# TP : Arbre binaire

Voici une structure de données que vous utiliserez plus tard, mais que vous pouvez programmer dés maintenant.

**Dessin des arborescences binaires**

Pour pouvoir tester vos résultats, vous pourrez utiliser une fonction

*std : :string SortieLatex(AB Ar) ;*

que vous devrez déclarer dans votre header, qui est implémentée dans le fichier *SortieLatex.o*

**exemple :**

– le code du *main* du fichier *AB.cpp* qui fabrique une arborescence *A0* que vous voulez dessiner.

```
int main() {
  // fabrication de A0
  AB A0=new Sommet(0), A1=new Sommet(1), A2=new Sommet(2),
        A3=new Sommet(3),A4=new Sommet(4), A5=new Sommet(5);
  A3->GrefferSAG(A4);
  A3->GrefferSAD(A5);
  A0->GrefferSAG(A2);
  A0->GrefferSAD(A4);
  // impression
  std::cout<<SortieLatex(A0);
  return 1;
}
```

– l'ordre de compilation de ce fichier

```
$ g++ SortieLatex.o AB.cpp
```

– Vous envoyez le string produit par cette fonction dans un fichier *machin.tex* (c'est à dire dont le suffixe sera *.tex*[1])

```
$ ./a.out > machin.tex
```

– enfin vous compilez ce ficher *.tex* en un fichier *.pdf*

```
pdflatex machin.tex
```

et le contenu du fichier *machin.pdf* sera le dessin de l'arborescence.

1. vous choisirez vous même le nom du fichier au lieu de *machin*

**travail demandé**

Il s'agit de représenter un arbre binaire, étiqueté sur les sommets par des objets de type *valeur*. Idéalement, il faudrait utiliser un type paramétré, mais pour simplifier la programmation, nous utiliserons la définition de type

```
typedef int Valeur;
```

pour étiqueter par des *int*.

1. soient les déclarations et définitions suivantes : (dans une *struct*, tous les champs et toutes les méthodes sont *public* par défaut).

   ```
   typedef Sommet* AB;

   struct Sommet {
     Valeur racine;
     AB SAG, SAD;
   // la suite aux questions suivantes
   ```

   – Où se trouve l'etiquette d'un sommet ?
   – comment voit-on que l'arborescence est binaire ?

2. On définit maintenant un constructeur

   ```
   Sommet(Valeur v);
   ```

   qui fabrique une arborescence réduite à un seul sommet étiqueté par la valeur $v$.
   Écrire la méthode correspondante.

3. Écrire la méthode du constructeur par recopie

   ```
   Sommet(Sommet& s);
   ```

   qui recopie toute l'arborescence de racine le sommet $s$.

4. Écrire la méthode

   ```
   bool FeuilleP();
   ```

   qui indique si l'objet avec lequel est invoquée la méthode est réduit à une feuille.

5. Écrire les deux méthodes

   ```
   void SupprimerSAG();
   void SupprimerSAD();
   ```

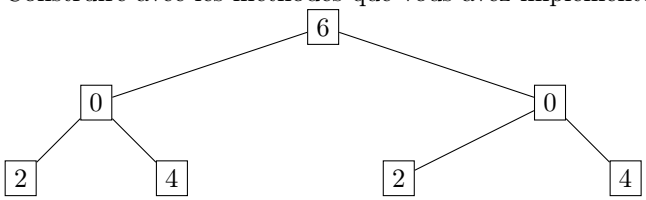
   qui suppriment respectivement les sous arborescences gauche et droite de l'objet avec lesquelles elles sont invoquées

6. puis les deux méthodes

   ```
   void GrefferSAG(AB p);
   void GrefferSAD(AB p);
   ```

   qui remplacent les sous arborescences respectivement gauche et droite de l'objet avec lesquelles elles sont invoquées par le paramètre $p$.

7. Construire avec les méthodes que vous avez implémentées l'arborescence

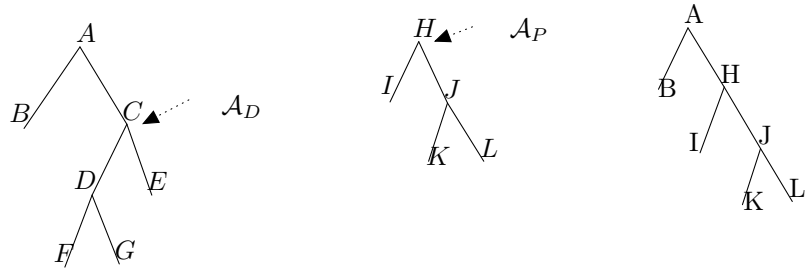



8. On veut maintenant rajouter un champs *Pere* tel que chaque sommet aie un pointeur sur son père. Modifier toutes les méthodes en conséquence.

9. Écrire alors la méthode

   ```
   void RemplacerPourLePerePar(AB);
   ```

   telle que après l'invocation $\mathcal{A}_D \rightarrow RemplacerPourLePerePar(\mathcal{A}_P)$ l'arborescence de gauche soit devenue l'arborescence de droite.



# TP : Cours 1 et 2

# reconnaissance de classes $\Theta$

1. (a) écrire des fonctions
   - *void* $f_1(int\ n)$ dont la complexité soit dans $\Theta(n)$,
   - *void* $f_9(int\ n)$ dont la complexité soit dans $\Theta(n^9)$,
   - *void* $f_2(int\ n)$ dont la complexité soit dans $\Theta(2^n)$
   - et *void* $f_3(int\ n)$ dont la complexité soit dans $\Theta(3^n)$.

   A partir de quelle valeur de $n$ sentez vous la différence entre l'exécution des fonctions
   - $f_1(n)$ et $f_9(n)$ ?
   - $f_9(n)$ et $f_2(n)$ ?
   - $f_2(n)$ et $f_3(n)$ ?

   y a-t-il une valeur de $n$ pour laquelle vous sentez une différence entre
   - l'exécution de la fonction $f_9(n)$ et l'exécution successive des fonctions $f_1(n)$ et $f_9(n)$ ?
   - l'exécution de la fonction $f_2(n)$ et l'exécution successive des fonctions $f_9(n)$ et $f_2(n)$ ?
   - l'exécution de la fonction $f_3(n)$ et l'exécution successive des fonctions $f_2(n)$ et $f_3(n)$ ?

   (b) réécrire éventuellement ces fonctions $f_1,\ f_9,\ f_2,\ f_3$ sans tricher : par exemple les répétitives doivent aller de 1 à $n$ et pas de 1 à $n^9$ ou à $2^n$.

   **Indications :**
   Pour les fonctions de complexité polynomiale, imbriquez des répétitives.
   Pour les fonctions de complexité exponentielle, utilisez des appels récursifs.

2. Vous récupérerez dans l'espace pédagogique les fichiers
   - fonctionsMysterieuses.h
   - fonctionsMysterieuses.o
   - SolutionsFonctionsMysterieuses.cpp

   Le fichier fonctionsMysterieuses.o est la compilation des six fonctions
   déclarées dans fonctionsMysterieuses.h.
   Chacune de ces fonctions est le produit par une constante (à chaque fois différente) d'une des fonctions

$$log(n),\ \sqrt{n},\ n^2,\ n^5,\ 2^n,\ 3^n$$

   Il s'agit de déterminer la classe de chacune de ces fonctions mystérieuses.

   **indications pour trouver la solution :**
   Le fichier SolutionsFonctionsMysterieuses.cpp vous fournit un *main* qui vous permet de tester ces fonctions à la main.
   Son rôle est de vous aider pour vos ordres de compilation.
   Il est plutôt conseillé d'écrire un programme pour tester automatiquement les valeurs renvoyées par chaque fonction quand on fait varier $n$.
   Je vous suggère de commencer par trouver la plage significative dans lequel faire varier le paramètre $n$.
   Ensuite, il vous faut essayer, pour chacune des fonctions $\Phi(n)$ à laquelle vous songez, de regarder si le *float* $\frac{f_?(n)}{\Phi(n)}$ (en appelant $f_?$ la fonction mystérieuse que vous étudiez) se stabilise à une valeur (non nulle) quand $n$ croit dans sa plage significative.
   Cette valeur stable vous donnera la constante multiplicative (qui est toujours soit un entier soit l'inverse d'un entier).
   Attention au cas où des nombres négatifs apparaissent : cela signifie en général que soit $f_?(n)$, soit $\Phi(n)$ ont dépassé la valeur *int* maximale autorisée.
   Pour les fonctions qui renvoie des réels, ne vous étonnez pas de petites erreurs d'arrondi.

3. (a) écrivez une fonction *int Incrementer(int n, bool T[])*, qui
      - étant donné un tableau $T$ de dimension $n$ qui représente un entier $n_T$ écrit en base 2 avec $n$ bits,[2] tel que $T[0]$ vaut 0
      - modifie ce tableau $T$ pour représenter le nombre $n_T + 1$
      - et renvoie le nombre d'**affectations** effectuées dans le tableau $T$ (on ne s'occupe pas du nombre d'affectations des variables auxiliaires).

      Écrire cette fonction de façon à minimiser ce nombre d'affectations (on ne s'occupe pas non plus du nombre de tests).

   (b) Écrire ensuite une fonction *int Compte(int n)* qui compte le nombre moyen d'affectations que l'on effectue quand on incrémente **tous** les tableaux qui représentent les entiers entre 0 et $2^n - 1$.
      Tester pour les valeurs de $n$ de 1 à 28 et conclure.

2. par exemple le tableau | 0 | 1 | 0 | représente le nombre 2

# TP : Les tris

1. implémentez les différents tris vus en cours et en TD.
   Pour le tri panier, vous testerez votre algorithme avec une valeur minimum de 0 et une valeur maximum qui sera la taille du tableau.
2. écrivez pour chacun de ces tris une version dans laquelle s'affiche chaque étape de l'algorithme.
3. A partir de quelle taille de tableau devient-il évident que ces tris [3] n'appartiennent pas aux mêmes classes de complexité ?
   Pour le tri rapide vous testerez **deux** pires des cas et un cas généré aléatoirement. Vous testerez aussi si dans le cas généré aléatoirement, les deux modifications proposées en TD.
4. analysez empiriquement la complexité en moyenne pour l'algorithme de recherche issu du tri rapide (on ne comptera que les affectations effectuées dans *déplacer*).

# TP : les tas

1. implémenter la classe tas vue en cours
2. écrire une fonction *TasVersAB* qui prenne en entrée un tas et qui renvoie une arborescence binaire dessinable grâce à la fonction *SortieLatex* vue au TP Arbres Binaires.
3. écrire une version de l'algorithme de tri par tas dans laquelle s'affiche chaque étape de l'algorithme.En tester les limites.

# TP : les ABR

1. implémentez la *struct ABR* comme héritant de la *struct AB* vue au TP Arbres Binaires [4].
2. un ABR $A_1$ est dit *de domaine moins étendu* qu'un ABR $A_2$ si et seulement si
   – la plus petite valeur de $A_1$ est plus grande que la plus petite valeur de $A_2$ et
   – la plus grande valeur de $A_1$ est plus petite que la plus grande valeur de $A_2$

3. n'affichez évidement pas alors chaque étape de l'algorithme
4. et que vous finirez éventuellement d'implémenter maintenant

Écrire une fonction (non récursive) qui teste si $A_1$ est de domaine moins étendu que $A_2$

3. un ABR est dit *inclus* dans un autre ABR quand toutes les valeurs du premier apparaissent dans le deuxième.
   Écrivez une méthode *inclus* de complexité minimale, analysez la complexité de votre algorithme et justifiez en la minimalité.