Institut Galilée - Université Paris 13

Structure de Données Avancées
Rapport

Travaux Pratiques - TP 03

Étudiants :
Tiziri OULD HADDA

Enseignants :
M. Olivier BODINI
M. Julien DAVID
M. Sergey DOVGAL

30 décembre 2019

Table des matières

1 Developpement des structures de données

1.1 Tas bianaire

Dans cette partie, j'ai implémenté dans le cas de ce TP un tas de type Tas min.

1.1.1 Choix et implémentation

Afin de coder dans une et même structure les deux types de tas binaires, j'ai placer un champs *is_fixed_size* dans celle ci afin de les différencier. Il peut valoir soit FALSE ou TRUE tel qu'elles ont été définies lors des précédents TP.

```
struct tas{
char is_fixed_size;
int size;
int capacity;
int* data;
};
```

le champs *size* et *capacity*, permettent de suivre l'état du tas et de les utiliser plus tard pour appliquer une stratégie de dimensionnement.

1.1.2 Fonctions de bases

Afin de faciliter l'accès au structures, j'ai rajouter les fonctions suivantes pour retrouver en fonction de l'indice le père et le fils gauche et droit du noeud.

```
int  tas_fils_gauche_noeud(tas* t, int i);
int  tas_fils_droit_noeud(tas* t, int i);
int  tas_parent_noeud(tas* t, int i);
```

La fonction qui suit positionne une clé dans le tas en partant d'une feuille vers la racine. Cette fonction est récursive et prend en argument uniquement le poiteur vers le tas et l'indice (position) du noeud a verifier et a mettre en place si besoin.

```
void tas_verifier_et_swaper_si_besoin_vers_le_haut(tas* t, int i);
```

La fonction suivante fait l'inverse de la précédente. Elle part d'un noeud i et parcours vers le bas (en direction des fils) en procédant à des permutation récusivement si besoin.

```
void tas_entasser(tas* t, int i);
```

1.1.3 Ajouter une clé

Ici on utilise le champs *is_fixed_size* pour savoir si on fait une extension ou pas. s'il s'agit d'un tas a taille fixe on remontera une erreur pour dire que le tas est plein. On commence par ajouter le nouvel élement à la fin puis on swap si besoin. jusqu'a respecter les régles d'un tas min.

```
char tas_ajouter_cle(tas* t, int valeur)
{
  char memory_allocation = FALSE;
  if(t->is_fixed_size == TRUE){
    if(t->size == t->capacity){
      printf("Tas complet\n");
      return memory_allocation;
    }
  }else{
    if(t->size == t->capacity){
      memory_allocation = TRUE;
      t->capacity *= 2;
      t->data = (int *) realloc(t->data, sizeof(int) * t->capacity);
    }
  }
  t->data[t->size] = valeur;
  tas_verifier_et_swaper_si_besoin_vers_le_haut(t, t->size);
  t->size++;
  return memory_allocation;
}
```

1.1.4 Exraire min

Pareil pour extraire, on verifie d'abort si besoin de faire un realloc. Pour extraire le min (racine), on commence par permuter la racine avec le dernier élement puis en entasse le premier jusqu'a ce qu'il trouve sa place.

```
char tas_extraire_min(tas* t){
  char memory_allocation = FALSE;
  int nbrElems = t->size;
  if(nbrElems==0){
    printf("le tableau est vide\n");
    return FALSE;
  }
  if(t->is_fixed_size == FALSE){
    if(nbrElems <= (t->capacity/4) ){
      t->capacity = t->capacity / 2;
      t->data = (int*) realloc(t->data, sizeof(int) * t->capacity);
      memory_allocation = TRUE;
    }
  }
  tas_permuter_noeuds(t, 0, nbrElems - 1);
  t->data[nbrElems - 1] = -1;
  tas_entasser(t, 0);
  t->size--;
  return memory_allocation;
}
```

1.2 Tas bianomial

L'implementation que j'ai réalisée dans ce TP3 est basée sur le livre "Introduction à l'algorithmique, Thomas Cormen...".

1.2.1 Choix et implémentation

Le tas binomial est défini par la structure suivante :

```
struct noeud_tas_binomial{
  struct noeud_tas_binomial* pere;
  struct noeud_tas_binomial* fils;
  struct noeud_tas_binomial* frere;
  int    cle;
  int    degre;
};
struct tas_binomial{
  struct noeud_tas_binomial * tete;
};
typedef struct tas_binomial tas_binomial;
typedef struct noeud_tas_binomial arbre_binomial;
```

Le noeud d'un tas binomial est un arbre binomial. Que ce soit l'arbre binomial ou le tas binomial, d'une manière indirect, ils sont créés par la fonction *union* du tas binomial. Nous définissont un arbre binomial dans notre implémentation comme étant à la fois le noeud du tas binomial et aussi la tete d'un tas binomial avec un seul noeud.

1.2.2 Fonction d'Ajout d'une clé

Dans cette fonction, on crée un nouvel tas binomial vide auquel on rajoute un noeud (B_0) contenant la clé à ajouter, puis, on fait une union entre ce tas et le tas t (destination). L'union consiste à faire la fusionner ses deux tas on reliant le derinier noeud du premier tas binomial à la tete du second. un nouveau tas est alloué en mémoire pointant sur la nouvelle chaine. les structures données en arguments sont supprimer (uniquement la strucuture "struct noeud_tas_binomial", pas l'espace mémoire pointé par têtes).

Une fois ces deux tas binomiaux fusionnés, on procède tel que décrit dans le livre à l'union on respectant chaque cas présenté dans le livre. Le pointeur vers nouveau tas binomial est retourné par la fonction "tas_binomial_inserer(...)", différent de l'adresse donnée en argument car celle ci a été libérée dans l'operation de fusion.

```
tas_binomial* tas_binomial_inserer(tas_binomial* t, struct noeud_tas_binomial* x){
  tas_binomial* t1 = tas_binomial_creer();
  x->pere = NULL;
  x->frere = NULL;
  x->fils = NULL;
  x->degre = 0;
  t1->tete = x;
  return tas_binomial_union(t, t1);
}
```

1.2.3 La fonction extraire min

Pour cette fonction, on supprimer le lien dans le tas du noeud ayant la clé de la racine la plus petite. le poiteur x_min sauvegarde son adresse et puis un nouveau tas est créé en inversant l'ordre des enfants de x_min. Puis on fait l'union avec ce nouveau tas.

```
struct noeud_tas_binomial* tas_binomial_extraire_min(tas_binomial** t){
  //trouver le min ET LE supprimer
  struct noeud_tas_binomial* x_min = tas_binomial_supprimer_min(*t);
  tas_binomial* t1 = tas_binomial_creer();
  //inverser l ordre chainee des enfants de x, et faire pointer tete[t1] sur la tete d
  struct noeud_tas_binomial* iter = x_min->fils;
  struct noeud_tas_binomial* avant_iter = NULL;
  struct noeud_tas_binomial* tmp;
  while(iter != NULL){
    tmp = iter->frere;
    iter->frere = avant_iter;
    //incrementer iterateur;
    avant_iter = iter;
    iter = tmp;
  }
  t1->tete = avant_iter;
  #if AFFICHAGE_EXT
  printf("\nT1 : tas extrait avec inversion ordre des fils\n");
  tas_binomial_afficher(t1);
  printf("\nT2 : tas principal sans l'element extrait\n");
  tas_binomial_afficher(*t);
  #endif
  *t = tas_binomial_union(*t, t1);
  #if AFFICHAGE_EXT
  printf("\nT  : Resultat de extraire min apres union\n");
  tas_binomial_afficher(*t);
  #endif
  return x_min;
}
```

1.2.4 Programme de test du tas binomial

les fonctions d'ajout et suppression de clé, ont été difficiles à implémenter. Pour trouver les causes des erreurs observé au cours du développement, j'ai rajouter des printf pour debuguer mon programme et fait beaucoup de tests. J'ai crée des define comme "AFFICHAGE_EXT" pour suivre ces fonctions dans toutes les étapes (une par une) afin de détecter la source de ces erreurs.

Programme de test

```
int main(int argc, char ** argv){
  printf("********** Programme de test arbre binomial ****************\n");
  tas_binomial* TasBinomial = tas_binomial_creer();
  int max = 7;
  printf("Nombre d'éléments à inserer : %d\n\n", max);
  for (int i = 1; i <= max; i++){
    printf("********** Insertion de la valeur %d dans le tas ****************\n", i
    TasBinomial = tas_binomial_inserer_cle(TasBinomial, i);
    tas_binomial_afficher(TasBinomial);
  }
  printf("********** Extraire min ****************\n");
  tas_binomial_extraire_min(&TasBinomial);
  printf("********** Resultat ****************\n");
  tas_binomial_afficher(TasBinomial);
  return EXIT_SUCCESS;
}
```

2 Analyse des expériences réalisées

Comme dans le TP1 et TP2, j'ai réalisé des script pour faciliter le test avec les expériences demandés dans le sujet du TP2, en variant aussi le paramètre p. On choisira $p > 1$ (=2 dans mes expérices) pour faire l'ajout uniquement, sinon $p = 0.5$ pour faire l'ajout et la suppression. J'ai rajouter un autre argument (entier) au main pour définir l'ordre d'insertion à chaque iteration qui vaut

- 0 : dans le cas d'un ordre croissant,
- 1 : pour l'ordre décroissant,
- 2 : pour l'ordre aléatoire.

```
#include "binary_binomial_heap.h"
#include "analyzer.h"
#define ORDRE_CROISSANT   0
#define ORDRE_DECROISSANT 1
#define ORDRE_ALEATOIRE   2
```

J'ai aussi rajouter des mesures pour les deux types de tas binaires et auusi pour le tas binomial, avec pour chacun son propre analyseur (*struct analyzer*).

```
  if(argc > 1){
    ordre = atoi(argv[1]);
    printf("Changement de l'ordre des valeur à inserer : %d\n",ordre);
  }
  ...
  for(i = 0; i < 1000000 ; i++){
    random = rand()/RAND_MAX;
    if(random < 0.5){
      if(ordre == ORDRE_DECROISSANT){
```

```
    valeur = 1000000 - i;
  }else if(ordre == ORDRE_ALEATOIRE){
    valeur = rand() % 1000000;
  }else {
    valeur = i;
  }
  // Ajout d'un élément et mesure du temps pris par l'opération.
  ...
 }else{
  // Suppression d'un élément et mesure du temps pris par l'opération.
  ...
```

Remarques :

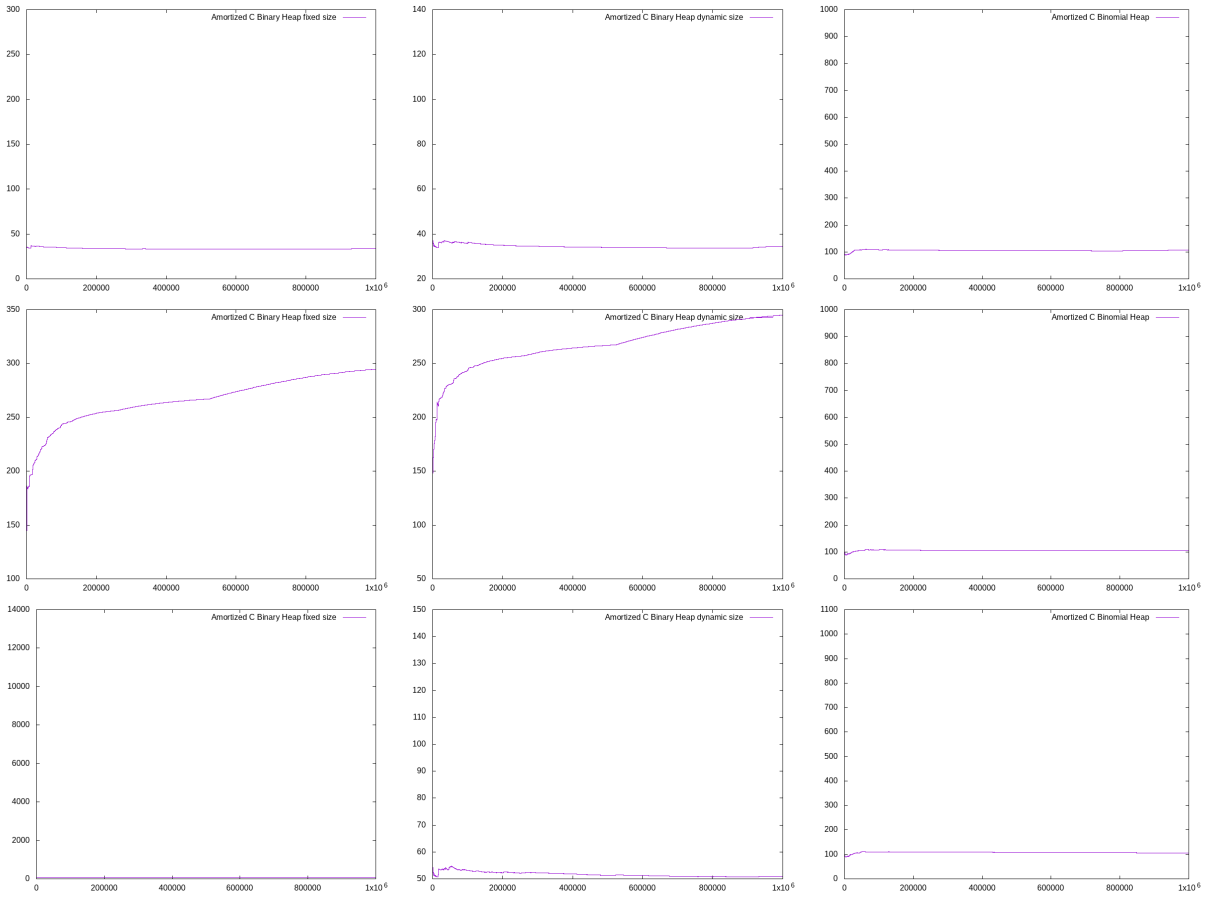

FIGURE 1 – Cout amorti - Ajout de valeurs croissantes(haut)/décroissant(milieu)/aleatoire(bas) dans un tas binaire a taille fixe (gauche),taille dynamique(milieu) et tas binomial(droite)

On remarque suite aux expériences effectuées que l'ajout de valeurs croissantes dans un tas binaire à un cout très faible par rapport à un ordre décroissant (pire cas). Ceci est du au faite que nous avons implémenter un tas binaire min. l'ajout de valeurs croissantes dans ce type de tas (contrairement à l'ajout de valeurs décroissantes) se fait directement de la racine au feuille sans aucune operation supplémentaire. Le tas binomial reste quand à lui uniforme par rapport à l'ajout et à la suppression, et ce, quel que soit l'odre des valeurs insérées dans celui ci. Dans ce type de structure, les noeud son alloués dynamiquement et

individuellement puis ils sont chainé avec les autres élements on jouant uniquement sur des pointeurs. Il n'a donc pas d'operation de reallocation mémoire ni de copie, ce qui fait le gain important de ce type de structure.

Dans les tas binomiaux, le cout amorti est plus important lors de l'ajout et la suppression quand le nombre d'élements n est petit, car c'est à ce moment la qu'on peut avoir beacoup d'opérations d'union (et aussi le nombre d'opération est petit donc cout amorti proche du cout réel).

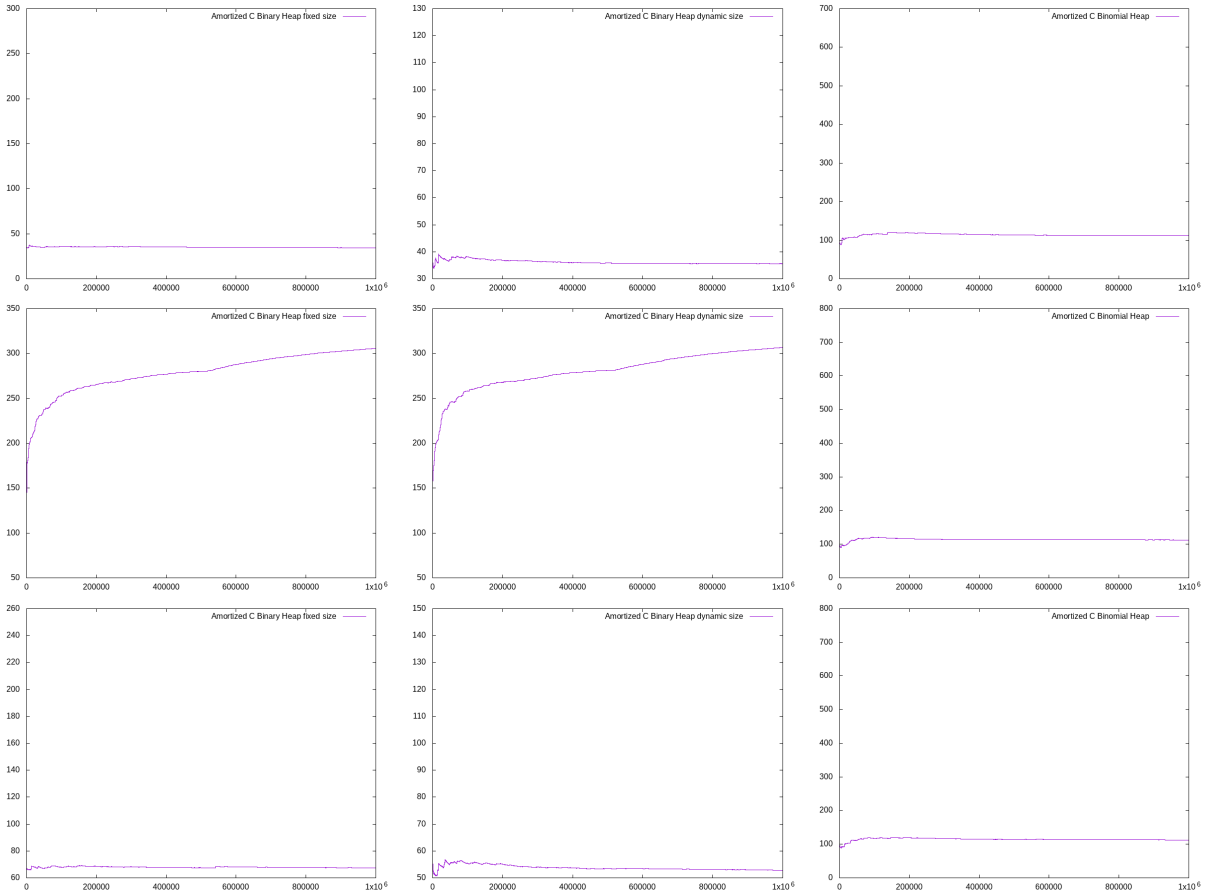

FIGURE 2 – Cout amorti - Suppression de valeurs croissantes(haut)/décroissant(milieu)/aleatoire(bas) dans un tas binaire a taille fixe (gauche),taille dynamique(milieu) et tas binomial(droite)

3 Etude théorique

3.1 Cout amorti incrément d'un compteur binaire

La fonction de potentielle pour l'incrémentation d'un compteur binaire est définie ://

- n = le nombre totale de bits.
- On suppose que la i^{me} opération réinitialisatio t_i bits
- On remarque que $\phi(i) <= \phi(i-1) - t_i + 1$
- Avec cette inégalité on se trouve devant deux cas de figures :
  - Si $\phi(i) = 0$, Alors : $\phi(i-1) = t_i$

— Si $\phi(i) > 0$, alors $\phi(i) = \phi(i-1) - t_i + 1$
— la différence de potentielle :
$\phi(i) - \phi(i-1) <= (\phi(i-1) - t_i + 1) - \phi(i-1) = -t_i + 1$
— le cout amorti de l'incrément d'un compteur binaire est :

$$c(i) = c(i) + \phi(i) - \phi(i-1) \quad (1)$$
$$= (t_i + 1) + (-t_i + 1) \quad (2)$$
$$c(i) = 2 \quad (3)$$

3.2 Cout amorti pour insertion dans un tas binomial

On imagine un tas comme étant représenté par un nombre binaire ou chaque bit i de ce nombre désigne l'existance ou pas de l'arbre binomial d'ordre i (B_i). L'union correspondera dans ce cas à faire une addition de cette représentation de chaque tas binaire.
0001011 : représente un tas : $B_0B_1B_3$
0000001 : représente un tas : B_0
0001100 : l'union des deux tas : B_2B_3

4 Conclusion

Chaque structure de données est utilisé dans un contexte particulier :
— Le Tas min (resp. max) doit être utilisé, à mon sens, de préférence dans un ordre croissant (resp. décroissant) de valeurs à l'insertion. On a vu dans le cas de valeurs aléatoire que on ne pert pas beaucoup dans le cout amorti.
— L'avantage du tas binomial est que le cout est constant quel que soit l'ordre de valeurs d'insertion. De plus, il n'a pas de gaspillage mémoire, comme le tas binaire, mais parcontre nécessite un malloc/free d'un noeud à chaque insertion/suppression.

5 ANNEXE

5.1 Résultat programme de test du tas binomial

```
********** Programme de test arbre binomial ****************
Nombre d'éléments à inserer : 7

********** Insertion de la valeur 1 dans le tas ****************
----------------------------------------
arbre {
 degre   : 0,
 cles    : 1 ,
}
----------------------------------------
********** Insertion de la valeur 2 dans le tas ****************
----------------------------------------
arbre {
 degre   : 1,
 cles    : 2 , 1 ,
}
----------------------------------------
********** Insertion de la valeur 3 dans le tas ****************
----------------------------------------
arbre {
 degre   : 1,
 cles    : 2 , 1 ,
}
arbre {
 degre   : 0,
 cles    : 3 ,
}
----------------------------------------
********** Insertion de la valeur 4 dans le tas ****************
----------------------------------------
arbre {
 degre   : 2,
 cles    : 4 , 2 , 3 , 1 ,
}
----------------------------------------
********** Insertion de la valeur 5 dans le tas ****************
----------------------------------------
arbre {
 degre   : 2,
 cles    : 4 , 2 , 3 , 1 ,
}
arbre {
 degre   : 0,
 cles    : 5 ,
```

```
}
----------------------------------------
********** Insertion de la valeur 6 dans le tas ****************
----------------------------------------
arbre {
 degre   : 2,
 cles    : 4 , 2 , 3 , 1 ,
}
arbre {
 degre   : 1,
 cles    : 6 , 5 ,
}
----------------------------------------
********** Insertion de la valeur 7 dans le tas ****************
----------------------------------------
arbre {
 degre   : 2,
 cles    : 4 , 2 , 3 , 1 ,
}
arbre {
 degre   : 1,
 cles    : 6 , 5 ,
}
arbre {
 degre   : 0,
 cles    : 7 ,
}
----------------------------------------
********** Extraire min ****************
********** Resultat ****************
----------------------------------------
arbre {
 degre   : 1,
 cles    : 4 , 3 ,
}
arbre {
 degre   : 2,
 cles    : 6 , 7 , 5 , 2 ,
}
----------------------------------------
```