprzedniej kompilacji,
3. info oraz debug - w celach naukowych.

Procedura kompilacji:

- make
- ./main (bądź przekierowanie strumienia z pliku i do pliku).

7 Wnioski

Implementacji struktury, posiadającej wektor `std::unique_ptr<>` pozwala na realizację postawionych założeń. W związku z użyciem optymalizacji, wydajność oferowanego rozwiązania poprawiła się około 20 razy w stosunku do pierwotnej wersji.

Literatura

[1] Stephen Prata, Helion, *Język C++. Szkoła programowania. Wydanie VI.* 2013.

[2] Scott Meyers, Promise, *Skuteczny nowoczesny C++.* 2015.

[3] Kurt Guntheroth, Promise, *C++ Optymalizacja kodu.* 2016.

[4] Mark A. Dye, Rick McDonald, Antoon W. Rufi, Wydawnictwo Naukowe PWN, *Akademia Sieci Cisco Semestr 1 Podstawy Sieci.* 2013.

[5] Evi Nemeth, Garth Snyder, Trent R. Hein, Ben Whaley, Dan Mackin, Helion, *Unix i Linux. Przewodnik administratora systemów. Wydanie V.* 2018.