T.C
GALATASARAY ÜNİVERSİTESİ
MÜHENDİSLİK VE TEKNOLOJİ FAKÜLTESİ

INF 224 Veri Yapısı ve Algoritmalar
Proje Raporu

Hazırlayan
DORUK BULUT

Aralık, 2021

Table des Matières

1) Introduction

La vitesse des services sanitaires rencontre beaucoup de difficultés. Des milliers de personnes meurent à cause d'une intervention tardive. Le temps d'action dépend sur plusieurs aspects comme le nombre de spécialiste médicale,l'équipement nécessaire, et la vitesse à laquelle le patient arrive à l'hôpital.

Le but de ce projet est d'améliorer le temps de trajet d'un patient à l'aide d'une application. Cette application permet d'automatiser le choix d'hôpital que doivent faire les chauffeurs d'ambulance, en leur donnant l'hôpital qui est le plus proche mais aussi le plus vide. De plus, il permet d'accéder à des informations sur les clients après avoir les localiser.

Le projet ne considère que les paramètres de distance et le rapport de plénitude étant donné que les ambulances peuvent circuler librement sur les routes. Finalement, l'application a été programmée avec la langage C.

Dans un premier temps, nous allons examiner l'architecture du projet,et puis, les résultats obtenus après des essais sur plusieurs échantillons.

2) L’architecture du Projet

Pour résoudre le problème il fallait trouver un moyen de représenter les régions et les hôpitaux dans la langage C,avec un temps de recherche de ces informations qui serait minimal.

A ce point, l'utilisation d’arbres, plus spécialement des Arbres Binaire AVL(Adelson-Velsky-Landis), et la liste chaînée était les structures les plus efficaces devant les autres. Des explications seront données plus tard dans ce rapport.

```
//struct Hospital

struct Host{
    int host_id;
    int num_patient;
    struct Host*  next;
};

//struct Area


struct Area{

    int id;
    struct Host* head;
    struct Area* r;
    struct Area* l;
    int height;
};
```

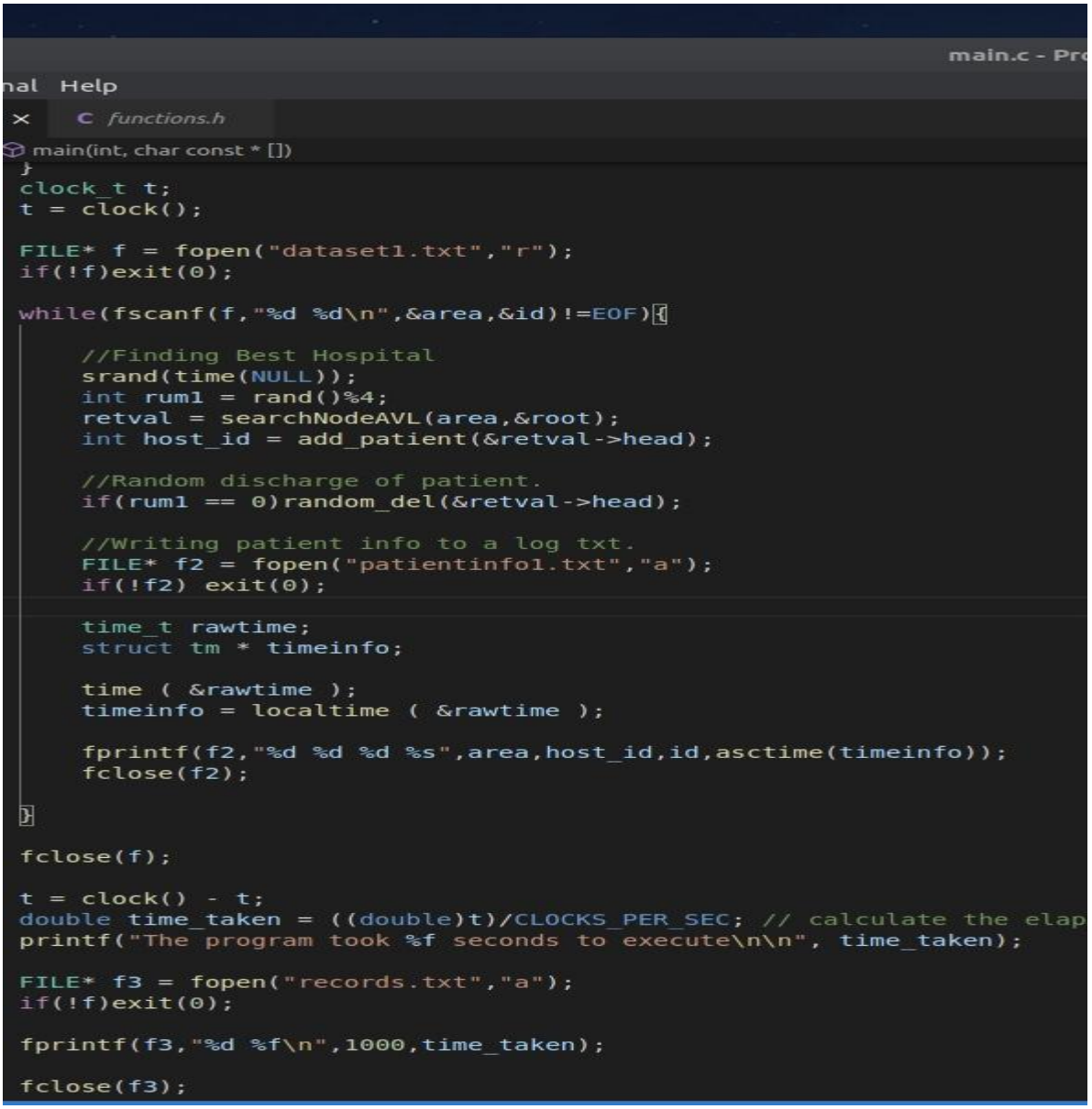
L’arbre AVL a été choisi pour représenter les différentes régions d’une ville en fonction des ces numéros d’identifiant. Comme les arbres AVL sont en équilibre , le temps de

recherche est optimisé: on évite de ce retrouver avec une liste chaînée des identifiants, ce qui n'est pas le cas pour les arbres binaires simples.En remarque aussi qu'on pourrait implémenter les tableaux qui prendra un temps O(1) pour retrouver les régions. Mais les tableaux étant statiques, il est coûteux d'ajouter de nouvelles régions.

Chaque nœud de cet arbre possède un attribut "ID" et le pointeur "head" a une liste chaînée. Ce dernier est là pour représenter les hôpitaux appartenant à la région, avec les attribut "ID" et le nombres actuelles de patients.

Le programme marche en fonction des données de clients : l'algorithme reçoit la numéro d'identifiants de la région et du patient auquel est appelé l'ambulance. Ce dernier est une dossier txt qui joue le rôle d'un dossier "log".
Ensuite, elle fait une recherche sur l'arbre AVL et retrouve les hôpitaux de la région correspondant. Enfin, elle retrouve l'hôpital qui est le plus vide parmi les autres (Les hôpitaux sont représentés sous forme de liste chaînée dans les nœuds de l'arbre) et enregistre la date d'enregistrement, le numéro d'identifiant et l'hôpital choisie sur un autre "log file " en

format “.txt”.


main.c - Pro
nal Help
functions.h
main(int, char const * [])
clock_t t;
t = clock();
FILE* f = fopen("dataset1.txt","r");
if(!f)exit(0);
while(fscanf(f,"%d %d\n",&area,&id)!=EOF){
//Finding Best Hospital
srand(time(NULL));
int rum1 = rand()%4;
retval = searchNodeAVL(area,&root);
int host_id = add_patient(&retval->head);
//Random discharge of patient.
if(rum1 == 0)random_del(&retval->head);
//Writing patient info to a log txt.
FILE* f2 = fopen("patientinfo1.txt","a");
if(!f2) exit(0);
time_t rawtime;
struct tm * timeinfo;
time (&rawtime);
timeinfo = localtime (&rawtime);
fprintf(f2,"%d %d %d %s",area,host_id,id,asctime(timeinfo));
fclose(f2);
}
fclose(f);
t = clock() - t;
double time_taken = ((double)t)/CLOCKS_PER_SEC; // calculate the elap
printf("The program took %f seconds to execute\n\n", time_taken);
FILE* f3 = fopen("records.txt","a");
if(!f)exit(0);
fprintf(f3,"%d %f\n",1000,time_taken);
fclose(f3);

Run Terminal Help

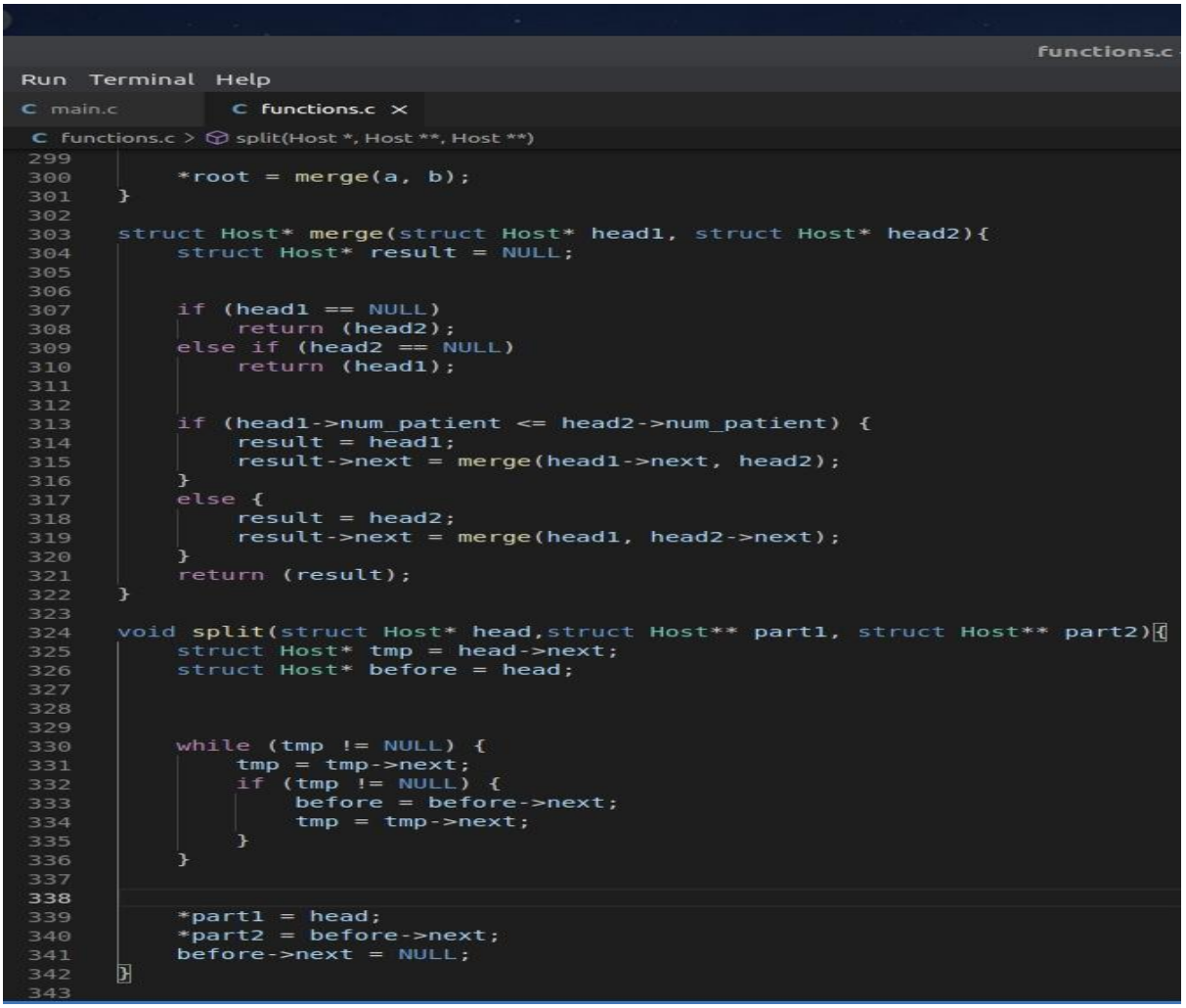
main.c | functions.c | patientinfo4.txt

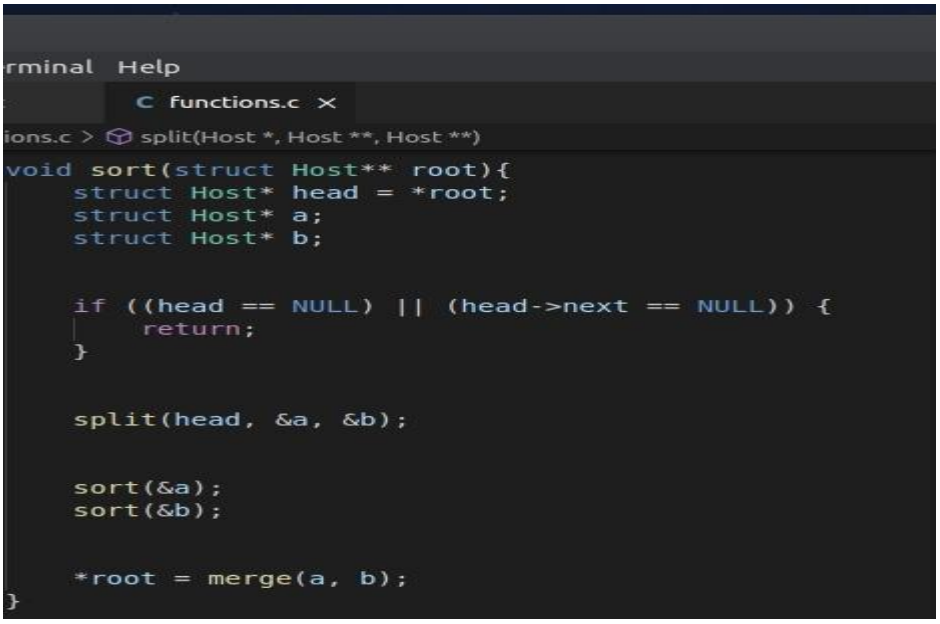
patientinfo4.txt

```
996158   14 4 996158 Sun Dec 19 18:14:30 2021
996159   10 8 996159 Sun Dec 19 18:14:30 2021
996160   6 3 996160 Sun Dec 19 18:14:30 2021
996161   11 4 996161 Sun Dec 19 18:14:30 2021
996162   3 2 996162 Sun Dec 19 18:14:30 2021
996163   3 3 996163 Sun Dec 19 18:14:30 2021
996164   20 8 996164 Sun Dec 19 18:14:30 2021
996165   19 9 996165 Sun Dec 19 18:14:30 2021
996166   3 4 996166 Sun Dec 19 18:14:30 2021
996167   11 8 996167 Sun Dec 19 18:14:30 2021
996168   4 7 996168 Sun Dec 19 18:14:30 2021
996169   16 7 996169 Sun Dec 19 18:14:30 2021
996170   8 3 996170 Sun Dec 19 18:14:30 2021
996171   8 3 996171 Sun Dec 19 18:14:30 2021
996172   10 7 996172 Sun Dec 19 18:14:30 2021
996173   18 5 996173 Sun Dec 19 18:14:30 2021
996174   14 5 996174 Sun Dec 19 18:14:30 2021
996175   15 3 996175 Sun Dec 19 18:14:30 2021
996176   14 1 996176 Sun Dec 19 18:14:30 2021
996177   15 10 996177 Sun Dec 19 18:14:30 2021
996178   18 2 996178 Sun Dec 19 18:14:30 2021
996179   12 6 996179 Sun Dec 19 18:14:30 2021
996180   12 7 996180 Sun Dec 19 18:14:30 2021
996181   20 7 996181 Sun Dec 19 18:14:30 2021
996182   16 6 996182 Sun Dec 19 18:14:30 2021
996183   4 2 996183 Sun Dec 19 18:14:30 2021
996184   19 3 996184 Sun Dec 19 18:14:30 2021
996185   19 2 996185 Sun Dec 19 18:14:30 2021
996186   17 3 996186 Sun Dec 19 18:14:30 2021
996187   20 6 996187 Sun Dec 19 18:14:30 2021
996188   9 8 996188 Sun Dec 19 18:14:30 2021
996189   3 5 996189 Sun Dec 19 18:14:30 2021
996190   2 7 996190 Sun Dec 19 18:14:30 2021
996191   14 1 996191 Sun Dec 19 18:14:30 2021
996192   5 2 996192 Sun Dec 19 18:14:30 2021
996193   4 3 996193 Sun Dec 19 18:14:30 2021
996194   8 5 996194 Sun Dec 19 18:14:30 2021
996195   4 4 996195 Sun Dec 19 18:14:30 2021
996196   14 5 996196 Sun Dec 19 18:14:30 2021
996197   2 1 996197 Sun Dec 19 18:14:30 2021
996198   14 4 996198 Sun Dec 19 18:14:30 2021
996199   9 8 996199 Sun Dec 19 18:14:30 2021
996200   17 2 996200 Sun Dec 19 18:14:30 2021
996201   13 7 996201 Sun Dec 19 18:14:30 2021
```

A ce point la il reste à trouver l'hôpital le plus vide. On pourrait penser à une recherche linéaire sur la liste chaînée. Mais il est plus pratique d'effectuer un triage en ordre croissant. Ainsi, l'hôpital le plus vide serait le premier élément de la liste qui serait facilement retrouvable.

Tri par fusion,(Merge Sort) est L'algorithme choisi pour effectuer la tâche. Elle est préférable à cause de sa performance sur les listes chaînées ; Tri par insertion, par sélection, à bulle, rapide donnent de résultats médiocre avec leurs temps d’ exécutions qui est 0(N^2). Cependant, le tri en tas reste impossible pour les listes chaînées.


functions.c
Run Terminal Help
main.c
functions.c
functions.c > split(Host *, Host **, Host **)
299
300 *root = merge(a, b);
301 }
302
303 struct Host* merge(struct Host* head1, struct Host* head2){
304 struct Host* result = NULL;
305
306
307 if (head1 == NULL)
308 return (head2);
309 else if (head2 == NULL)
310 return (head1);
311
312
313 if (head1->num_patient <= head2->num_patient) {
314 result = head1;
315 result->next = merge(head1->next, head2);
316 }
317 else {
318 result = head2;
319 result->next = merge(head1, head2->next);
320 }
321 return (result);
322 }
323
324 void split(struct Host* head,struct Host** part1, struct Host** part2){
325 struct Host* tmp = head->next;
326 struct Host* before = head;
327
328
329
330 while (tmp != NULL) {
331 tmp = tmp->next;
332 if (tmp != NULL) {
333 before = before->next;
334 tmp = tmp->next;
335 }
336 }
337
338
339 *part1 = head;
340 *part2 = before->next;
341 before->next = NULL;
342 }
343


rminal Help
functions.c
ions.c > split(Host *, Host **, Host **)
void sort(struct Host** root){
struct Host* head = *root;
struct Host* a;
struct Host* b;
if ((head == NULL) || (head->next == NULL)) {
return;
}
split(head, &a, &b);
sort(&a);
sort(&b);
*root = merge(a, b);
}

3) Résultats

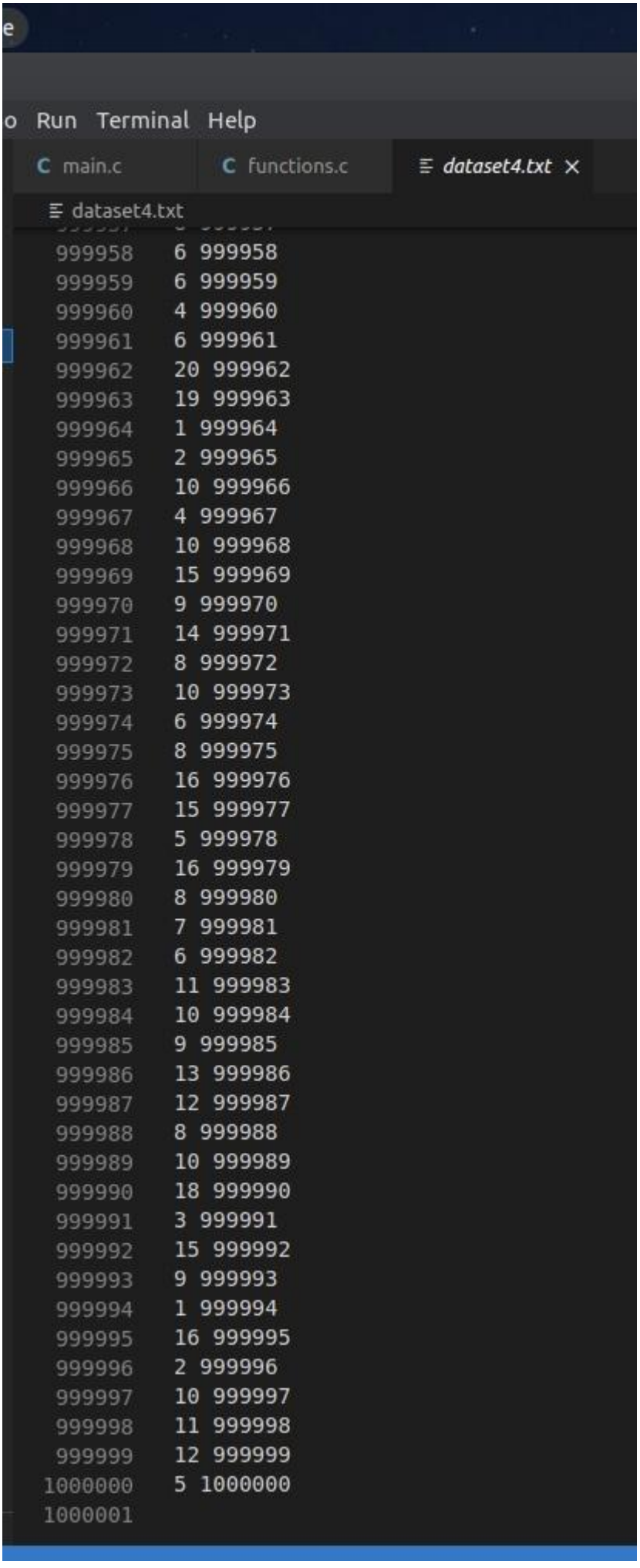
Après avoir construit l'architecture, il restait à tester l’algorithme en saisissant des clients.

Tout d'abord, des rapports de plénitude aléatoire et une chance ¼ de décharge pour l'hôpital retrouver son mis en place afin de simuler la vie courante.

Ensuite, des échantillons de différents nombres et une fonction C de chronométrage ont été créés pour mesurer la performance .

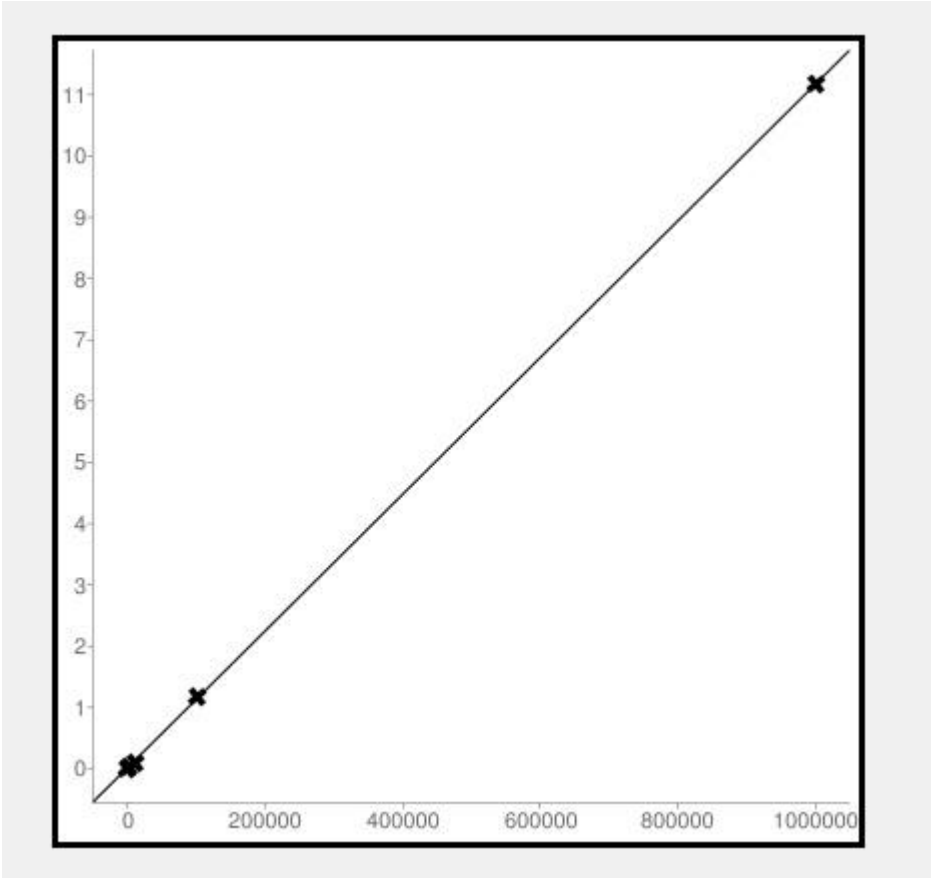
Théoriquement, la recherche sur l’arbre AVL prend un temps O(logN) et le tri par fusionnement O(NLogN). Donc, pour **un** patient, la performance d’algorithme serait O(NLogN). Par contre, le nombre d'hôpitaux et de régions sont finies dans une ville, voir même ça ne dépasse pas 10^3. Ainsi, nous pouvons considérer que l'opération de recherche et de tri sont des constantes, et fait en O(1) pour un client. Ceci permet de conclure que pour M clients, la performance de l’algorithme est linéaire : M* O(1).

Pour tester la théorie, quatre base de données ont été avec 10^3, 10^4,10^5,10^6 clients avec un “ID” attribuer et des régions aléatoires. La performance de l’algorithme est enregistrée en secondes pour chaque base. Les résultats ont été interprétés sous forme d’un graphique sur Microsoft Excel.


e
o Run Terminal Help
main.c
functions.c
dataset4.txt
dataset4.txt
999958 6 999958
999959 6 999959
999960 4 999960
999961 6 999961
999962 20 999962
999963 19 999963
999964 1 999964
999965 2 999965
999966 10 999966
999967 4 999967
999968 10 999968
999969 15 999969
999970 9 999970
999971 14 999971
999972 8 999972
999973 10 999973
999974 6 999974
999975 8 999975
999976 16 999976
999977 15 999977
999978 5 999978
999979 16 999979
999980 8 999980
999981 7 999981
999982 6 999982
999983 11 999983
999984 10 999984
999985 9 999985
999986 13 999986
999987 12 999987
999988 8 999988
999989 10 999989
999990 18 999990
999991 3 999991
999992 15 999992
999993 9 999993
999994 1 999994
999995 16 999995
999996 2 999996
999997 10 999997
999998 11 999998
999999 12 999999
1000000 5 1000000
1000001

records.txt

```
1000000 11.168635 secs.
100000 1.174690 secs.
10000 0.102920 secs.
1000 0.018274 secs.
```



Finalement nous trouvons des résultats similaires à la théorie.

4) Conclusion

Pour conclure, cet algorithme propose une solution efficace pour le problème concerné. Comme le programme a besoin d’un etree d'information du patient, sa meilleurs performance dépendra toujours du nombre de patients saisit. De plus, même s' il aura une demande d’un million de personnes en même temps, le test nous montre qu’il prend dix secondes pour localiser tous les patients.Cependant, l’algorithme peut être amélioré en faisant intervenir plus de variable en ce qui concerne le choix de l'hôpital comme la route et le trafic (en retrouvant l’informations à l'aide de GPS).