- Examen -

15 janvier 2021

Attention, il s'agit d'un examen d'algorithmique donc pas de langage C
Le barème est donné à titre indicatif

Vos réponses sont à rendre sur 3 copies distinctes:
- Copie 1: sujets **1**
- Copie 2: sujet **2.1** et **2.2**
- Copie 3: sujet **2.3** et **2.4**

1.Complexité (4 points)

On veut écrire l'algorithme de la fonction qui prend en paramètre un tableau trié d'entiers et retourne une des postions contenant 0 (on suppose qu'il y en a au moins une). Par exemple, pour le tableau

-19 -13 -10 -3 0 2 3 10 11 16

le résultat est 4 et pour le tableau

-18 -16 -14 -5 -5 -3 0 0 2 10

le résultat est 6 ou 7.

L'algorithme suivant peut être utilisé pour résoudre le problème:

```
procédure trouver_zero(tab : tableau d'entiers, size : entier)
variables
   d,f,m,pos : entier
début
  d ← 0
  f ← size-1
  pos ← -1
  tant que pos=-1 faire
    m ← (d+f)/2
    si tab[m] = 0 alors
      pos ← m
    sinon
      si tab[m] < 0 alors
        d ← m+1
      sinon
        f ← m-1
      fsi
    fsi
  ftq
  retourne pos
fin
```

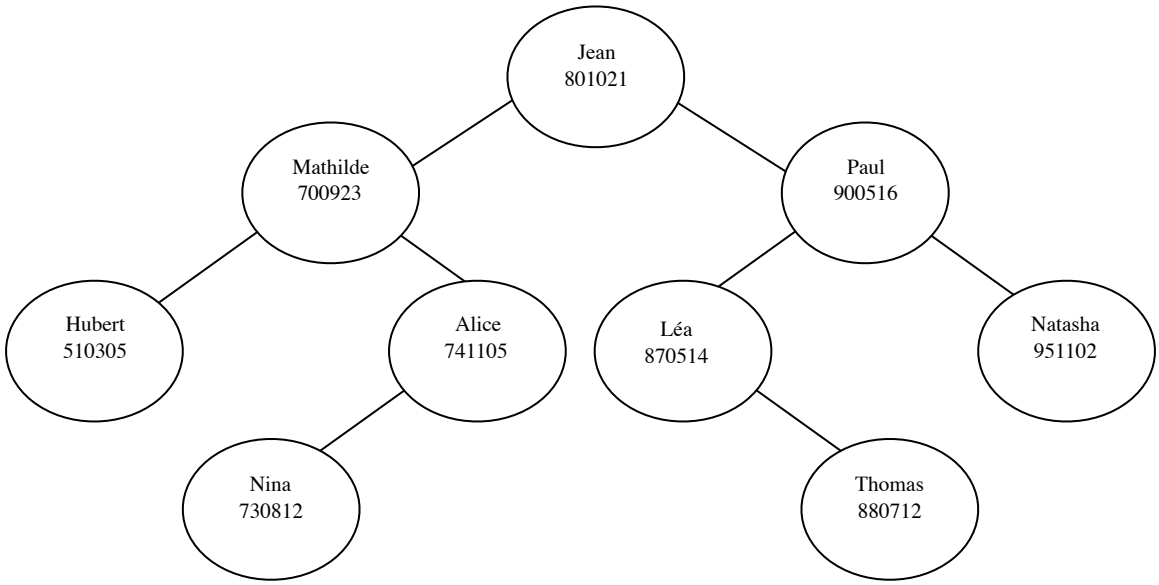
1. Déterminer la complexité asymptotique de cette procédure, justifier votre réponse.
2. Proposer un algorithme récursif pour résoudre le même problème; la complexité asymptotique de cet algorithme doit être identique à celle de la fonction itérative ci-dessus.

3.Une variante des ABR (2+3+3+2+3+3 = 16 points)

On souhaite répertorier un ensemble de personnes dans un arbre binaire de recherche (**ABR**) trié en fonction de leur date de naissance. Chaque nœud de l'arbre contient le nom de la personne, et sa date de naissance sous la forme AAMMJJ.

On suppose qu'il existe au moins une personne.

Exemple :



On utilise le type

```
Personne = <nom : chaîne, date : entier>
```

pour représenter les personnes et on peut accéder aux champs d'une variable `pers` de type `Personne` en utilisant les opérations

```
pers.nom
pers.date
```

Les opérations pour les arbres et les listes sont rappelées à la fin du sujet.

On demande d'écrire les algorithmes des fonctions/procédures qui permettent de résoudre les questions suivantes. Il faut également compléter le profile de chaque fonction/ procédures avec le mode (Entrée/Sortie/EntréeSortie) de chaque paramètre. **L'efficacité des algorithmes est prise en compte dans l'évaluation, un algorithme correct mais inefficace permettra d'obtenir seulement une partie des points**. Pour chaque question vous pouvez utiliser des fonctions/procédures auxiliaires.

1. Ecrire une fonction qui retourne vrai si et seulement si une `personne` est présente dans l'arbre:

```
fonction existe( arb:ABR[Personne], personne:Personne):booléen
```

Indication: La fonction ne doit pas explorer inutilement des parties de l'arbre.

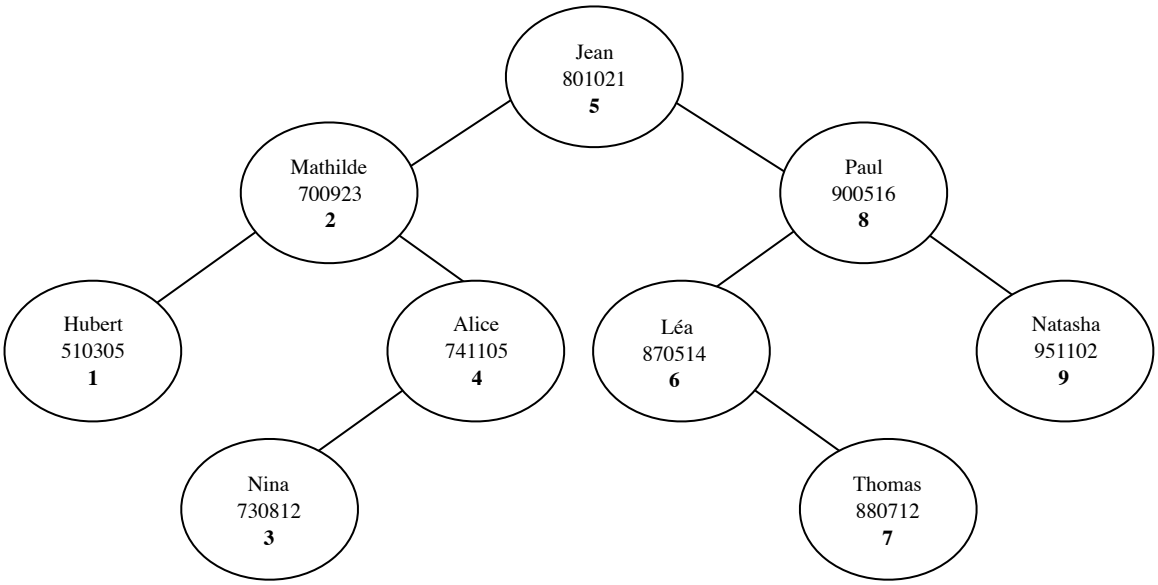
2. Ecrire une fonction qui permet de retourner la i-ème personne la plus âgée:

```
fonction ieme( arb:ABR[Personne], i:entier) : Personne
```

Dans notre exemple la 4ème personne est Alice et la 6ème est Léa.

Indication: On peut obtenir une complexité asymptotique O(N) avec N=nombre noeuds de l'arbre.

3. Afin d'améliorer l'algorithme de recherche de i-ème élément nous utilisons un nouveau type d'arbre, les ABRRs. Un ABRR est un ABR dans lequel chaque noeud contient une information supplémentaire, appelée le rang. Ce rang indique à quelle position on trouverait le noeud dans la liste infixée de ses clés. Par exemple, la version avec rangs de l'arbre ci-dessus est la suivante:



Les primitives de ces ABRR sont celles des ABR classiques avec `Cons` modifiée et deux primitives supplémentaires :

```
fonction   Cons(↓X:E,↓fg:ABRR[E],↓fd:ABRR[E],↓rank:int) : ABRR[E]
fonction   rang (↓A:ABRR[E]) : entier
procedure  ModifRang (↕A:ABRR[E], ↓rang:entier)
```

Ecrire une fonction qui transforme un ABR classique en un ABRR avec exactement la même structure que l'ABR classique mais avec des rangs correspondants aux positions des noeuds dans la liste infixée des clés:

```
fonction ajouterRang(  A:ABR[E]) : ABRR[E]
```

Indication: On peut obtenir une complexité asymptotique O(N) avec N=nombre noeuds de l'arbre.

4. Ecrire une fonction qui retourne le nombre de personnes présentes dans un ABRR:

```
fonction taille(A:ABRR[E]) : entier
```

Indication: On peut obtenir une complexité asymptotique O(H) avec H=hauteur de l'arbre.

5. Ecrire une fonction qui retourne le nombre de personnes nées avant une date donnée:

```
fonction chercher(A:ABRR[E], date:entier) : entier
```

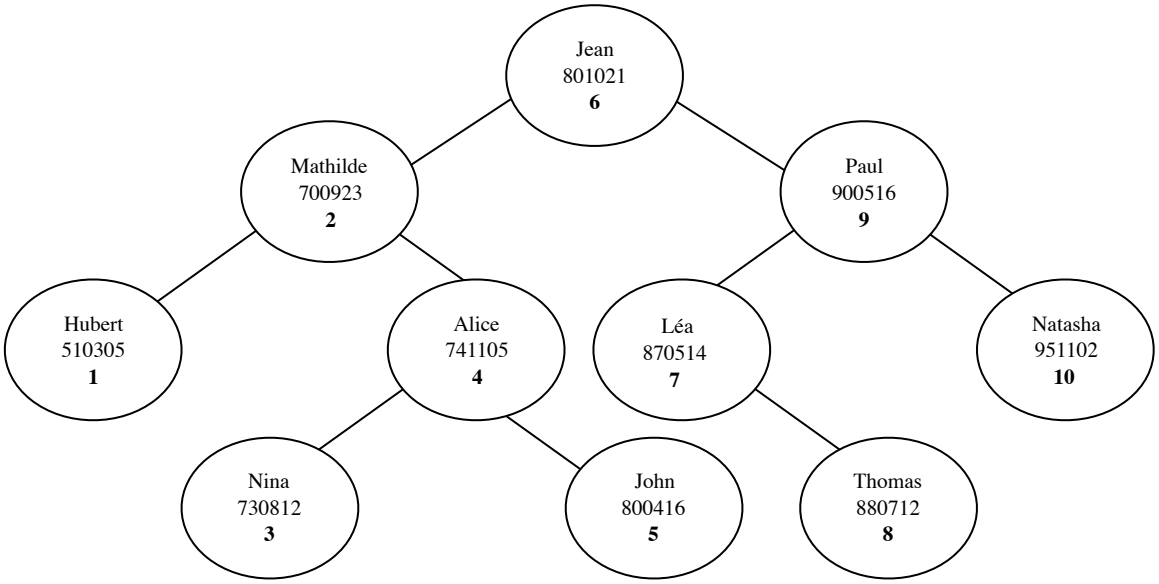
Par exemple, le résultat pour l'exemple ci-dessus et une date `870605` est `6`.

Indication: La fonction ne doit pas explorer inutilement des parties de l'arbre.

6. Ecrire une procédure d'insertion d'une personne dans un `ABRR`. Les rangs doivent bien évidement être mis à jour si nécessaire.

```
procedure inserer(  arb:ABRR[Personne], personne:Personne)
```

Par exemple, en insérant John dans l'arbre précédent nous obtenons:



***Rappel**:*

```
fonction   ConsVide () : Liste[E]
fonction   Cons (↓X : E, ↓L : Liste[E]) : Liste[E]
fonction   EstVide (↓L : Liste[E]) : booléen
fonction   Tete (↓L : Liste[E]) : E
fonction   Reste (↓L : Liste[E]) : Liste[E]
procedure  ModifTete (↕L : Liste[E], ↓X : E)
procedure  ModifReste (↕L : Liste[E], ↓NouvReste : Liste[E])

fonction   ConsVide() : ABR[E]
fonction   Cons(↓X : E, ↓fg : ABR[E], ↓fd : ABR[E]) : ABR[E]
fonction   EstVide(↓A : ABR[E]) : booléen
fonction   Racine (↓A : ABR[E]) : E
fonction   FG (↓A : ABR[E]) : ABR[E]
fonction   FD (↓A : ABR[E]) : ABR[E]
procedure  ModifRacine (↕ A : ABR[E], ↓ X : E)
procedure  ModifFG(↕A : ABR[E], ↓NouvFG : ABR[E])
procedure  ModifFD(↕A : ABR[E], ↓NouvFD : ABR[E])
```