NF16 : FINAUX

TD

On peut créer un objet d'une structure dans une fonction sans utiliser de malloc si la structure est bien définie et qu'elle n'a pas besoin d'allocation dynamique

structure FILE

```
typedef struct file {
    int tab [ MAXF ];
    int tete ; // Premier element a defiler
    int queue ; // Premier emplacement libre
} file ;
```

- La file est vide si tete==queue
- La file est pleine si la queue se trouve juste en dessous de la tête

> pleine si Tete[F]=(Queue[F]+1)%Longueur[F]

structure PILE

```
typedef struct pile {
    int sommet ; // Indice de l ’ element au sommet de la pile
    int tab [ MAXP ];
} pile ;
```

Arbre k-aire

```
typedef struct Cellule{
    int cle;
    struct Cellule *père;
    struct Cellule *filsG;
    struct Cellule *filsD;
}Cellule;
```

Arbre binaire de recherche

Parcours d'un arbre binaire

1. Parcours préfixe

- racine > guche > droite

2. Parcours postfixe

- gauche > droite > racine

3. Parcours infixe

- gauche > racine > droite

Supprimer un noeud

- si aucun fils on supprime en modifiant le père
- si un fils on supprime et on rattache le fils au père
- si deux fils on remplace le noeud par le successeur (ou le prédecesseur ?)

> si x possède un sous-arbre droit non vide, le successeur est le plus petit élément de cet arbre si x possède un sous-arbre gauche non vide, le prédecesseur est le plus grand élément de cet arbre

Complexité des opérations de base

Fonction	Liste quelconque	Liste triée	ABR
Ajouter	O(1)	O(n)	O(h)
Supprimer	O(n)	O(n)	O(h)
Rechercher	O(n)	O(n)	O(h)
Intersecte	O(n)	O(n)	O(h)

Dans le cas général, on peut avoir un ABR de hauteur n (si on insère par ordre croissant). Dans le cas d'un arbre complet on a h=log2(n).

AVL

Définition : Un arbre AVL est un arbre binaire de recherche équilibré, c'est-à-dire que pour chaque nœud de l'arbre, la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit est au plus égale à 1.

- nombre max de noeuds : (2^h+1) - 1

Rotation droite

> Complexité O(1)

```
Rotd(x:Racine)
    Si gauche[x]!=NIL:
        y:=gauche[x]
        C:=droit[y]
        pere[y]:=pere[x]
        Si pere[x]!=NIL:
            Si gauche[pere[x]]=x:
                gauche[pere[x]]=y;
            Sinon:
```

```
                droit[pere[x]]=y;
        droit[y]:=x
        pere[x]:=y
        gauche[x]:=c
        Si C!=NIL:
            pere[C]:=x
        retourner y
```

Rotation gauche simplifiée

```
nœud *Rotg(*nœud N){
    if(N->fdroit!=NULL){
        nœud *y=N->fdroit; // fils droit du noeud
        nœud *c=y->fgauche; // fils gauche du fils

        y->fgauche=N;
        N->fdroit=c;
        return y;
    }
}
```

Variation hauteur d'un noeud après Rotation (droite)

h'(v)=h(v)+DELTA si h(g)>h(d) ou (h(g)=h(d) et DELTA=1)
0 sinon

or DELTA =

- -1 si h(B)>max(h(D),h(C))
- 1 si h(D)>max(h(B),h(C))
- 0 sinon

h=1+max(1+h(B),1+h(C),h(D))
h'=1+max(h(B),1+h(C),1+h(D))

Algorithme récursif d'enracinage de x

> Complexité O(hauteur)

```
enraciner(r:Racine; e:entier)
    Si(cle[r]=e)
        retourner r
    Sinon si(cle[r]>e)
        enraciner(gauche[r],e) // remonte e au fils gauche
        retourner rotd(r) // enracine fils gauche
    Sinon si(cle[r]<e)
        enraciner(droit[r],e) // remonte e au fils droit
        retourner rotg(r) // enracine fils droit
```

Equilibre d'un noeud

h(gauche[x])-h(droit[x])

eqx=1 + max(h(B),h(C)) - h(D)

Rotd qui retourne DELTA

```
Rotd ( x : Noeud ; Delta : entier )
// ...
    eqx := eq [ x ]
    eqy := eq [ y ]
    eq [ x ] := eqx -1 - max (0 , eqy )
    Si eq [ x ] >= 0
        eq [ y ] := eqy -1
    Sinon
        eq [ y ] := eqx -2+ min (0 , eqy )
    Delta := 0
    Si eqx <= 0
        Delta := 1
    Si eq [ y ] >=0
        Delta := -1
```

Rotation gauche droite

- CNS : X a un fils gauche Y et Y a un fils droit Z

1. Rotation gauche sur y
2. Rotation droite sur x

```
Rotgd ( x : Noeud ; Delta : entier )
    Si gauche [ x ] != NIL et droit [ gauche [ x ]] != NIL
        eqx := eq [ x ]
        gauche [ x ] := rotg ( gauche [ x ] , D1 ) // Rot gauche sur y
        eq [ x ] := eq [ x ]+ D1 // equilibre parent quand fils gauche varie de D1
        Si ( eqx > 0 ou ( eqx = 0 et D1 = 1) ) // variation pere quand fils varie
            Delta := D1
        Sinon
            Delta := 0
        z := rotd (x , D2 ) // totation droite sur x
        Delta := Delta + D2 // Delta total
        retourner z // z est la racine
```

Insérer un noeud puis équilibrer un AVL

```
insere_avl(x:Noeud, n:Noeud, delta:entier):
    si x = NIL:
        delta:=1
        retourner n

    delta:=0
    si (cle[n] < cle[x]):
        gauche[x]:=insere_avl(gauche[x], n, D1)
         si(eq[x]>0 ou (eq[x]=0 et D1=1)):
            delta:=D1
        eq[x] := eq[x] + D1

    si (cle[n] > cle[x]):
        droit[x]:=insere_avl(droit[x], n , D1)
        si(eq[x]<0 ou (eq[x]=0 et D1=1)):
            delta:=D1
        eq[x] := eq[x] - D1

    si |eq[x]|=2:
        x:=Reequilibre(x, D2)
        delta := delta + D2
```

Complexité :

- Insertion du sommet / descente : O(h)
- Rechercher du sommet |eq[x]| / remontée : O(h)
- équilibre : O(1)

> L'algo est en O(h) --> dans un AVL on a h<1.44log2(n) donc O(log2n)

Passer de string à integer

```
while(expression[k]>='0' && expression[k]<='0'){
    nb=nb*10;
    nb=nb+(exp[k]-'0');
    k++;
}
```

STRUCTURE TAS

TAS BINAIRE :

- Le noeud parent d'un élément i est à l'indice (i-1)//2 // Li/2
- Le fils gauche est à l'indice 2i+1 // 2i
- Le fils droit est à l'indice 2i+2 // 2i+1 --> pour passer d'un ABR à un tas on utilise donc l'ordre de parcours en niveau

> tas-min : chaque noeud est inférieur ou égale à ses enfants tas-max : chaque noeud est supérieur ou égale à ses enfants

Propriétés :

- Racine(A)=Retourner(1)
- Pere(i)=Retourner(i//2)
- Gauche(i)=Retourner(2i)
- Droit(i)=Retourner(2i+1)
- A[i]<=A[Pere(i)] dans ce cours
- les éléments entre [taille[A]/2]+1 et taille[A] sont des feuilles

1. Une liste triée représente TOUJOURS un tas
2. UN tas ne représente PAS TOUJOURS une liste triée

> NOMBRE DE NOEUD MIN : 2**h**
> **NOMBRE DE NOEUX MAX : 2**(h+1)-1

- entasser un tas max : **O(log n)**
  --> car parcourt de la racine aux feuilles donc O(h), or tas = arbre parfait donc = O(log2(n))

```
void entasser(int arr[], int n, int i){
    int max=i; // le plus grand est la racine
    int left=2*i+1;
    int right=2*i+2;

    // si gauche plus grand que racine
    if(left<n && arr[left]>arr[max]){
        max=left;
    }
    if(right<n && arr[right]>arr[max]){
        max=right;
    }

    if(max!=i){
        // on échange de racine
        swap(&arr[i],&arr[max]);
        // on vérifie récursivement avec la nouvelle racine
        entasser(arr,n,max);
    }
}
```

- construire : **O(n)**

```
void construire(int arr[], int n)[
    for(int i=n/2-1;i>=0;i--){ // on part du dernier noeud non feuille et on
remonte
        entasser(arr, n, i)
    }
]
```

- supprimer le max : **O(log n)**

```
void supprimer_max(int arr[], int *n){
    int max=arr[0]; // le max est la racine
    arr[0]=arr[n-1]; // on remplace par la dernière feuille
    (*n)--;
    entasser(arr, n, 0);
    return max;
}
```

Insérer une valeur (tas min)

```
void insérer(int *arr[], int *n, int val){
    // on commence par insérer à la fin
    arr[*n]=&val;
    *n++;

    // ensuite on remonte tant que le père est plus grand
    int i=*n-1; // indice nouvel élément
    while(i>0 && arr[i]<arr[(i-1)/2]){
        int temp = arr[i];
        arr[i]=arr[(i-1)/2];
        arr[(i-1)/2]=temp;
        i=(i-1)/2;
    }
}
```

- tri par tas : **O(n log(n))**

```
void triTas(int arr[], int n){
    construire(arr,n);
    for(int i=n-1;i>0;i--){
        swap(&arr[0], &arr[i]);
        entasser(arr,i,0)
    }
}
```

TRIS

TRI PAR SELECTION

On recherche le min du tableau à chaque itération et on le place au début du compteur

```
Tri_Selection(T:Tableau; n:entier)
    i,j,pos_min : entier
```

```
    Pour i:=1 à n-1:
        pos_min:=i
        Pour j:=i+1 à n
            Si T[j]<T[pos_min]: // recherche du min
                pos_min:=j
        Echanger(T,i,pos_min) // on échange les deux
```

Complexité : somme de i=1 à n-1(n-i) = somme de i=1 à n-1(n) = (n(n-1))/2 donc O(n^2)

TRI A BULLES

On prend l'élément en position n, tant qu'il est plus petit que son prédécesseur, on l'échange de position avec jusqu'à la première position. On prend le nouvel élément en position n et on l'échange jusqu'à la position 2 etc...

```
Tri_bulle(T:tableau; n:entier)
    i,j:entier
    // tab_trie:bool
    Pour i:=n-1 à 1 par pas de -1:
        // tab_trie:=vrai
        Pour j:=n à i+1 par pas de -1:
            Si T[j]<T[j-1]:
                Echanger(T,j,j-1)
                // tab_trie:=faux
        // Si tab_trie:
            // Fin
```

Complexité : dans le pire des cas en O(n^2) comme au dessus. Le meilleur cas est aussi en OMEGA(n^2) mais is on rajoute tous les commentaires alors on aura OMEGA(n)

TRI PAR INSERTION

Part d'un tableau de taille i-1 trié et remonte le ième élément à la bonne position poir avoir un tableau trié de taille i

```
Tri_Insertion(T:Tableau; i:entier)
    j:entier
    // tableau de taille 1 déjà trié
    Si i>1:
        Tri_Insertion(T,i-1) // tab de taille i-1 trié
        j:=1
        Tant que(j>1 et T[j-1]>T[j])
            Echanger(T,j,j-1) // décale à gauche jusqu'à bonne position
            j:=j-1
```

Complexité :
Soit C(n) le nombre d'exec de la boucle TantQue dans le pire des cas (tableau trié décroissant), à chaque appel

récursif, l'élément à l'indice i doit être remonté jusqu'à 1 ce qui nécessite i-1 permutations.
C(n)=0 (si n<=1) sinon (n-1)+C(n-1)
donc C(n) = somme de k=1 à n-1(k) = (n(n-1))/2
Donc O(n^2)

TRI PAR DENOMBREMENT

```
Tri_Dénombrement(A,B,k)
    Pour i=1 à k faire C[i]:=0
    Pour j=1 à longueur[A] faire:
        C[A[j]]:=C[A[j]]+1
    // C[i] contient le nb d'éléments de A égaux à i
    Pour i=2 à k faire:
        C[i]:=C[i]+C[i-1]
    Pour j:=longueur[A] à 1 faire:
        B[C[A[j]]]:=A[j]
        C[A[j]]:=C[A[j]]-1
```

- **Complexité** : O(n+k) avec n taille de A

Tri rapide

```
Tri_Rapide(A,p,r)
    si p<r alors:
        q:=Partitionner(A,p,r)
        Tri_Rapide(A,p,q)
        Tri_Rapide(A,q+1,r)

Partitionner(A,p,r)
    x:=A[p]; i:=p-1; j:=r+1
    Tant que vrai faire:
        répéter j:=j-1 jusqu'à A[j]<=x
        répéter i:=i+1 jusqu'à A[i]>=x
        si i<j alors échanger(A[i],A[j])
            sinon retourner j
```

- **Pire des cas** : O(n^2)
- **Meilleur des cas** : O(n log(n))

TRI FUSION

On divise successivement le tableau. Par exemple si on a t[7] alors on aura 7 sous-tableaux qu'on va ensuite fusionner grâce à la fonction Interclassement.

```
Tri_fusion(T:tableau; i,j:entier)
    k:entier
```

```
    Si i<j:
        k:=(i+j)/2
        Tri_fusion(T,i,k)
        Tri_fusion(T,k+1,j)
        Interclassement(T,i,j,k)
```

- Interclassement : fusionne les deux moitiés en les triant (compare valeur par valeur de chaque moitié)
  Complexité interclassement : O(j-i+1) --> trie le tableau entre i et j en sachant qu'il est trié de i à k et de k+1 à j
  Complexité de l'ensemble des interclassements : O(n)
- il y a ln(n)/ln(2) niveaux de décomposition
- il y a ln(n)/ln(2) niveaux d'interclassement
  Complexité tri fusion : O(nln(n))

DEFINITIONS

- un tri est **stable** s'il ne modifie pas l'ordre initial de deux éléments de clés égales.
- un tri est **interne** s'il s'effectue sur des données présentes en mémoire centrale.
- un tri **externe** s'effectue sur des données résidant en mémoire secondaire.

ANNALES

insertion d'un noeud et mise à jour du delta de hauteur + rééquilibrage

```
Arbre insert(Arbre A, int cle, int *delta){
    int D1, D2;
    *delta=0;
    if(*A==NULL){
        *delta=1;
        return creerNoeud(cle);
    }
    if(cle>A->cle){
        A->droit=insert(A->droit,cle,delta);
        if(A->equilibre<0 || (A->equilibre==0 && D1==1)){
            *delta=D1;
        }
        A->equilibre-=D1;
    }
    else if(cle<A->cle){
        A->gauche=insert(A->gauche,cle,D1);
        if(A->equilibre>0 || (A->equilibre==0 && D1==1)){
            *delta=D1;
        }
        A->equilibre+=D1;
    }
    // on rééquilibre si besoin
    if(A->equilibre>=2 || A->equilibre<=2){
        A=reequilibre(A, &D2);
        (*delta)+=D2;
    }
```

```
    return A;
}
```

complexité spatiale

- int : O(1)
- tableau : O(n)
- on regarde les variables temporaires et on additionne leur complexité spatiale

> par exemple algo de tri : O(n) car on utilise un tableau temporaire de taille n appels récursifs : profondeur de récursion en O(log n)

VRAC DU COURS

- Arbre complet d'arité k : arbre d'arité k pour lequel toutes les feuilles ont la même profondeur et tous les nœuds internes ont pour degré k
- nombre de noeuds internes d'un arbre binaire complet de hauteur h vaut 2^h - 1
- arbre parfait : Arbre binaire dont les feuilles sont situées sur deux niveaux au plus, l'avant dernier niveau est complet, et les feuilles du dernier niveau sont regroupées le plus à gauche possible

• Dans Partitionner :
• Les indices i et j ne font jamais référence à un élément de A hors de l'intervalle [p..r].
• L'indice j n'est pas égal à r quand Partitionner se termine : le découpage n'est jamais trivial.
• Chaque élément de A[p..j] est inférieur ou égal à chaque élément de A[j + 1 .. r] quand Partitionner se termine.

toute la partie sur les graphes ?