Victor MIGNOT
Pascal QUACH

# TP4

Arbres Binaire de Recherche

## Modification post-soutenance :

Après la soutenance et suivant votre conseil, nous avons après la soutenance apporté une modification au niveau de la création de l'arbre. Au lieu d'utiliser la fonction *malloc* pour allouer dynamiquement de la mémoire pour un pointeur vers l'arbre (donnant donc un double pointeur), nous avons ici fixé une variable ABR de type T_Arbre.

## Choix des structures et fonctions supplémentaires

Dans notre cas, nous avons gardé la structure de nœud traditionnelle (sans pointeur vers le nœud parent afin de respecter la consigne.

```
int validation_entree_int(int cond, int inf, int sup); // Structure basique pour choix
void afficher_infos_choix_menu(int choix_menu, const char * CHOIX_MENU_TEXTE[]); // Afficher les informations du sous-menu
int cle(T_Inter intervalle); // Renvoie la clé d'un noeud
T_Noeud * parcours(T_Inter intervalle, T_Noeud *noeud_parcours); // On renvoie FG ou FD selon l'intervalle
int droite_intervalle(T_Noeud* noeud); // Renvoie borne_sup d'un intervalle d'un noeud
int intervalle_chevauche(T_Noeud *noeud_1, T_Arbre *ABR); // On parcourt l'arbre pour vérifier qu'aucun n'intervalle ne se
    chevauche
void afficher_reservation(T_Noeud *noeud); // Affichage réservation
void afficher_date(T_Inter intervalle); // Affichage date "JJ/MM"
int nombre_de_fils(T_Noeud *noeud); // Renvoie le nombre de fils d'un noeud
T_Noeud* plus_proche_successeur(T_Arbre ABR); // Renvoie l'intervalle min de l'arbre (sous-arbre droit)
T_Noeud* plus_proche_predecesseur(T_Arbre ABR); // Renvoie l'intervalle max de l'arbre (sous-arbre gauche)
T_Noeud* recherche_pere(T_Noeud *ABR, T_Inter intervalle, int id_entreprise); // Renvoie le père du noeud recherché
```

*Figure 1- Liste des fonctions implémentées*

Dans le même principe, notre type **T_Inter** est une structure composée de deux entiers étant représentant la borne inférieure et la borne supérieure de notre intervalle.

Pour les fonctions supplémentaires que nous avons implémentées, voici une liste de leur prototype et leur utilité :

- ***int validation_entree_int(int cond, int inf, int sup)*** :
  Cette fonction permet de vérifier qu'un entier entré par l'utilisateur appartient bien à un intervalle précis. Cette fonction a été mise en place pour éviter du code redondant.

- ***void afficher_infos_choix_menu(int choix_menu, const char * CHOIX_MENU_TEXTE[])*** :
  Cette fonction permet d'afficher les infos des différents choix de menu. Elle a été choisie pour éviter le code redondant lors de l'affichage du menu.

- ***int cle(T_Inter intervalle) :***
  Retourne la borne inférieure de l'intervalle passé en paramètre. Utilisée pour éviter du code redondant.

- ***int droite_intervalle(T_Noeud* noeud)*** :
  Retourne la borne supérieure de l'intervalle du nœud passé en paramètre. Encore une fois, cette fonction a été implémentée pour éviter du code redondant.

- ***T_Noeud * parcours(T_Inter intervalle, T_Noeud *noeud_parcours)*** :
  Fonction permettant de parcourir l'arbre en fonction d'un intervalle donné. A chaque appel de cette fonction, le nœud du parcours passe à la hauteur inférieure. Ici, il s'agit d'une fonction utilisée pour éviter du code redondant.

- ***int intervalle_chevauche(T_Noeud *noeud_1, T_Arbre *ABR)*** :
  Cette fonction a été implémentée dans le but de déterminer si l'intervalle d'un nouveau nœud se chevauchait avec un intervalle déjà présent au sein d'un nœud de l'ABR. Ici, la fonction retourne une valeur servant de booléen afin de déterminer si l'on peut ajouter le nouveau nœud à l'ABR. Cette fonction a été implémentée dans le but de connaître facilement les différentes raisons de potentiels bugs dans le programme.

- ***void afficher_reservation(T_Noeud *noeud)*** :
  Formate les informations de la réservation pour les rendre lisibles à l'utilisateur. Cette fonction a été mise en place pour éviter le code redondant.

- ***void afficher_date(T_Inter intervalle)*** :
  Formate la borne supérieure et inférieure de l'intervalle pour la mettre au format de date traditionnel et ainsi la rendre facilement lisible pour l'utilisateur. Elle a été mis en place pour éviter le code redondant.

- ***int nombre_de_fils(T_Noeud *noeud)*** :
  Cette fonction sert à déterminer le nombre de fils qu'à un nœud au sein de l'ABR. Ce code était en effet redondant dans les fonctions principales. Ici, la fonction renvoie un entier correspondant au nombre de fils (à savoir 0, 1, 2).

- ***T_Noeud* plus_proche_successeur(T_Arbre ABR)*** :
  Sert à déterminer le plus petit intervalle du sous arbre droit de l'arbre passé en paramètre. Cette fonction sert principalement à décomposer la fonction de suppression de nœud qui est assez longue et complexe.

- ***T_Noeud* plus_proche_predecesseur(T_Arbre ABR)*** :
  Semblable à la fonction précédente, toutefois, celle si retourne le plus grand intervalle du sous arbre gauche de l'arbre passé en paramètre.

- ***T_Noeud* recherche_pere(T_Noeud *ABR, T_Inter intervalle, int id_entreprise)*** :
  Cette fonction a comme utilité de renvoyer le père du nœud recherché au sein de l'arbre binaire de recherche passé en paramètre de la fonction. Celle-ci retourne donc un pointeur vers le nœud père.

## Complexité des fonctions principales :

```
T_Noeud* creer_noeud(int id_entreprise, T_Inter intervalle);
```

*Figure 2 - Création d'un nœud*

La fonction ***creer_noeud*** n'est composé que d'un appel à la fonction ***malloc*** et d'affectations de valeur. Sa complexité dépend donc de la complexité de la fonction ***malloc.*** Si on considère que celle-ci a une complexité constante alors celle de la fonction est :

$$O(1)$$

```
void ajouter_noeud(T_Arbre* ABR, T_Noeud* noeud);
```

*Figure 3 - Ajout d'un nœud dans l'ABR*

La fonction ***ajouter_noeud*** parcourt les différentes profondeurs de l'arbre à l'aide d'une boucle itérative pour insérer le nœud en tant que feuille. On considère h la hauteur de l'arbre.

$$O(h)$$

```
T_Noeud* recherche(T_Arbre ABR, T_Inter intervalle, int id_entreprise);
```

*Figure 4 - Recherche d'un nœud dans l'ABR*

La fonction ***recherche*** parcourt aussi les différentes profondeurs de l'ABR selon la valeur de la clé à l'aide d'une boucle itérative. Ici, on considère p comme la hauteur profondeur du nœud recherché, celui-ci correspond à la hauteur h de l'arbre si le nœud est une feuille (dans le pire des cas p = h).

$$O(h)$$

```
void suppr_noeud(T_Arbre *ABR, T_Inter intervalle, int id_entreprise);
```

*Figure 5 - Suppression d'un nœud*

La fonction ***suppr_noeud*** est constitué de l'appel des fonctions ***recherche*** (en O(p) avec p la profondeur du nœud) et ***recherche_pere*** (en O(p)) et d'affectations. Encore une fois dans le pire des cas p = h. Nous tenons toutefois à signaler que notre fonction n'est pas totalement effective et ne fonctionne que pour des feuilles ou des nœuds à un seul fils.

$$O(h)$$

```
void modif_noeud(T_Arbre ABR, T_Inter intervalle, int id_entreprise, T_Inter nouv_intervalle);
```

*Figure 6 – Modification d'un nœud*

La fonction ***modif_noeud*** n'est constitué que d'appel à des fonctions précédemment citées en O(h) ou O(1). Ces fonctions ne sont pas récursives et il n'y a pas de structure itérative, seulement une structure conditionnelle.

$$O(h)$$

```
void affiche_abr(T_Arbre ABR);
```

*Figure 7 – Affichage de l'arbre*

La fonction ***affiche_abr*** est définie de manière récursive pour parcourir et afficher tous les nœuds de l'ABR. SI on considère n le nombre de nœud composant l'arbre.

$$O(n)$$

```
void affiche_entr(T_Arbre ABR, int id_entreprise);
```

*Figure 8 - Affichage des réservations d'une entreprise*

Comme la fonction précédente, ***affiche_entr*** est définie pas récursivité. Celle-ci parcourt tous les nœuds de l'ABR et n'affiche que les réservations de l'entreprise correspondante.

$$O(n)$$

```
void detruire_arbre (T_Arbre *ABR);
```

*Figure 9 - Libération de la mémoire allouée pour les nœuds de l'ABR*

La fonction ***detruire_arbre*** est aussi définie de manière récursive, celle-ci libère de nœuds en nœuds la mémoire qui leur est allouée selon un parcours postfixe. Il y a donc n appels imbriqués de la fonction.

$$O(n)$$