

## INGÉNIERIE LOGICIELLE ET INTEGRATION DES SYSTÈMES INFORMATIQUES

# Algorithmique et programmation avancée

## Compte rendu des Tri

## 

# *Encadré par :*

# **Mr. BEKKHOUCHA**

# *Réalisé par* :

* **IJJIG HIBA**
* **ABID SALAH EDDINE**

***ANNÉE UNIVERSITAIRE* : *2020-2021***

I. INTRODUCTION:

Dans ce tp nous allons présenter les différentes méthodes de tri :

* Tri à bulles.
* Tri par insertion.
* Tri sélection-Échange.
* Tri rapide.
* Tri par extraction.

Au début nous allons donner l’idée générale pour chaque méthode, ensuite nous allons passer à l’algorithme et puis nous donnerons le programme et on va citer la complexité pour chaque méthode.

II.structures de données:

Mise en œuvre à l’aide des tableaux:

1. **typedef struct**
2. {
3. **int** tab[NBMAX];
4. **int** nbElem;
5. }MaTable;
6. Mise en œuvre à l’aide des pointeurs:
7. **typedef struct** Cellule
8. {
9. **int** elem;
10. **struct** Cellule \*suivant;
11. **struct** Cellule \*precedent;
12. }Cellule;

III. Tri à Bulles:

A.Analyse:

Cette méthode consiste à parcourir les éléments en permutant chaque élément par son successeur s’il est plus grand jusqu’à la fin de la liste afin de placer le max en dernier rang et après on place avant ce dernier l’élément plus petit et plus grand que les autres éléments, et on continue ainsi jusqu’à l’obtention d’une liste complétement triée.

Exemple:

nous prenons la liste suivante: 8 11 3 15 0

**1ère itération:**

(8 11 3 15 0) puisque 8<11 on fait rien.

(8 3 11 15 0) puisque 11>3 on les permute.

On fait rien puisque 11<15

(8 3 11 0 15) puisque 11>3 on les permute.

**2ème itération:**

(3 8 11 0 15) puisque 8>3 on les permute.

(3 8 0 11 15) 8<11 on fait rien et 11>0 on les échange et on passe à l’itération suivante.

**3ème itération:**

(3 0 8 11 15) 3<8 on fait rien et 8>0 on les échange et on passe à la prochaine itération.

**4ème itération:**

(0 3 8 11 15) 3>0 on les échange et finalement on obtient une liste complètement triée.

B.Analyse fonctionnel:

triABullesTab(T:MaTable)

Variable:

ind1,ind2:entier.

Tri:boolean.

Début:

tri← faux;

tant que (non(tri)) alors:

Début:

tri← vrai;

pour ind1 allant de T.nbElem à 1 par -1 faire:

pour ind2 allant de 1 à ind2-1 faire:

si(T.tab[ind2]>T.tab[ind2+1])alors:

Début:

echange(T.tab[ind2],T.tab[ind2+1]);

tri← faux;

Finsi

Fin tant que.

Fin.

Cette fonction permet d’échanger 2 variables entières.

echange(a,b:entier)

Variables:

aux entiers.

Début:

aux=a;

a=b;

b=aux;

Fin.

C.Programme:

a.échange:

1. **/\*\***
2. **Nom :echange.**
3. **Entrée:deux pointeurs.**
4. **Rôle :échanger 2 éléments entre eux.**
5. **\*\*/**
6. **void** echange(**int** \*a,**int** \*b)
7. {
8. **int** aux;
9. aux=\*a;
10. \*a=\*b;
11. \*b=aux;
12. }**///fin echange(int \*a,int \*b).**
13. b.mise en oeuvre à l’aide du tableau:
14. **/\*\* Nom: triABullesTab**
15. **Entree : un pointeur vers la structure de type tableau**
16. **Sortie : un tableau trié**
17. **Rôle : Cette fonction permet de trier le tableau en**
18. **se basant sur le principe du tri\_a\_bulles**
19. **cité ci-dessus (rapport).**
20. **\*\*/**
21. **void** triABullesTab(MaTable \*table)
22. {
23. **int** ind1,ind2,trie;
24. trie=0;
25. **///tant que le tableau n'est pas trié.**
26. **while**(!trie)
27. {
28. trie=1;
29. **///la position où on s'arrête.**
30. **for**(ind1=table->nbElem-1;ind1>=0;ind1--)
31. **///on parcours la table.**
32. **for**(ind2=0;ind2<ind1;ind2++)
33. **///si l'element est sup à son successeur.**
34. **if**(table->tab[ind2]>table->tab[ind2+1])
35. {
36. **///on échange et on met tri à 0.**
37. echange(&table->tab[ind2],&table->tab[ind2+1]);
38. trie=0;
39. }**///fin if(table->tab[ind2]>table->tab[ind2+1]).**
40. }**///fin while(!trie).**
41. }**///fin triABullesTab(MaTable \*table).**
42. c.mise en œuvre à l’aide du pointeur:

1. Cellule\*triABullesPoint(Cellule \*liste)
2. {
3. Cellule \*crt1,
4. \*crt2,
5. \*dernierElem;
6. **int** trie=0;
7. **while**(!trie)
8. {
9. trie=1;
10. dernierElem=dernier(liste);
11. **///la position où on s'arrête.**
12. **for**(crt1=dernierElem;crt1->precedent;crt1=crt1->precedent)
13. {
14. **///on parcours la table.**
15. **for**(crt2=liste;crt2!=crt1;crt2=crt2->suivant)
16. **///si l'element est sup à son successeur.**
17. **if**(crt2->elem>crt2->suivant->elem)
18. {
19. **///on échange et on met tri à 0.**
20. echange(&crt2->elem,&crt2->suivant->elem);
21. trie=0;
22. }
23. }**///fin for.**
24. }**///fin while(!trie).**
25. **return** (Cellule\*)liste;
26. }**///fin triABullesPoint(Cellule \*liste).**

D.Complexité:

Dans le meilleur cas avec des données déjà triées on effectue n-1 comparaisons c’est à dire de complexité O(n), mais dans le cas d’un tableau inversement trié on effectue

comparaisons c’est à dire O() .

IV. Tri par insertion

A.Analyse:

Le premier élément constitue une liste à lui tout seul, on range les deux premiers éléments pour constituer une liste triée à deux éléments, après on range les 3 premiers pour obtenir une liste triée à trois éléments...

Exemple:

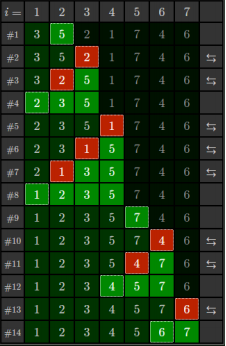


Figure : Trace de l'exécution de l'algorithme du tri par insertion.

B.Analyse fonctionnel:

triInsert(T:MaTable)

Variables:

i,j:entiers.

Début:

pour i allant de 1 à T.nbElem faire:

j← i;

tant que((T.tab[j-1]>T.tab[j])et j>0) faire:

echange(T.tab[j],T.tab[j-1]);

j← j-1;

Fin tant que.

Fin pour.

Fin

C.Programme:

1. a.mise en oeuvre à l’aide du tableau:
2. **/\*\* Nom: triInsertion**
3. **Entree :un pointeur vers la structure de type tableau**
4. **Sortie : un tableau trié**
5. **Rôle : Cette fonction permet de trier le tableau en**
6. **se basant sur le principe du tri par insertion.**
7. **.**
8. **\*\*/**
9. **int** triInsert(MaTable \*table)
10. {
11. **int** ind1,ind2;
12. **///si la table n'existe pas.**
13. **if**(!table) **return** (**int**)0;
14. **///sinon on commence le tri.**
15. **for**(ind1=1;ind1<table->nbElem;ind1++)
16. {
17. ind2=ind1;
18. **///tant que l'élément est inf à son précedeur.**
19. **while**((table->tab[ind2]<table->tab[ind2-1])&&(ind2>0))
20. { **///on les échange.**
21. echange(table->tab+ind2,table->tab+ind2-1);
22. ind2--;
23. }**///fin while((table->tab[ind2]<table->tab[ind2-1])&&(ind2>0)).**
24. }**///fin for(ind1=1;ind1<table->nbElem;ind1++).**
25. **return** (**int**)1;
26. }**///triInsert(MaTable \*table).**
27. b.mise en oeuvre à l’aide du pointeur:
28. **/\*\* Nom: triInsertPoint**
29. **Entree :un pointeur vers la liste**
30. **Sortie : une liste triée.**
31. **Rôle : Cette fonction permet de trier une liste en**
32. **se basant sur le principe du tri par insertion.**
33. **\*\*/**
34. Cellule \*triInsertPoint(Cellule \*liste)
35. {
36. Cellule \*crt1,
37. \*crt2;
38. **///on commence le tri.**
39. **for**(crt1=liste->suivant;crt1;crt1=crt1->suivant)
40. {
41. crt2=crt1;
42. **///tant que l'élément est inf à son précedeur.**
43. **while**(crt2->elem<crt2->precedent->elem)
44. {
45. **///on les échange.**
46. echange(&crt2->elem,&crt2->precedent->elem);
47. crt2=crt2->precedent;
48. **if**(!crt2->precedent) **break**;
49. }**///fin while(crt2->elem<crt2->precedent->elem).**
50. }**///fin for(crt1=liste->suivant;crt1;crt1=crt1->suivant).**
51. **return** (Cellule\*)liste;
52. }**///fin triInsertPoint(Cellule \*liste).**

D.Complexité:

Dans le meilleur cas avec des données déjà triées on effectue n comparaisons c’est à dire de complexité O(n), mais dans le cas d’un tableau inversement trié on effectue (n-1)+(n-2)+...+1 comparaisons et échanges qui fait c’est à dire O() .

V. Tri par sélection-echange

A.Analyse:

Pour réaliser un tri croissant avec cette méthode, on cherche l’indice du minimum du tableau à partir de la position courante, si cet indice est différent de notre position on fait l’échange, si non on se déplace directement vers la case suivante et on répète le même traitement.

**exemple:**

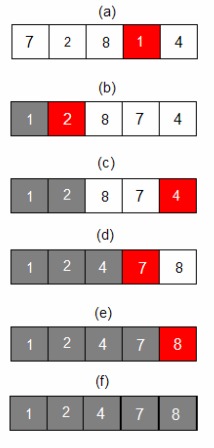
****

Figure : Trace de l'exécution de l'algorithme du tri par sélection-échange..

B.Analyse fonctionnel:

Tri\_par\_selection (table:MaTable)

variables:

ind,tmp,ind\_min:entiers

Début:

pour (ind allant de 1 à (NbElem-1)) faire

début

ind\_min← emp\_min(table,ind);

si(ind\_min<>ind)

tmp← table->tab[ind];

table->tab[ind]<- table->tab[ind\_min];

table->tab[ind\_min]<- tmp;

finSi

fin

Fin

emp\_min(table:MaTable,indice:entier)

varieble:

crt,ind\_min:entiers

début

ind\_min← indice;

pour (crt allant de (indice+1)à NbElem) faire

début

si(table→ tab[crt]<table→ tab[ind\_min]) alors

ind\_min←crt;

fin

retourner(ind\_min);

fin

C.Programme:

1. a.mise en oeuvre à l’aide du tableau:
2. **int** emp\_min (MaTable t,**int** pos)
3. {
4. **int** crt,,//pour le parcour
5. ind\_min;//pour garder l'indice du min
6. ind\_min=pos;//initialisation
7. //recherche du min
8. **for**(crt=(pos+1);crt<t->nbElem;crt++)
9. {
10. **if**(t->tab[crt]<t->tab[ind\_min]) ind\_min=crt;
11. }**///fin for(crt=(pos+1);crt<t->nbElem;crt++)**
12. **return** (**int**)ind\_min;
13. }**///fin int emp\_min (MaTable t,int pos)**
14. **void** tri\_par\_selection(MaTable \*t)
15. {
16. **int** crt,//pour le parcour
17. tmp,//pour l'echange
18. ind\_min;//pour garder l'indice du min
19. **for**(crt=0;crt<(t->nbElem-1);crt++)
20. {
21. //recherche du min
22. ind\_min=emp\_min(\*t,crt);
23. **if**(ind\_min=!crt)
24. {
25. tmp=t->tab[crt];
26. t->tab[crt]=t->tab[ind\_min];
27. t->tab[ind\_min]=tmp;
28. }**///fin if(ind\_min=!crt)**
29. }**/// fin for(crt=0;crt<(t->nbElem-1);crt++)**
30. }**///fin void tri\_par\_selection(MaTable \*t)**
31. b.mise en oeuvre à l’aide du pointeur:
32. Cellule \*emp\_min\_point(Cellule \*lst)
33. {
34. Cellule \*crt,//pour le parcour
35. \*min;//pour garder le min
36. min=lst;//initialisation
37. //recherche du min
38. **for**(crt=lst->suivant;crt;crt=crt->suivant)
39. {
40. **if**((crt->elem)<(min->elem)) min=crt;
41. }**///fin for(crt=lst->suivant;crt;crt=crt->suivant)**
42. **return** (Cellule\*)min;
43. }**///fin Cellule \*emp\_min\_point(Cellule \*lst)**
44. Cellule \*tri\_par\_selection\_point(Cellule \*liste)
45. {
46. Cellule \*crt,//pour le parcour
47. \*min;//pour garder le min
48. **int** tmp;//pour l'echange
49. **for**(crt=liste;crt;crt=crt->suivant)
50. {
51. min=emp\_min(crt);
52. **if**(min!=crt)
53. {
54. tmp=min->elem;
55. min->elem=crt->elem;
56. crt->elem=tmp;
57. }
58. }
59. **return** (Cellule\*)liste;
60. }

D.Complexité:

Pour trier n éléments, le tri par sélection effectue n(n-1)/2 comparaisons.

Sa complexité est donc Θ(n2).

VI. Tri rapide (quicksort)

A.Analyse:

Le principe de ce tri est d’ordonner une table de n éléments en choisissant un pivot, autour duquel on va organiser les autres éléments de telle sorte que les éléments qui sont inférieurs du pivot se trouvent dans les positions i telle que i < p : la position du pivot.

On refait ce traitement avec les deux sous tables , jusqu’à se trouver avec n sous table (n est la taille initiale du tableau).

**exemple :**

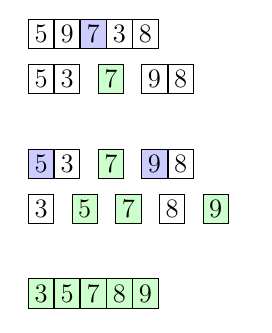


Figure : Trace de l'exécution de l'algorithme du tri rapide.

Au début le pivot est 7:

on le compare avec les autres éléments pour les bien placer .

Après l'exécution de la première étape, on obtient deux sous tables. On refait avec eux la même chose avec de nouveaux pivot (9 et 5) jusqu’à ce qu’on obtient des sous tables d’un seul élément.

B.Analyse fonctionnel:

**triRapide(T:MaTable,i:entier,j:entier)**

**Variables:**

**s:entier.**

**Début:**

**si(i<j) alors:**

**Début:**

**s=partition(T,i,j);**

**triRapide(T,i,s);**

**triRapide(T,s+1,j);**

**Fin si.**

**Fin.**

**partition(T:MaTable,gche:entier,dt:entier):entier**

**Variables:**

**pivot:entier.**

**Début:**

**pivot← T.tab[gche];**

**tant que (gche<dt) faire:**

**Début**

**echange(T.tab[gche],T.tab[dt]);**

**tant que(T.tab[gche]<=pivot) faire**

**gche← gche+1;**

**tant que(T.tab[dt]>pivot) faire**

**dt← dt-1;**

**fin tant que.**

**retourner (dt).**

**Fin**

C.Programme:

1. a.mise en oeuvre à l’aide du tableau:
2. **int** partition(MaTable \*T,**int** gche,**int** dt)
3. {
4. **int** pivot;
5. pivot=T->tab[gche];
6. **while**(gche<dt)
7. {
9. **///on echange le gauche avec le droit.**
10. echange(&T->tab[gche],&T->tab[dt]);
11. **///si le gauche est inferieur au pivot.**
12. **while**(T->tab[gche]<=pivot) gche++;
13. **///si le droit est supérieur au pivot.**
14. **while**(T->tab[dt]>pivot) dt--;
15. }**///fin while(i<j).**
16. **return** (**int**)dt;
17. }**///fin partition(MaTable \*T,int i,int j).**
18. **void** triRapide(MaTable \*T,**int** i,**int** j)
19. {
20. **int** s;
21. **if**(i<j)
22. {
23. **///la partition.**
24. s=partition(T,i,j);
25. **///le tableau gauche.**
26. triRapide(T,i,s);
27. **///le tableau droit.**
28. triRapide(T,s+1,j);
29. }**///fin if(i>j).**
30. }**///fin triRapide(MaTable \*T,int i,int j).**
31. b.mise en oeuvre à l’aide du pointeur:
32. Cellule\*Repartition(Cellule\*tete, Cellule\*queue, **int** pivot)
33. {
34. **int** trier = 1;
35. Cellule\*gauche = tete, \*droit = queue;
36. **while** (gauche != droit)
37. {
38. **while** (gauche->elem < pivot)gauche = gauche->suivant;
39. **while** ((droit->elem >= pivot) && (droit != gauche))droit = droit->precedent;
40. **if** (gauche != droit)
41. {
42. echange(gauche, droit);
43. trier = 0;
44. }//fin du if (gauche != droit)
45. }//fin du while (gauche != droit)
46. **if** (trier == 1)**return**((Cellule\*)tete->suivant);
47. **return**((Cellule\*)gauche);
48. }//fin de la fonction Repartition(Cellule\*tete, Cellule\*queue, int pivot)
49. **void** QuickSortList(Cellule\*tete, Cellule\*queue)
50. {
51. Cellule\*ptcrt;
52. **int** pivot;
53. **if** (tete != queue)
54. {
55. **if** (tete->elem < tete->suivant->elem)pivot = tete->suivant->elem;
56. **else** pivot = tete->elem;
57. ptcrt = Repartition(tete, queue, pivot);
58. QuickSortList(tete, ptcrt->precedent);
59. QuickSortList(ptcrt, queue);
60. }//fin du if (tete != queue)
61. }//fin de QuickSortList(Cellule\*tete, Cellule\*queue)

VII. Tri par extraction

**L'idée qui sous-tend cet algorithme consiste à voir le tableau comme un** [**arbre binaire**](https://fr.wikipedia.org/wiki/Arbre_binaire)**. Le premier élément est la racine, le deuxième et le troisième sont les deux descendants du premier élément. En général, chaque élément de position ‘i’ a pour fils gauche celui qui se trouve dans la position ‘2i’ et comme fils droit celui de la position ‘2i+1’.**

**Après la construction de l’arbre avec les éléments de la liste on cherche chaque fois le plus petit élément at on l'insère à la racine puis on l’échange avec l’élément le plus loin en diminuant le nombre d’éléments et on reprend cette étape jusqu’à ordonner tous les éléments**

**Exemple :**

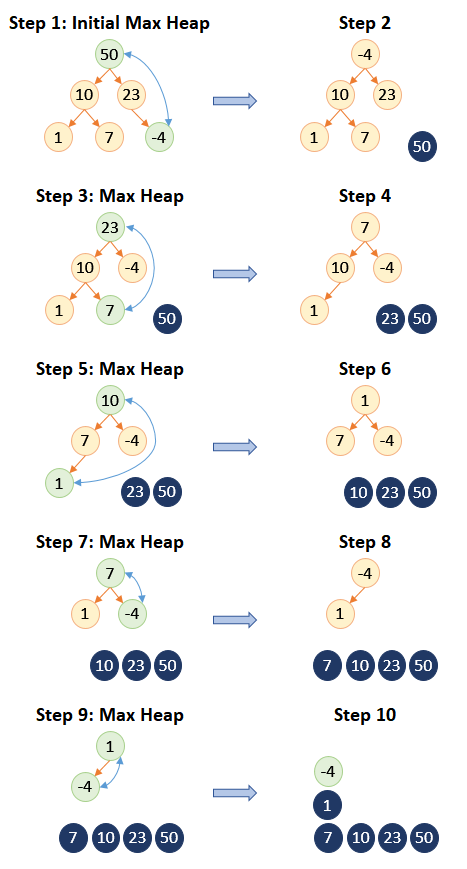
****

Figure : Trace de l'exécution de l'algorithme du tri par extraction.

B.Analyse fonctionnel:

**heapSort(T:MaTable)**

**Variables:**

**i:entier.**

**Début:**

**pour i allant de T.nbElem/2 à 1 par par -1 faire:**

**construireHeap(T,T.nbElem,i);**

**pour i allant de T.nbElem à 1 par -1 faire:**

**Début:**

**echange(T.tab[1],T.tab[i]);**

**construireHeap(T,i,1);**

**Fin pour.**

**Fin.**

**construireHeap(T:MaTable,n:entier,i:entier)**

**Variables:**

**gche,dt,indMax:entier.**

**Début:**

**indMax← i;**

**gche← 2\*i;**

**dt← 2\*i+1;**

**si(gche<n et T.tab[gche]>T.tab[indMax]) alors:**

**indMax← gche;**

**si(dt<n et T.tab[dt]>T.tab[indMax]) alors:**

**indMax← dt;**

**si(indMax<>i) alors:**

**Début:**

**echange(T.tab[i],T.tab[indMax]);**

**construireHeap(T,n,indMax);**

**Fin si.**

**Fin.**

C.Programme:

1. a.mise en oeuvre à l’aide du tableau:
2. **void** construireHeap(MaTable\*T,**int** n, **int** i)
3. {
4. **int** indMax=i,
5. gche =2\*i+1,
6. dt=2\*i+2;

/// chercher le max.

1. **if** (gche<n&&(T->tab[gche]>T->tab[indMax]))
2. indMax=gche;
3. **if** (dt<n&&T->tab[dt] > T->tab[indMax])
4. indMax = dt;
5. **if** (indMax != i)
6. {
7. echange(&T->tab[i],&T->tab[indMax]);
8. // on construit recursivement.
9. construireHeap(T, n, indMax);
10. }///fin if(indMax!=i).
11. }///fin construireHeap(MaTable\*T,int n,int i).
12. **void** heapSort(MaTable \*T, **int** n)
13. {
14. **int** i,j=n-1;
    * + - 1. ///on construit le heap pour tous les noeuds.
15. **for** (i=(n/2)-1;i>=0;i--)
16. {
17. afficher\_Liste(\*T);
18. construireHeap(T,n,i);
19. afficher\_Liste(\*T);
20. }
21. ///on tri.
22. **for**(i=n-1;i>0;i--)
23. {
    * 1. //on echange l’indice avec le debut.
      2. echange(&T->tab[0],&T->tab[i]);
      3. //on reconstruit le heap.
24. construireHeap(T,i,0);
25. }
26. }//heapSort(MaTable \*T,int n).

b.mise en oeuvre à l’aide du pointeur:

1. /\*----------------------------------------------------------------------
2. le nom de la fonction:RechercheFils.
3. les entrées:Cellule\*pere:la Cellule pere
4. int pospere :la position de Cellule pere dans la liste
5. les sorties:la fonction retourne la Cellule fils de position 2\*pospere
6. dans la liste.
7. le role:elle permet de rechercher la Cellule fils de pere si elle existe.
8. ------------------------------------------------------------------------\*/
9. Cellule\*RechercheFils(Cellule\*pere, **int** pospere)
10. {
11. **int** ind;
12. Cellule\*fils = pere;
13. **for** (ind = pospere; ind < 2 \* pospere; ind++)
14. {
15. **if** (fils == NULL)**break**;
16. fils = fils->suivant;
17. }
18. **return**((Cellule\*)fils);
19. }
20. /\*-----------------------------------------------------------------
21. le nom de la fonction:Descent.
22. les entrées:Cellule\*Debut,\*fin: les Cellules de debut et fin de sous
23. arbre a traiter
24. int posDebut: la position de Cellule debut dans la liste
25. les sorties:sous arbre ordonne
26. le role:elle permet de trier le sous liste\_arbre de telle maniere
27. qu'il soit verticalement ordonee.
28. --------------------------------------------------------------------\*/
29. **void** Descent(Cellule\*Debut,**int** posDebut)
30. {
31. Cellule\*Racine = Debut,\*fils=RechercheFils(Debut,posDebut);
32. **int** desent = 1;
33. **while** ((fils) && desent)
34. {
35. **if** (fils->suivant)
36. **if** (fils->elem < fils->suivant->elem)fils = fils->suivant;
37. **if** (Racine->elem < fils->elem)
38. {
39. echange(&fils->elem, &Racine->elem);
40. Racine = fils;
41. posDebut \*= 2;
42. fils = RechercheFils(fils, posDebut);
43. }//fin du if (Racine->elem < fils->elem)
44. **else** desent = 0;
45. }//fin du while ((fils) && desent)
46. }//fin de la fonction Descent(Cellule\*Debut, Cellule\*fin,int posDebut)
47. /\*---------------------------------------------------------------------
48. le nom de la fonction : create\_Heap.
49. les entrées : Cellule\*fin: la dernier element a traiter dans la liste.
50. int pos:la position de Cellule fin dans la liste
51. les sorties : un heap
52. le role : elle permet de trier l'arbre liste de telle maniere qu'il
53. soit verticalement ordone.
54. ----------------------------------------------------------------------\*/
55. **void** create\_Heap(Cellule\*fin,**int** pos)
56. {
57. Cellule\*deb=fin;
58. **while** (deb)
59. {
60. Descent(deb, pos);
61. pos--;
62. deb = deb->precedent;
63. }//fin du while (deb)
64. }//fin de la fonction create\_Heap(Cellule\*fin,int pos)
65. /\*---------------------------------------------------------------------
66. le nom de la fonction : tri\_extraction
67. les entrées : Cellule\*debut,\*fin:les Cellule de debut et la fin de la liste.
68. les sorties : la fonction retourne liste triée
69. le role:Cette fonction sert à trier une liste en utilisant la méthode de
70. tri par extraction.
71. ------------------------------------------------------------------------\*/
72. **void** tri\_extraction(Cellule\*debut, Cellule\*fin,**int** posfin)
73. {
74. Cellule\*crt = fin;
75. **while** (crt)
76. {
77. create\_Heap(crt, posfin);
78. echange(&crt->elem, &debut->elem);
79. crt = crt->precedent;
80. posfin--;
81. }//fin du while (crt)
82. }//fin de la fonction tri\_extraction(Cellule\*debut, Cellule\*fin,int posfin)

D.Complexité:

**Dans le pire des cas, la complexité du tri rapide est en Θ(n2). Par contre, dans le cas moyen, cet algorithme a une complexité en Θ(n log n).**

VIII.Conclusion:

**Pour ce Tp nous avons réalisé les différentes méthodes de tri pour un tableau et pour une liste doublement chaînée, au début nous avons donné l’idée pour chaque méthode et après nous avons élaboré son algorithme en les traduisant en langage C en ajoutant la complexité pour chaque algorithme.**