CS353 – Algorithmique
Grille de rendu du TP 2

Adrian Bonnet et Owen Rougé

```
struct Client * creerClient(int numeroTel, int nbAppel,int cout)
{
	client * liste = NULL;
	if ((liste = malloc(sizeof(client))) == NULL)  return NULL;
	liste->numero = numeroTel;
	liste->nbAppel = nbAppel;
	liste->prixAppel = cout;
	liste->gauche = NULL;
	liste->droite = NULL;
	return liste;
}

struct Client * chercher (struct Client * abr,int numeroTel)
{
	client * parcourt = NULL;
	parcourt = abr;

	if (parcourt == NULL)                                   return NULL;

	while (parcourt->numero != numeroTel)
	{
		if (parcourt->numero < numeroTel)
		{
			if (parcourt->droite == NULL)           return NULL;
			else                                    parcourt = parcourt->droite;
		}
		else if (parcourt->numero > numeroTel)
		{
			if (parcourt->gauche == NULL)           return NULL;
			else                                    parcourt = parcourt->gauche;
		}
	}
	return parcourt;
}
```

```
struct Client *inserer(struct Client ** abr, int numeroTel, int prixAppel)
{
   client * parcourt = NULL;
   client * insertion = NULL;
   parcourt = *abr;

        if( (insertion = chercher(parcourt,numeroTel) ) != NULL) {
                insertion->nbAppel += 1;
                insertion->prixAppel += prixAppel;
                return *abr;
        }

        if (parcourt == NULL)
        {
                if((*abr = malloc(sizeof(client))) == NULL)          return NULL;
                *abr = creerClient(numeroTel, 1, prixAppel);
                return *abr;
        }
        while (parcourt->numero != numeroTel)
        {
                if (parcourt->numero < numeroTel)
                {
                        if (parcourt->droite == NULL)
                        {
                                if((insertion = malloc(sizeof(client))) == NULL)    return NULL;
                                insertion = creerClient(numeroTel, 1, prixAppel);
                                parcourt->droite = insertion;
                                return *abr;
                        }
                        else                                    parcourt = parcourt->droite;
                }
                else if (parcourt->numero > numeroTel)
                {
                        if (parcourt->gauche == NULL)
                        {
                                if((insertion = malloc(sizeof(client))) == NULL)    return NULL;
                                insertion = creerClient(numeroTel, 1, prixAppel);
                                parcourt->gauche = insertion;
                                return *abr;
                        }
                        else                                    parcourt = parcourt->gauche;
                }
        }
        return NULL;
}
```

```
struct Client *supprimerClient(struct Client ** abr, int numeroTel)
{
	client * parcourt = NULL;
	client * suppression = NULL;
	client * temp = NULL;
	parcourt = *abr;

	if (parcourt == NULL)                                                    return NULL;

	//cas racine a supprimer
	if(parcourt->numero == numeroTel){
		if (parcourt->gauche == NULL) {
			suppression = *abr;
			*abr=parcourt->droite;                  // si la racine a un fils droite
			return suppression;
		}
		else if(parcourt->droite == NULL){
			suppression = *abr;
			*abr=parcourt->gauche;                  // si la racine a un fils gauche
			return suppression;
		}
		else{
			// si la racine a deux fils
			suppression = parcourt;
			parcourt = parcourt->droite;
			if(parcourt->gauche == NULL) {
				parcourt->gauche = suppression->gauche;
				*abr = parcourt;
				return suppression;
			}
			while(parcourt->gauche != NULL)            parcourt= parcourt->gauche;
			temp = parcourt->gauche;
			parcourt->gauche = temp->droite;
			temp->gauche = suppression->gauche;
			temp->droite = suppression->droite;
			*abr = temp;
			return suppression;
		}
	}
```

```
while (parcourt->numero != numeroTel)		// on cherche l'élément à supprimer
{
	if (parcourt->numero > numeroTel)
	{
		if (parcourt->gauche == NULL)		return NULL;
		else
		{
			suppression = parcourt;
			parcourt = parcourt->gauche;
		}
	}
	else if (parcourt->numero < numeroTel)
	{
		if (parcourt->droite == NULL)		return NULL;
		else
		{
			suppression = parcourt;
			parcourt = parcourt->droite;
		}
	}
}

// ici parcourt continent l'élément à supprimer et suppression contient son père

// si parcourt a un ou 0 fils
if ((parcourt->gauche == NULL) || (parcourt->droite == NULL))
{
	// si parcourt n'a qu'un fils à droite
	if (parcourt->gauche == NULL)
	{
		// si parcourt est à gauche de suppression
		if (suppression->numero < numeroTel)
			suppression->droite = parcourt->droite;
		// si parcourt est à droite de suppression
		else		suppression->gauche = parcourt->droite;
		return parcourt;
	}
	// si parcourt n'a qu'un fils à gauche
	else
	{
		// si parcourt est à gauche de suppression
		if (suppression->numero < numeroTel)
			suppression->droite = parcourt->gauche;
		// si parcourt est à droite de suppression
		else		suppression->gauche = parcourt->gauche;
		return parcourt;
	}
}
```

```
        // si parcourt a deux fils
        else
        {
                parcourt = parcourt->droite;
                if(parcourt->gauche == NULL) {
                        parcourt->gauche = suppression->droite->gauche;
                        temp = suppression->droite;
                        suppression->droite = parcourt;
                        return temp;
                }
                while(parcourt->gauche != NULL)         parcourt= parcourt->gauche;
                temp = parcourt->gauche;
                parcourt->gauche = temp->droite;
                parcourt = suppression->droite;
                temp->gauche = parcourt->gauche;
                temp->droite = parcourt->droite;
                suppression->droite = temp;
                return parcourt;
        }

}
```

```
void parcourirInfixe(struct Client * abr)
{
	if (abr != NULL)
	{
		if (abr->gauche != NULL)		parcourirInfixe(abr->gauche);
		printf("numero : %d \tcout : %d \tnbAppel : %d\n", abr->numero, abr->prixAppel,
abr->nbAppel);
		if (abr->droite != NULL)		parcourirInfixe(abr->droite);
	}
}

struct Client * creerArbre()
{
	client * liste = creerClient(15, 0, 0);					// couche 1
	liste->gauche = creerClient(12, 0, 0);					// couche 2
	liste->droite = creerClient(20, 0, 0);
	liste->gauche->gauche = creerClient(8, 0, 0);				// couche 3
	liste->gauche->droite = creerClient(14, 0, 0);
	liste->droite->gauche = creerClient(16, 0, 0);
	liste->droite->droite = creerClient(21, 0, 0);
	liste->gauche->gauche->droite = creerClient(10, 0, 0);			// couche 4
	liste->gauche->droite->gauche = creerClient(13, 0, 0);
	liste->droite->gauche->droite = creerClient(17, 0, 0);
	return liste;
}
```

```
int test ()                    // section de test de nos fonctions sur un arbre plus petit
{
        printf("------ SECTION TEST -------\n");

        client * listeTest=NULL;
        listeTest = creerArbre();

        printf("parcourt infixe de l'ABR cree\n");
        parcourirInfixe(listeTest);

        printf("\ntest de chercher\n");

        client * var = NULL;

        for (int j = 6; j <= 22 ; j ++)
        {
                var = chercher (listeTest, j);
                if (var == NULL)      printf("valeur %d non présente dans l'ABR\n", j);
                else                  printf("valeur %d trouvée : %d\n", j, var->numero);
        }

        printf("\ntest de inserer\n");

        inserer(&listeTest, 5, 0);
        inserer(&listeTest, 19, 0);
        inserer(&listeTest, 22, 0);
        printf("ajout de 5, 19 et 22\n");
        parcourirInfixe(listeTest);

        printf("\ntest de supprimer\n");

        supprimerClient(&listeTest, 10);
        supprimerClient(&listeTest, 14);
        supprimerClient(&listeTest, 21);
        supprimerClient(&listeTest, 20);
        printf("suppression de 10, 14, 21\n");
        parcourirInfixe(listeTest);

        printf("------ SECTION TEST -------\n");
        return 0;
}
```

Temps total d’exécution du main (déclaration des variables, affectation de l’arbre binaire de recherche, exécution de la facturation et de la suppression, exécution de la section de test)	
NBCLIENT 2000 NBLOGLINE 200000	**0,045 secondes**
NBCLIENT 20000 NBLOGLINE 2000000	**0,39 secondes**
NBCLIENT 200000 NBLOGLINE 20000000	**15,47 secondes**

Conclusion :

Pour les mêmes valeurs, la gestion de la mémoire par une liste chaînée était drastiquement moins efficace. La méthode de de l’arbre binaire de recherche réduit le temps d’exécution du dernier cas d’une heure approximative à 15 secondes.

La méthode utilisée possède la même complexité dans le pire de cas (qui s’apparente alors à une liste chaînée). Or cas particulier, l’amélioration est due au fait que la méthode utilisée réduit la distance à parcourir dans la structure de données pour trouver l’élément recherché.

La complexité O(h) est comprise entre O (log(n)) dans le meilleur des cas, et O(n) dans le pire des cas.