Ana Carolina da Silva Carvalho de Sousa
Carlos Adir Ely Murussi Leite

1 Structures de données

Questão 1

Dans le fichier type_def.h, le type "personne" a été créé. Pour cela, un enregistrement a été réalisé avec trois caractéristiques définies: nom, prenom et numss (numéro de sécurité). Toutes les fonctionnalités sont de type (chaine de caractère, c'est-à-dire des strings.

```
Enregistrement personne
    chaine de caractère nom;
    chaine de caractère prenom;
    chaine de caractère numss;
Fin Enregistrement
```

Questão 2

Ensuite, une structure de données "vectPersonne" a été définie au format vectoriel. Sa taille est définie dans une constante égale à 10 000 et ses éléments sont du type personne.

Questão 3

Le type elementListe sera un élément d'une liste chaînée contenant une personne et deux pointeurs, un précédent et un suivant. Pour cela, vous devez faire un enregistrement:

```
Enregistrement elementListe
    personne p;
     lien elementListe suivant;
     lien elementListe precedent;
Fin Enregistrement
```

Questão 4

Le type "noeud" sera un élément d'un arbre. Pour ce faire, il doit stocker une entité de type "personne", plus deux liens "FilsGauche" et "FilsDroit". Pour cela nous devons faire un enregistrement:

```
Enregistrement noeud
    personne p;
    lien noeud FilsGauche;
    lien noeud FilsDroit;
Fin Enregistrement
```

2 Fonctions dans le fichier Utilitaires.cpp

Question 5: Fonction initTabNomPrenom

```
Description:
Cette fonction lit deux textes, "Nom.text" et "Prenom.text", qui contiennent dans leur première
ligne le nombre de noms ou prénoms, les noms et prénoms. La fonction stocke ces noms et prénoms
dans une liste de strings. Les entrées de fonction sont deux listes de strings et les nombres
de noms et de prénoms existant dans les textes. La fonction renvoie un booléen uniquement
à des fins de test.

Spécification:
Fonction : Test, Listenom, Listeprenom, nbNom, nbPrenom <-
initTabNomPrenom(fichierNoms, fichierPrenoms)
Parametres: Chaine de caractere fichierNoms, fichierPrenoms
Resultats : Vecteur de chaine Listenom, Listeprenom
            Entier nbNom, nbPrenom
            Booléen Test
```

Question 6: Fonction genererRepertoire

```
Description:
Cette fonction utilise les fonctions "initTabNomPrenom" et "genererPersonne" pour écrire
un nombre de personnes (donné en tant que constante "QTD_PERSONNES") dans le document
"repertoire.text".La fonction renvoie un booléen pour des raisons de test.
Le retourne vrai si il a reussi creer le fichier, faux sinon.

Spécification:
Fonction : Test <-  genererRepertoire()
Paramètres: 0;
Résultats : Booléen Test
```

Question 7: Fonction egalitePersonne

```
Description:
Cette fonction pendre deux enregistrements de personne et compare leurs noms,prénoms et numéros
de sécurité sociale. C'est une fonction booléenne qui retourne vraie si les informations
sont égales ou fausses sinon.

Spécification:
Fonction : Comparaison <-  egalitePersonne (p1, p2)
Paramètres: personne p1, p2;
Résultats : Booléen Comparaison
```

Question 8: Fonction comparerPersonne

```
Description:
Cette fonction compare deux personnes basées sur ses noms, après prénom et numéro de sécurité
sociale. C'est une fonction booléenne qui retourne vraie si la personne 1 -c'est-à-dire première
paramètre - est avant l'autre et faux sinon.

Spécification:
Fonction : Comparaison <-  comparerPersonne (p1, p2)
Paramètres: personne p1, p2;
Résultats : Booléen Comparaison
```

3 Creation et utilisation d'un répertoire(Tableau)

Question 9 - Fonction afficherTableau

```
Description:
Cette fonction permet d'afficher le nom, le prénom et le numéro de sécurité sociale de chaque
personne d'un Tableau de personnes.

Spécification:
Fonction  : 0 <- afficherTableau (tableau, taille_t)
Paramètres: Vectpersonne tableau;
            Entier taille_t;
Résultats : 0
```

Question 10 - Fonction ajouterTableau

```
Description:
Cette fonction ajoute dans Tableau un élément de type personne au bon endroit, c'est-à-dire par
rapport à l'ordre de son nom, de son prénom et de son numéro de sécurité sociale.

Spécification:
Fonction  : 0 <-  ajouterTableau (tableau, taille_t, p)
```

```
Paramètres: Vectpersonne tableau;
            Entier taille_t;
            personne p;
Résultats : 0;
```

Question 11 - Fonction rechercherTableau

```
Description:
Cette fonction recherche une personne, donnée comme paramètre,
dans un tableau de personnes et retourne
* vrai si il existe dans le tableau
* faux sinon.

Spécification:
Fonction  : Existe <- rechercherTableu (tableau, taille_t, p)
Paramètres: Vectpersonne tableau;
            Entier taille_t;
            personne p;
Résultats : Booléen Existe;
```

Question 12 - Fonction supprimerTableu

```
Description:
Cette fonction vous permet de supprimer une personne, donnée en entrée,
d'un tableau également donné en entrée. Cette suppression est faite à
partir de comparaison avec la fonction "egalitePersonne".

Spécification:
Fonction  : 0 <-  supprimerTableau (tableau, taille_t, p)
Paramètres: Vectpersonne tableau;
            Entier taille_t;
            personne p;
Résultats : 0;
```

Question 13 - Fonction TableuLectureRepertoire

```
Description:
Lecture des informations du fichier "repertoire.txt", donnée comme entrée,
pour créer  un tableau trié avec les éléments du type personne.

Spécification:
Fonction  : numero <- tableuLectureRepertoire(vect, repertoire_text)
Paramètres: Vectpersonne vect;
            Chaine de caractere repertoire_text;
Résultats : entier numero
```

4 Creation et utilisation d'un répertoire(Liste)

Question 9 - Fonction afficherListe

```
Description:
Cette fonction permet d'afficher le nom, le prénom et le nombre de
sécurité sociale de chaque personne d'un Liste de personnes.
Comme la fonction seulement affiche, elle renvoyé rien

Spécification:
Fonction  : 0 <- afficherListe (liste)
Paramètres: lien ElementListe liste;
Résultats : 0
```

Question 10 - Fonction ajouterListe

```
Description:
Cette fonction ajoute un élément de type personne au bon
endroit dans une liste donné à travers des paramètres,
c'est-à-dire par rapport à l'ordre de son nom, de son prénom
et de son nombre de sécurité sociale.
La fonction renvoyé un booléen qui dire si elle a pu ajouter
la personne dans la liste:
* Vrai si elle a pu ajouter
* Faux si non

Spécification:
Fonction  : reussi <- ajouterListe (liste, p)
Paramètres: lien ElementListe liste;
            personne p; doublement enchaînée
Résultats : booléen reussi;
```

Question 11 - Fonction rechercherListe

```
Description:
Cette fonction recherche une personne, donnée comme paramètre, dans une
liste de personnes.
* Vrai si il y a la personne dans la liste
* Faux sinon

Spécification:
Fonction  : existe <- rechercherListe (liste, p)
Paramètres: lien ElementListe liste;
            personne p;
Résultats : booléen existe;
```

Question 12 - Fonction supprimerListe

```
Description:
Cette fonction vous permet de supprimer une personne, donnée en entrée,
d'un Liste également donné en entrée. Cette suppression est faite à
partir de la recherche du nom, du prénom et du numéro de sécurité de la
personne.

Spécification:
Fonction  : liste <- supprimerListe (liste, p)
Paramètres: lien ElementListe liste;
            personne p;
Résultats : lien ElementListe liste;
```

Question 13 - Fonction LectureRepertoireListe

```
Description:
La fonction faire la lecture des informations du fichier "repertoire.txt", donnée comme entrée, pour

Spécification:
Fonction  : liste <- LectureRepertoireListe(repertoire_text)
Paramètres: Chaine de caractere repertoire_text;
Résultats : lien ElementListe liste;
```

5 Création et utilisation d'un répertoire(Arbre)

Question 9 - Afficher

Pour montrer nous avons trois manières qui sont

- Prefixe

- Infixe
- Posfixe

Comme nous voulons afficher la arbre comme ordonnée, alors nous irons utiliser la méthode d'affichage *infixe*, bien sur que les autres méthodes d'affichage ont implémentées et sont dans les fichiers(exactement au repertoire.cpp).

Le algorithme que nous avons utilisé est ici-dessous. Pour montrer nous avons utilisé la récursion.

Algorithm 1 AfficherArbreInfixe

Entrée: noeud *racine*
Sortie: $\emptyset$

Algorithme debut AFFICHERARBREINFIXE( )
  si $racine \neq nulle$ **alors**
    $afficherArbreInfixe(racine.filsGauche)$
    $afficherPersonne(racine.personne)$
    $afficherArbreInfixe(racine.filsDroit)$
  fin si
fin Algorithme debut

Question 10 - Ajouter

Pour ajouter à la arbre, nous devons faire la comparaison a tout le moment.

La idée est que nous irons recevoir la nouvelle personne à ajouter, e comparer si est avant ou après la personne que est au noeud. Si la personne est avant, nous irons appeler la fonction récursivement au noeud à gauche, et si est après, on fait le même chose mais avec le noeud à droit. Quand la fonction trouve un noeud avec le suivant qui est nulle, alors la fonction crée un nouveau noeud.

C'est la idée principale dans le code, mais comme nous utilisons les pontiers, la fonction renvoyé le pontier du noeud suivant. C'est important seulement quand nous créons un nouveau noeud.

Algorithm 2 AjouterArbre

Entrée: lien noeud *racine*
  personne p
Sortie: lien noeud *proxracine*

Algorithme debut AJOUTERARBRE( )
  si $racine = nulle$ **alors**
    $proxracine \leftarrow creeNoeud(p)$
    {Ici nous prenons le lien pour le nouveau noeud}
  sinon
    si $comparerPersonne(p, racine.p)$ **alors**
      {Si p est avant que $racine.p$}
      $(racine.filsGauche) \leftarrow ajouterArbre(racine.filsGauche, p)$
    sinon si $comparerPersonne(racine.p, p)$ **alors**
      {Si p est apres que $racine.p$}
      $(racine.filsGauche) \leftarrow ajouterArbre(racine.filsGauche, p)$
    sinon
      {Si p est la meme que $racine.p$}
      $Ecrire$("*La personne est deja dans la arbre*")
    fin si
    $proxracine \leftarrow racine$
  fin si
  renvoyer *proxracine*
fin Algorithme debut

Question 11 - Recherche

Pour rechercher, nous faisons presque le même chose que nous avons fait à la fonction AjouterArbre, pour savoir où ajouter le nouveau noeud:

1. Si la racine est nulle, la personne n'est pas dans l'arbre

2. Si la personne est avant, on va récursivement à gauche
3. Si la personne est après, on va récursivement à droit
4. Si la personne est la même que racine, nous avons trouvé.

Algorithm 3 rechercherArbre

Entrée: lien noeud *racine*
personne *p*
Sortie: boolean *trouve*

```
1:  Algorithme debut RECHERCHERARBRE( )
2:      si racine = nulle alors
3:          trouve ← faux
4:      sinon si comparerPersonne(p, racine.p) alors
5:          {Si p est avant que racine.p}
6:          trouve ← rechercherArbre(racine.filsGauche, p)
7:      sinon si comparerPersonne(racine.p, p) alors
8:          {Si p est apres que racine.p}
9:          trouve ← rechercherArbre(racine.filsGauche, p)
10:     sinon
11:         {Si p est la meme que racine.p}
12:         trouve ← vrai
13:     fin si
14:     renvoyer trouve
15: fin Algorithme debut
```

Question 12 - Supprimer

Entre toute les fonctions que on a utilise ici, c'est la plus difficile de faire l'algorithme.

La idée principale est que quand nous faisons la suppression d'un élément dans une arbre, nous irons appeler la fonction de supprimer récursivement jusqu'à trouver le élément, ou un lien nul qui signifie qui l'élément n'est pas dedans.

Quand nous trouvons le élément à supprimer, nous pouvons avoir 3 cas:

- La racine à gauche est nulle
- La racine à droit est nulle
- Les deux racines ne sont pas nulles.

Ici nous irons partager le problème en 3:

Cas 1 - racine à gauche est nulle

Pour faire ça, il suffit que nous renvoyons la racine à droit, parce qui le noeud au-dessus va prendre le noeud à gauche et oublier le noeud que nous voulons supprimer.

Cas 2 - racine à droit est nulle

Presque le même chose que le cas 1, mais nous renvoyons la racine à gauche au noeud au-dessous et perdre la référence au noeud que nous voulons supprimer.

Cas 3 - Aucune racine est nulle

Cette cas est le plus difficile. Nous devons prendre le noeud plus grand possible, mais que est aussi plus petit que le noeud que nous voulons supprimer. Pour faire ça, laissez nous appeler les noeud pour les noms:

- k est le noeud parent de a
- a est le noeud qui nous voulons supprimer
- b est le fil à gauche de a
- c est le fil à droit de a

- e est le noeud fils tout à droit de b(alors, e n'a pas fils à droit)
- d est le parent de e
- f est le fil à gauche de e

Les étapes sont

1. Faire c être le fil à droit de e
2. Faire f être le fil à droit de d
3. Faire b être le fil à gauche de e
4. Faire e être le fil de k(renvoyer le adresse de e)

Algorithm 4 supprimerArbre

Entrée: lien noeud *racine*
personne p
Sortie: lien noeud *proxracine*

```
1:  Variable Locales:
2:  lien noeud auxiliar
3:  Algorithme debut SUPPRIMERARBRE( )
4:      si racine = nul alors
5:          proxracine ← nul
6:          {Ne a pas pu etre trouvé}
7:      sinon
8:          si comparerPersonne(p, racine.p) alors
9:              {Si p est avant que racine.p}
10:             (racine.filsGauche) ← ajouterArbre(racine.filsGauche, p)
11:             proxracine ← racine
12:         sinon si comparerPersonne(racine.p, p) alors
13:             {Si p est apres que racine.p}
14:             (racine.filsGauche) ← ajouterArbre(racine.filsGauche, p)
15:             proxracine ← racine
16:         sinon
17:             {Ici nous avons trouvé le noued pour supprimer}
18:             si racine.filsGauche = nul alors {Cas 1}
19:                 proxracine ← racine.filsDroit
20:             sinon si racine.filsDroit = nul alors {Cas 2}
21:                 proxracine ← racine.filsGauche
22:             sinon{Cas 3}
23:                 auxiliar ← racine.filsGauche
24:                 si auxiliar.filsDroit = nul alors
25:                     auxiliar.filsDroit ← racine.filsDroit
26:                     proxracine ← auxiliar
27:                 sinon
28:                     tant que (auxiliar.filsDroit).filsDroit ≠ nul faire
29:                         auxiliar ← auxiliar.filsDroit
30:                     fin tant que
31:                     proxracine ← auxiliar.filsDroit
32:                     proxracine.filsDroit ← racine.filsDroit
33:                     auxiliar.filsDroit ← proxracine.filsGauche
34:                     proxracine.filsGauche ← racine.filsGauche
35:                 fin si
36:             fin si
37:             supprimer racine
38:         fin si
39:     fin si
40:     renvoyer proxracine
41: fin Algorithme debut
```

Question 13 - lectureRepertoire

Pour lire la arbre dans a fichier, nous utilisons seulement la lecture d'un noeud en le fichier, et après ça nous utilisons la fonction *ajouterPersonneArbre* pour mettre le noeud dans la arbre.

La principaux difficulté ou complexité c'est sur la fonction ajouter, qui est déjà expliqué.

Algorithm 5 lectureRepertoireArbre

Entrée: string $nameFile$
Sortie: lien noeud $racine$

```
1:  Algorithme debut LECTUREREPERTOIREARBRE( )
2:      openFile(nameFile)
3:      si errorOpeningFile alors
4:          Ecrire("Nepeutpaslirelefichier")
5:      sinon
6:          n ← lireFile() {Lire la quantité de personnes}
7:          pour i ← 0 a n − 1 faire
8:              p.nom ← lireFile()
9:              p.prenom ← lireFile()
10:             p.numss ← lireFile()
11:             racine = ajouterArbre(racine, p)
12:         fin pour
13:     fin si
14:     renvoyer racine
15: fin Algorithme debut
```

6 Tests et conclusion

Comparaison entre les temps d'exécution

Table 1: Tableau des temps

Fonctions /Temps (s)	Tableau	Liste	Arbre
l'affichage	0,025202	0,102134	0,082211
la recherche	0,384166	0,373161	0,017844
la suppression	0,528774	0,352255	0,029143
lectureRepertoire	0,002340	3,941888	0,050309

Comparaison entre les Affichages

En ce qui concerne l'affichage, les temps d'exécution diffèrent de l'ordre de 10^{-2} secondes. En perspective avec la différence de temps d'exécution des autres fonctions, cette fonction n'a pas présenté de divergence très importante entre les trois types de stockage.

Comparaison entre les Recherches

Dans ce test, nous pouvons voir que le temps d'exécution de l'arbre est beaucoup plus court que les deux autres structures. En effet, dans l'arbre, chaque itération élimine la moitié des éléments de la recherche.

Comparaison entre les Suppressions

Pour la fonction de suppression, l'exécution la plus grande est Tableau car la suppression d'un élément nécessite le déplacement de tous les éléments suivantes. Le deuxième temps d'exécution le plus grande est la liste liée, car même si elle doit aller élément par élément, la suppression ne nécessite que le déplacement du pointeur. Pour l'arbre, nous trouvons le temps d'exécution le plus court. En effet, la recherche est binaire et la suppression ne nécessite que le déplacement du pointeur.

Comparaison entre les LecturesRepertoires

On constate que le temps d'exécution de la liste est plus grand que celui du tableau e de celui de l'arbre.

Cela est dû au fait de trouver le bon endroit pour insérer chaque élément dans la structure du stock. Pour la structure Tableau, seuls les index des éléments sont parcourus. Pour la liste liée, tous les éléments doivent être parcourus pour trouver le bon endroit. Pour l'arbre, une branche est sélectionnée (à gauche ou à droite) à chaque itération, ce qui réduit considérablement le temps d'exécution par rapport à la liste liée.