Contents

1 Exo n°1

1.1 Question n°1

1.1.1 Implémentation

Un arbre sera donc composé :

- un tableau (tab)
- La taille du tableau.
- L'index/Valeur de la racine (sinon on pourrait le retrouver en parcourant le tableau à la recherche de la case contenant la valeur -1 (Soit une comp. temp. O(n) au pire cas, n étant la taille du tableau)).
- Les noeuds seront donc représentés par leur index.

1.1.2 CreerArbre O(1)

```
1 Arbre CreerArbre(int taille)
2 {
3   Arbre a = new Arbre();
4   a.tab = int[taille];
5   a.taille = taille;
6   a.racine = -1; //On ne sait pas encore l'index de la racine lors de l
     'initialisation de l'arbre
7   return a;
8 }
```

1.1.3 Racine O(1) vu qu'on stock son index

```
1   int Racine(Arbre a)
2 {
3   return a.racine;
4 }
```

1.1.4 FilsNoeud O(n)

```
1 Liste FilsNoeud(Arbre a,int indexNoeud)
2 {
3   Liste l = CreerListe();
4   for (int i = 0; i < a.taille; i ++)
5     {
6       if(a.tab[i] == indexNoeud)
7         {
8           l.insererTeteListe(i);
9         }
10    }
11  return l;
12 }
```

1.1.5 SontFreres O(1)

```
1 bool SontFreres(Arbre a, int noeud_x, int noeud_y)
2 {
3   return a.tab[noued_x] == a.tab[noued_y];
4 }
```

1.1.6 Hauteur O(n^2)

```
int Hauteur(Arbre a)
{
  int h = 0;
  for (int i = 0; i < a.taille; i++)
    {
      int noeudActuelle = i;
      int pere = a.tab[i];
      int h_actuelle = 0;
      while (noeudActuelle != Racine(a))
        {
          noeudActuelle = pere;
          pere = a.tab[pere];
          h_actuelle++;
        }
      if(h_actuelle > h)
        {
          h = h_actuelle;
        }
    }
  return h;
}
```

1.1.7 SousArbre $O(n^2)$

```
    Arbre SousArbre(Arbre a,int noeud_x)
  {
    Arbre b = CreerArbre(a.taille)

     for (int i = 0; i < a.taille; i++)
      {
        int noeudActuelle = i;
        int pere = a.tab[i];
        int sauv_pere = pere;
        while (noeudActuelle != noeud_x || noeudActuelle != racine(a))
          {
            noeudActuelle = pere;
            pere = a.tab[pere];
          }
      if(i == noeud_x)
        {
          b.tab[i] = -1;
        }
      else if(noeudActuelle == noeud_x)
        {
          b.tab[i] = sauv_pere;
        }
      }
return b
  }
```

1.2 Question n°2

Si on utilise cette nouvelle implémentation, on aura (n est la taille de l'arbre):

- toujours O(1) pour la création de l'arbre
- O(1) pour savoir si 2 noeuds sont frères (on compare juste les pères)
- O(i) pour la liste des fils : i étant le nbr de fils, pour cela il suffira de stocker dans une liste le premier fils du noeud, puis de récupérer le frère de son premier fils et ceux tant que frères n'est pas null.
- O(1) pour récupérer la racine (notre arbre sera composé que d'un unique noeud représentant la racine, il suffit donc de retourner cette variable)
- O(n) au pire cas –> O(h) pour la hauteur, h étant justement la hauteur de l'arbre, en effet il suffit d'incrémenter une variable tout en parcourant l'arbre de père en fils à l'aide d'une boucle while (la condition et si fils == null et dans ce cas la variable à comme valeur de base 0).
- O(n) au pire cas –> O(a) : a étant la taille du sous arbre souhaité. (On va devoir recopier dans un nouvelle arbre chaque noeud (qui seront eux aussi des copies pour éviter de modifier l'arbre original) avec une première boucle qui va récupérer chaque premier fils et pour chacun de ces noeuds, on va récupérer tous ses frères sans oublier le noeud x qui aura comme père -1).

2 Exo n°2

2.1 Question n°1

On a : -8 -3->8 -3->(8 7) -3->(8->9 7) -3->(4->(9 8) 7)

2.2 Question n°2

On pourrait utiliser un tableau si on connait la taille max atteignable de notre tas min, cependant à la différence de la table PERE ou la valeur des noeuds était l'index, ici notre tas min peut avoir plusieurs fois la même valeur et donc nous allons être obligé de stocker à la fois le pere mais aussi la valeur du noeud, mais cela nous permet de mettre n'importe quel noeud dans n'importe quel case (on pourrait aussi stocker filsG et filsD mais dans ce cas on se rapproche plus de la question suivante).

2.3 Question n°3

Vu qu'on ne connait pas la place prise par notre tas, utilisait un tableau statique serait une mauvaise idée vu que si notre tableau est trop petit, il faudra tout recopier dans un tableau plus grand (ce qui va prendre beaucoup de temps). Une meilleure solution serait de stocker dans le tas une seule Cellule (la racine), et chaque cellule à accès à son père, son filsG et son filsD (qui sont des cellules) et une valeur .

2.4 Question n°4

Type du dessus en rajoutant aussi le nombre de noeud (la taille) de la cellule et sa hauteur (pour bien avoir une complexité O(log(n)) pour l'insertion);

2.5 Question n°5

2.5.1 construire-tas O(1)

```
1  Tas construire-tas()
2 {
3   Tas t = new Tas();
4   t.racine.pere = -1;
5   return t;
6 }
```

2.5.2 inserer O(log(n))

```
1  Tas inserer(Tas t,T val)
2  {
3    Cellule a = t.racine;
4    while (FilsGauche(a) != null || FilsDroit(a) != null)
5      {
6        Cellule g = FilsGauche(a);
7        Cellule d = FilsDroit(a);
8
9        if((d.nbrNoeud == pow(2,d.hauteur+1)-1) && g.hauteur == d.hauteur
     )
10         {
11           a.nbrNoeud++;
12           a = FilsGauche(a);
13         }
14       else
15         {
16           a.nbrNoeud++;
17           a = FilsDroit(a);
18         }
19     }
20   a.nbrNoeud++;
21   a.hauteur++;
22   Cellule h = a;
23   while(h.pere != null)
24     {
25       if (h.pere.hauteur < h.hauteur +1)
26         {
27           h.pere.hauteur = h.hauteur+1;
28         }
29       h = h.pere;
30     }
31   if(FilsGauche(a) == null)
32     {
33
34       a.filsGauche = CreerCellule();
35       FilsGauche(a).valeur = val+1;
36       FilsGauche(a).pere = a;
37       remplacerCle(FilsGauche(a),val);
38     }
39   else
40     {
41       a.filsDroit = CreerCellule();
42       FilsDroit(a).valeur = val+1;
```

```
      FilsDroit(a).pere = a;
      remplacerCle(FilsDroit(a),val);
    }
  return t;
}
```

2.5.3 entasser-min O(h) = O(log(n))

```
Tas entasserMin (Tas t,cellule r)
{
  Cellule min;
  Cellule g = FilsGauche(r);
  Cellule d = FilsDroite(r);
  if (g != NULL && g.valeur <= r.valeur)
    {
      min = g;
    }
  else
    {
      min = r;
    }
  if (d != NULL and d.valeur <= min.valeur)
    {
      min = d;
    }
  if (min != r)
    {
      T tempo = min.valeur;
      min.valeur = r.valeur;
      r.valeur = tempo;
      entasserMin(t,min);
    }
  return t;
}
```

2.5.4 minimum O(1)

Juste retourner la valeur de la racine.

2.5.5 remplacerCle O(h(x)) ou O(h-h(x))

```
Tas remplacerCle(Tas t,Cellule r,T val)
{
  if(r.valeur < val)
    {
      r.valeur = val;
      entasserMin(t,r);
    }

  else
    {
      if (r.valeur > val)
```

```
        {
          r.valeur = r.pere.valeur;
          r.pere.valeur = val + 1;
          remplacerCle(r.pere,val);
        }
    }
return t;
}
```

2.5.6 retirerElement O(h)

```

retirerElement(Tas t,cellule r)
{
  Cellule c = t.racine;
  while (filsGauche(c) != null && filsDroite(c) != null) {

    if(FilsGauche(c).nbrNoeud == pow(2,FilsGauche(c).hauteur+1)-1)
      {
        c.nbrNoeud--;
        c = FilsDroite(c);
      }
    else
      {
        c.nbrNoeud--;
        c = FilsGauche(c);
      }
  }

  if(c == FilsGauche(Pere(c)))
    {
      Pere(c).hauteur--;
    }
  r.valeur = c.valeur;
  free(c);
  entasserMin(t,r);
}
```

2.6 Question n°6

Pour récupérer la maximum dans un tas min, il va falloir prendre le maximum entre toutes les feuilles, c'est à dire une complexité O(n).

2.7 Question n°7 O(n*log(n))

```
T[] tri(Tas t)
{
  n = t.racine.nbrNoeud;
  T[] = new T[n];
  for (int i = 0; i < n; i++)
    {
      t[i] = minimum(t);
```

```
      retirerElement(t,t.racine);
    }
  return t;
}
```

2.8 Question n°8

Les opérations de la file de priorité sont : insérer,maximum,extraire-max et remplacer, ce qui correspond respectivement avec un tas-max à insérer, maximum, retirerElement(t,t.racine), remplacerCle.

3 Exo n° 3

3.1 Question n° 1

Il est assez simple de représenter une expression arithmétique avec un arbre binaire, (surtout avec la notation polonaise –> (2*3) devient (* 2 3)) On va stocker dans la racine la dernière opérations à effectuer, dans son sous arbre gauche l'opération à effectuer avant celle ci et qui va donner l'opérante gauche, de même avec avec la partie droite pour l'opérante droite. Et ceux juqu'au feuilles ou on aura les constantes. Il nous suffira donc d'appeler récursivement depuis la racine et ceux jusqu'au opérations avant les feuilles une méthode **résolution**,puis remonter petit à petit, si on suit cette méthode il n'y a même pas besoin de spécifier les priorités.

3.2 Question n°2

On va utiliser un parcours en profondeur(suffixe). Si on utilise un parcours préfixe, ou infixe on ne va pas connaître l' opérantes de droite (infixe) voir les deux (préfixe). De même pour le parcours en longueur.

```
float eval(Arbre a)
{
  float g,d,retour;
  if(FilsGauche(a).valeur == *operations*)
    {
      g = eval(ArbreGauche(a));
    }
  else
    {
      g = FilsGauche(a).valeur;
    }
  if(FilsDroit(a).valeur == *operations*)
    {
      d = eval(ArbreDroit(a));
    }
  else
    {
      d = FilsGauche(a).valeur;
    }
  if(a.racine.valeur == '+')
    {
      retour = g+d;
    }
  return retour;
}
```

3.3 Question n°3

Ce sera le même principes avec au noeud soit True Ou false et comme noeud interne Soit OR, AND (2 fils) soit NOT (1 fils), la seule différence est que notre arbre binaire n'est pas forcément parfait ni presque parfait.

```
bool eval(Arbre a)
{
  bool g,d,retour;
  if(FilsGauche(a).valeur == *operations*)
    {
      g = eval(ArbreGauche(a));
    }
  else
    {
      g = FilsGauche(a).valeur;
    }
  if(FilsDroit(a) != null)
    {
      if( FilsDroit(a).valeur == *operations*)
        {
          d = eval(ArbreDroit(a));
        }
      else
        {
          d = FilsGauche(a).valeur;
        }
    }
  if(a.racine.valeur == "AND")
    {
      retour = g && d;
    }
  if(a.racine.valeur == "NOT")
    {
      retour = not g;
    }
  return retour;
}
```

4 Exo n°4

4.1 Question n°1

Récurrence :::

Initialisation ::: hauteur 0, c'est à dire que x = racine donc il ne peut y avoir qu'un chemin (x -> x)

Hérédité ::: Supposons que la propriété est vrai pour tout x de hauteur <= n, c'est à dire qu'il n'existe qu'un seul chemin entre x et la racine r que l'on appellera chm. Prenons x_1 fils de x, x_1 a donc de hauteur n+1, Prouvons qu'il n'existe qu'un seul chemin entre x_1 et r. La seule direction possible pour x_1 allant vers la racine est de passer par son père x, ensuite il ne pourra prendre que le chemin uniqur chm, donc proposition vérifié pour x_1.

4.2 Question n°2

Récurrence :::

Initialisation ::: noeud 1, c'est à dire qu'il n'y a que la racine, qui est une feuille.

Hérédité ::: Supposons que la propriété est vrai pour tout arbre ayant <= n noeud, c'est à dire qu'il y a au moins une feuille dans l'arbre T que l'on va notée f. Ajoutons lui un nouveau noeud x, Prouvons qu'il la propriété est toujours vérifier. Si x est fils de d, alors f ne sera plus feuille mais x deviendra feuille puisque qu'il n'a pas d'enfants. Sinon f restera feuille.

4.3 Question n°3

Récurrence :::

Initialisation ::: noeud 1, c'est à dire qu'il n'y a que la racine r On a donc V = {r} et E = $\emptyset$ => |V| -1 = |E|

Hérédité ::: Supposons que la propriété est vrai pour tout arbre ayant <= n noeud. Ajoutons lui un nouveau noeud x, Prouvons que la propriété est toujours vérifier. Notons p le père de x, V' et E' l'ensemble et relations binaires avant l'ajout de x On à donc maintenant $V = V' \bigcup x$ et $E = E' \bigcup (p, x)$. Donc |V| = |V'| +1 et |E| = |E'| +1 (|V| = |E|).

4.4 Question n°4

Récurrence :::

Initialisation ::: hauteur = 0, c'est à dire qu'il y a 1 feuille pour 1 noeud (1 = 2*1-1)

Hérédité ::: Supposons que la propriété est vrai pour tout arbre ayant une hauteur <= h. Prouvons que la propriété est toujours vérifier pour tout arbre binaire de hauteur > h. Soit T l'arbre de hauteur h+1 et soit f le nbr de feuilles de l'arbre de hauteur h. On à donc maintenant 2*f nbr de feuilles (noté f_1) (vu que chaque noeud précédent à maintenant 2 fils) et comme noeuds totale 2*f-1 + f_1. Soit nbr de noeuds = f_1 -1 +f_1 => 2*$f_1$$-1

4.5 Question n°5

Récurrence :::

Initialisation ::: hauteur = 0, c'est à dire qu'il y a qu'une feuille (la racine) ($1 <= 2^0 = 1$)

Hérédité ::: Supposons que la propriété est vrai pour tout arbre ayant une hauteur <= h. Prouvons que la propriété est toujours vérifier pour tout arbre binaire de hauteur > h. Soit T l'arbre de hauteur h+1 et soit f le nbr de feuilles de l'arbre de hauteur h. Le nbr de feuilles f_1 de l'arbre T sera au maximum f*2 (puisque l'arbre est binaires) Donc f $\leq 2^h$ ==> 2*f $\leq 2^{h+1}$ ==> $f_1 \leq 2^{hauteur(T)}$

5 Exo n°5

5.1 Question n°1

5.1.1 12345678 et 53247681 :

- L'arbre binaire n'existe pas

5.1.2 12345678 et 432527861

- L'arbre binaire existe (le parcours en longueur donne : 1/ 2 6/ 3 5 7 8/4)

5.2 Question n°2

5.2.1 Construction Arbre :

- J'ai pris la notation de python pour obtenir des sous-String, sinon il faudrait créer une fonction qui renvoie une sous string en fonction de l'index de départ et d'arrivé et sauvegarder la taille à chaque fois.
- FindIndexDans(String,char) va renvoyer la position du char dans le String en parcourant le string

- Dans cette exo un noeud est représenté par un entier valeur et par 2 noeuds (filsG et filsD), pas besoin du père ici.
- Un arbre est donc seulement un noeud que l'on appelle racine.

```
1  Arbre CreationRacine(String pre,String post)
2  {
3  Arbre a = CreerArbre();
4  a.racine.val = pre[0];
5  CreationArbreInterne(a.racine,pre[1:],post[:-1])
6  return a;
7  }
8
9  void  CreationArbeInterne(Noeud n,String pre,String post)
10 {
11  if(taille(pre) > 1)
12    {
13      g = CreerNoeud();
14      g.val = pre[0];
15      n.filsG = g;
16      d = CreerNoeud();
17      d.val = post[-1];
18      n.filsD = d;
19
20      int posInterPost = FindIndexDans(post,pre[0]);
21      int posInterPre = FindIndexDans(pre,post[-1]);
22
23      CreationArbreInterne(g,pre[1:postInterPre],post[0:posInterPost]);
24      CreationArbreInterne(d,pre[postInterPre+1:],post[postInerPost
    +1:-1]);
25    }
26  else if (taille(pre)> 0)
27    {
28      g = CreerNoeud();
29      g.val = pre[0];
30      n.filsG = g;
31    }
32 }
```

on aurait pu avoir un probleme si notre arbre n'est pas binaire, en effet si on avait eu 12345678 et 432528761, on aurait du avoir 6 –>7 –> 8 et non pas. Il faudrait donc un peu modifier l'algo