Sorbonne Université

Devoir de Programmation

Algorithmique Avancée

François Malenfer (28706664), Danaël Carbonneau (28709878)

Implémentation de structures de données de recherche (en OCaml)

Enseignant : Antoine Genitrini

Master Informatique, Semestre 1, septembre - janvier 2023 - 2024

Table des matières

1 Échauffement

Le code correspondant à cette section se trouve dans le fichier *int128.ml*. En plus des fonctions demandées par le sujet, nous y avons ajouté d'autres fonctions utilitaires pour manipuler les entiers 128 dans la suite du projet :

of_str convertit une chaîne de caractères (au format des clés fournies dans le jeu de données aléatoires) en entier 128.

to_str convertit un entier 128 bits en une chaîne de caractères (sous le même format).

list_of_file permet de récupérer une liste d'entiers 128 bits depuis un fichier présentant des clés au bon format.

1.1 Représentation d'une clé 128 bits

OCaml nous donne accès au module Int32, qui permet d'avoir des entiers codés sur exactement 32 bits. Nous allons les utiliser dans un tuple de 4 entiers de taille 32 bits qui font la décomposition de notre entier 128 bits.

```
open Int32;;
type entier128 = (Int32.t * Int32.t * Int32.t * Int32.t);;
```

Pour implémenter nos prédicats, nous avons choisi d'écrire une fonction compare, qui compare des bits de poids forts vers ceux de poids faible les entiers 32 bits qui composent nos entier 128 bits. Ce choix nous permet de factoriser le code pour nos deux prédicats de comparaison.

```
let cmp (cle1 : t) (cle2 : t) : int =
  let (a1, b1, c1, d1) = cle1 and (a2, b2, c2, d2) = cle2 in
  if a1=a2 then
    if b1=b2 then
      if c1 = c2 then
        (Int32.unsigned_compare d1 d2)
      else
        (Int32.unsigned_compare c1 c2)
    else
      (Int32.unsigned_compare b1 b2)
  else
    (Int32.unsigned_compare a1 a2)
```

1.2 Le prédicat inf

Ce prédicat peut être implémenté en vérifiant si le résultat de compare est négatif. Nous avons également implémenté une fonction *inf2*, qui permet de manipuler des clés sous forme de type option.

```
let inf (cle1 : t) (cle2 : t) : bool = (cmp cle1 cle2) < 0
```

1.3 Le prédicat eg

De manière analogue, le prédicat peut être implémenté en vérifiant si le résultat de compare est nul.

```
let inf (cle1 : int128) (cle2 : int128) : bool = (cmp cle1 cle2) = 0
```

2 Structure 1 : Tas priorité min

Dans cette section, nous allons étudier deux manières d'implémenter des tas minimum : une utilisant une structure de tableau, ce qui est la manière la plus usuelle de représenter les tas minimum, et une utilisant une structure arborescente, permettant d'implémenter nos algorithmes dans un style purement fonctionnel. Le code se trouve dans les fichiers *tas_min_tab.ml* et *tas_min_arbre.ml*.

Nos structure sont les suivantes :

```
1 (*indice dernier element * taille du tableau * tableau*)
2 type heapArray = int ref * int ref * (Int128.t option) Array.t;;
3
4 (* Noeud of  rang * ndescendants * elt * fg * fd *)
5 type  heapTree = E | L of Int128.t | N of int * int *  Int128.t *  heapTree *
    heapTree;;
```

Le tas sous forme de tableau est représenté à l'aide de types option, ce qui permet de ne pas devoir le copier dans un tableau plus petit à chaque suppression, mais également de pouvoir l'agrandir d'un étage à chaque fois qu'on atteint la taille limite du tableau (ce qui permet d'éviter de trop faire cette couteuse opération de copie) : on rend possible le fait d'avoir des cases vides dans le tableau. Les deux entiers contenus dans la structure sont des références afin de rester cohérents avec la mutabilité du tableau (on veut pouvoir leur réaffecter de nouvelles valeurs). Dans le tableau, l'arbre est représenté en considérant le lien suivant entre les cases : L'indice d'un père, par rapport à son fils d'indice i est $(i-1)/2$, l'indice du fils droit d'un père i est $2*i+1$, celui de son fils gauche est $2*i+2$.

Pour le tas sous forme d'arbre, nous avons choisi de faire un constructeur feuille permettant de savoir facilement, dans nos match, lorsque nous arrivons au bout de l'arbre. Afin de pouvoir naviguer dans l'arbre, il nous est également nécessaire de retenir plusieurs informations sur chaque nœud afin de nous aiguiller lors des ajouts et des suppressions : le rang et le nombre de descendants. Cette manière d'implémenter la structure s'inspire de celle présentée dans *Purely functionnal data structures*, de Chris Okasaki [3] pour les *Leftist Heaps*, notamment pour la notion de rang, qui est la distance la plus courte d'un nœud vers un nœud vide (en nombre de nœuds).

Nos deux structures, en plus des fonctions de manipulations demandées par le sujet, sont munies d'une fonction *to_dot*, qui permet, grâce au langage dot, de visualiser sous forme d'arbre le résultat de nos opérations. Les différents arbres obtenus sont dans le dossier *Images/graphes*

2.1 Implémenter les 3 fonctions fondamentales d'un tas min

2.1.1 Tas min sous forme de tableau

Pour notre fonction *Ajout*, il suffit de faire une insertion à la fin du tableau grâce à l'indice maintenu (opération $O(1)$), puis de remonter dans l'arbre afin de faire des permutations tant que la clé du fils est plus petite que celle du père.

Pour *SupprMin*, on récupère le premier élément du tableau, qui est, par définition et construction du tas, le minimum dans la structure, puis on récupère l'élément se situant à la dernière case où un élément a été inséré, on le met dans la case d'indice 0, et on descend dans l'arbre en échangeant la clé courante avec celle de son fils ayant le plus petit élément (s'il est inférieur), puis en recommençant, si besoin, dans le fils où on a fait l'insertion.

Pour *AjoutsIteratifs* nous pouvons créer un tas vide de la taille de notre liste, puis y faire tous nos ajouts avec la fonction définie précédemment

2.1.2 Tas min sous forme d'arborescence

Pour notre fonction *Ajout*, on parcourt l'arbre à l'aide du rang pour trouver le prochain nœud vide en faisant le long du parcours les inversions de clés permettant de garder la propriété minimale du tas

Pour *SupprMin*, on parcourt l'arbre à l'aide du rang et du nombre de descendants pour trouver le dernier nœud ajouté dans le tas, losqu'on la trouvé, on fait remonter l'élément et on fait appel aux fonctions *reeq_tas_gauche* et *reeq_tas_droite* pour replacer correctement l'élément remonté (toujours le minimum du tas récupéré avec l'appel récursif) dans le tas afin de garder sa propriété minimale

Pour *AjoutsIteratifs*, il suffit de parcourir la liste et d'ajouter ses éléments uns par uns au tas (en commençant avec un tas vide.

2.2 Construction

Un algorithme permettant de construire un tas en temps constant a été présentée par l'informaticien Robert W. Floyd en 1964 dans une communication de l'Association for Computing Machinery[1], puis reprise, notamment, par Donald E. Knuth dans *The Art of Computer Programming, Vol. III Sorting and Searching*[2].

Algorithm 1 Heapify

Require: n un tas
  if n n'est pas une feuille et qu'un de ses fils est plus petit que la clé de n **then**
    f est le fils de n avant la clé la plus petite
    interchanger cle(f) et cle(n)
    heapify (f)
  end if

Algorithm 2 Construction

Require: l, une liste de clés
  t = transformation de l en une structure d'arbre parfait
  for k un nœud dans t en partant du dernier dans l'arbre parfait jusqu'à la racine **do**
    heapify (k)
  end for
  return Distances[n][m]

Le principe est donc de d'abord s'assurer de la structure de l'arbre globale (sous forme d'arbre parfait tassé à gauche), puis de partir des feuilles pour "heapify" (faire tas) les sous arbres qui composent le résultat final dans un parcours Bas Haut Droite Gauche.

La forme de cet algorithme s'adapte assez bien au style de programmation fonctionnel dans la mesure où les modifications sont locales à l'arbre étudié au moment où on le heapify.

2.2.1 Implémentation par tableau

Pour l'implémentation par tableau, il nous suffit de convertir la liste en tableau (par une primitive fournie par le module *Array*), puis de faire les remontées grâce aux indices depuis la fin de ce dernier.

2.2.2 Implémentation par arborescence

Pour l'arborescence, bien que les appels à heapify soient assez simples à situer (dès qu'on créé un nouvel arbre tassé à gauche, on le heapify, ce qui forme alors bien un tas, qu'on peut retourner), la question de créer un arbre parfait tassé à gauche depuis une liste est moins évidente.

Pour résoudre ce problème, nous utilisons le fait qu'à un nombre d'éléments donné, il n'y a qu'une seule forme d'arbre parfait tassé à gauche possible : il nous est possible donc, de savoir, pour un nœud donné, en fonction de la taille qui a été passée en argument, de savoir la taille de sa descendance droite et de sa descendance gauche : on peut alors faire deux appels récursifs demandant cette taille de tas pour obtenir deux fils aux tailles souhaitées : de là, il nous suffit de les combiner avec un nouvel élément tiré de la liste passée en argument, puis de faire appel à heapify sur notre nouveau nœud, pour obtenir un tas minimal de la taille souhaitée.

```
let rec make_tas (li : Int128.t list) (taille : int) :  (heapTree * Int128.t list)=
  if taille = 0 || taille < 0 then (E,li)
  else if taille = 1 then
    (*Cas d'arret : on veut faire un tas de taille 1, on renvoie une feuille du 1er element de la liste et le reste *)
  else
    let hauteur = log2 taille in
    let hauteur_prec = hauteur -1 in
    let reste = taille - ((two_pow hauteur)-1) in
    if reste < ((two_pow hauteur)/2) then
      let nb_elem_gauche = reste+ (((two_pow (hauteur_prec+1)) -1)/2) in
      let nb_elem_droite = (((two_pow (hauteur_prec+1)) -1)/2) in
      let (fg,lr) = make_tas li nb_elem_gauche in
      let (fd,lr2) = make_tas lr nb_elem_droite in
      match lr2 with
      | [] -> failwith "invalid argument"
      | h::tl -> let hp =  N( (min (rank fg) (rank fd)) +1, taille -1, h, fg, fd )
        in ( (heapify hp), tl)
    else
      let nb_elem_gauche = ((two_pow hauteur)/2) + (((two_pow (hauteur_prec+1)) -1)/2)
      in
      let nb_elem_droite = (((two_pow (hauteur_prec+1)) -1)/2) + (reste - ((two_pow hauteur)/2))
      in
      (*meme principe que dans l'autre cas, mais avec d'autres nombres d'elements *)
```

2.3 Union

Pour réaliser l'union en temps lineaire de deux tas, il suffit de mettre tous leurs éléments dans une même liste (ou un même tableau), puis de faire appel à construction sur cette liste nouvellement créée.

2.3.1 Implémentation par tableau

On fait la concaténation des deux tableaux avec la primitive du module *Array Array.append*, sur laquelle on reprend le même fonctionnement que pour construction (on remonte depuis les feuilles pour heapify les nœuds sur le chemin vers la racine).

2.3.2 Implémentation par arbre

Pour pouvoir utiliser notre fonction construction, il nous faut construire une liste en temps linéaire contenant tous les éléments des deux listes. Nous avons écrit, pour cela, une fonction *heap_to_list* qui permet de transformer un tas en une liste de ses clés en temps linéaire. Ainsi, pour faire l'union, on relie nos deux tas par un nœud "fantôme" (*N(0,0,(0l,0l,0l,0l))*) qui sera en tête de la liste obtenue par un appel à *heap_to_list* . Il suffit alors d'appeler *construction* sur la liste privée de cette tête.

2.4 Preuves des différentes complexités

2.5 Vérification graphique des complexités temporelles

2.6 Vérification graphique pour l'union

3 File binomiale

3.1 Primitives et structure

3.2 Fonctions fondamentales

3.3 Vérification graphique de la complexité de construction

3.4 Vérification graphique de la complexité de union

4 Hachage

5 Arbre Binaire de Recherche

6 Étude expérimentale

Références

[1] Robert W. Floyd. Algorithm 245 : Treesort. *Commun. ACM*, 7(12) :701, dec 1964.

[2] Donald E. Knuth. *The Art of Computer Programming, Volume 3 : (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[3] Chris Okasaki. *Purely Functionnal Data Structures*. Cambridge University Press, 1998. page 17.