

Algorithmique (*Info0401*)

Dossier d'Algorithmique

*Dupont Corentin
Lacroix Owen*

Table des matières

Table des matières	2
I. Information	3
II. Généralité	3
III. Matrice	9
IV. Tableau	19
V. Trie	30
VI. Pile	39
VII. Liste chaînée	51
VIII. Arbre	72
IX. Huffman.....	89

I. Information

Certains codes en C ne sont pas exactement le même que celui écrit en algorithme car ils sont améliorés. De plus dans les codes algo nous ne traitons pas forcément les erreurs

II. Généralité

Nombre parfait

Algorithme:

Variables :

Entier : nb, somme, i ;

Début :

Afficher (« Entrez un nombre entier »);

Lire(nb);

somme <= 0;

Pour i allant de 1 à nb Faire:

Si nb % i = 0 Alors:

somme <= somme + i;

Fin Si

Fin pour

Si nb = somme Alors:

Afficher (« ce nombre est parfait ») ;

Sinon :

Afficher (« ce nombre n'est pas parfait ») ;

Fin Si

Fin

Code C :

```
#include<stdio.h>
```

```
int main(){
```

```
    int nb, i, y;
```

```
    int somme = 0;
```

```
    do{
```

```
        printf("Entrez une valeur max pour la recherche de nombre parfait: \n");
```

```
        scanf("%d", &nb);
```

```
    } while (nb < 0);
```

```
    for (y = 0; y < nb; y++){
```

```
        for (i = 1; i < y; i++) if(y % i == 0) somme += i;
```

```
        if (y == somme) printf("%d\n", y);
```

```
        somme = 0;
```

```
    }
```

```
    return 0;
```

```
}
```

Complexité : $O(N)$

```
Entrez une valeur max pour la recherche de number parfait:
```

```
28
```

```
0
```

```
6
```

Taille chaîne de caractère

<p><u>Algorithme:</u></p> <p><u>Variables :</u></p> <p>Char ch[100] ; Char *p; Entier: l;</p> <p><u>Début :</u></p> <p> Afficher("Entrez maximum 100 caractères") ; Lire(ch) ;</p> <p> <u>Pour</u> (p=ch; *p; p++); <u>Fin pour</u></p> <p> l <= p - ch;</p> <p> Afficher (" Longueur de la chaine : %d ", l) ; <u>Fin</u></p>	<p><u>Code C :</u></p> <pre>int main(){ char ch[100]; char *p; int l; printf("Entrez une chaine de caractere (max 100): \n"); scanf("%s", ch); for(p = ch; *p; p++); l = p - ch; printf("La taille de la chaine est: %d\n", l); return 0; }</pre>
--	--

Complexité : O(N)

```
Entrez une chaine de caractere (max 100):  
Bonjour  
La taille de la chaine est: 7
```

Fibonnaci (vecteur)

Algorithme :

Variables :

Entier : n, i, F[100] ;

Début :

Afficher (« Entrez la taille du tableau : ») ;

Lire(n) ;

F[0] <= 0 ;

Afficher(F[0]) ;

F[1] <= 1 ;

Afficher(F[1]) ;

Pour i allant de 2 à n Faire:

F[i] <= F[i-1] + F[i-2] ;

Afficher(F[i]) ;

Fin pour

Fin

Code C :

```
#include <stdio.h>
```

```
#define MAX 100
```

```
int main(){
```

```
    int F[MAX], i, size;
```

```
    do{
```

```
        printf("Entrez une taille <%d:\n", MAX);
```

```
        scanf("%d", &size);
```

```
    } while (size < 0 || size > MAX);
```

```
    F[0] = 0;
```

```
    printf("f(0) = %d \n", F[0]);
```

```
    F[1] = 1;
```

```
    printf("f(1) = %d \n", F[1]);
```

```
    for(i = 2; i < size; i++){
```

```
        F[i] = F[i-1] + F[i-2];
```

```
        printf("f(%d) = %d \n", i, F[i]);
```

```
    }
```

```
    return 0;
```

```
}
```

Complexité : O(N)

Entrez une taille <100:

17

f(0) = 0

f(1) = 1

f(2) = 1

f(3) = 2

f(4) = 3

f(5) = 5

f(6) = 8

f(7) = 13

f(8) = 21

f(9) = 34

f(10) = 55

f(11) = 89

f(12) = 144

f(13) = 233

f(14) = 377

f(15) = 610

f(16) = 987

Fibonnaci (3 variables)

Algorithme :

Variables :

Entier : n, nb1, nb2, nb3, i ;

Début :

n <= 0;

Afficher (« Entrez une valeur : ») ;

Lire(n);

nb1 <= 0;

Afficher (nb1);

nb2 <= 1

Afficher (nb2);

Pour i allant de 2 à n Faire:

nb3 <= nb1 + nb2;

nb1 <= nb2;

nb2 <= nb3;

Afficher (nb3);

Fin pour

Fin

Code C :

```
#include <stdio.h>
```

```
int main(){
```

```
    int i, size, nb1, nb2, nb3;
```

```
    printf("Entrez une valeur \n");
```

```
    scanf("%d", &size);
```

```
    nb1 = 0;
```

```
    printf("f(0) = %d \n", nb1);
```

```
    nb2 = 1;
```

```
    printf("f(1) = %d \n", nb2);
```

```
    for(i = 2; i < size; i++){
```

```
        nb3 = nb1 + nb2;
```

```
        nb1 = nb2;
```

```
        nb2 = nb3;
```

```
        printf("f(%d) = %d \n", i, nb3);
```

```
    }
```

```
    return 0;
```

```
}
```

Complexité : $O(N)$

Entrez une valeur

9

f(0) = 0

f(1) = 1

f(2) = 1

f(3) = 2

f(4) = 3

f(5) = 5

f(6) = 8

f(7) = 13

f(8) = 21

Fonction fibonnaci (récursive)

<p><u>Algorithme :</u></p> <p><u>Fonction</u> fib(x : entier) : entier</p> <p><u>Début :</u></p> <p> <u>Si</u> x <= 1 <u>Alors :</u></p> <p> Retourne x ;</p> <p> <u>Sinon :</u></p> <p> Retourne fib(x-1) + fic(x-2) ;</p> <p> <u>Fin Si</u></p> <p><u>Fin</u></p> <p><u>Variables :</u></p> <p>Entier : n;</p> <p><u>Début :</u></p> <p> <u>TantQue</u> n<0 <u>Faire :</u></p> <p> Afficher(« Fibonnaci de ? ») ;</p> <p> Lire(« %d », &n) ;</p> <p> <u>Fin tant que</u></p> <p> Afficher(« Fibonnaci de %d : %d », n, fib(n)) ;</p> <p><u>Fin</u></p>	<p><u>Code C :</u></p> <pre>#include <stdio.h> int fib(int n){ if (n <= 1) return n; return fib(n-1) + fib(n-2); } int main() { int nb; do{ printf("Fibonnaci de ? \n"); scanf("%d", &nb); } while (nb < 0); printf("Fib de %d : %d", nb, fib(nb)); return 0; }</pre>
---	---

Complexité : O(N)

```
Fibonnaci de ?
8
Fib de 8 : 21
```

Fonction d'Ackermann (récuratif)

<u>Algorithme :</u>	<u>Code C :</u>
<p>Fonction ackermann(entier m, entier n) : entier</p> <p><u>Si</u> (m==0){ retourner n+1 ; <u>Sinon Si</u> (n==0){ retourner ackermann(m-1,1) ; <u>Sinon</u> retourner ackermann(m-1, ackermann(m, n-1) ; <u>Fin Si</u> <u>Fin</u></p> <p><u>Début :</u> Afficher(«-Ackermann rec- \n ») ; Afficher(« A(2,2) =%d\n », ackermann(2,2)) ; Afficher(« A(3,3) =%d\n », ackermann(3,3)) ; <u>Fin</u></p>	<pre>#include <stdio.h> int ackermann(int m, int n){ if(m==0){ return n+1; }else if(n==0){ return ackermann(m-1,1); }else{ return ackermann(m-1,ackermann(m,n-1)); } } int main(){ printf("--Ackermann rec--\n"); printf("A(2,2) = %d\n", ackermann(2,2)); printf("A(3,3) = %d\n", ackermann(3,3)); return 0; }</pre>

Complexité : $O(2\log(N))$

```
--Ackermann rec--
A(2,2) = 7
A(3,3) = 61
```


III. Matrice

Intersection entre 2 images binaires

Algorithme:

Variables :

Définir : MAX = 2

Entier mat1, mat2, inter, i, j,

Début :

Afficher ("Première matrice binaire:");

Pour i allant de 0 à MAX Faire:

Pour j allant de 0 à MAX Faire:

Tant que (mat1[i][j]<0) OU
(mat1[i][j]>1) Faire:

Afficher("mat1[i][j]");

Lire(mat1[i][j]);

Fin tant que

Fin pour

Fin pour

Afficher ("Deuxième matrice
binaire:");

Pour i allant de 0 à MAX Faire:

Pour j allant de 0 à MAX Faire:

Tant que (mat2[i][j]<0) OU
(mat2[i][j]>1) Faire:

Afficher("mat2[i][j]");

Lire(mat2[i][j]);

Fin tant que

Fin pour

Fin pour

Pour i allant de 0 à 2 Faire:

Pour j allant de 0 à 2 Faire:

Si ((mat1[i][j]==mat2[i][j]) ET
mat1[i][j]==1) Alors :

inter[i][j]<=1;

Sinon Si ((mat1[i][j]==mat2[i][j])

ET mat1[i][j]==0) Alors :

inter[i][j] <=0;

Sinon

inter[i][j] <=0;

Fin si

Fin pour

Fin pour

Afficher("Matrice 1 :");

Pour i allant de 0 à MAX Faire:

Pour j allant de 0 à MAX Faire:

Afficher("mat1[i][j]");

Fin pour

Afficher ("")

Fin pour

Code C :

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#define MAX 2
```

```
int main() {
```

```
    int mat1[MAX][MAX];
```

```
    int mat2[MAX][MAX];
```

```
    int inter[MAX][MAX];
```

```
    int i,j;
```

```
    printf("Premiere matrice binaire: \n");
```

```
    for(i=0;i<MAX;i++){
```

```
        for(j=0;j<MAX;j++){
```

```
            do{
```

```
                printf("mat1[%d][%d]", i, j);
```

```
                scanf("%d",&mat1[i][j]);
```

```
            } while (mat1[i][j] < 0 || mat1[i][j] >
```

```
1);
```

```
        }
```

```
    }
```

```
    printf("Seconde matrice binaire: \n");
```

```
    for(i=0;i<MAX;i++){
```

```
        for(j=0;j<MAX;j++){
```

```
            do{
```

```
                printf("mat2[%d][%d]", i, j);
```

```
                scanf("%d",&mat2[i][j]);
```

```
            } while (mat2[i][j] < 0 || mat2[i][j] >
```

```
1);
```

```
        }
```

```
    }
```

```
    for(i=0;i<MAX;i++){
```

```
        for(j=0;j<MAX;j++){
```

```
            if((mat1[i][j]==mat2[i][j]) &&  
mat1[i][j]==1){
```

```
                inter[i][j]=1;
```

```
            }else if ((mat1[i][j]==mat2[i][j]) &&  
mat1[i][j]==0) {
```

```
                inter[i][j]=0;
```

```
            }else {
```

```
                inter[i][j]=0;
```

```
            }
```

```
        }
```

```
    }
```

```
    printf("Matrice 1: \n");
```

```
    for(i=0;i<MAX;i++){
```

```
        for(j=0;j<MAX;j++){
```

<pre> Afficher("Matrice 2 :"); Pour i allant de 0 à MAX Faire: Pour j allant de 0 à MAX Faire: Afficher("mat2[i][j]"); Fin pour Afficher ("") Fin pour Afficher("Matrice intersection :"); Pour i allant de 0 à MAX Faire: Pour j allant de 0 à MAX Faire: Afficher("inter[i][j]"); Fin pour Afficher ("") Fin pour Fin </pre>	<pre> printf("%d ", mat1[i][j]); } printf("\n"); } printf("Matrice 2: \n"); for(i=0;i<MAX;i++){ for(j=0;j<MAX;j++){ printf("%d ", mat2[i][j]); } printf("\n"); } printf("Matrice intersection : \n"); for(i=0;i<MAX;i++){ for(j=0;j<MAX;j++){ printf("%d ", inter[i][j]); } printf("\n"); } return 0; } </pre>
---	---

```

Premiere matrice binaire:
mat1[0][0]0
mat1[0][1]0
mat1[1][0]1
mat1[1][1]1
Seconde matrice binaire:
mat2[0][0]1
mat2[0][1]0
mat2[1][0]0
mat2[1][1]1
Matrice 1:
0 0
1 1
Matrice 2:
1 0
0 1
Matrice intersection :
0 0
0 1

```

Union entre 2 images binaires

Algorithme:

Variables :

Définir : MAX = 2

Entier mat1, mat2, inter, i, j,

Début :

Afficher ("Première matrice binaire:");

Pour i allant de 0 à MAX Faire:

Pour j allant de 0 à MAX Faire:

Tant que (mat1[i][j]<0) OU (mat1[i][j]>1) Faire:

Afficher("mat1[i][j]");

Lire(mat1[i][j]);

Fin tant que

Fin pour

Fin pour

Afficher ("Deuxième matrice binaire:");

Pour i allant de 0 à MAX Faire:

Pour j allant de 0 à MAX Faire:

Tant que (mat2[i][j]<0) OU (mat2[i][j]>1) Faire:

Afficher("mat2[i][j]");

Lire(mat2[i][j]);

Fin tant que

Fin pour

Fin pour

Pour i allant de 0 à 2 Faire:

Pour j allant de 0 à 2 Faire:

Si ((mat1[i][j] == mat2[i][j]) ET mat1[i][j] == 1) Alors:

inter[i][j] <= 1;

Sinon Si ((mat1[i][j]==mat2[i][j])

ET mat1[i][j]==0)

inter[i][j] <= 0;

Sinon

inter[i][j] <= 1;

Fin si

Fin pour

Fin pour

Afficher("Matrice 1 :");

Pour i allant de 0 à MAX Faire:

Pour j allant de 0 à MAX Faire:

Afficher("mat1[i][j]");

Fin pour

Afficher ("")

Fin pour

Afficher("Matrice 2 :");

Pour i allant de 0 à MAX Faire:

Code C :

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#define MAX 2
```

```
int main() {
```

```
    int mat1[MAX][MAX];
```

```
    int mat2[MAX][MAX];
```

```
    int inter[MAX][MAX];
```

```
    int i,j;
```

```
    printf("Premiere matrice binaire: \n");
```

```
    for(i=0;i<MAX;i++){
```

```
        for(j=0;j<MAX;j++){
```

```
            do{
```

```
                printf("mat1[%d][%d]", i, j);
```

```
                scanf("%d",&mat1[i][j]);
```

```
            } while (mat1[i][j] < 0 || mat1[i][j] >
```

```
1);
```

```
        }
```

```
    }
```

```
    printf("Seconde matrice binaire: \n");
```

```
    for(i=0;i<MAX;i++){
```

```
        for(j=0;j<MAX;j++){
```

```
            do{
```

```
                printf("mat2[%d][%d]", i, j);
```

```
                scanf("%d",&mat2[i][j]);
```

```
            } while (mat2[i][j] < 0 || mat2[i][j] >
```

```
1);
```

```
        }
```

```
    }
```

```
    for(i=0;i<2;i++){
```

```
        for(j=0;j<2;j++){
```

```
            if((mat1[i][j]==mat2[i][j]) &&
```

```
mat1[i][j]==1){
```

```
                inter[i][j]=1;
```

```
            }else if((mat1[i][j]==mat2[i][j]) &&
```

```
mat1[i][j]==0){
```

```
                inter[i][j]=0;
```

```
            }else {
```

```
                inter[i][j]=1;
```

```
            }
```

```
        }
```

```
    }
```

```
    printf("Matrice 1: \n");
```

```
    for(i=0;i<MAX;i++){
```

```
        for(j=0;j<MAX;j++){
```

```
            printf("%d ", mat1[i][j]);
```

```
        }
```

<u>Pour j allant de 0 à MAX Faire:</u> Afficher("mat2[i][j]"); <u>Fin pour</u> Afficher ("") <u>Fin pour</u> Afficher("Matrice union :"); <u>Pour i allant de 0 à MAX Faire:</u> <u>Pour j allant de 0 à MAX Faire:</u> Afficher("inter[i][j]"); <u>Fin pour</u> Afficher ("") <u>Fin pour</u> <u>Fin</u>	<pre> printf("\n"); } printf("Matrice 2: \n"); for(i=0;i<MAX;i++){ for(j=0;j<MAX;j++){ printf("%d ", mat2[i][j]); } printf("\n"); } printf("Matrice union : \n"); for(i=0;i<MAX;i++){ for(j=0;j<MAX;j++){ printf("%d ", inter[i][j]); } printf("\n"); } return 0; } </pre>
---	---

```

Premiere matrice binaire:
mat1[0][0]1
mat1[0][1]0
mat1[1][0]0
mat1[1][1]0
Seconde matrice binaire:
mat2[0][0]0
mat2[0][1]0
mat2[1][0]1
mat2[1][1]0
Matrice 1:
1 0
0 0
Matrice 2:
0 0
1 0
Matrice avec AND :
1 0
1 0

```

Complément d'une image

Algorithme:

Variables :

Définir : MAX = 2

Entier: mat[MAX][MAX], comp[MAX][MAX], i, j,

Début :

```

Afficher("Matrice");
Pour i allant de 0 à MAX Faire :
    Pour j allant de 0 à MAX Faire :
        Tant que (mat[i][j] < 0 OU
mat[i][j] > 1) Faire :
            Afficher("mat[i][j]");
            Lire(mat[i][j]);
        Fin tant que
    Fin pour
Fin pour

Pour i allant de 0 à MAX Faire :
    Pour J allant de 0 à MAX Faire :
        Si (mat[i][j] == 1) Alors :
            comp[i][j] <= 0;
        Sinon :
            comp[i][j] <= 1;
        Fin si
    Fin pour
Fin pour

Afficher("Matrice : ");
Pour i allant de 0 à MAX Faire :
    Pour j allant de 0 à MAX Faire :
        Afficher("comp[i][j]");
    Fin pour
Afficher("");
Fin pour

Afficher("Complémentaire : ");
Pour i allant de 0 à MAX Faire :
    Pour j allant de 0 à MAX Faire :
        Afficher("comp[i][j]");
    Fin pour
Afficher("");
Fin pour

```

Fin

Code C :

```

#include <stdio.h>
#include <stdlib.h>

#define MAX 2

int main() {

    int mat[MAX][ MAX];
    int comp[MAX][ MAX];
    int i, j;

    printf("Matrice: \n");
    for(i=0;i<MAX;i++){
        for(j=0;j<MAX;j++){
            do{
                printf("mat1 [%d] [%d]", i, j);
                scanf("%d", &mat[i][j]);
            } while (mat[i][j] < 0 || mat[i][j] > 1);
        }
    }

    for(i=0;i<MAX;i++){
        for(j=0;j<MAX;j++){
            if(mat[i][j]==1){
                comp[i][j]=0;
            }else {
                comp[i][j]=1;
            }
        }
    }

    printf("Matrice : \n");
    for(i=0;i<MAX;i++){
        for(j=0;j<MAX;j++){
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }

    printf("Complémentaire: \n");
    for(i=0;i<MAX;i++){
        for(j=0;j<MAX;j++){
            printf("%d ", comp[i][j]);
        }
        printf("\n");
    }

    return 0;
}

```

```
Matrice:
mat1[0][0]1
mat1[0][1]0
mat1[1][0]1
mat1[1][1]1
Matrice :
1 0
1 1
Complementaire:
0 1
0 0
```

Maximum et minimum entre 2 images

Algorithme:

Variables :

Définir : MAX = 2

Entier : mat1[MAX][MAX], mat2[MAX][MAX],
max[MAX][MAX], min[MAX][MAX], i, j ;

Début :

Afficher("Première matrice binaire: ");

Pour i allant de 0 à MAX Faire:

Pour j allant de 0 à MAX Faire:

Afficher("mat1 [i] [j]");

Lire(mat1 [i] [j]);

Fin pour

Fin pour

Afficher("Deuxieme matrice binaire:
");

Pour i allant de 0 à MAX Faire:

Pour j allant de 0 à MAX Faire:

Afficher("mat2[i] [j]");

Lire(mat2[i] [j]);

Fin pour

Fin pour

Pour i allant de 0 à MAX Faire:

Pour j allant de 0 à MAX Faire:

Si mat1 [i] [j] <= mat2[i] [j] Alors:
min[i] [j] <= mat2[i] [j];

Sinon

min[i] [j] <= mat1 [i] [j];

Fin si

Fin pour

Fin pour

Afficher("Matrice 1: ");

Pour i allant de 0 à MAX Faire:

Pour j allant de 0 à MAX Faire:

Afficher("mat1 [i] [j]");

Fin pour

Afficher("");

Fin pour

Afficher("Matrice 2: ");

Pour i allant de 0 à MAX Faire:

Pour j allant de 0 à MAX Faire:

Afficher("mat2[i] [j]");

Fin pour

Afficher("");

Fin pour

Afficher("Min: ");

Pour i allant de 0 à MAX Faire:

Pour j allant de 0 à MAX Faire:

Code C :

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#define MAX 2
```

```
int main() {
```

```
    int mat1[MAX][MAX];
```

```
    int mat2[MAX][MAX];
```

```
    int max[MAX][MAX];
```

```
    int min[MAX][MAX];
```

```
    int i,j;
```

```
    printf("Premiere matrice binaire: \n");
```

```
    for(i=0;i<MAX; i++){
```

```
        for(j=0;j<MAX; j++){
```

```
            printf("mat1[%d][%d]", i, j);
```

```
            scanf("%d",&mat1[i][j]);
```

```
        }
```

```
    }
```

```
    printf("Seconde matrice binaire: \n");
```

```
    for(i=0;i<MAX; i++){
```

```
        for(j=0;j<MAX; j++){
```

```
            printf("mat2[%d][%d]", i, j);
```

```
            scanf("%d",&mat2[i][j]);
```

```
        }
```

```
    }
```

```
    for(i=0;i<MAX; i++){
```

```
        for(j=0;j<MAX; j++){
```

```
            if(mat1[i][j] <= mat2[i][j]){
```

```
                max[i][j] = mat2[i][j];
```

```
            }else{
```

```
                max[i][j] = mat1[i][j];
```

```
            }
```

```
        }
```

```
    }
```

```
    for(i=0;i<MAX; i++){
```

```
        for(j=0;j<MAX; j++){
```

```
            if(mat1[i][j]>=mat2[i][j]){
```

```
                min[i][j]=mat2[i][j];
```

```
            }else{
```

```
                min[i][j]=mat1[i][j];
```

```
            }
```

```
        }
```

```
    }
```

```
    printf("Matrice 1: \n");
```

```
    for(i=0;i<MAX; i++){
```

```
        for(j=0;j<MAX; j++){
```

<pre> Afficher("min[i][j]"); Fin pour Afficher(""); Fin pour Afficher("Max: "); Pour i allant de 0 à MAX Faire: Pour j allant de 0 à MAX Faire: Afficher("Max[i][j]"); Fin pour Afficher(""); Fin pour Fin </pre>	<pre> printf("%d ", mat1[i][j]); } printf("\n"); } printf("Matrice 2: \n"); for(i=0;i<MAX; i++){ for(j=0;j<MAX; j++){ printf("%d ", mat2[i][j]); } printf("\n"); } printf("Min: \n"); for(i=0;i<MAX; i++){ for(j=0;j<MAX; j++){ printf("%d ", min[i][j]); } printf("\n"); } printf("Max: \n"); for(i=0;i<MAX; i++){ for(j=0;j<MAX; j++){ printf("%d ", max[i][j]); } printf("\n"); } return 0; } </pre>
---	---

Complexité : N^2

```

Premiere matrice binaire:
mat1[0][0]1
mat1[0][1]0
mat1[1][0]0
mat1[1][1]1
Seconde matrice binaire:
mat2[0][0]1
mat2[0][1]1
mat2[1][0]0
mat2[1][1]1
Matrice 1:
1 0
0 1
Matrice 2:
1 1
0 1
Min:
1 0
0 1
Max:
1 1
0 1

```


Multiplication matrice

Algorithme :

Variables :

définir : MAX = 3

Entier : i, j, k ;

Début :

```

Afficher("Première matrice binaire: ")
Pour i allant de 0 à MAX Faire :
    Pour j allant de 0 à MAX Faire :
        Afficher ("mat1 [i] [j]");
        Lire (mat1 [i] [j]);
    Fin Pour
Fin Pour

Afficher("deuxième matrice binaire: ")

Pour i allant de 0 à MAX Faire :
    Pour j allant de 0 à MAX Faire :
        Afficher ("mat2[i] [j]");
        Lire (mat2[i] [j]);
    Fin Pour
Fin Pour

Pour i allant de 0 à MAX Faire :
    Pour j allant de 0 à MAX Faire :
        mul[i] [j] ← 0;
        Pour k allant de 0 à MAX Faire :
            mul[i] [j] = mul[i] [j] + mat1 [i] [k] * mat2[k] [j] ;
        Fin Pour
    Fin Pour
Fin Pour
Afficher('Résultat :') ;
Pour i allant de 0 à MAX Faire :
    Pour j allant de 0 à MAX Faire :
        Afficher('mul[i] [j]') ;
    Fin Pour
Fin

```

Code C :

```

#include<stdio.h>

#define MAX 3

int main(){
    int mat1 [MAX][MAX], mat2[MAX][MAX],
    mul[MAX][MAX];

    int i,j,k;

    printf("Première matrice\n");
    for(i = 0; i < MAX; i++){
        for(j = 0; j < MAX; j++){
            printf("mat1 [%d] [%d] : " , i, j);
            scanf("%d",&mat1 [i] [j]);
        }
    }

    printf("Seconde matrice\n");
    for(i = 0; i < MAX; i++){
        for(j = 0; j < MAX; j++){
            printf("mat2[%d] [%d] : " , i, j);
            scanf("%d",&mat2[i] [j]);
        }
    }

    //Traitement
    for(i = 0; i < MAX; i++){
        for(j = 0; j < MAX; j++){
            mul[i] [j] = 0;
            for(k = 0; k < MAX; k++){
                mul[i] [j] += mat1 [i] [k] * mat2[k] [j];
            }
        }
    }

    printf("Résultat : \n");
    for(i = 0; i < MAX; i++){
        for(j = 0; j < MAX; j++){
            printf("%d\t",mul[i] [j]);
        }
        printf("\n");
    }
    return 0;
}

```

Complexité : N^3

Première matrice

mat1[0][0] : 2

mat1[0][1] : 3

mat1[0][2] : 4

mat1[1][0] : 5

mat1[1][1] : 6

mat1[1][2] : 7

mat1[2][0] : 8

mat1[2][1] : 9

mat1[2][2] : 1

Seconde matrice

mat2[0][0] : 1

mat2[0][1] : 0

mat2[0][2] : 2

mat2[1][0] : 5

mat2[1][1] : 6

mat2[1][2] : 8

mat2[2][0] : 9

mat2[2][1] : 7

mat2[2][2] : 0

Résultat :

53	46	28
----	----	----

98	85	58
----	----	----

62	61	88
----	----	----

IV. Tableau

Normalisation d'un tableau

Algorithme:

Variables :

Réel : t1, t2, max ;

Entier : taille, i ;

Début :

Afficher (« Entrez une taille : ») ;

Lire(taille);

Pour i allant de 1 à taille Faire :

Afficher (« Entrez une valeur
pour t1[i] : ») ;

Lire(t1[i]);

Fin pour

Afficher (« Affichage de t1: ») ;

Pour i allant de 1 à taille Faire:

Afficher (t1[i]) ;

Fin pour

max <= t1[0];

Pour i allant de 1 à taille Faire :

Sj t1[i] > max Alors:

max <= t1[i];

Fin si

Fin pour

Afficher (« Max : » + max) ;

Pour i allant de 1 à taille Faire:

t2[i] <= t1[i] / max;

Fin pour

Afficher (« Affichage de t2: »);

Pour i allant de 1 à taille Faire:

Afficher (t2[i]);

Fin pour

Fin

Code C :

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(){  
    int size, i;  
    float *tab, *tab2, max;
```

```
    do{
```

```
        printf("Entrer une taille : ");
```

```
        scanf("%d", &size);
```

```
    } while (size < 0);
```

```
    tab = malloc(size * sizeof(float));
```

```
    tab2 = malloc(size * sizeof(float));
```

```
    if(tab == NULL || tab2 == NULL){
```

```
        printf("Error!");
```

```
        exit(0);
```

```
    }
```

```
    for (i = 0; i < size; i++){
```

```
        printf("Entrer une valeur pour t[%d]: \n",
```

```
    i);
```

```
        scanf("%f", &tab[i]);
```

```
    }
```

```
    printf("Tableau 1: \n");
```

```
    for (i = 0; i < size; i++){
```

```
        printf("t1[%i] = %.2f\n", i, tab[i]);
```

```
    }
```

```
    max = tab[0];
```

```
    for (i = 0; i < size; i++){
```

```
        if(tab[i] >= max) max = tab[i];
```

```
    }
```

```
    printf("Max : %.2f", max);
```

```
    for (i = 0; i < size; i++){
```

```
        tab2[i] = tab[i] / max;
```

```
    }
```

```
    printf("Tableau 2: \n");
```

```
    for (i = 0; i < size; i++){
```

```
        printf("t2[%i] = %.2f\n", i, tab2[i]);
```

```
    }
```

```
}
```

Complexité : N

Concaténation de tableau

Algorithme:

Variables :

Entier : tabA[100], tabB[50];
Entier : n, m, i;

Début :

Afficher(" Entrer une taille pour le
tableau A avec n < 50 ");
Lire(n);

afficher(" Entrer une taille pour le
tableau B avec m < 50 ");
Lire(m);

Pour i allant de 1 à n Faire :
Afficher ("élément %d :", i);
Lire ("%d", tabA+i);

Fin pour

Pour i allant de 0 à m Faire :
Afficher ("élément %d :", i);
Lire ("%d", tabB+i);

Fin pour

Pour i allant de 0 à m Faire :
Afficher ("%d", *(tabA+i));

Fin pour

Pour i allant de 0 à m Faire :
Afficher ("%d", *(tabB+i));

Fin pour

//Copie de B à la fin de A :
Pour i allant de 1 à n Faire :
*(tabA+n+i) <= *(tabB+i);

Fin pour

n=n+m ;

Pour i allant de 1 à N Faire;
Afficher (" %d ", *(tabA+i));

Fin pour

Fin

Code C :

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define MAXA 100
#define MAXB 50
```

```
int main(){
```

```
    int tabA[MAXA], tabB[MAXB];
    int n, m, i;
```

```
    printf("Entrer une taille pour le tableau A
avec n < %d: \n", MAXA);
    scanf("%d", &n);
```

```
    printf("Entrer une taille pour le tableau B
avec m < %d: \n", MAXB);
    scanf("%d", &m);
```

```
    if(n < m){
        printf("Error N est plus petit que M !");
        exit(0);
    }
```

```
    printf("\nValeur tableau A: \n");
    for(i = 0; i < n ; i++){
        printf("tabA[%d]: ", i);
        scanf("%d", &tabA[i]);
    }
```

```
    printf("\nValeur tableau B: \n");
    for(i = 0; i < m ; i++){
        printf("tabB[%d]: ", i);
        scanf("%d", &tabB[i]);
    }
```

```
    printf("\nTableau A: \n");
    for(i = 0; i < n ; i++){
        printf("[%d] | ", *(tabA+i));
    }
```

```
    printf("\n");
    printf("\nTableau B: \n");
    for(i = 0; i < m ; i++){
        printf("[%d] | ", *(tabB+i));
    }
```

```
    for(i = 0; i < n ; i++){
        *(tabA+n+i) = *(tabB+i);
    }
```

```
    n = n + m;
```

	<pre> printf("\n"); printf("\nTableau A final: \n"); for(i = 0; i < n ; i++){ printf("[%d] ", *(tabA+i)); } return 0; } </pre>
--	--

Complexité : $O(N)$

```

PS C:\Users\owen1\Desktop\TPinfo0401\Tableau> & .\"Concatenation.exe"
Entrer une taille pour le tableau A avec n < 100:
3
Entrer une taille pour le tableau B avec m < 50:
7
Error N est plus petit que M !
PS C:\Users\owen1\Desktop\TPinfo0401\Tableau> & .\"Concatenation.exe"
Entrer une taille pour le tableau A avec n < 100:
7
Entrer une taille pour le tableau B avec m < 50:
5

Valeur tableau A:
tabA[0]: 1
tabA[1]: 45
tabA[2]: 6
tabA[3]: 85
tabA[4]: 42
tabA[5]: 21
tabA[6]: 1

Valeur tableau B:
tabB[0]: 25
tabB[1]: 65
tabB[2]: 95
tabB[3]: 75
tabB[4]: 35

Tableau A:
[1] | [45] | [6] | [85] | [42] | [21] | [1] |

Tableau B:
[25] | [65] | [95] | [75] | [35] |

Tableau A final:
[1] | [45] | [6] | [85] | [42] | [21] | [1] | [25] | [65] | [95] | [75] | [35] |

```

Tassement de tableau

Algorithme:

Variables :

Entier : a[50], x, *p1, *p2 ;

Début :

Afficher(" Dimension <50 ") ;
Lire(n) ;

Pour (p1=a ; p1<a+n ; p1++)

Faire :

Afficher("Elément %d", p1-a) ;
Lire("%d", p1) ;

Fin pour

Pour (p1=a ; p1<a+n ; p1++)

Faire :

Afficher("Elément %d", *p1) ;

Fin pour

Afficher("Entrer x") ;
Lire(x) ;

Pour (p1=p2=a ; p1<a+n ; p1++)

Faire :

*p2 = *p1 ;
Si (*p2 != x) Alors :
 p2++ ;

Fin si

Fin pour

n = p2 - a

Pour (p1=a ; p1<a+n ; p1++)

Faire :

Afficher("%d", *p1) ;

Fin pour

Fin

Code C :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX 10

int main(){
    int A[MAX]={0}, x, *P1, *P2, i;

    srand(time(NULL));
    x = rand()%10;
    P1 = A;
    P2 = A;

    for(i=0 ; i<MAX ; i++) *(A+i) = rand()%10;

    printf("\nx: %d\n", x);

    printf("\nTableau: \n");
    for( i=0 ; i<MAX ; i++){
        printf("[%d] | ", *(A+i));
    }

    for(i=0 ; i<MAX ; i++){
        *P1=*P2;
        if(*P2!=x)P1++;
        P2++;
    }

    printf("\nTableau sans %d : \n",x);
    for(i=0 ; i<(P1-A) ; i++) printf("[%d] | ",
*(A+i));

    return 0;
}
```

Complexité : $O(N)$

x: 7

Tableau:

[4] | [9] | [6] | [3] | [6] | [6] | [1] | [7] | [1] | [8] |

Tableau sans 7 :

[4] | [9] | [6] | [3] | [6] | [6] | [1] | [1] | [8] |

Inversion de tableau

Algorithme:

Variables :

Définir : MAX = 20

Entier : tab[MAX], size, tmp, i, *p1, *p2 ;

Début :

Tant que (size < 0 **OU** size > MAX) Faire:
Afficher("Entrez une taille <%d:",
MAX);

Lire(size);

Fin tant que

Pour i allant de 0 à size Faire:

Afficher("tab[%d]: ", i)

Lire(tab+i) ;

Fin pour

Afficher("Tableau :")

Pour i allant de 0 à size Faire:

Afficher("[%d] ", *(tab+i));

Fin pour

Pour j allant de n à 0 Faire:

P1 <= tab+i

P2 <= (tab+size-i-1);

tmp <= *p1

*p1 <= *p2

*p2 <= tmp

Fin pour

Afficher("Tableau inversé :")

Pour i allant de 0 à size Faire:

Afficher("[%d], *(tab+i));

Fin pour

Fin

Code C :

```
#include <stdio.h>
```

```
#define MAX 20
```

```
int main(){
```

```
    int tab[MAX], i, tmp, size, *p1, *p2;
```

```
    do{
```

```
        printf("Entrez une taille <%d: \n", MAX);
```

```
        scanf("%d", &size);
```

```
    } while (size < 0 || size > MAX);
```

```
    for(i = 0; i < size; i++){
```

```
        printf("tab[%d]: ", i);
```

```
        scanf("%d", (tab+i));
```

```
    }
```

```
    printf("\nTableau: \n");
```

```
    for(i = 0; i < size; i++){
```

```
        printf("[%d] | ", *(tab+i));
```

```
    }
```

```
    for(i = 0; i < size / 2; i++){
```

```
        p1 = (tab+i);
```

```
        p2 = (tab+size-i-1);
```

```
        tmp = *p1;
```

```
        *p1 = *p2;
```

```
        *p2 = tmp;
```

```
    }
```

```
    printf("\nTableau inverse: \n");
```

```
    for(i = 0; i < size; i++){
```

```
        printf("[%d] | ", *(tab+i));
```

```
    }
```

```
    return 0;
```

```
}
```

Complexité : N

```
Entrez une taille <20:
7
tab[0]: 7
tab[1]: 5
tab[2]: 1
tab[3]: 12
tab[4]: 59
tab[5]: 34
tab[6]: 45

Tableau normal:
[7] | [5] | [1] | [12] | [59] | [34] | [45] |
Tableau inverse:
[45] | [34] | [59] | [12] | [1] | [5] | [7] |
```


Majorité

Algorithme :

Variables :

Définir : MAX = 10

Entier : i, j, nb, max = 0, pos = -1, size, tab[MAX];

Début :

Tant que size < 0 OU size > MAX Faire :

Afficher(« Entrez une taille
<%d », MAX) ;

Lire(« %d », &size) ;

Fin tant que

Pour i allant de 0 à size Faire :

Afficher(« tab[i] : », i) ;

Lire(« %d », & tab[i]) ;

Fin pour

Pour i allant de 0 à size Faire :

nb <= 0;

Pour j allant de 0 à size Faire :

Si tab[i] = tab[j] Alors :

nb <= nb + 1 ;

Fin si

Fin pour

Si nb > max Alors :

max <= nb ;

pos <= i ;

Fin si

Fin pour

Si pos = -1 Alors :

Afficher(« Pas de majorité ») ;

Sinon :

Afficher(« Majorité : %d »,

tab[pos]) ;

Fin si

Fin

Code C :

```
#include <stdio.h>
```

```
#define MAX 10
```

```
int main() {
```

```
    int i, j, max = 0, pos = -1, size, tab[MAX];
```

```
    do {
```

```
        printf("Entrez une taille <%d)", MAX);
```

```
        scanf("%d", &size);
```

```
    } while (size < 0 || size > MAX);
```

```
    for(i = 0; i < size; i++){
```

```
        printf("tab[%d] : ", i);
```

```
        scanf("%d", &tab[i]);
```

```
    }
```

```
    for(i = 0; i < size; i++){
```

```
        int nb = 0;
```

```
        for(j = 0; j < size; j++){
```

```
            if(tab[i] == tab[j]) nb++;
```

```
        }
```

```
        if(nb > max){
```

```
            max = nb;
```

```
            pos = i;
```

```
        }
```

```
    }
```

```
    if(pos == -1 ) printf("Pas de majorite");
```

```
    else printf("Majorite : %d", tab[pos]);
```

```
    return 0;
```

```
}
```

Complexité : N^2

```
Entrez une taille <10 : 7
tab[0] : 1
tab[1] : 1
tab[2] : 7
tab[3] : 7
tab[4] : 2
tab[5] : 7
tab[6] : 7
Majorite : 7
```

Trie

Algorithme :

Variables :

Définir : MAX = 10
Entier : tab[MAX], tab2[MAX], tabFinal[MAX],
i, j, tmp, sizeTab1, sizeTab2, sizeTab3

Début :

Tant que sizeTab1 < 0 ou sizeTab1 > MAX Faire :
Afficher("Taille du second tableau
%d" MAX:);
Lire("%d" &sizeTab1);
Fin Tant que

Tant que sizeTab2 < 0 ou sizeTab2 > MAX Faire :
Afficher("Taille du second tableau
%d" MAX:);
Lire("%d" &sizeTab2);
Fin Tant que

Afficher("Saisie du premier tableau : ");
Pour i allant de 0 à sizeTab1 Faire :
Afficher("tab[%d", i);
Lire("%d", &tab[i]);
Fin Pour

Afficher("Saisie du second tableau : ");
Pour i allant de 0 à sizeTab2 Faire :
Afficher("tab2[%d", i);
Lire("%d", &tab2[i]);
Fin Pour

Pour i allant de 0 à sizeTab1 Faire :
tabFinal[i] ← tab[i];
Fin Pour

sizeTab3 ← sizeTab1 + sizeTab2;

Pour j allant de sizeTab1 à sizeTab3 &&
i allant de 0 à sizeTab2 Faire ;
tabFinal[j] ← tab2[i]
Fin Pour

Pour j allant de 1 à sizeTab3 Faire :
Pour i allant de 0 à sizeTab3 Faire :
Si tabFinal[i] > tabFinal[i+1] Alors :
tmp ← tabFinal[i];

Code C :

```
#include<stdio.h>
```

```
#define MAX 20
```

```
int main(){
```

```
int tab[MAX], tab2[MAX], tabFinal[MAX];  
int i, j, tmp, sizeTab1, sizeTab2, sizeTab3;
```

```
do{  
printf("Taille du premier tableau (<%d):  
", MAX);  
scanf("%d", &sizeTab1);  
} while (sizeTab1 < 0 || sizeTab1 > MAX);
```

```
do{  
printf("Taille du second tableau (<%d):  
", MAX);  
scanf("%d", &sizeTab2);  
} while (sizeTab2 < 0 || sizeTab2 > MAX);
```

```
printf("Saisi du premier tableau: \n");  
for(i = 0; i < sizeTab1; i++){  
printf("tab[%d] :", i);  
scanf("%d", &tab[i]);  
}
```

```
printf("Saisi du second tableau: \n");  
for(i = 0; i < sizeTab2; i++){  
printf("tab2[%d] :", i);  
scanf("%d", &tab2[i]);  
}
```

```
for(i = 0; i < sizeTab1; i++){  
tabFinal[i] = tab[i];  
}
```

```
sizeTab3 = sizeTab1 + sizeTab2;
```

```
for(i = 0, j = sizeTab1; j < sizeTab3 && i <  
sizeTab2; i++, j++){  
tabFinal[j] = tab2[i];  
}
```

```
for(j=1;j<=sizeTab3;j++){  
for(i=0;i<sizeTab3-1;i++){  
if (tabFinal[i] > tabFinal[i+1]) {  
tmp = tabFinal[i];  
tabFinal[i] = tabFinal[i+1];  
tabFinal[i+1] = tmp;
```

<pre> tabFinal[i]=tabFinal[i+1] ; tabFinal[i+1]=tmp ; Fin Si Fin Pour Fin Pour Afficher("Tableau trié et fusionné : ") ; Pour i allant de 0 à sizeTab3 Faire ; Afficher("[%d]", tabFinal[i]) ; Fin Pour Fin </pre>	<pre> } } printf("Tableau trie et fusionne : \n"); for(i = 0; i < sizeTab3; i++){ printf("[%d] ", tabFinal[i]); } return 0; } </pre>
---	--

```

Taille du premier tableau (<20): 7
Taille du second tableau (<20): 4
Saisi du premier tableau:
tab[0] :4
tab[1] :5
tab[2] :6
tab[3] :2
tab[4] :1
tab[5] :3
tab[6] :4
Saisi du second tableau:
tab2[0] :5
tab2[1] :7
tab2[2] :8
tab2[3] :9
Tableau trie et fusionne :
[1] | [2] | [3] | [4] | [4] | [5] | [5] | [6] | [7] | [8] | [9] |

```

Conversion décimale vers binaire

Algorithme :

Variables :

tab[16], val, i : entier

Début :

Afficher("Entrez une valeur");

Lire("%d", &val);

Tant Que val < 0 Faire :

 Tant Que val > 0 Faire ;

 i = 0

 tab[i] ← val%2;

 val ← val/2;

 i++

 Fin Tant Que

 Tant que i >= 0 Faire :

 Ecrire("[%d] | ", tab[i]);

 Fin Tant Que

Fin Tant Que

Fin

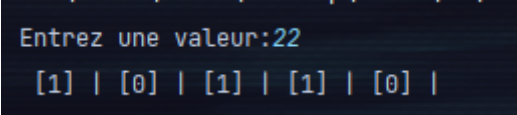
Code C :

```
int main() {
    int tab[16] = {0}, val, i;

    //Initialisation de la valeur à convertir
    do{
        printf("Entrez une valeur: ");
        scanf("%d", &val);
    } while (val < 0);

    //Conversion binaire
    for(i=0; val>0; i++){
        tab[i]=val%2;
        val=val/2;
    }

    //Affichage du tableau contenant le
    chiffre convertit en binaire
    for (i = i-1; i>=0; i--) {
        printf("[%d] | ", tab[i]);
    }
}
```



```
Entrez une valeur:22
[1] | [0] | [1] | [1] | [0] |
```

V. Trie

Trie à bulle

Algorithme :

Tableau : t ;
Entier : i, j, tmp, taille, finish = 1;

Début

Afficher(« Entrez une taille du tableau ») ;

Lire(taille);

Lire(tab)

Afficher(tab);

Tant que finish = 1 Faire:

finish <= 0;

Pour j allant de 0 à taille - 1 Faire :

Si tab[j]>tab[j+1] Alors :

tmp<=tab[j+1] ;

tab[j+1] <=tab[j];

tab[j] <=tmp;

finish <= 1;

Fin si

Fin pour

Fin tant que

Afficher(tab) ;

Fin

Code C :

```
int main() {
    int tab[10], i, j, tmp, size, finish = 1;

    do {
        printf("Entrez une taille <20\n");
        scanf("%d", &size);
    } while (size <= 0);

    for (i = 0; i < size; i++){
        printf("tab[%d]: ", i);
        scanf("%d", &tab[i]);
    }

    printf("Tableau:\n");
    for (i = 0; i < size; i++){
        printf("[%d] | ", tab[i]);
    }
    printf("\n");

    while (finish){
        finish = 0;
        for (j = 0; j < size-1; j++){
            if(tab[j] > tab[j+1]){
                tmp = tab[j+1];
                tab[j+1] = tab[j];
                tab[j] = tmp;

                finish = 1;
            }
        }
    }

    printf("Tableau trié:\n");
    for (i = 0; i < size; i++){
        printf("[%d] | ", tab[i]);
    }

    return 0;
}
```

Complexité : $O(N^2)$

Avant trie a bulle:

[25] | [41] | [2] | [3] | [9] | [8] |

Après trie a bulle:

[2] | [3] | [8] | [9] | [25] | [41] |

Trie couleur

Algorithme :

Tableau : t ;
Entier : i, j, k, taille;
Définir :

- BLEU = 1
- BANC = 2
- ROUGE = 3

Début

Afficher(« Entrez une taille du tableau ») ;
Lire(taille) ;
Lire(tab) ;
Afficher(tab) ;
i = 0
j = 0 ;
k = taille ;

Tant que (k != j) Faire :

Si(tab[j+1] = BLEU) Alors :

Echanger(tab[j+1], tab[i+1]) ;
i <= i + 1 ;
j <= j+1 ;

Sinon si (tab[j+1] = ROUGE) Alors :

Echanger(tab[j+1], tab[k]) ;
k <= k - 1 ;

Sinon :

j <= j+1 ;

Fin si

Fin tant que

Fin

Code C :

```
#include <stdio.h>
```

```
#define SIZE 9
```

```
int main() {  
    //Bleu = 1 | Blanc = 2 | Rouge = 3  
    int tab[SIZE] = {1, 3, 2, 2, 3, 1, 2, 3, 1};  
    int x, i = 0, j = 0, k = SIZE, temp;
```

```
    //Affichage avant
```

```
    printf("Avant trie : \n");  
    for(x = 0 ; x < SIZE ; x++){  
        printf("[%d] | ", tab[x]);  
    }  
    printf("\n");
```

```
    //Tri
```

```
    while(j <= k){  
        if(tab[j] == 2){  
            j++;  
        }  
        else if(tab[j] == 1){  
            temp = tab[i];  
            tab[i] = tab[j];  
            tab[j] = temp;  
            i++;  
            j++;  
        }  
        else{  
            temp = tab[j];  
            tab[j] = tab[k];  
            tab[k] = temp;  
            k--;  
        }  
    }  
}
```

```
    //Affichage
```

```
    printf("Après trie : \n");  
    for(x = 0 ; x < SIZE ; x++) {  
        printf("[%d] | ", tab[x]);  
    }  
}
```

```
    return 0;
```

```
}
```

Complexité : $O(N^2)$

Avant trie Hollandais :

[1] | [3] | [2] | [2] | [3] | [1] | [2] | [3] | [1] |

Après trie Hollandais :

[1] | [1] | [1] | [2] | [2] | [2] | [3] | [3] | [0] |

Trie rapide

Algorithme :

Fonction fast(Entier : *tab, Entier : size) : vide*

Entier : wall, current, pivot, tmp

Début

Si (size < 2) Alors :

Retourne() ;

Fin si

pivot <= tab[size-1] ;

wall <= current <= 0 ;

Tant que (current < size) Faire :

Si (tab[current] <= pivot) Alors :

Si (wall != current) Alors :

tmp <= tab[current] ;

tab[current] <= tab[wall] ;

tab[wall] <= tmp ;

Fin si

wall++ ;

Fin si

current++ ;

Fin tant que

fast(tab, wall-1) ;

fast(tab+wall-1, size-wall+1) ;

Fin

main

Entier : i, tab[20] = {10,5,8,9,3,4,6,1,2,7};

Define : SIZE 10

Début

Afficher("Avant le trie rapide");

Pour i allant de 0 à SIZE Faire:

Afficher("[%d] | ", tab[i]);

Fin pour

fast(tab, SIZE);

Afficher("Après le trie rapide");

Pour i allant de 0 à SIZE Faire:

Afficher("[%d] | ", tab[i]);

Fin pour

Fin

Code C :

```
#include <stdio.h>
```

```
#define SIZE 10
```

```
/**
```

```
 * Fonction recursive pour faire le trie ra-  
pide
```

```
 * @param tab le tableau
```

```
 * @param size la taille du tableau
```

```
 */
```

```
void fast(int *tab, int size) {
```

```
    int wall, current, pivot, tmp;
```

```
    if (size < 2) return;
```

```
    // On prend comme pivot l'element le  
    plus à droite
```

```
    pivot = tab[size - 1];
```

```
    wall = current = 0;
```

```
    while (current < size) {
```

```
        if (tab[current] <= pivot) {
```

```
            if (wall != current) {
```

```
                tmp=tab[current];
```

```
                tab[current]=tab[wall];
```

```
                tab[wall]=tmp;
```

```
            }
```

```
            wall ++;
```

```
        }
```

```
        current ++;
```

```
    }
```

```
    fast(tab, wall - 1);
```

```
    fast(tab + wall - 1, size - wall + 1);
```

```
}
```

```
int main() {
```

```
    int i, tab[20] = {10,5,8,9,3,4,6,1,2,7};
```

```
    printf("Avant le trie rapide: \n");
```

```
    for(i=0 ; i < SIZE ; i++){
```

```
        printf("[%d] | ",tab[i]);
```

```
    }
```

```
    //traitement
```

```
    fast(tab, SIZE);
```

```
    printf("\nAprès le trie rapide: \n");
```

```
    for(i=0 ; i < SIZE ; i++){
```

```
        printf("[%d] | ",tab[i]);
```

```
    }
```

```
}
```

Complexité : $O(N^2)$

Avant le trie rapide:

[10] | [5] | [8] | [9] | [3] | [4] | [6] | [1] | [2] | [7] |

Après le trie rapide:

[1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] |

Trie par insertion

Algorithme :

Entier : i, j, tmp, tab[20] = {10,5,8,9,3,4,6,1,2,7};
Define : SIZE 10

Début

Afficher("Avant le trie insertion");

Pour i allant de 0 à SIZE Faire:

Afficher("[%d] | ", tab[i]);

Fin pour

Pour i allant de 0 à SIZE Faire:

j <= i;

Tant que (j > 0 ET tab[j-1] > tab[j]) Faire:

Tmp <= tab[j];

tab[j] <= tab[j-1];

tab[j-1] <= tmp;

j--;

Fin tant que

Fin pour

Afficher("Après le trie insertion");

Pour i allant de 0 à SIZE Faire:

Afficher("[%d] | ", tab[i]);

Fin pour

Fin

Code C :

```
#include <stdio.h>
```

```
#define SIZE 10
```

```
int main() {  
    int i, j, tmp, tab[20] = {10,5,8,9,3,4,6,1,2,7};
```

```
    //Affichage
```

```
    printf("Avant le trie insertion: \n");
```

```
    for(i=0 ; i < SIZE ; i++){  
        printf("[%d] | ",tab[i]);  
    }
```

```
    //traitement
```

```
    for(i=1; i<= SIZE-1; i++){  
        j = i;  
        while (j > 0 && tab[j-1] > tab[j]) {  
            tmp = tab[j];  
            tab[j] = tab[j-1];  
            tab[j-1] = tmp;  
            j--;  
        }  
    }
```

```
    //Affichage
```

```
    printf("\nAprès le trie insertion: \n");
```

```
    for(i=0 ; i < SIZE ; i++){  
        printf("[%d] | ",tab[i]);  
    }  
}
```

Complexité : $O(N^2)$

```
Avant le trie insertion:
```

```
[10] | [5] | [8] | [9] | [3] | [4] | [6] | [1] | [2] | [7] |
```

```
Après le trie insertion:
```

```
[1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] |
```

Trie fusion

Algorithme :

Déclarations :

a[SIZE_A] ← {0,1,2,4,5,8,9,512} : entier
 b[SIZE_B] ←
 {3,4,5,6,8,10,11,42,65,89,101} : entier
 fullLength ← SIZE_A + SIZE_B : entier
 res[fullLength] : entier
 i, alIndex, blIndex : entier

Début :

Afficher(« Tableau 1 :\n ») ;
 Pour i allant de 0 à SIZE_A faire :
 Afficher(« [%d] | », a[i]) ;
 Fin Pour
 Afficher(« \n ») ;
 Afficher(« Tableau 2 :\n ») ;
 Pour i allant de 0 à SIZE_B faire :
 Afficher(« [%d] », b[i]) ;
 Fin Pour
 Afficher(« \n ») ;
 alIndex ← 0 ;
 blIndex ← 0 ;
 Pour i allant de 0 à fullLength faire :
 Si (alIndex ≥ SIZE_A) alors :
 res[i] ← a[alIndex] ;
 alIndex ++ ;
 Sinon Si (blIndex ≥ SIZE_B) alors :
 res[i] ← b[blIndex] ;
 blIndex ++ ;
 Sinon
 res[i] ← a[alIndex] ;
 alIndex ++ ;
 Fin Si
 Fin Pour
 Afficher(« \nAprès trie fusion:\n ») ;
 Pour i allant de 0 à fullLength faire :
 Afficher(« [%d] | », res[i]) ;
 Fin Pour

Fin

Code C

```
#define SIZE_A 8
#define SIZE_B 11

int main() {

    //Variables
    int a[SIZE_A] = {0, 1, 2, 4, 5, 8, 9, 512};
    int b[SIZE_B] = {3, 4, 5, 6, 8, 10, 11, 42, 65, 89, 101};
    int fullLength = SIZE_A + SIZE_B;
    int res[fullLength];
    int i, alIndex, blIndex;

    //Affichage tableau 1
    printf("Tableau 1:\n");
    for (i = 0; i < SIZE_A; i++){
        printf("[%d] | ", a[i]);
    }
    printf("\n");

    //Affichage tableau 2
    printf("Tableau 2:\n");
    for (i = 0; i < SIZE_B; i++){
        printf("[%d] | ", b[i]);
    }
    printf("\n");

    //Traitement
    for (i = 0, alIndex = 0, blIndex = 0; i < fullLength; i++) {
        if (alIndex >= SIZE_A) {
            res[i] = b[blIndex];
            blIndex ++;
        } else if (blIndex >= SIZE_B) {
            res[i] = a[alIndex];
            alIndex ++;
        } else if (a[alIndex] > b[blIndex]) {
            res[i] = b[blIndex];
            blIndex ++;
        } else {
            res[i] = a[alIndex];
            alIndex ++;
        }
    }

    //Affichage final
    printf("\nAprès trie fusion:\n");
    for (i = 0; i < fullLength; i++){
        printf("[%d] | ", res[i]);
    }

    return 0;
}
```

Complexité : $O(n \log(n))$

```
Tableau 1:  
[0] | [1] | [2] | [4] | [5] | [8] | [9] | [512] |  
Tableau 2:  
[3] | [4] | [5] | [6] | [8] | [10] | [11] | [42] |  
  
Après trie fusion:  
[0] | [1] | [2] | [3] | [4] | [4] | [5] | [5] | [6] | [8] | [8] | [9] | [10] | [11] | [42] | [65] | [89] | [101] | [512]
```

VI. Pile

Structure d'une pile

<u>Algorithme :</u>	<u>Code C</u>
<u>//structure d'un élément de la pile :</u> <u>Début</u> struct cell *next ; entier value ; <u>Fin</u> CELL ; <u>//structure de la pile :</u> <u>Début</u> CELL *first ; <u>Fin</u> PILE ;	 typedef struct Cell{ int value; struct Cell *next; } Cell; typedef struct Pile{ Cell *first; } Pile;

Fonction créer

<u>Algorithme :</u>	<u>Code C :</u>
Fonction create() : Pile* <u>Début</u> PILE *pile = allouer(*pile) ; pile → first = NULL ; Retourner(pile) ; <u>Fin</u>	 /** * Méthode permettant de créer une pile * @return une pile */ Pile* create(){ Pile *pile = malloc(sizeof(*pile)); pile→first = NULL; return pile; }

Fonction empiler

Algorithme :

Fonction push(Pile* pile, entier val) : vide

Début

CELL *cell = allouer(tailleDe(*cell)) ;

Si (pile != NULL && cell != NULL) Alors :

Cell→value <= val ;

Cell→next <= pile→first ;

Pile→first <= cell ;

Fin si

Fin

Code C :

```
/**
 * Méthode permettant d'empiler une valeur dans
 * une pile
 * @param pile la pile
 * @param value la valeur
 */
void push(Pile* pile, int value){
    Cell *cell = malloc(sizeof(*cell));

    //On verifie que la pile et la nouvelle cellule
    n'est pas NULL
    if(pile == NULL || cell == NULL) exit(1);

    //On empile la valeur
    cell→value = value;
    cell→next = pile→first;
    pile→first = cell;
}
```

```
--Empiler--
Entrez une valeur :
3
Pile avant :
2
5

Pile apres :
3
2
5
```


Fonction dépiler

Algorithme :

Fonction pop(Pile* pile) : vide

Début

Si (pile != NULL) Alors :

CELL *cellToUnstack = pile→first ;

Si (cellToUnstack != NULL) Alors :

pile→first <= cellToUnstack→next ;

deallocuer(cellToUnstack) ;

Fin si

Fin si

Fin

Code C :

```
/**
 * Méthode permettant de dépiler une pile
 * @param pile la pile
 */
int pop(Pile* pile){
    //On verifie que la pile n'est pas NULL
    if (pile == NULL) exit(1);
    int val = 0;

    //On dépile la première valeur de la liste
    Cell *cellToUnstack = pile→first;
    val = pile→first→value;
    pile→first = cellToUnstack→next;
    free(cellToUnstack);

    return val;
}
```

--Depiler--

Pile avant :

3

2

5

Pile apres :

2

5

Fonction vider

Algorithme :

Fonction clear(Pile *pile) : vide

Début

Tant Que (pile→first != NULL) Faire :

 pop(pile) ;

FinTantQue

Fin

Code C :

```
/**
 * Méthode permettant de vider une pile
 * @param pile la pile
 */
void clear(Pile* pile){
    //On verifie que la pile n'est pas NULL
    if(pile == NULL) exit(1);
    //Tant que la cellule courante n'est pas null on dépile
    la pile
    while(pile→first != NULL){
        pop(pile);
    }
}
```

```
-Vider--
Pile avant :
40
30
20
10

Pile apres :
```

Fonction sommet

Algorithme :

Fonction top(Pile *pile) : entier

Début

Si (pile→first != NULL) Alors :

Retourner(pile→first→value) ;

Sinon :

Retourner(-1) ;

Fin si

Fin

Code C :

```
/**
 * Méthode permettant de retourner le sommet d'une
 * pile
 * @param pile la pile
 * @return le sommet de la pile
 */
int top(Pile* pile){
    return pile == NULL ? -1 : pile→first→value;
}
```

--Sommet--

Pile :

40

30

20

10

Sommet : 40

Fonction pile_vide

Algorithme :

Fonction isEmpty(Pile *pile) : entier

Début

Retourner (pile→first == NULL ? 1 : 0) ;

Fin

Code C :

```
/**
 * Méthode permettant de vérifier si une pile est vide
 * @param pile la pile
 * @return 0 ou 1
 */
int isEmpty(Pile* pile){
    return pile→first == NULL ? 1 : 0;
}
```

```
--Vide ?--
Pile :
10
30
40

La pile n'est pas vide
```

```
--Vide ?--
Pile :

La pile est vide
```

Fonction recherche et restaure

Algorithme :

Fonction searchAndRestore(Pile* p, entier : val) : vide

Début

Si (p != NULL) Alors :

 Pile *finalPile = create() ;

 Cell *val = pile→first ;

Si (finalPile != NULL **ET** val != NULL) Alors :

Tant que (val != NULL) Faire :

Si (val→value != val) Alors :

 push(finalPile, val→value) ;

Fin si

 val = val→next ;

Fin tant que

Fin si

Fin si

 *pile = *finalPile ;

Fin

Code C :

```
/**
 * Méthode permettant de restaurer une pile sans une
 * valeur donnée
 * @param pile la pile
 * @param value la valeur
 */
void searchAndRestore(Pile *pile, int value){
    //On verifie que la pile n'est pas null
    if(pile == NULL) exit(1);
    Pile *finalPile = create();
    Cell *val = pile→first;
    //On verifie que la nouvelle pile n'est pas null
    if(finalPile == NULL) exit(1);

    //Tant que la cellule courante n'est pas null on verifie
    si sa valeur n'est pas égale à la valeur donnée, si c'est le
    cas on empile la valeur dans la nouvelle pile
    while (val != NULL){
        if(val→value != value) push(finalPile, val→value);
        val = val→next;
    }
    *pile = *finalPile;
}
```

--Rechercher et restaurer--

Entrez une valeur :

20

Pile avant :

40

30

20

10

Pile apres :

10

30

40

Fonction taille

Algorithme :

Fonction size(Pile* pile) : vide

Entier : cpt;

Début

cpt = 0;

Si (pile != NULL) Alors :

Cell *cur = pile→first ;

Si (cur != NULL) Alors :

Tant que (cur != NULL) Faire:

cpt <= cpt + 1;

cur = cur→next;

Fin tant que

Fin si

Fin si

Fin

Code C :

```
/**
 * Méthode permettant de retourner la taille
 * d'une pile
 * @param pile la pile
 * @return la taille de la pile
 */
int size(Pile* pile){
    int size = 0;
    //On verifie que la pile n'est pas NULL
    if (pile == NULL) exit(1);

    //Tant que la cellule courante n'est pas
    null on increment la variable size de 1
    Cell *current = pile→first;
    while (current != NULL){
        size++;
        current = current→next;
    }
    return size;
}
```

```
--Taille--
Pile :
40
30
20
10

Taille : 4
```

Fonction contient

Algorithme :

Fonction contient(Pile* pile, entier : value) : entier

Début

Si (pile != NULL) Alors :

Cell *cur = pile→first ;

Si (cur != NULL) Alors :

Tant que (cur != NULL) Faire:

Si(cur→value = value) Alors:

Retourner(1);

Fin si

cur = cur→next;

Fin tant que

Retourner(0);

Fin si

Fin si

Fin

Code C :

```
/**
 * Méthode permettant de vérifier si une
 * valeur est contenue dans la pile
 * @param pile la pile
 * @param value la valeur recherchée
 * @return 0 ou 1
 */
int contient(Pile *pile, int value){
    //On vérifie que la pile n'est pas NULL
    if (pile == NULL) exit(1);
    //Tant que la cellule courante n'est
    pas null on vérifie si sa valeur est égale
    à la valeur recherché
    Cell *current = pile→first;
    while (current != NULL){
        if(current→value == value) return 1;
        current = current→next;
    }
    return 0;
}
```

Entrez une valeur :

100

Pile :

40

30

20

10

100 n'est pas présent dans la pile

--Trouver un element--

Entrez une valeur :

30

Pile :

40

30

20

10

30 est présent dans la pile

Fonction égale

Algorithme :

Fonction equals(Pile* p1, Pile* p2) : entier

Début

Si (p1 != NULL **ET** p2 != NULL) Alors :

Si (size(p1) = size(p2) Alors:

Cell p1_val <= p1→first;

Si (p1_val != NULL) Alors:

Tant que (p1_val != NULL) Faire:

Si (!contain(p2, p1_val→value) Alors:

Retourner(0);

Fin tant que

Retourner(1);

Fin si

Fin si

Fin si

Fin

Code C :

```
/**
 * Méthode permettant de vérifier si deux
 * piles sont égales
 * @param p1 la première pile
 * @param p2 la seconde pile
 * @return 0 ou 1
 */
int equals(Pile *p1, Pile *p2){
    //On vérifie que les deux piles ne sont pas
    NULL
    if (p1 == NULL || p2 == NULL) exit(1);
    //On vérifie que les deux piles ont la même
    taille
    if(size(p1) != size(p2)){
        printf("The size of the two stacks are not
        equal");
        return 0;
    }
    //Tant que la cellule courante de la pile 1
    n'est pas null on vérifie si sa valeur est con-
    tenu dans la deuxième pile
    Cell *p1_value = p1→first;
    if(p1_value == NULL) exit(1);
    while (p1_value != NULL){
        if (!contain(p2, p1_value→value)) return
        0;
        p1_value = p1_value→next;
    }
    return 1;
}
```

```
--Egalite--
Pile 1 :
3
2
5

Pile 2 :
3
2
1

Les deux piles ne sont pas égale
```

```
--Egalite--
Pile 1 :
1
3
2

Pile 2 :
3
2
1

Les deux piles sont égale
```


Fonction afficher

<u>Algorithme :</u>	<u>Code C :</u>
<p>Fonction afficher(Pils *pile) : vide</p> <p><u>Début</u></p> <p><u>Si</u>(pile != NULL) <u>Alors</u>:</p> <p> Cell *current <= list→first ;</p> <p> <u>Tant Que</u> (current != NULL) <u>Faire</u> :</p> <p> Afficher(« %d \n », current→value) ;</p> <p> current <= current→next ;</p> <p> <u>FinTantQue</u></p> <p> Afficher(« \n ») ;</p> <p> <u>Fin si</u></p> <p><u>Fin</u></p>	<pre>/** * Méthode permettant d'afficher une pile * @param pile la pile */ void show(Pile* pile){ //On verifie que la pile n'est pas NULL if (pile == NULL) exit(1); //Tant que la cellule courante n'est pas null on affiche son contenu Cell *current = pile→first; while (current != NULL){ printf("%d\n", current→value); current = current→next; } printf("\n"); }</pre>

```
--Affichage--
Pile :
40
30
20
10
```

Fonction d'Ackermann (pile)

Algorithme :

Fonction ack(entier : n, entiere: m) : entier

pile p;
entier n,m;

Début

create (p);

push(p,m);

push (p,n);

Tant que (!isEmpty(p)) Faire:

n <= top(p);

depiler(p);

Si (isEmpty(p)) Alors:

m <= top (p);

pop(p);

Sinon

retourner(n);

Fin si

Si (m==0) Alors

push (p, n+1);

Sinon si(n==0) Alors:

push(p, m-1);

push(1);

Sinon

push(p, m-1);

push(p, m);

push(p, n-1);

Fin si

Fin si

Fin Tant que

retourner(top (p));

Fin

Code C :

```
/**
 * Méthode permettant de calculer Ackermann
 * @param n val 1
 * @param m val 2
 * @return Ackermann(va1, val2)
 */
int ack(int n, int m){
    Pile *p = create();
    push(p, m);
    push(p, n);

    while (isEmpty(p) == 0){
        n = top(p);
        pop(p);
        if(isEmpty(p) == 0){
            m = top(p);
            pop(p);
        } else {
            return n;
        }
        if(m==0){
            push(p, n + 1);
        } else if(n==0){
            push(p, m - 1);
            push(p, 1);
        } else {
            push(p, m - 1);
            push(p, m);
            push(p, n - 1);
        }
    }
    return top(p);
}
```

--Ackermann--

A(2,2) = 7

A(3,3) = 61

VII. Liste chaînée

Structure d'une liste chaînée

<p>Algorithme : //structure d'un élément de la liste : <u>Début</u> struct Cell *next ; entier value ; <u>Fin</u> Cell;</p> <p>//structure de la liste : <u>Début</u> Cell *first ; <u>Fin</u> List ;</p>	<p>Code C : /** * Méthode permettant de créer une liste chaînée * @return une liste chaînée */ List* create(){ List *list = malloc(sizeof(*list)); Cell *cell = malloc(sizeof(*cell)); if (list == NULL cell == NULL) { printf("Memory allocation problem of the list or cell !"); exit(1); } cell->value = -1; cell->next = NULL; list->first = cell; return list; }</p>
--	--

Fonction créer

<p>Algorithme : Fonction initialiser() : List* <u>Début</u> List *list <= allouer(tailleDe(*list)); Cell *cell <= allouer(tailleDe(*cell)) ; cell->value <= 0 ; cell->next <= NULL ; list->first <= cell ; Retourner(list) ; <u>Fin</u></p>	<p>Code C : /** * Méthode permettant de créer une liste chaînée * @return une liste chaînée */ List* create(){ List *list = malloc(sizeof(*list)); Cell *cell = malloc(sizeof(*cell)); //On vérifie que la liste ou la cellule n'est pas NULL if (list == NULL cell == NULL) { printf("Memory allocation problem of the list or cell !"); exit(1); } //On initialise la première cellule de la liste cell->value = -1; cell->next = NULL; list->first = cell; return list; }</p>
---	---

Fonction taille

Algorithme :**//structure d'un élément de la liste :**Début

```
struct Cell *next ;  
entier value ;
```

Fin

Cell;

//structure de la liste :Début

Cell *first ;

Fin

List ;

Code C :

```
/**  
 * Méthode permettant de calculer la taille  
 d'une liste chaînée  
 * @param list la liste  
 * @return la taille de la liste  
 */  
int size(List* list){  
    int size = 0;  
    //On vérifie que la liste n'est pas NULL  
    if (list == NULL){  
        printf("List cannot be NULL !");  
        exit(1);  
    }  
    //On parcourt la liste tant que la cellule  
    courante n'est pas null on incrémente la va-  
    riable size de 1  
    Cell *current = list->first;  
    while (current != NULL){  
        size++;  
        current = current->next;  
    }  
    return size;  
}
```

```
--Taille de la liste--  
10 -> 30 -> 100 -> 50 -> 78 -> NULL  
Taille de la liste : 5
```

Fonction ajouter en fin de liste

Algorithme :

```
Fonction insert(List *list, entier val) : void
    Cell *new ← allouer(sizeof(*new)) ;
    Si (list == NULL ET new == NULL) Alors :
        Afficher(« List cannot be NULL or/and
Memory allocation problem of the new cell
!);
        exit(1);
    Si (list → first → value == -1 ET size(list) == 1)
list → first → value ← val;
    Sinon
        new → value ← val;
        new → next ← NULL;
        Cell* current ← list → first;
        Tant que (current → next != NULL)
current ← current → next;
        current → next ← new;
        Fin Tant Que
    Fin Si
Fin
```

Code C :

```
/**
 * Méthode permettant d'insérer un élément
à la fin d'une liste chaînée
 * @param list la liste
 * @param val la valeur à ajouter
 */
void insert(List *list, int val) {
    Cell *new = malloc(sizeof(*new));
    //On vérifie que la liste ou la nouvelle cel-
lule n'est pas NULL
    if (list == NULL || new == NULL) {
        printf("List cannot be NULL or/and
Memory allocation problem of the new cell
!");
        exit(1);
    }
    //Si la taille de la liste est égale à 1 et que
la valeur de la première cellule est -1 alors
on remplace -1 par la valeur à ajouter
    if(list→first→value == -1 && size(list) == 1)
list→first→value = val;
    else {
        //Sion on ajoute la valeur à la fin de la
liste
        new→value = val;
        new→next = NULL;

        Cell* current = list→first;
        while(current→next != NULL) current =
current→next;

        current→next = new;
    }
}
```

```
--Insertion fin de liste--
Entrez une valeur :
30
10 -> NULL
10 -> 30 -> NULL
```

Fonction ajouter à un rang quelconque

Algorithme :

```

Fonction insertElementByKey(List *list, entier
key, entier val) : void
    Si (list == NULL) alors :
        Afficher(« List cannot be NULL ! ») ;
        exit(1) ;
    Fin Si
    Si (key > size(list) OU key < 0) alors :
        Afficher(« Index overflow :%d », key) ;
        exit(1) ;
    Fin Si
    Si (key == size(list)) alors :
        insert(list ; val) ;
    Sinon
        Cell *new = allouer(sizeof(*new)) ;
    Fin Si
    Si (new == NULL) alors :
        Afficher(« Memory allocation problem
of the new cell ! ») ;
        exit(1) ;
    Fin Si
    new → value ← val ;
    Cell *current ← list → first ;
    Pour i allant de 2 à key - 1 faire :
        current ← current → next ;
    Fin Pour
    new → next ← current → next ;
    current → next ← new ;
Fin

```

Code C :

```

/**
 * Méthode permettant d'insérer un élément
 * à un rang précis d'une liste chaînée
 * @param list la liste
 * @param key le rang
 * @param val la valeur à ajouter
 */
void insertElementByKey(List *list, int key, int
val) {
    //On vérifie que la liste n'est pas NULL
    if (list == NULL) {
        printf("List cannot be NULL !");
        exit(1);
    }
    //On vérifie que le rang ne dépasse pas
    la taille de la liste ou que le rang ne soit pas
    négatif
    if (key > size(list) || key < 0) {
        printf("Index overflow: %d", key);
        exit(1);
    }
    //On vérifie le est agales à la taille de la
    liste on appelle la méthode pour inserer à la
    fin de la liste
    if (key == size(list)) insert(list, val);
    else{
        //Sinon on ajoute à la valuer au rang
        demandé
        Cell *new = malloc(sizeof(*new));
        //On vérifie que la nouvelle cellule n'est
        pas NULL
        if (new == NULL) {
            printf("Memory allocation problem of
the new cell !");
            exit(1);
        }
        new→value = val;
        Cell *current = list→first;
        for(int i = 2; i <= key - 1; i++){
            current = current→next;
        }
        new→next = current→next;
        current→next = new;
    }
}

```

```
--Insertion a un rang--  
Entrez une valeur :  
15  
Entrez un rang :  
2  
10 -> 20 -> 60 -> NULL  
15 insere au rang 2:  
10 -> 15 -> 20 -> 60 -> NULL
```

Fonction supprimer à un rang quelconque

Algorithme :

```

Fonction deleteElementByKey(List *list, entier
key) : void
    entier i ;
    Cell *del, *prev ;
    Si (list == NULL) alors :
        Afficher(« List cannot be NULL ! ») ;
        exit(1) ;
    Fin Si
    Si (key > size(list) OU key < 0) alors :
        Afficher(« Index overflow : %d », key) ;
        return ;
    Fin Si
    del ← list → first ;
    prev ← list → first ;
    Pour i allant de 2 à key faire :
        prev ← del ;
        del ← del → next ;
    Si (del == NULL) alors
        break ;
    Fin Si
    Si (del != NULL) alors
        Si (del == list → first) alors
            list → first ← list → first → next ;
        Fin Si
        prev → next ← del → next ;
        del → next ← NULL ;
        liberer(del) ;
    Fin Si
Fin

```

Code C :

```

/**
 * Méthode permettant de supprimer un élé-
 * ment à un rang précis d'une liste chaînée
 * @param list la liste
 * @param key le rang
 * @param val la valeur à supprimer
 */
void deleteElementByKey(List *list, int key) {
    int i;

    Cell *del, *prev;

    //On vérifie que la liste n'est pas NULL
    if (list == NULL) {
        printf("List cannot be NULL !");
        exit(1);
    }

    //On vérifie que le rang ne dépasse pas
    la taille de la liste ou que le rang ne soit pas
    négatif
    if (key > size(list) || key < 0) {
        printf("Index overflow: %d", key);
        return;
    }

    //On position la cellule précédente et à
    supprimer au début de la liste
    del = list→first;
    prev = list→first;

    //On supprime l'élément au rang de-
    mandé
    for(i=2; i<=key; i++){
        prev = del;
        del = del→next;

        if(del == NULL)
            break;
    }
    if(del != NULL){
        if(del == list→first)
            list→first = list→first→next;

        prev→next = del→next;
        del→next = NULL;
        free(del);
    }
}

```



```
--Supression a un rang--  
Entrez un rang :  
3  
Element supprimer au rang 3 :  
10 -> 15 -> 20 -> 30 -> NULL  
10 -> 15 -> 30 -> NULL
```

Fonction supprimer la liste

Algorithme :

Fonction delete(List *list) : vide

Début

Tant Que (list→first != NULL) Faire :

Cell *del <= list→first ;

list→first <= list→first→next ;

free(del) ;

Fin Tant Que

free(liste) ;

Fin

Code C :

```
/**
 * Méthode permettant de supprimer une
 * liste
 * @param list la liste
 */
void delete(List *list){
    //On vérifie que la liste n'est pas NULL
    if (list == NULL){
        printf("List cannot be NULL !");
        exit(1);
    }
    //On parcourt la liste pour chaque cellule
    //on la de-alloque
    while(list→first != NULL){
        Cell *del = list→first;
        list→first = list→first→next;
        free(del);
    }
    //On de-alloque la liste
    free(list);
    printf("The list has been deleted !");
    exit(0);
}
```

```
--Suppresion d'une liste--
The list has been deleted !
```

Fonction rechercher

Algorithme :

```
Fonction find(List -list, entier val) : entier
Si (list == NULL) alors :
    Afficher(« List cannot be NULL ! ») ;
    exit(1) ;
Fin Si
Cell *current ← list → first ;
Tant que (current != NULL) faire
    Si (current → value == val) alors :
        retourner 1 ;
    Fin Si
    Current ← current → next ;
Fin Tant que
Fin
```

Code C :

```
/**
 * Méthode permettant de chercher si une
 * valeur est présente dans une liste chaînée
 * @param list la liste
 * @param val la valeur à chercher
 * @return 1 ou 0
 */
int find(List *list, int val){
    //On vérifie que la liste n'est pas NULL
    if (list == NULL){
        printf("List cannot be NULL !");
        exit(1);
    }
    //On parcourt la liste tant que la cellule
    //courante n'est pas null
    //Si la valeur de la cellule courante est
    //égale à la valeur recherchée on retourne 1
    //sinon 0
    Cell *current = list→first;
    while (current != NULL){
        if(current→value == val) return 1;
        current = current→next;
    }
    return 0;
}
```

```
--Trouver un element--
Entrez une valeur :
20
10 -> 20 -> 30 -> 40 -> 50 -> NULL
20 est present dans la liste
```

```
--Trouver un element--
Entrez une valeur :
11
10 -> 20 -> 30 -> 40 -> 50 -> NULL
11 n'est pas present dans la liste
```

Fonction fusion de 2 listes

Algorithme :

```

Fonction *fusion(List *l1, List *l2) : List
    List *l3 ← create() ;
    Si (l1 == NULL OU l2 == NULL OU l3 == NULL)
    alors :
        Afficher(« List cannot be NULL ! ») ;
        exit(1) ;
    Fin Si
    entier i, alIndex, blIndex ;
    entier fullLength ← size(l1) + size(l2) ;
    Cell *p1 ← l1 → first ;
    Cell *p2 ← l2 → first ;
    alIndex ← 0 ;
    blIndex ← 0 ;
    Pour i allant de 0 à fullLength faire :
        Si (alIndex ≥ size(l1)) alors :
            insert(l3, p2 → value) ;
            p2 ← p2 → next ;
            alIndex ++ ;
        Sinon Si (blIndex ≥ size(l2)) alors :
            insert(l3, p1 → value) ;
            p1 ← p1 → next ;
            blIndex ++ ;
        Sinon Si (p1 → value > p2 → value)
        alors :
            insert(l3, p2 → value) ;
            p2 ← p2 → next ;
            blIndex ++ ;
        Sinon
            insert(l3, p1 → value) ;
            p1 ← p1 → next ;
            alIndex ++ ;
        Fin Si
    Fin Pour
    retourner l3 ;
Fin

```

Code C :

```

/**
 * Methode permettant de fusionner deux
 * listes
 * @param l1 la première liste
 * @param l2 la seconde liste
 * @return la liste fusionnée
 */
List *fusion(List *l1, List *l2){
    List *l3 = create();

    //On vérifie bien qu'aucune list n'est NULL
    if (l1 == NULL || l2 == NULL || l3 == NULL ){
        printf("List cannot be NULL !");
        exit(1);
    }

    int i, alIndex, blIndex;
    int fullLength = size(l1) + size(l2);
    //On se place au début des deux listes
    puis on les fusionnes
    Cell *p1 = l1->first;
    Cell *p2 = l2->first;

    for (i = 0, alIndex = 0, blIndex = 0; i <
    fullLength; i++) {
        if (alIndex >= size(l1) ){
            insert(l3, p2->value);
            p2 = p2->next;
            blIndex ++;
        } else if (blIndex >= size(l2)){
            insert(l3, p1->value);
            p1 = p1->next;
            alIndex ++;
        } else if (p1->value > p2->value) {
            insert(l3, p2->value);
            p2 = p2->next;
            blIndex++;
        } else {
            insert(l3, p1->value);
            p1 = p1->next;
            alIndex++;
        }
    }
    return l3;
}

```

```
--Fusion de deux liste--  
Liste 1 :  
1 -> 5 -> 7 -> 9 -> NULL  
Liste 2 :  
0 -> 2 -> 3 -> 4 -> NULL  
Liste fusionne :  
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 7 -> 9 -> NULL
```

Fonction afficher

Algorithme :

Fonction afficher(List *list) : vide

Début

Cell *current <= list→first ;

Tant Que (current != NULL) Faire :

Afficher(« %d → \n », current→value) ;

current <= current→next ;

FinTantQue

Afficher(« NULL ») ;

Fin

Code C :

```
/**
 * Méthode permettant d'afficher le contenu d'une liste
 * @param list la liste
 */
void show(List *list){
    //On vérifie que la liste n'est pas NULL
    if (list == NULL){
        printf("List cannot be NULL !");
        exit(1);
    }
    //On vérifie que la taille de la liste n'est pas égale à 0
    if(size(list) == 0){
        printf("List is empty\n");
        return;
    }
    //On parcourt la liste pour chaque cellule on affiche son contenu
    Cell *current = list→first;
    while (current != NULL){
        printf("%d → ", current→value);
        current = current→next;
    }
    printf("NULL\n");
}
```

--Affichage--

10 -> 30 -> 100 -> 50 -> 78 -> NULL

Fonction incrémenter liste binaire

Algorithme :

```

Fonction incrementListBinary(List *list) : List*
    entier bin[128], i = 0, value = 0, temps,
    decimalValue, pow, last_digit, bit = 0 ;
    List *listFinal = create() ;
    Si (list == NULL OU listFinal == NULL) alors :
        Afficher(« List cannot be NULL ! ») ;
        exit(1) ;
    Fin Si
    Cell *current ← list → first ;
    Tant Que (current != NULL) faire :
        Si (current → value != 0 ET current →
        value != 1) alors :
            Afficher(« This is not a binary chained
            list. \n ») ;
            exit(0) ;
        Fin Si
        current ← current → next ;
    Fin Tant Que
    current ← list → first ;
    Tant Que (current != NULL) faire :
        value ← value * 10 ;
        value ← value + current → value ;
        current ← current → value ;
    Fin Tant Que
    temp ← value ;
    decimalValue ← 0 ;
    pow ← 1 ;
    Tant Que (temp) faire :
        last_digit ← temp % 10 ;
        temp ← temp / 10 ;
        decimalValue ← decimalValue +
        (last_digit * pow) ;
        pow ← pow * 2 ;
    Fin Tant Que
    decimalValue++ ;
    Pour i allant de 0 à decimalValue faire :
        bin[i] ← decimalValue % 2 ;
        decimalValue ← decimalValue / 2 ;
        bit++ ;
    Fin Pour
    Pour i allant de bit - 1 à 0 faire :
        insert(listFinal, bin[i]) ;
    Fin Pour
    retourner listFinal ;
Fin

```

Code C :

```

/**
 * Méthode permettant d'incrémenter de 1
 * une liste chaînée binaire
 * @param list la liste binaire
 * @return la nouvelle liste binaire incrémentée de 1
 */
List *incrementListBinary(List* list){
    int bin[128], i = 0, value = 0, temp, decimalValue, pow, last_digit, bit = 0;
    List *listFinal = create();

    //On vérifie bien qu'aucune list n'est NULL
    if (list == NULL || listFinal == NULL){
        printf("List cannot be NULL !");
        exit(1);
    }

    //On parcourt la liste pour vérifier si il s'agit
    bien d'une liste binaire
    Cell *current = list→first;
    while (current != NULL){
        if(current→value != 0 && current→value != 1){
            printf("This is not a binary chained list.
            \n");
            exit(0);
        }
        current = current→next;
    }

    //On se remet au début de la liste puis on
    la parcourt pour créer un seul entier avec
    toutes les cellules de la liste
    current = list→first;
    while (current != NULL){
        value *= 10;
        value += current→value;
        current = current→next;
        i++;
    }

    temp = value;
    decimalValue = 0;
    pow = 1;

    //Convertit le nombre binaire initiale en
    décimale
    while (temp) {
        last_digit = temp % 10;
        temp = temp / 10;
        decimalValue += last_digit * pow;
    }
}

```

	<pre> pow = pow * 2; } //On incrémente la valeur décimale de 1 decimalValue++; //On re convertit la nouvelle valeur dec- male en binaire for(i = 0; decimalValue > 0; i++){ bin[i] = decimalValue % 2; decimalValue = decimalValue / 2; bit++; } //Insertion du nouveau nombre binaire dans la nouvelle liste for (i = bit - 1; i >= 0; i--){ insert(listFinal, bin[i]); } return listFinal; } </pre>
--	---

```

--Incrementation liste binaire--
1 -> 1 -> 1 -> 1 -> NULL
1 -> 0 -> 0 -> 0 -> 0 -> NULL

```


Fonction décrémenter liste binaire

Algorithme :

```

Fonction *decrementListBinary(List* list) : List
    Entier bin[128], i = 0, value = 0, temps,
    decimalValue, pow, last_digit, bit = 0 ;
    List *listFinal ← creat() ;
    Si (list == NULL OU listFinal == NULL) alors :
        Afficher(« List cannot be NULL ! ») ;
        exit(1) ;
    Fin Si
    Cell *current ← list → first ;
    Tant Que (current != NULL) faire :
        Si (current → value != 0 ET current →
        value != 1) alors :
            Afficher(« This is not a binary chained
            list\n ») ;
            exit(0) ;
        Fin Si
        current ← current → next ;
    Fin Tant Que
    current ← list → first ;
    Tant Que (current != NULL) faire :
        value ← value*10 ;
        value ← value + current → value ;
        current ← current → next ;
        i++ ;
    Fin Tant Que
    temp ← value ;
    decimalValue ← 0 ;
    pow ← 1 ;
    Tant Que (temp) faire :
        last_digit ← temp % 10 ;
        temp ← temp / 10 ;
        decimalValue ← decimalValue +
        last_digit * pow ;
        pow ← pow*2 ;
    Fin Tant Que
    decimalValue-- ;
    i ← 0 ;
    Tant que decimalValue > 0 faire :
        bin[i] ← decimalValue%2 ;
        decimalValue ← decimalValue/2 ;
        bit++ ;
        i++ ;
    Fin Tant Que
    Pour i allant de bit -1 à 0 faire
        insert(listFinal, bin[i]) ;
        i-- ;
    Fin Pour
    return listFinal ;
Fin

```

Code C :

```

/**
 * Méthode permettant de décrémenter de
 * 1 une liste chaînée binaire
 * @param list la liste binaire
 * @return la nouvelle binaire décrémentée
 * de 1
 */
List *decrementListBinary(List* list){
    int bin[128], i = 0, value = 0, temp, deci-
    malValue, pow, last_digit, bit = 0;
    List *listFinal = create();

    //On vérifie bien qu'aucune list n'est NULL
    if (list == NULL || listFinal == NULL){
        printf("List cannot be NULL !");
        exit(1);
    }

    //On parcourt la liste pour vérifier si il s'agit
    bien d'une liste binaire
    Cell *current = list→first;
    while (current != NULL){
        if(current→value != 0 && cur-
        rent→value!= 1){
            printf("This is not a binary chained list.
            \n");
            exit(0);
        }
        current = current→next;
    }

    //On se remet au début de la liste puis on
    la parcourt pour créer un seul entier avec
    toutes les cellules de la liste
    current = list→first;
    while (current != NULL){
        value *= 10;
        value += current→value;
        current = current→next;
        i++;
    }

    temp = value;
    decimalValue = 0;
    pow = 1;

    //Convertit le nombre binaire initiale en
    décimale
    while (temp) {
        last_digit = temp % 10;
        temp = temp / 10;
        decimalValue += last_digit * pow;
    }
}

```

```

        pow = pow * 2;
    }

    //On decrement la valeur décimale de 1
    decimalValue--;

    //convertit la nouvelle valeur decimale
    en binaire
    for(i = 0; decimalValue > 0; i++){
        bin[i] = decimalValue % 2;
        decimalValue = decimalValue / 2;
        bit++;
    }

    //Insertion du nouveau nombre binaire
    dans la nouvelle liste
    for (i = bit - 1; i >= 0; i--){
        insert(listFinal, bin[i]);
    }

    return listFinal;
}

```

```

--Decrementation liste binaire--
1 -> 1 -> 0 -> 1 -> 0 -> NULL
1 -> 1 -> 0 -> 0 -> 1 -> NULL

```

Fonction trier

Algorithme :

```
Fonction sort(List *list) : void
    entier stop;
    Cell *current, *temp ← NULL
    Si (list == NULL) alors
        Afficher("List cannot be NULL !");
        exit(1);
    Fin Si
    stop ← 0;
    current ← list → first;
    Tant Que (current → next != temp) faire:
        Si (current → value > current → next →
value) faire:
            entier tmp ← current → value;
            current → value ← current → next →
value;
            current → next → value ← tmp;
            stop ← 1;
        Fin Si
        current ← current → next;
    Fin Tant Que
    temp ← current;
    Tant Que (stop) faire:
        Fin Tant Que
    Fin
```

Code C :

```
/**
 * Methode permettant de trier une liste
 dans l'ordre croissant
 * @param list la liste
 */
void sort(List *list){
    int stop;
    Cell *current, *temp = NULL;

    //On vérifie que la liste n'est pas NULL
    if (list == NULL){
        printf("List cannot be NULL !");
        exit(1);
    }

    //Tant que stop n'est pas à 1 on continue
    de parcourir la liste et de la trier
    do{
        stop = 0;
        current = list→first;
        while (current→next != temp){
            if(current→value > cur-
rent→next→value){
                int tmp = current→value;
                current→value = cur-
rent→next→value;
                current→next→value = tmp;
                stop = 1;
            }
            current = current→next;
        }
        temp = current;
    } while (stop);
}
```

```
Liste non triee :
50 -> 20 -> 10 -> 30 -> 40 -> NULL
Liste triee :
10 -> 20 -> 30 -> 40 -> 50 -> NULL
```

Fonction 2 listes égales

Algorithme :

```
Fonction equals(List *l1, List *l2) : entier
Si (l1==NULL) OU (l2==NULL) Alors
    Ecrire ("List cannot be NULL !");
    exit(1);
Cell *currentL1 ← l1 → first;
Tant Que (currentL1!=NULL) Faire
    Si (find(l2, currentL1→value)==0) Faire
        retourne 0;
    Fin Si
    currentL1 ← currentL1→next;
Fin Tant Que
Retourne 1;
Fin;
```

Code C :

```
/**
 * Methode permettant de vérifier si deux
 * liste sont égales
 * @param l1 la première liste
 * @param l2 la seconde liste
 */
int equals(List *l1, List *l2){
    //On vérifie que la liste n'est pas NULL
    if (l1 == NULL || l2 == NULL){
        printf("List cannot be NULL !");
        exit(1);
    }
    //Tant que la cellule courante n'est pas
    //null, on vérifie si sa valeur est contenu dans
    //la seconde liste
    Cell *currentL1 = l1->first;
    while (currentL1 != NULL){
        if(find(l2, currentL1->value) == 0) return
        0;
        currentL1 = currentL1->next;
    }
    return 1;
}
```

```
--Egale--
Liste 1 :
4 -> 7 -> 8 -> NULL
Liste 1 :
4 -> 7 -> 8 -> NULL
Les deux listes sont égale
```

```
--Egale--
Liste 1 :
4 -> 5 -> 8 -> NULL
Liste 2 :
4 -> 7 -> 8 -> NULL
Les deux listes ne sont pas égale
```

Concaténation de 2 listes

Algorithme :

```
Fonction concatenation(List *l1, List *l2) :  
void  
  Si (l1==NULL) OU (l2==NULL) Alors  
    Ecrire ("List cannot be NULL !");  
    exit(1);  
  Cell *lastL1 ← l1 → first;  
  Tant que (lastL1!=NULL) Faire  
    Si (lastL1 → next!=NULL) Faire  
      lastL1 ← lastL1 → next;  
    Sinon break;  
  Fin Tant Que;  
  lastL1 → next = l2 → first;  
Fin
```

Code C :

```
/**  
 * Methode permettant de concaténer  
 deux listes  
 * @param l1 la première liste  
 * @param l2 la deuxième liste  
 */  
void concatenation(List *l1, List *l2){  
  
  //On vérifie que les listes ne sont pas NULL  
  if (l1 == NULL || l2 == NULL){  
    printf("List cannot be NULL !");  
    exit(1);  
  }  
  
  //On se positionne au début de la pre-  
  mière liste puis on la parcourt jusqu'à arriver  
  à la dernière cellule puis on lui ajoute la se-  
  conde liste  
  Cell *lastL1 = l1->first;  
  while (lastL1 != NULL){  
    if(lastL1->next != NULL) lastL1 = lastL1-  
    >next;  
    else break;  
  }  
  lastL1->next = l2->first;  
}
```

```
--Concatenation--  
Liste 1 :  
52 -> 23 -> 56 -> NULL  
Liste 2 :  
4 -> 7 -> 8 -> NULL  
52 -> 23 -> 56 -> 4 -> 7 -> 8 -> NULL
```

Fonction estExtraite

Algorithme :

```

Fonction isExtract(List *l1, List *l2): entier
    Si (l1 == NULL OU l2 == NULL) Alors
        Afficher("List cannot be NULL !");
        exit(1);
    Fin Si
    Si (size(l2) > size(l1)) alors
        Afficher("The second list extracted
cannot be larger than the original list");
        exit(0);
    Fin Si
    Si (size(l2) > size(l1)) equals (l1, l2) alors
        entier cpt ← 0;
        entier result ← 0;
        Cell *firstElmtL1 ← l1 → first;
        Cell *firstElmtL2 ← l2 → first;
        Tant Que (firstElmtL1 != NULL) faire :
            Si (firstElmtL2 != NULL) alors :
                Si (firstElmtL2 → value == firstElmtL1
→ value) alors ;
                    firstElmt2 ← firstElmt2 → next;
                    cpt++;
                Sinon
                    cpt ← 0;
                    firstElmt2 ← l2 → first;
            Fin Si
        Fin Si
        Si (cpt == size(l2)) alors
            result ← 1;
            break;
        Fin Si
        firstElmt1 ← firstElmt1 → next;
    Fin Pour
    Retourner result;
Fin

```

Code C :

```

/**
 * Methode permettant de vérifier si une liste
est extraite
 * @param l1 liste originale
 * @param l2 la liste à extraire
 */
int isExtract(List *l1, List *l2){
    if (l1 == NULL || l2 == NULL){
        printf("List cannot be NULL !");
        exit(1);
    }

    if(size(l2) > size(l1)){
        printf("The second list extracted cannot
be larger than the original list");
        exit(0);
    }
    if (size(l2) > size(l1)) equals(l1, l2);

    int cpt = 0;
    int result = 0;

    Cell *firstElmtL1 = l1->first;
    Cell *firstElmtL2 = l2->first;

    while (firstElmtL1 != NULL){
        if(firstElmtL2!= NULL){
            if(firstElmtL2->value == firstElmtL1-
>value){
                firstElmtL2 = firstElmtL2->next;
                cpt++;
            } else {
                cpt = 0;
                firstElmtL2 = l2->first;
            }
        }
        if(cpt == size(l2)){
            result = 1;
            break;
        }
        firstElmtL1 = firstElmtL1->next;
    }

    return result;
}

```

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> NULL  
2 -> 3 -> 4 -> NULL  
est extraite
```

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> NULL  
2 -> 3 -> 4 -> 23 -> NULL  
n'est pas extraite
```

VIII. Arbre

Structure d'un arbre

<u>Algorithme :</u> enum colors{NONE = -1, NOIR = 0, ROUGE = 1}; Structure Node entier value ; entier color ; struct Node *left ; struct Node *right ;	<u>Code C :</u> enum colors{NONE = -1, NOIR = 0, ROUGE = 1}; typedef struct Node{ int value; int color; struct Node *left; struct Node *right; } Node;
---	--

Fonction creerNoeud

<u>Algorithme :</u> <u>Fonction</u> createNode(entier val, enum colors color) : Node* Node* newNode ← allouer(sizeof(Node)) ; Si (newNode == NULL) alors Afficher(« A node cannot be NULL ») ; exit(1) ; Fin Si newNode → left ← NULL ; newNode → right ← NULL ; newNode → value ← val ; Si (color == NOIR) alors : newColor → color ← 0 ; Sinon Si (color == ROUGE) alors newNode → color ← 1 ; Sinon Si (color == NONE) alors : newColor → color ← -1 ; Fin Si Retourner newNode ; Fin	<u>Code C :</u> /** * Permet de créer un noeud * @param val valeur du noeud * @param color couleur du noeud pour les arbres R&N * @return un noeud */ Node* createNode(int val, enum colors color){ Node* newNode= malloc(sizeof(Node)); if(newNode == NULL){ printf("A node cannot be NULL"); exit(1); } newNode->left = NULL; newNode->right = NULL; newNode->value = val; if(color == NOIR) newNode->color = 0; else if(color == ROUGE) newNode->color = 1; else if(color == NONE) newNode->color = -1; return newNode; }
--	---

Fonction insertion

Algorithme :

```
Fonction insert(Node *tree, entier value,
enum colors color) : void
    Si (value < tree → value ET tree → left !=
NULL) alors :
        insert(tree → left, value, color) ;
    Sinon Si (value < tree → value ET tree →
left == NULL) alors :
        tree → left ← createNode(value,
color) ;
    Sinon Si (value > tree → value ET tree →
right != NULL) alors :
        insert(tree → right, value, color) ;
    Sinon Si (value > tree → value ET tree →
right == NULL) alors :
        tree → right ← createNode(value,
color) ;
    Fin Si
Fin
```

Code C :

```
/**
 * Permet d'insérer un noeud dans un arbre
 * @param tree l'arbre
 * @param value la valeur du noeud à insé-
rer
 * @param color la couleur du noeud à insé-
rer pour les arbres R&N
 */
void insert(Node *tree, int value, enum col-
ors color){
    /* On construit l'arbre pour que les valeur
inférieur soit à gauche et supérieur à droite
*/
    if (value < tree->value && tree-
>left!=NULL)
        insert(tree->left, value, color);

    else if (value < tree->value && tree-
>left==NULL)
        tree->left = createNode(value, color);

    else if (value > tree->value && tree-
>right!=NULL)
        insert(tree->right, value, color);

    else if (value > tree->value && tree-
>right==NULL)
        tree->right = createNode(value, color);
}
```

=> Insertion d'un noeud (14, NOIR) <=

```
Pacours infixe :
17 -> ROUGE
13 -> NOIR | [Fils droit : 17]
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]
9 -> NOIR
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]
5 -> NOIR
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]
2 -> NOIR
```

```
Pacours infixe :
17 -> ROUGE | [Fils gauche : 14]
14 -> NOIR
13 -> NOIR | [Fils droit : 17]
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]
9 -> NOIR
7 -> NOIR | [Fils gauche : 5, Fils droit : 11]
5 -> NOIR
3 -> ROUGE | [Fils gauche : 2, Fils droit : 7]
2 -> NOIR
```

Fonction rotationDroite

Algorithme :

```
Fonction rightRotation(Node* node) : Node*  
    Node* tmp ← node → left ;  
    node → left ← tmp → right ;  
    tmp → right ← node ;  
    node ← tmp ;  
    retourner node ;  
Fin
```

Code C :

```
/**  
 * Permet de faire une rotaion droite de  
 * l'arbre sur un noeud  
 * @param node le noeud sur lequel on ap-  
 * plique la rotation  
 * @return l'arbre avec la rotaton droite ef-  
 * fectuée  
 */  
Node* rightRotation(Node* node){  
    Node* tmp = node->left;  
    node->left = tmp->right;  
    tmp->right = node;  
    node = tmp;  
    return node;  
}
```

```
Pacours suffixe :  
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]  
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]  
13 -> NOIR | [Fils droit : 17]  
17 -> ROUGE | [Fils gauche : 14]  
14 -> NOIR  
9 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]  
5 -> NOIR  
2 -> NOIR
```

```
Pacours infixe :  
17 -> ROUGE | [Fils gauche : 14]  
14 -> NOIR  
13 -> NOIR | [Fils droit : 17]  
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]  
9 -> NOIR  
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]  
5 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]  
2 -> NOIR
```

```
Pacours postfixe :  
14 -> NOIR  
17 -> ROUGE | [Fils gauche : 14]  
13 -> NOIR | [Fils droit : 17]  
9 -> NOIR  
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]  
5 -> NOIR  
2 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]  
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]
```

```
Pacours suffixe :  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 7]  
7 -> NOIR | [Fils gauche : 5, Fils droit : 11]  
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]  
13 -> NOIR | [Fils droit : 17]  
17 -> ROUGE | [Fils gauche : 14]  
14 -> NOIR  
9 -> NOIR  
5 -> NOIR  
2 -> NOIR
```

```
Pacours infixe :  
17 -> ROUGE | [Fils gauche : 14]  
14 -> NOIR  
13 -> NOIR | [Fils droit : 17]  
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]  
9 -> NOIR  
7 -> NOIR | [Fils gauche : 5, Fils droit : 11]  
5 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 7]  
2 -> NOIR
```

```
Pacours postfixe :  
14 -> NOIR  
17 -> ROUGE | [Fils gauche : 14]  
13 -> NOIR | [Fils droit : 17]  
9 -> NOIR  
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]  
5 -> NOIR  
7 -> NOIR | [Fils gauche : 5, Fils droit : 11]  
2 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 7]
```

Fonction rotationGauche

Algorithme :

```
Fonction leftRotation(Node* node) : Node*  
    Node* tmp ← node → right ;  
    node → right ← tmp → left ;  
    tmp → left ← node ;  
    node ← tmp ;  
    retourner node ;  
Fin
```

Code C :

```
/**  
 * Permet de faire une rotaion gauche de  
 * l'arbre sur un noeud  
 * @param node le noeud sur lequel on ap-  
 * plique la rotation  
 * @return l'arbre avec la rotaton gauche ef-  
 * fectuée  
 */  
Node* leftRotation(Node* node){  
    Node* tmp = node->right;  
    node->right = tmp->left;  
    tmp->left = node;  
    node = tmp;  
    return node;  
}
```

```
Pacours sufuxe :  
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]  
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]  
13 -> NOIR | [Fils droit : 17]  
17 -> ROUGE | [Fils gauche : 14]  
14 -> NOIR  
9 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]  
5 -> NOIR  
2 -> NOIR
```

```
Pacours infixe :  
17 -> ROUGE | [Fils gauche : 14]  
14 -> NOIR  
13 -> NOIR | [Fils droit : 17]  
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]  
9 -> NOIR  
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]  
5 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]  
2 -> NOIR
```

```
Pacours postfixe :  
14 -> NOIR  
17 -> ROUGE | [Fils gauche : 14]  
13 -> NOIR | [Fils droit : 17]  
9 -> NOIR  
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]  
5 -> NOIR  
2 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]  
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]
```

```
Pacours sufuxe :  
11 -> ROUGE | [Fils gauche : 7, Fils droit : 13]  
13 -> NOIR | [Fils droit : 17]  
17 -> ROUGE | [Fils gauche : 14]  
14 -> NOIR  
7 -> NOIR | [Fils gauche : 3, Fils droit : 9]  
9 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]  
5 -> NOIR  
2 -> NOIR
```

```
Pacours infixe :  
17 -> ROUGE | [Fils gauche : 14]  
14 -> NOIR  
13 -> NOIR | [Fils droit : 17]  
11 -> ROUGE | [Fils gauche : 7, Fils droit : 13]  
9 -> NOIR  
7 -> NOIR | [Fils gauche : 3, Fils droit : 9]  
5 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]  
2 -> NOIR
```

```
Pacours postfixe :  
14 -> NOIR  
17 -> ROUGE | [Fils gauche : 14]  
13 -> NOIR | [Fils droit : 17]  
9 -> NOIR  
5 -> NOIR  
2 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]  
7 -> NOIR | [Fils gauche : 3, Fils droit : 9]  
11 -> ROUGE | [Fils gauche : 7, Fils droit : 13]
```

Fonction minimum

Algorithme :

```
Fonction minimum(Node* tree, entier mini) :
Entier
    Si (tree == NULL) alors :
        retourner mini ;
    Sinon
        entier min ← tree → value ;
        entier minLeft ← minimum(tree → left,
min) ;
        entier minRight ← minimum(tree →
right, min) ;
        Si (minLeft < min) min ← minLeft ;
        Si (minRight < min) min ← minRight ;
        Retourner min ;
    Fin Si
Fin
```

Code C :

```
/**
 * Permet de retourner la plus petite valeur
 * présente dans l'arbre
 * @param tree l'arbre dont on veut la plus
 * petite valeur
 * @param mini la valeur minimum actuelle
 * (appelle récursive)
 * @return
 */
int minimum(Node* tree, int mini) {
    if (tree == NULL)
        return mini;
    else {
        int min = tree->value;
        int minLeft = minimum(tree->left, min);
        int minRight = minimum(tree->right,
min);

        if (minLeft < min) min = minLeft;
        if (minRight < min) min = minRight;
        return min;
    }
}
```

Pacours infixe :

```
17 -> ROUGE
13 -> NOIR | [Fils droit : 17]
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]
9 -> NOIR
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]
5 -> NOIR
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]
2 -> NOIR
```

Minimum : 2

Fonction maximum

Algorithme :

```
Fonction maximum(Node* tree, entier  
maxi) : Entier  
  Si (tree == NULL) alors :  
    retourner maxi ;  
  Sinon  
    entier max  $\leftarrow$  tree  $\rightarrow$  value ;  
    entier maxLeft  $\leftarrow$  maximum(tree  $\rightarrow$  left,  
max) ;  
    entier maxRight  $\leftarrow$  maximum(tree  $\rightarrow$   
right, max) ;  
    Si (maxLeft < max) max  $\leftarrow$  maxLeft ;  
    Si (maxRight < max) max  $\leftarrow$  maxRight ;  
    Retourner max ;  
  Fin Si  
Fin
```

Code C :

```
/**  
 * Permet de retourner la plus grande valeur  
 * présente dans l'arbre  
 * @param tree l'arbre dont on veut la plus  
 * grande valeur  
 * @param maxi la valeur maximale actuelle  
 * (appelle récursive)  
 * @return  
 */  
int maximum(Node* tree, int maxi) {  
  if (tree == NULL)  
    return maxi;  
  else {  
    int max = tree->value;  
    int maxLeft = maximum(tree->left, maxi);  
    int maxRight = maximum(tree->right,  
max);  
  
    if (maxLeft > max) max = maxLeft;  
    if (maxRight > max) max = maxRight;  
    return max;  
  }  
}
```

Pacours infixe :

```
17 -> ROUGE  
13 -> NOIR | [Fils droit : 17]  
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]  
9 -> NOIR  
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]  
5 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]  
2 -> NOIR
```

Maximum : 17

Fonction contient

Algorithme :

```
Fonction contains(Node* tree, entier value) :  
entier  
    Si (tree == NULL) alors :  
        retourner 0 ;  
    Fin Si  
    Si (tree == value) alors :  
        retourner 1 ;  
    Fin Si  
    Si (contains(tree → left, value)) alors :  
        retourner 1 ;  
    Fin Si  
    retourne contains(tree → right, value) ;  
Fin
```

Code C :

```
/**  
 * Permet de vérifier si une valeur est pré-  
 * sente dans l'arbre  
 * @param tree l'arbre  
 * @param value la valeur à chercher  
 * @return 1 => présente 0 => non présente  
 */  
int contains(Node* tree, int value) {  
    if(tree == NULL)  
        return 0;  
    if(tree->value == value)  
        return 1;  
    if(contains(tree->left, value))  
        return 1;  
    return contains(tree->right, value);  
}
```

```
Pacours infixe :  
17 -> ROUGE  
13 -> NOIR | [Fils droit : 17]  
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]  
9 -> NOIR  
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]  
5 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]  
2 -> NOIR
```

```
13 est-il present dans l'arbre ? Oui  
21 est-il present dans l'arbre ? Non
```


Fonction hauteur

Algorithme :

```
Fonction height(Node* tree) : entier
  Si (tree == NULL) alors :
    retourner 0 ;
  Sinon
    entier hg ← height(tree → left) ;
    entier hd ← height(tree → right) ;
    Si (hg > hd) alors :
      retourner (hg+1) ;
    Sinon
      retourner (hd+1) ;
  Fin Si
Fin Si
Fin
```

Code C :

```
/**
 * Permet de calculer la hauteur d'un arbre
 * @param tree l'arbre dont on veut la hauteur
 * @return la hauteur de l'arbre
 */
int height(Node* tree) {
  if (tree == NULL) return 0;
  else {
    int hg = height(tree->left);
    int hd = height(tree->right);
    return hg > hd ? (hg + 1) : (hd + 1);
  }
}
```

Pacours infixe :

```
17 -> ROUGE
13 -> NOIR | [Fils droit : 17]
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]
9 -> NOIR
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]
5 -> NOIR
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]
2 -> NOIR
```

Hauteur : 4

Fonction hauteurNoir :

Algorithme :

```
Fonction countBlackHeight(Node* tree) :
entier
  Si (tree == NULL) alors
    retourner 0 ;
  Fin Si
  entier hl ← countBlackHeight(tree →
right) ;
  entier hr ← countBlackHeight(tree → left) ;
  Si (tree → color == NOIR) alors :
    entier add ← 1 ;
  Sinon
    entier add ← 0 ;
  Fin Si
  Si (hl == -1 OU hr == -1 OU hl != hr) alors :
    retourner -1 ;
  Sinon
    retourner (hl + add) ;
  Fin Si
Fin
```

Code C :

```
/**
 * Permet de calculer la hauteur noire d'un
 * arbre R&N
 * @param tree l'arbre dont on veut la hau-
 * teur noire
 * @return la hauteur noire
 */
int countBlackHeight(Node* tree) {
  if (tree == NULL) return 0;
  int hl = countBlackHeight(tree->right);
  int hr = countBlackHeight(tree->left);
  int add = tree->color == NOIR ? 1 : 0;
  if (hl == -1 || hr == -1 || hl != hr)
    return -1;
  else
    return hl + add;
}
```

```
Pacours infixe :
17 -> ROUGE
13 -> NOIR | [Fils droit : 17]
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]
9 -> NOIR
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]
5 -> NOIR
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]
2 -> NOIR
```

```
Hauteur noir : 2
```


Fonction estRN

Algorithme :

```
Fonction isRN(Node* tree) : entier
    Si (tree → color == ROUGE ET (tree → left
    == NULL OU tree → right == NULL OU tree →
    right → color == ROUGE OU tree → left →
    color == ROUGE)) alors :
        retourner 0 ;
    Sinon Si ((tree → right == NULL ET tree →
    left == NULL) OU (tree → left == NULL ET tree
    → right == NULL)) alors :
        retourner 0 ;
    Sinon Si (countBlackHeight(tree) > 0)
    alors :
        retourner 1 ;
    Sinon
        retourner 0 ;
    Fin si
Fin
```

Code C :

```
/**
 * Permet de vérifier si un arbre est bien un
 * arbre bicolore (rouge et noir)
 * @param tree l'arbre
 * @return 0 => n'est pas R&N 1 => est R&N
 */
int isRN(Node* tree){
    if(tree->color == ROUGE && (tree->left ==
    NULL || tree->right == NULL || tree->right-
    >color == ROUGE || tree->left->color ==
    ROUGE)){
        return 0;
    }
    else{
        if((tree->right == NULL && tree->left ==
    NULL) || (tree->left == NULL && tree->right
    == NULL)){
            return 0;
        }
        else{
            if(countBlackHeight(tree) > 0)
                return 1;
            else
                return 0;
        }
    }
}
```

```
Pacours infixe :
17 -> ROUGE
13 -> NOIR | [Fils droit : 17]
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]
9 -> NOIR
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]
5 -> NOIR
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]
2 -> NOIR
```

Est-il un arbre R&N ? Oui

Fonction nombreNode

Algorithme :

Fonction countNodes(Node *tree) : entier
Si (tree == NULL) alors :
 retourn 0 ;
Fin Si
 retourner (1+ countNodes(tree → right) +
countNodes(tree → left) ;
Fin

Code C :

```
/**  
 * Permet de compter le nombre du noeud  
 d'un arbre  
 * @param tree * @return nombre de noeud  
 de l'arbre  
 */  
int countNodes(Node *tree) {  
    if (tree == NULL) return 0;  
    return 1 + countNodes(tree->right) +  
countNodes(tree->left);  
}
```

Pacours infixe :

```
17 -> ROUGE  
13 -> NOIR | [Fils droit : 17]  
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]  
9 -> NOIR  
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]  
5 -> NOIR  
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]  
2 -> NOIR
```

Nombre de noeud : 8

Fonction supprimer

Algorithme :

Fonction delete(Node *tree) : void
Si (tree != NULL) alors :
 delete(tree → right) ;
 delete(tree → left) ;
 free(tree) ;
Fin Si
Fin

Code C :

```
/**  
 * Permet de supprimer un arbre  
 * @param tree l'arbre à supprimer  
 */  
void delete(Node *tree){  
    if (tree != NULL){  
        delete(tree->right);  
        delete(tree->left);  
        free(tree);  
    }  
}
```

Fonction affichage pour les parcours

Code C :

```
/**
 * Permet d'afficher un noeud avec sa valeur, sa couleur et ses fils
 * @param node le noeud à afficher
 */
void display(Node *node){
    if(node->color == 0 || node->color == 1){
        if(node->left != NULL && node->right == NULL){
            printf("%d -> %s | [Fils gauche : %d]\n", node->value, convert(node->color), node->left->value);
        } else if (node->left == NULL && node->right != NULL){
            printf("%d -> %s | [Fils droit : %d]\n", node->value, convert(node->color), node->right->value);
        } else if (node->left != NULL && node->right != NULL){
            printf("%d -> %s | [Fils gauche : %d, Fils droit : %d]\n", node->value, convert(node->color), node->left->value, node->right->value);
        } else {
            printf("%d -> %s\n", node->value, convert(node->color));
        }
    } else {
        if(node->left != NULL && node->right == NULL){
            printf("%d | [Fils gauche : %d]\n", node->value, node->left->value);
        } else if (node->left == NULL && node->right != NULL){
            printf("%d | [Fils droit : %d]\n", node->value, node->right->value);
        } else if (node->left != NULL && node->right != NULL){
            printf("%d | [Fils gauche : %d, Fils droit : %d]\n", node->value, node->left->value, node->right->value);
        } else {
            printf("%d\n", node->value);
        }
    }
}
```

Fonction conversion

Algorithme :

Fonction convert(entier val) : char*

Si (val==0) Alors

Retourner « NOIR » ;

Sinon Si (val==1) Alors

Retourner « ROUGE » ;

Fin Si

Retourner « NULL » ;

Fin

Code C :

```
/**
 * Permet de retourner la couleur d'un
 * noeud
 * @param val valeur de la couleur (0 ou 1)
 * @return la couleur (string)
 */
char* convert(int val) {
    if (val == 0) return "NOIR";
    else if (val == 1) return "ROUGE";
    return "NULL";
}
```

Fonction parcoursSufixé

Algorithme :

Fonction sufixe(Node *tree) : void

Si (tree != NULL) alors :

display(tree) ;

sufixe(tree → right) ;

sufixe(tree → left) ;

Fin Si

Fin

Code C :

```
/**
 * Permet de faire le parcours sufixé d'un
 * arbre
 * @param tree l'arbre que l'on veut parcourir
 */
void sufixe(Node *tree){
    if (tree != NULL){
        display(tree);
        sufixe(tree->right);
        sufixe(tree->left);
    }
}
```

Parcours sufixe :

```
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]
13 -> NOIR | [Fils droit : 17]
17 -> ROUGE
9 -> NOIR
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]
5 -> NOIR
2 -> NOIR
```

Fonction parcoursInfixé

Algorithme :

```
Fonction infixe(Node *tree) : void
  Si (tree != NULL) alors :
    infixe(tree → right) ;
    display(tree) ;
    infixe(tree → left) ;
  Fin Si
Fin
```

Code C :

```
/**
 * Permet de faire le parcours infixé d'un
 * arbre
 * @param tree l'arbre que l'on veut parcourir
 */
void infixe(Node *tree){
  if (tree != NULL){
    infixe(tree->right);
    display(tree);
    infixe(tree->left);
  }
}
```

```
Parcours infixé :
17 -> ROUGE
13 -> NOIR | [Fils droit : 17]
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]
9 -> NOIR
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]
5 -> NOIR
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]
2 -> NOIR
```

Fonction parcoursPostfixé

Algorithme :

```
Fonction postfixe(Node *tree) : void
    Si (tree!=NULL) Alors
        postfixe(tree→ right) ;
        postfixe(tree→ left) ;
        display(tree) ;
    Fin Si
Fin
```

Code C :

```
/**
 * Permet de faire le parcours postfixé d'un
 * arbre
 * @param tree l'arbre que l'on veut parcourir
 */
void postfixe(Node *tree){
    if (tree != NULL){
        postfixe(tree->right);
        postfixe(tree->left);
        display(tree);
    }
}
```

```
Parcours postfixe :
17 -> ROUGE
13 -> NOIR | [Fils droit : 17]
9 -> NOIR
11 -> ROUGE | [Fils gauche : 9, Fils droit : 13]
5 -> NOIR
2 -> NOIR
3 -> ROUGE | [Fils gauche : 2, Fils droit : 5]
7 -> NOIR | [Fils gauche : 3, Fils droit : 11]
```

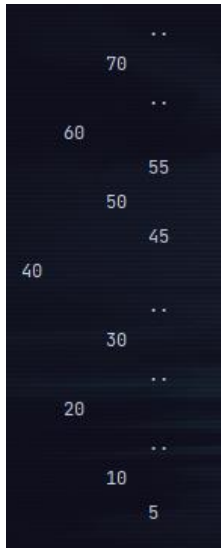
Fonction afficher graphiquement

Algorithme :

```
Fonction formatTree(Node *tree, entier
level, entier levelMax) : void
    entier i ;
    Si (tree != NULL) alors :
        formatTree(tree → right, level + 1,
levelMax) ;
        Pour i allant de 0 à level faire :
            Afficher(«   ») ;
        Fin Pour
        Si (tree → color == ROUGE OU tree →
color == NOIR) alors :
            Afficher(« %s (%d) \n », convert(tree
→ color), tree → value) ;
        Sinon
            Afficher(« %d \n », tree → value) ;
        Fin Si
        formatTree(tree → left, level + 1 ,
levelMax) ;
        Sinon
            Si (level < levelMax) alors :
                formatTree(NULL, level + 1 ,
levelMax) ;
            Pour i allant de 0 à level faire :
                Afficher(«   ») ;
            Fin Pour
            Afficher(« ..\n ») ;
            formatTree(NULL, level + 1, levelMax) ;
        Fin Si
    Fin Si
Fin
```

Code C :

```
/**
 * Permet d'afficher un arbre de façon gra-
 * phique
 * @param tree l'arbre
 * @param level la hauteur minimum = 0
 * @param levelMax la hauteur de l'arbre
 */
void formatTree(Node *tree, int level, int lev-
elMax){
    int i;
    if(tree != NULL){
        formatTree(tree->right, level + 1, lev-
elMax);
        for(i=0; i < level; i++){
            printf("   ");
        }
        if(tree->color == ROUGE || tree->color
== NOIR){
            printf("%s (%d)\n", convert(tree-
>color), tree->value);
        } else {
            printf("%d\n", tree->value);
        }
        formatTree(tree->left, level + 1, lev-
elMax);
    }else{
        if(level < levelMax){
            formatTree(NULL, level + 1, levelMax);
            for(i=0; i < level; i++){
                printf("   ");
            }
            printf(".. \n");
            formatTree(NULL, level + 1, levelMax);
        }
    }
}
```



IX. Huffman

Structure pour Huffman

<p>Algorithme : struct Node : char symbole ; struct Node *pere ; struct Node *left ; struct Node *right ; Fin Node ;</p> <p>struct Data : char symbole ; entier poids ; Fin Node ;</p> <p>struct Forest : struct Node *tree ; entier poids ; Fin Forest ;</p>	<p>Code C : typedef struct Node{ char symbole; struct Node *pere; struct Node *left; struct Node *right; } Node;</p> <p>typedef struct Data{ char symbole; int poids; } Data;</p> <p>typedef struct Forest{ struct Node *tree; int poids; } Forest;</p>
--	---

Fonction initialisation des données

<p>Algorithme : Début : Data* data← allouer(size *sizeof(Data)) ; entier i ; Pour i allant de 0 à size Faire data[i].symbole← symboles[i] ; data[i].poids← poids[i] ; Fin Pour Retourner data ; Fin</p>	<p>Code C : Data* initData(const char *symboles, const int *poids, int size){ Data* data = malloc(size * sizeof(Data)); int i; for(i = 0; i < size; i++){ data[i].symbole = symboles[i]; data[i].poids = poids[i]; } return data; }</p>
---	--

Fonction de la forêt

Algorithme :

```
Début :  
Forest *forest ← allouer(size * sizeof(Forest)) ;  
Pour i allant de 0 à size Faire  
    forest[i].tree ← allouer(sizeof(Node)) ;  
    tree ← forest[i].tree ;  
    tree → pere ← NULL ;  
    tree → left ← NULL ;  
    tree → right ← NULL ;  
    tree → symbole ← data[i].symbole ;  
    forest[i].poids ← data[i].poids ;  
Fin Pour  
Retourner forest ;  
Fin
```

Code C :

```
Forest* initForest(Data *data, Node* tree, int  
size){  
    Forest *forest = malloc(size * sizeof(Forest))  
    ;  
    for(int i=0; i<size; i++){  
        forest[i].tree = malloc(sizeof(Node));  
        tree = forest[i].tree;  
        tree->pere = NULL;  
        tree->left=NULL;  
        tree->right=NULL;  
        tree->symbole = data[i].symbole;  
        forest[i].poids = data[i].poids;  
    }  
    return forest;  
}
```

Fonction père

Algorithme :

```
Début :  
Node *racine ← allouer(sizeof(Node)) ;  
racine → pere ← NULL ;  
racine → left ← left ;  
racine → roght ← right ;  
left → pere ← racine ;  
roght → pere ← racine ;  
retourner racine ;  
Fin
```

Code C :

```
Node* pere(Node* left, Node* right){  
    Node *racine = malloc(sizeof(Node));  
    racine->pere = NULL;  
    racine->left = left;  
    racine->right = right;  
    left->pere = racine;  
    right->pere = racine;  
  
    return racine;  
}
```

Fonction huffman

<p>Algorithme :</p> <p>Variables :</p> <p>i, j ,n=sizeof(*forest) : entier</p> <p>Début :</p> <p>Tant que (n>1) Faire</p> <p> findIndice(forest, i, j) ;</p> <p> tree← pere(forest[i].tree, forest[j].tree) ;</p> <p> forest[i].poid←</p> <p> forest[i].poids+forest[j].poids ;</p> <p> forest[j].poids← forest[n].poids ;</p> <p> forest[j].tree← forest[n].tree ;</p> <p> forest[i].tree← tree ;</p> <p> n- ;</p> <p>Fin Tant Que</p> <p>Fin</p>	<p>Code C :</p> <pre>void huffman(Forest *forest, Node *tree){ int i, j; int n = sizeof(*forest); while(n > 1){ findIndice(forest, i, j); tree = pere(forest[i].tree, forest[j].tree); forest[i].poids = forest[i].poids + forest[j].poids; forest[j].poids = forest[n].poids; forest[j].tree = forest[n].tree; forest[i].tree = tree; n--; } }</pre>
--	---

Fonction affichage des données

<p>Algorithme :</p> <p>Variables :</p> <p>Début :</p> <p>Pour i allant de 0 à size Faire</p> <p> Ecrire (« %c », data[i].symbole) ;</p> <p>Fin Pour</p> <p>Pour i allant de 0 à size Faire</p> <p> Ecrire (« %d », data[i].poids) ;</p> <p>Fin Pour</p> <p>Fin</p>	<p>Code C :</p> <pre>void showData(Data *data, int size){ for(int i = 0; i < size; i++){ printf(" %c ", data[i].symbole); } printf("\n"); for(int i = 0; i < size; i++){ printf(" %d ", data[i].poids); } printf("\n"); }</pre>
---	--

Tableau des donnees:

```
| A | | F | | G | | N | | O | | P |
| 10 | | 8 | | 3 | | 25 | | 32 | | 7 |
```