Algorithmique et Programmation 3

→
_____ *

Convention à adopter :

- les formules de type mathématique on les notera en *italique*;
- les termes à définir seront précédés d'une étoile et écrit en **gras** + surligné (***Comme ça**)
- astuce :
- Ctrl + **B** (mettre en gras)
- Ctrl + U (mettre en surlignage)
- Ctrl + *I* (mettre en italique)
- les mot-clefs seront écrits en **gras (Procédure, Début, Si, Pour, Fin, etc...)**
- les commentaires pour décrire la fonction/procédure seront soulignés aussi (et oui fallait être pionnier)
- Pour les titres; No Séance (Titre1) / Titre partie (I, II, III)(Titre2)
- symboles: ↓, ↑, ↕, ←

_____
__*

// afficher en prefix un arbre "arb"
procedure prefix (↓ arb : arbre [E])
debut

- **si** non EstVide(arb) **alors**
- *afficher*(Racine(arb))
- prefix(FG(arb))
- prefix(FD(arb))

fsi
fin

// afficher en postfix un arbre "arb"
procedure postfix (↓ arb : arbre [E])
debut

- **si** non EstVide(arb) **alors**
- postfix(FG(arb))
- postfix(FD(arb))
- *afficher*(Racine(arb))

fsi
fin

C(N) = 1 + C(Nfg) + C(Nfd) + 1
= 2 + C(Nfgfg) + C(Nfdfg) + C(Nfgfd) + C(Nfdfd) = N

Nfg+Nfd = N
Nfgfg + Nfdfg + Nfgfd + Nfdfd = N
...
1+1+...+1=N

fonction estComplet(↓ arb : arbre [E]) : booléen
debut
- res ← vrai
- **si** EstVide(arb) **alors**
- res ← vrai
- **sinon** // non Estvide(arb)
- diff ← hauteur(FG(arb)) - hauteur(FD(arb))
- cfg ← estComplet(FG(arb))
- cfd ← estComplet(FD(arb))
- res ← cfg **et** cfd **et** diff=0

fsi
- retourner res

fin

fonction estComplet(↓ arb : arbre [E]) : booléen
debut
- res ← vrai
- **si** non EstVide(arb) **alors**
- meme_hauteur ← hauteur(FG(arb)) = hauteur(FD(arb))
- cfg ← estComplet(FG(arb))

- cfd ← estComplet(FD(arb))
- res ← cfg **et** cfd **et** meme_hauteur

fsi
- retourner res

fin

C(N) = 1 +1 + C(Nfg) + C(Nfd) + Nfg + Nfd

fonction estComplet(↓ arb : arbre [E]) : booléen
debut
- res ← vrai
- **si** non EstVide(arb) **alors**
- res ← estComplet(FG(arb)) **et** estComplet(FD(arb)) **et** (hauteur(FG(arb)) = hauteur(FD(arb)))

fsi
- retourner res

fin

Devoir: l'algo de estComplet en O(N)

ABR
======

fonction chercher (↓ val : entier; ↓ arb : Arbre [E]) : booléen
début
- res ← faux
- **si** EstVide(arb) **alors**
- res ← faux
- **sinon** // non Estvide(arb)
- **si** val = Racine(arb) **alors**
- res ← vrai
- **sinon**
- res ← chercher(FG(arb))
- **si** non res **alors** // res = faux
- res ← chercher(FD(arb))
- **fsi**

- **fsi**

fsi
- retourner res

fin

O(H) = O(N)

// chercher ***val*** dans arbre binaire de recherche ***arb***
fonction chercher (↓ val : entier; ↓ arb : ABR [E]) : booléen
début
- res ← faux
- **si** EstVide(arb) **alors**
- res ← faux
- **sinon** **//** non Estvide(arb)
- **si** val = Racine(arb) **alors**
- res ← vrai
- **sinon** // val < Racine(arb) **ou** val > Racine(arb)
- **si** val < Racine(arb) **alors**
- res ← chercher(FG(arb))

  sinon // val > Racine(arb)
- res ← chercher(FD(arb))
- **fsi**
- **fsi**

fsi
- retourner res

fin

// chercher le minimum dans arbre binaire de recherche ***arb*** pas vide
fonction min (↓ arb : ABR [entier]) : entier
début
- **si** EstVide(FG(arb)) **alors**
- res ← Racine(arb)
- **sinon**

- res ← min(FG(arb))

fsi

- retourner res

fin

// chercher le maximum dans arbre binaire de recherche ***arb*** pas vide
fonction max (↓ arb : ABR [entier]) : entier
début

- **si** EstVide(FD(arb)) **alors**
- res ← Racine(arb)
- **sinon**
- res ← max(FD(arb))

fsi

- retourner res

fin

// chercher le minimum dans arbre binaire de recherche ***arb*** pas vide (en iteratif)
fonction minIt (↓ arb : ABR [entier]) : entier
début

- **tant que** non EstVide(FG(arb)) **faire**
- arb ← FG(arb)
- **ftq**
- retourner Racine(arb)

fin

// inserer ***val*** dans arbre binaire de recherche ***arb***
procedure inserer (↓ val : entier; ↕ arb : ABR [E])
début

- **si** EstVide(arb) **alors**
- arb ← Cons(val, ConsVide(), ConsVide())
- **sinon**

si val < Racine(arb) **alors**
fg ← FG(arb)
inserer(val, fg)
ModifierFG(arb, fg)

- // inserer(val, FG(arb))
- **sinon** // val >= Racine(arb)

- // inserer(val, FD(arb))
- fd ← FD(arb)
- inserer(val, fd)
- ModifierFD(arb, fd)

fsi
fsi
fin

// **procedure** modifierFG(↕ arb:Arbre, ↓ fg:Arbre)
```
void modifierFG(tree arb, tree fg){
   arb -> fg = fg;
}// racine(arb) = 5
```

// procedure inserer (↓ val : entier; ↕ arb : ABR [E])
```
void changerVal(int x, int nv){
   x = nv;
}
void main(){
  n = 3;
  changerVal(n,10);
  // n = 3
}
```

```
void inserer(int val, tree* arb){
     if( isEmpty(arb) ){
         arb = cons( val, consVide(), consVide())
     } else {
        ...
     }
}
void main(){
  n = consVide();
  inserer(10, &n);
  // n = ?
}
```

// supprimer ***val*** dans arbre binaire de recherche ***arb***
procedure supprimer (↓ val : entier; ↕ arb : ABR [E])
début
- **si** non EstVide(arb) **alors**
- **si** val < Racine(arb) **alors**
- //supprimer(val, FG(arb))

```
        fg ← FG(arb)
        supprimer(val, fg)
        ModifierFG(arb, fg)
sinon // val >= Racine(arb)
        si val > Racine(arb) alors
            // supprimer(val, FD(arb) )
            fd ← FD(arb)
            supprimer(val, fd)
            ModifierFD(arb, fd)
        sinon // val = Racine(arb)
            si EstVide(FG(arb)) et  EstVide(FD(arb))  alors
                Liberer(arb)
            sinon // fg=/=0 ou fd =/=0
                si EstVide(FD(arb))  alors
                    fg ← FG(arb)
                    ModifFG(arb, ConsVide())
                    Liberer(arb)
                    arb ← fg
                sinon // fd=/=0
                    si EstVide(FG(arb))  alors
                        fd ← FD(arb)
                        ModifFD(arb, ConsVide())
                        Liberer(arb)
                        arb ← fd
                    sinon // fg=/=0 et fd =/=0
                        min ← supprimerMin( FD(arb) )
                        ModifRacine( arb, min)
                    fsi
                fsi
            fsi
        fsi
fsi
```

fin

// supprimer le min dans arbre binaire de recherche ***arb***
fonction supprimerMin (↕ arb : ABR [E]) : entier
début

- **si** EstVide(FG(arb)) **alors**
- min ← Racine(arb)
- temp ← arb
- arb ← FD(arb)
- ModifFD(temp, ConsVide())
- Liberer(temp)
- **sinon**
- fg ← FG(arb)
- min ← supprimerMin(fg)
- ModifFG(arb, fg)
- **fsi**
- retourner min

fin

inserer: O(H) = O(N)
chercher: O(H)
supprimer: O(H)
supprimerMin: O(H)

procedure rotationDroite(↕ abr: Arbre[E])
debut
fg ← FG(arb)
ModifFG(arb, FD(fg))
ModifFD(fg, arb)
arb ← fg
fin

procedure rotationGauche(↕ abr: Arbre[E])
debut
fd ← FD(arb)
ModifFD(arb, FG(fd))

```
   ModifFG(fd, arb)
   arb ← fd
fin
```

O(1)

```
// inserer val dans AVL arb
procedure inserer (↓ val : entier; ↕ arb : AVL [E])
début
    • si EstVide(arb) alors
    •  arb ← Cons( val, ConsVide(), ConsVide() )
    • sinon
       si  val < Racine(arb) alors
              fg ← FG(arb)
              inserer(val, fg)
              ModifierFG(arb, fg)
    • // inserer( val, FG(arb) )
    • sinon //  val >= Racine(arb)
    • // inserer( val, FD(arb) )
    • fd ← FD(arb)
    • inserer(val, fd)
    • ModifierFD(arb, fd)
       fsi
    fsi

    fe ← hauteur(FG(arb)) - hauteur(FD(arb))  // dans un AVL, |fe| <= 1

    si fe >1 et val < Racine(FG(arb)) alors // LL
       rotationDroite(arb)
    fsi
    si fe >1 et val >= Racine(FG(arb)) alors // LR
       // rotationGauche(FG(arb))
       fg ← FG(arb)
       rotationGauche(fg)
       ModifFG(arb,fg)
       rotationDroite(arb)
    fsi
    si fe <-1 et val > Racine(FD(arb)) alors // RR
       rotationGauche(arb)
    fsi
    si fe <-1 et val <= Racine(FD(arb)) alors // RR
       // rotationDroite(FD(arb))
       fd ← FD(arb)
```

```
        rotationDroite(fg)
        ModifFD(arb,fd)
        rotationGauche(arb)
    fsi
fin
```

O(log N)

HEAP / Tas

```
// construire le tableau tab a partir de l'arbre arb.
// a partir de la position pos dans tab
// retourner la taille de l'arbre (c.a.d. la derniere case de tab)
fonction arbre2tab(↓ arb:Arbre[E], ↕ tab:tableau[E]), pos:entier ) : entier
debut
  res ← pos
  si non EstVide(arb) alors
     tab[pos] ← Racine(arb)
     si non EstVide(FG(arb)) alors
        res ← arbre2tab(FG(arb), tab, pos*2+1)
     fsi
     si non EstVide(FD(arb)) alors 2 * pos +1
        res ← arbre2tab(FD(arb), tab, pos*2+2)
     fsi
  fsi
  returner res
fin
```

```
// construire un arbre binaire arb en utilisant le tableau tab a partir de la position pos
fonction tab2arbre(↓ tab:tableau[E]), N:entier, pos:entier ) : Arbre[E]
debut
  arb ← ConsVide()
  si  pos < n  alors
      arb ← Cons(tab[pos],
                 tab2arbre(tab, N, 2*pos+1),
                 tab2arbre(tab, N, 2*pos+2) )
  fsi
  retourner arb
fin
```

```
fonction max(tas:tableau entiers[N], N:entier, taille:entier) : entier
debut
  retourne tas[0]
fin
// O(1)
```

```
// fonction qui retourne le max et le supprime
fonction supprimerMax(↕ tas:tableau entiers[N], N:entier, ↕ taille:entier) : entier
debut
  max ← tas[0]
  tas[0] ← tas[taille-1]
  taille ← taille-1
  swapDown(tas, N, taille, 0)
  retourne max
fin
// O(log N)


procedure swapDown(↕ tas:tableau entiers[N], N:entier, ↕ taille:entier, pos:entier)
debut
  si pos < taille/2 alors  // pas feuille
     fg ← 2*pos + 1
     fd ← 2*pos + 2
     si fd < taille alors  // fd pas vide (donc fg pas vide)
        si tas[fg] > tas[fd] alors
           posMax ← fg
        sinon
           posMax ← fd
        fsi
     sinon // fd vide (fg pas vide)
        posMax ← fg
     fsi
     si tab[pos] < tab[posMax] alors
        temp ← tab[pos]
        tab[pos] ← tab[posMax]
        tab[posMax] ← temp
        swapDown(tas, N, taille, posMax)
     fsi
  fsi
fin


procedure inserer(↕ tas:tableau entiers[N], N:entier, ↕ taille:entier, val:entier)
debut
    tas[taille] ← val
    taille ← taille+1
    siftUp(tas, N, taille, taille-1)
fin




procedure siftUp(↕ tas:tableau entiers[N], N:entier, ↕ taille:entier, pos:entier)
debut
  si pos>0 alors  // en dessous de la racine
     parent ← (pos-1)/2
     si tab[pos] > tab[parent] alors
        temp ← tab[pos]
```

```
            tab[pos] ← tab[parent]
            tab[parent] ← temp
            siftUp(tas, N, taille, parent)
        fsi
    fsi
fin
```

```
procedure heapSort(↕ tas:tableau entiers[N], N:entier)
debut
    pour taille de N à 1 faire
        temp ← supprimerMax(tas, N, taille)
        tas[taille-1] ← temp
    fsi
fin
```

```
procedure heapSort(↕ tas:tableau entiers[N], N:entier)
debut
    tab2heap(tas, N)
    taille ← N
    pour i de 0 à N-1 faire
        temp ← supprimerMax(tas, N, taille)
        tas[taille-1] ← temp
        taille ← taille-1
    fsi
fin
```

```
procedure tab2heap(↕ tas:tableau entiers[N], N:entier)
debut
    pour i decroissant N/2-1 à 0 faire
        swapDown(tas, N, N, i)
    fsi
fin
```

```
procedure qs(↕ tab:tableau entiers[N], N:entier)
debut
    quicksort(tab,N,0,N-1)
fin
```

```
procedure quicksort(↕ tab:tableau entiers[N], N:entier, deb:entier, fin:entier)
debut
            pivot ← partager(tab,N,deb,fin)
            quicksort(tab,N, deb, pivot)
            quicksort(tab,N, pivot+1, fin)
fin
```

```
procedure partager(↕ tab:tableau entiers[N], N:entier, deb:entier, fin:entier)
debut
        val ← tab[deb]
        gauche ← deb-1
        droite ← fin+1
        tant que gauche < droite faire
                gauche ← gauche+1
                tant que tab[gauche] < val faire
                        gauche ← gauche+1
                ftq
                droite ← droite-1
                tant que tab[droite] >= val faire
                        droite ← droite-1
                ftq
                si gauche < droite faire
                   // swap(tab,gauche,droite)
                   temp ← tab[gauche]
                   tab[gauche] ← tab[droite]
                   tab[droite] ← temp
                fsi
        ftq
        retourner droite
fin
```