

LISTES

```
typedef struct maille  
{
```

```
    elt_t elt;  
    struct maille *next;
```

```
}maille_t;
```

```
maille_t *creer1Maille( elt_t e , maille_t *l){
```

```
    maille_t* new= (maille_t*)malloc(sizeof(maille_t*));  
    assert(new!=NULL);  
    new -> elt =e;  
    new-> next=l;  
    return new;  
}
```

```
void inserer1Elt( elt_t e, maille_t **l){
```

```
    if VIDE((*l)){  
        *l=creer1Maille(e,*l);  
        return ;  
    }  
    if(mode==tete)inserer1EltEnTete(e,l);  
    else if(mode==queue)inserer1EltEnQueue(e,*l);  
    else if(mode==ordo)inserer1EltEnOrdo(e,l);
```

```
}
```

```
void inserer1EltEnTete(elt_t e, maille_t **l){
```

```
    *l=creer1Maille(e,*l);
```

```
}
```

```
void inserer1EltEnQueue(elt_t e,maille_t *l){
```

```
    if(VIDE(SUIVANT(l))){  
        SUIVANT(l)=creer1Maille(e,SUIVANT(l));  
        return;  
    }
```

```
    inserer1EltEnQueue(e,SUIVANT(l));
```

```
}
```

```
void afficherListe( maille_t *l){
```

```
    if VIDE(l)return;
```

```
    afficherElt(&(l->elt));
```

```
    printf("=>");
```

```
    afficherListe(SUIVANT(l));
```

```
}
```

```
int nbElts(maille_t *l){
```

```
    if VIDE(l) return 0;
```

```
    return nbElts(SUIVANT(l))+1;
```

```
}
```

```
maille_t *dernierEltListe(maille_t *l){
```

```
    if VIDE(l) return NULL;
```

```
    if VIDE(SUIVANT(l)) return l;
```

```
    return dernierEltListe(SUIVANT(l));
```

```
}
```

```
void supprimerPremier( maille_t **l){
```

```
    if VIDE(l)return ;
```

```
    maille_t *temp =*l;
```

```
    *l=SUIVANT((*l));
```

```
    free(temp);
```

```
}
```

```
void supprimerDernier(maille_t **l){
```

```
    if VIDE((*l))return;
```

```
    if(SUIVANT((*l))==NULL){
```

```
        maille_t *aux=*l;
```

```
        *l=NULL;
```

```
        free(aux);
```

```
        return;
```

```
    }
```

```
    supprimerDernier(&SUIVANT((*l)));
```

```
}
```

```
maille_t *copierListe(maille_t *l){
```

```
    if VIDE(l) return NULL;
```

```
    return creer1Maille(ELT(l),copierListe(SUIVANT(l)));
```

```
}
```

ABR

```
typedef struct noeudABR noeudABR_t; //définition du type nœud
struct noeudABR // définition du type arbre
{
    elt_t elt;
    struct noeudABR * gauche;
    struct noeudABR * droite ;
};

void displayABRCroissant( noeudABR_t * a)
{
    if VIDE(a) return ;
    displayABR(GAUCHE(a));
    afficherElt(&(a->elt));
    printf("=>");
    displayABR(DROIT(a));
}

noeudABR_t * insererABR(elt_t e, noeudABR_t * a)
{
    if VIDE(a) return creer1Noeud(e,NULL,NULL);
    if(a->elt >e) GAUCHE(a)=insererABR(e,GAUCHE(a));
    else DROIT(a)=insererABR(e,DROIT(a));
    return a;
}

noeudABR_t * estDansABR (elt_t e, noeudABR_t * a)
{
    if VIDE(a) return NULL;
    if(a->elt==e) return a;
    if(a->elt >e) return estDansABR(e,GAUCHE(a));
    else return estDansABR(e,DROIT(a));
}

int hauteurABR(const noeudABR_t * a)
{
    if VIDE(a) return -1;
    return 1+MAX(hauteurABR(GAUCHE(a)),hauteurABR(DROIT(a)));
}

int hauteurABR(const noeudABR_t * a)
{
    if (a == NULL) return -1;
    if (hauteurABR(a->gauche) > hauteurABR(a->droite))
        return 1+hauteurABR(a->gauche);
    else
        return 1+hauteurABR(a->droite);
}
```

```
int nbNoeudsABR(const noeudABR_t * a)
```

```
{
    if VIDE(a) return 0;
    return 1+nbNoeudsABR(DROIT(a)) + nbNoeudsABR(GAUCHE(a)) ;
}
```

```
noeudABR_t * supprimerABR(noeudABR_t * a)
```

```
{
    if VIDE(a) return NULL;
    supprimerABR(GAUCHE(a));
    supprimerABR(DROIT(a));
    free(a);
    return NULL;
}
```

```
noeudABR_t * copierABR (const noeudABR_t * a)
```

```
{
    if(VIDE(a)) return NULL;
    return creer1Noeud(a->elt,copierABR(GAUCHE(a)),copierABR(DROIT(a)));
}
```

```
int egalABR (const noeudABR_t * a1, const noeudABR_t * a2)
```

```
{
    if(VIDE(a1) && VIDE(a2)) return 1;
    if(VIDE(a1) && PAS_VIDE(a2) || VIDE(a2) && PAS_VIDE(a1)) return 0;
    if (a1->elt == a2->elt){
        if( egalABR(GAUCHE(a1), GAUCHE(a2))==1 && egalABR(DROIT(a1),
DROIT(a2))==1)return 1;
        else return 0;
    }else return 0;
}
```

```
noeudABR_t * creer1Noeud(elt_t e, noeudABR_t * g, noeudABR_t * d)
```

```
{
    noeudABR_t * p;
    p = (noeudABR_t *)malloc(sizeof(noeudABR_t));
    if (p != NULL)
    {
        p->elt = e;
        p->gauche = g;
        p->droite = d;
    }
}
```

```
    return p;
}
```

AVL

```
typedef struct noeudAVL noeudAVL_t;
struct noeudAVL
{
    elt_t elt ;
    int bal;
    struct noeudAVL * gauche ;
    struct noeudAVL * droite ;
};
```

```
noeudAVL_t * rotG(noeudAVL_t *A){
```

```
    noeudAVL_t * B=DROIT(A);
    DROIT(A)=GAUCHE(B);
    GAUCHE(B)=A;
    BAL(A)+=1-MIN(0,BAL(B));
    BAL(B)+=1+MAX(BAL(A),0);
    return B;
```

```
}
```

```
noeudAVL_t * rotD(noeudAVL_t *A){
```

```
    noeudAVL_t * B=GAUCHE(A);
    GAUCHE(A)=DROIT(B);
    DROIT(B)=A;
    BAL(A)+=-1-MAX(BAL(B),0);
    BAL(B)+=-1-MIN(0,BAL(A));
    return B;
```

```
}
```

```
noeudAVL_t * Equilibrer(noeudAVL_t *A){
```

```
    if(A->bal ==2){
        if(GAUCHE(A)->bal== -1) GAUCHE(A)=rotG(GAUCHE(A));
        A=rotD(A);
    }
    if(A->bal ==-2){
        if(DROIT(A)->bal==1) DROIT(A)=rotD(DROIT(A));
        A=rotG(A);
    }
    return A;
```

```
}
```

```
void majBal (noeudAVL_t *A){
```

```
    BAL(A)=hauteurAVL(GAUCHE(A))-hauteurAVL(DROIT(A));
```

```
}
```

```
T_noeudAVL * miniAVL (T_noeudAVL * a){  
if (a == NULL || a->gauche==NULL) return a;  
return miniAVL (a->gauche);  
}
```

```
T_noeudAVL * maxiAVL (T_noeudAVL * a){  
while (a->droite!= NULL){  
a=a->droite;  
}  
Return a;  
}
```

MINIMIER

on utilise le Heapsort, le principe consiste à organiser l'ensemble à trier en tas. Complexité de $O(n \log(n))$

Dans son état initial le premier élément forme un tas, ensuite on insère la seconde valeur puis on effectue des permutations jusqu'à ce que la condition maximière est validée

```
int est1Minimier(T_Elt T[], int n)
```

```
{  
    int i = n;  
    if(i == 0) return 1;  
    while(i--)  
        if(T[Pere(i)] > T[i])  
            return 0;  
    return 1;  
}
```

```
void remonterMinimier(T_Elt t[], int k)
```

```
{  
    if(existePere(k) && t[k]<t[Pere(k)]) {  
        permuter(t, k, Pere(k));  
        remonterMinimier(t, Pere(k));  
    }  
}
```

```

void permuter(T_Elt t[], int a, int b) {
    T_Elt aux = t[a];
    t[a] = t[b];
    t[b] = aux;
}

```

```

void descendreMinimier(T_Elt t[], int iDebut, int iFin)
{
    int j, Fini = est1Feuille(iDebut, iFin);
    resultat.nbAffectations++;
    while(!Fini) {
        j = FilsG(iDebut);
        if(existeFilsD(iDebut, iFin) && t[j] > t[FilsD(iDebut)]) {
            j=FilsD(iDebut);
        }
        if(t[iDebut] <= t[j]) {
            Fini = 1;
        } else {
            permuter(t, iDebut, j);
            iDebut = j;
            Fini = est1Feuille(iDebut, iFin);
        }
    }
}

```

```

void transfEnMinimier_v2(T_Elt t[], int n)
{
    int i;
    for (i = (n/2)-1; i >= 0; i--) {
        descendreMinimier(t, i, n);
    }
}

```

```

void transfEnMinimier(T_Elt t[], int n)
{
    int i;
    for(i=1;i<n;i++) {
        remonterMinimier(t, i);
    }
}

```

```

void triArbre(T_Elt t[], int n){
    while (n>0){

        n--;

        permuter(t,0,n);

        descendreMinimier(table,0,n);

    }
}

```

}

HUFFMAN

Principe :

La 1° étape, comptabilise le nombre d'occurrences de chaque caractère du document à traiter.

La 2° étape, construit l'arbre de codage de Huffman :

- On part des feuilles (associées aux caractères) et qui portent comme information leur nombre d'occurrences.
- On associe ensuite deux nœuds ayant le nombre d'occurrences le plus faible, pour former un nouveau nœud interne dont la valeur est la somme des valeurs de ses fils.
- On réitère ce processus avec les feuilles et les nœuds internes restants jusqu'à ne plus en avoir qu'un seul, la racine.

La 3° étape, détermine le codage de chaque caractère en parcourant l'arbre de codage depuis les feuilles. En effet, le chemin de la feuille à la racine, détermine le code du caractère :

- À la branche de gauche, on associe un bit à 1, et à l'autre un bit à 0 (ou inversement, car peu importe).
- Enfin, on utilise le codage ainsi obtenu pour coder chaque caractère du document à traiter.


C'est un code préfixe, en effet chaque code d'un caractère ne peut en aucun cas être le préfixe du code d'un autre caractère. Ainsi, la propriété « préfixe » garantit que le message pourra être décodé sans difficulté et sans ambiguïté.

Utilisation d'un minimier indirect car :

Condition minimière : la clé (valeur) de chaque nœud est inférieure ou égale à celle de ses deux fils s'ils existent. Indirect :

les nœuds du minimier ne contiennent pas directement la clé définissant l'ordre de priorité des nœuds, ! la clé d'un nœud du minimier est en fait un indice donnant accès à sa valeur présente dans un (autre) tableau.

Notation de Landau

$$T(m) = aT\left(\frac{m}{b}\right) + o(m^d)$$


- si $d < \log_b a \Rightarrow$ cas 1

$$T(m) = \Theta(m^{\log_b a})$$

- si $d = \log_b a \Rightarrow$ cas 2

$$T(m) = \Theta(m^{\log_b a} \log m)$$

- si $d > \log_b a \Rightarrow$ cas 3

$$T(m) = \Theta(g(m))$$

avec $\log_b a = \frac{\lg a}{\lg b}$ $\lg(1) = 0$

Méthode itérative

$$T(n) = 2 \times T(n/2) + cn$$

$$T(n) = 2 \times (2 \times T(n/4) + cn/2) + cn = 4 \times T(n/4) + 2cn$$

$$T(n) = 4 \times (2 \times T(n/8) + cn/4) + 2cn = 2^3 \times T(n/2^3) + 3cn$$

$$T(n) = 2^i \times T(n/2^i) + i \times cn$$

on cherche quand $n/2^i = 1$ ou $0 \leq i \leq \lfloor \log_2 n \rfloor$

$$T(n) = n \times T(1) + \lfloor \log_2 n \rfloor \times cn = \lfloor \log_2 n \rfloor \times cn$$

on a donc $T(n) = (n \log n)$

$$T(n) = 2 \times T(n-1) + 1 \text{ pour } n > 0 \text{ et } T(0) = 0$$

$$T(n) = 2 \times (2 \times T(n-2) + 1) + 1 = 2^2 \times T(n-2) + 1 + 2$$

$$T(n) = 2^2 \times (2 \times T(n-3) + 1) + 1 + 2 = 2^3 \times T(n-3) + 1 + 2 + 2^2$$

$$T(n) = 2^i \times T(n-i) + 1 + 2 + 2^2 + \dots + 2^{i-1}$$

$$T(n) = 2^n \times T(0) + 1 + 2 + 2^2 + \dots + 2^{n-1} \text{ lorsque } i = n$$

$$T(n) = 1 + 2 + 2^2 + \dots + 2^{n-1}$$

$$T(n) = 2^n - 1$$

Effet d'une amélioration (1/2)

Soit N , la taille maximale des données que l'on peut traiter aujourd'hui en 1 heure.

Quelle taille pourra-t-on traiter en 1 heure avec le même programme lorsque les ordinateurs seront 100 et 1000 fois plus rapides ?

Exemple 1 : $T(n) = \Theta(n^2)$

Aujourd'hui : $k \times N^2 = 1h$

Demain : $k/100 \times N'^2 = 1h$

$$\rightarrow N' = 10N$$

Exemple 2 : $T(n) = \Theta(2^n)$

Aujourd'hui : $k \times 2^N = 1h$

Demain : $k/100 \times 2^{N'} = 1h$

$$\rightarrow N' = N + \log_2 100$$

soit

$$N' = N + 6,64$$

Si on a N^2 on aura $N = \sqrt{xN}$

Si on a 2^n on aura $N = \log_2 xN$

Nombre de comparaisons			
Algorithme	Minimum (Ω)	Maximum (O)	Moyenne(Θ)
Tri rapide	$n \log n$	n^2	$n \log n$
Tri fusion	$n \log n$	$n \log n$	$n \log n$
Tri par tas	$n \log n$	$n \log n$	$n \log n$
Tri par insertion	n	n^2	n^2
Tri par selection	n^2	n^2	n^2
tri à bulle	n	n^2	n^2
Recherche max Arbre Part Ord	1	1	1
Recherche ABR 1000 elt	1	25	/
Exponentiation rapide	$\log n$	$\log n$	$\log n$

En analyse d'algorithme, l'opération de base d'un tri est	La comparaison d'un élément du tableau à un autre élément
Le tri par la méthode du tri par sélection d'une table de 10 éléments nécessite	45 comparaisons dans le cas le plus défavorable Formule ($n(n-1)/2$) -> comparaison $n-1$ échanges dans le pire cas
La complexité du tri par sélection est	$\Theta(n^2)$ et $O(n^2)$
Le tri par la méthode du tri par insertion d'une table de 10 éléments nécessite	Cas favorable : il y a $n-1$ comparaisons et au plus n affectations Cas défavorable : $n/2$ affectations et comparaisons 9 comparaisons dans le cas le plus favorable
La complexité du tri par insertion est	$O(n^2)$
La complexité du tri fusion est	$\Theta(n \log(n))$
Une recherche dans un arbre AVL contenant 1000 éléments effectuée au plus	10
La recherche de l'élément de valeur maximale dans un arbre partiellement ordonné contenant n éléments est en	$O(1)$
Un programme dont la complexité est en $\Theta(2N)$ résout en 1 h un problème de taille N sur un ordinateur, quelle taille de problème pourra-t-on traiter, avec le même programme exécuté sur un ordinateur, 1000=k fois plus rapide	$N+10$ $= N+\log_2(k)$
La complexité du tri par tas d'un tableau n éléments est en	$\Theta(n \log(n))$ et $\Omega(n \log n)$

Soit la déclaration : <code>char *n = "12";</code> alors	<code>n[1]</code> est un caractère qui vaut '2'
--	---

<code>int * p = (int *)calloc (3, sizeof(int)) ;</code>	<code>*p+2</code> vaut 2
---	--------------------------

La recherche de l'élément de valeur maximale dans un arbre partiellement ordonné contenant n éléments est en	$O(1)$
--	--------

Un programme dont la complexité est en $\Theta(2N)$ résout en 1 h un problème de taille N sur un ordinateur, quelle taille de problème pourra-t-on traiter, avec le même programme exécuté sur un ordinateur, 250 fois plus rapide	$=N+8$
La complexité du tri par tas d'un tableau n éléments est en	$\Theta(n \log(n))$ et $\Omega(n \log n)$

