Université Paris-Nord
Institut Galilée

Algorithmique, arbres et graphes
Licence 2 - Année 2005-2006

TD 7 & 8 : Files de priorités et tris par tas

Préliminaires

Nous étudions ici la structure abstraite de *minimier*, et son utilisation pour représenter une file de priorité ou pour trier des éléments selon une clé (on parle alors de*tris par tas*). Cette structure est un arbre binaire et soit e une position dans l'arbre nous utiliserons les fonctions *pere(m,e)*, *filsG(m,e)*, *filsD(m,e)* pour désigner les positions du père de e dans l'arbre, et des fils gauche et droit de e. Les positions sont totalement ordonnées et nous noterons *rac* la position du premier élément, *dernier* celle du dernier élément, *succ(e)* le successeur de la position e dans cet ordre, *pred(e)* le prédécesseur de e. On utilisera de plus une fonction *plusGrand(e,f)* qui renvoie vrai si la position e est supérieure à la position f dans cet ordre.

Les éléments sont rangés dans l'arbre en suivant cet ordre : la racine est *rac*, puis les successeurs sont rangés par niveaux de gauche à droite. Tous les éléments ont une *clé* (par exemple un entier) et satisfont tous la propriété de *dominance* suivante : dans un minimier la clé d'un élément est toujours plus petite ou égale à celle de ses fils [1].

On supposera dans ce qui suit que *rac* est constant alors que *dernier* dépend du nombre d'éléments du minimier. Dans ce qui suit un minimier sera représenté par un tableau m d'éléments et la position *dernier* du dernier élément.

On notera *echange(m,e,f)* la fonction qui échange les éléments de position e et f dans l'arbre. Cette fonction est utilisée en particulier dans les deux principales opérations de mises à jour :

- *maintientMinimierBas(m, r, dernier)* qui est utilisée quand la clé de l'élément r a peut être augmenté. il faut alors descendre l'élément en r par échanges successifs jusqu'à ce que la propriété de dominance soit satisfaite.
- *maintientMinimierHaut (m, d)* qui est utilisée quand la clé de l'élément d a peut être diminué. il faut alors remonter l'élément en d par échanges successifs jusqu'à ce que la propriété de dominance soit satisfaite.

L'insertion d'un nouvel élément *el* se fait en agrandissant le minimier m, en mettant en dernière position le nouvel élément, puis en faisant une mise à jour du minimier. La fonction d'insertion s'écrit alors *insere(m, dernier, el)* et renvoie *succ(dernier)*.

La suppression de l'élément à la racine se fait en le remplaçant par le dernier élément, en réduisant le minimier et en faisant une mise à jour du minimier. La fonction de suppression s'écrit alors *supprimeRac(m,dernier)* et renvoie *pred(dernier)*.

Exercice 1 (maintenir un minimier.)

Voici deux exemples d'arbres binaires qui ne sont pas des minimiers car la propriété de dominance n'est pas respectée pour un des noeuds. Les noeuds sont donnés dans l'ordre à partir de la racine :

- $cle(rac) = 1., 4., 3., 8., 0., 5., 4.$
- $cle(rac) = 18., 4., 3., 8., 9., 5., 4.$

Donner à chaque fois, en notant n la position de l'élément fautif ainsi que la suite d'instructions permettant de maintenir le minimier, et dessiner les deux minimiers mis à jour.

Correction

1. l'élément fautif est celui de clé = 0. Il faut donc appliquer *maintientMinimierHaut(m,n)* ce qui revient à faire les échanges suivants :
   - *echange(n, pere(n))*
   - *echange(pere(n), pere(pere(n))*

   On obtient alors $cle(rac) = 0., 1., 3., 8., 4., 5., 4.$
2. l'élément fautif est celui de clé = 18., c'est à dire la racine. Il faut donc appliquer *maintientMinimierBas(m,n)* ce qui revient à faire les échanges suivants :

[1] En inversant la relation d'ordre on obtient un *maximier*.

– *echange(n, filsD(n))* (car *cle(filsD(n)) < cle(filsG(n))*)
– *echange(filsD(n), filsD(filsD(n)))* (car *cle(filsD(filsD(n))) < cle(filsG(filsD(n))*)
On obtient alors $cle(rac) = 3., 4., 4., 8., 9., 5., 18.$

fin Correction

Exercice 2 (L'ordre des positions et les fonctions associées.)

La manière classique de présenter l'implémentation d'un minimier est de prendre rac=1 et de considérer l'ordre direct *: les noeuds sont numérotés 1 , 2, ... , dernier. Cependant il peut être utile de conserver l'ordre direct et de prendre rac* $\neq 1$*, en particulier lorsqu'on utilise un langage comme le C où les tableaux sont indexés à partir de 0. Plus généralement ceci permet de ranger les éléments du minimier à un endroit quelconque dans un tableau. Il peut être également utile de le ranger en ordre* inverse *c'est à dire dans l'ordre rac, rac-1, ... , dernier. En particulier nous verrons plus loin que l'on peut ainsi ranger un nombre constant n d'éléments dans deux files de priorités représentées par un seul tableau de n éléments.*

On sait que dans un minimier en ordre direct avec $rac = 1$*, on a :*
– $pere(m, e) = \lfloor e/2 \rfloor$
– $filsG(m, e) = 2e$ *;* $filsD(m, e) = 2e + 1$
– $succ(e) = e + 1$ *;* $pred(e) = e - 1$
– $plusGrand(e, f) = e > f$

1. *Ecrire* $pere(m, e), filsG(m, e), filsD(m, e)$ *pour* $rac = 0$*, et plus généralement pour* $rac \neq 1$*, quand l'ordre est* direct*. Dessiner l'arbre des positions en ordre direct pour* rac $= 0$ *et* dernier=5 *ainsi que pour* rac $= 5$ *et* dernier=10*.*
2. *Ecrire* $succ(e), pred(e), plusGrand(e, f), pere(m, e), filsG(m, e), filsD(m, e)$ *pour* $rac = n$ *quelconque lorsque l'ordre est* inverse*. Dessiner l'arbre des positions en ordre inverse pour* rac $= 5$ *et* dernier=0*.*

Correction.
Evidemment il faut passer par de petis exemples simples pour trouver et vérifier les solutions.
On note x_d la représentation de la position x pour $rac = d$ en ordre direct et x^f celle de x pour $rac = f$ en ordre inverse. Ces représentations indiquent l'emplacement (entre 0 et $N - 1$), dans le tableau m représentant le minimier, de la position x.

1. – On a en ordre direct $x_d = x_1 + d - 1$ (la position de la racine s'ecrit $rac_1 = 1$ et $rac_d = d$ donc $rac_d = rac_1 + d - 1$). On a donc $pere_d(x_d) = pere_1(x_1) + d - 1 = \lfloor x_1/2 \rfloor + d - 1 = \lfloor (x_d - d + 1)/2 \rfloor + d - 1$.
Si $(x_d - d + 1)$ est pair on a $pere_d(x_d) = (x_d - d + 1)/2 + d - 1 = (x_d - d + 1 + 2d - 2)/2 = (x_d + d - 1)/2$ qui est pair et donc $pere_d(x_d) = \lfloor (x_d + d - 1)/2 \rfloor$.
Si $(x_d - d + 1)$ est impair on a $\lfloor (x_d - d + 1)/2 \rfloor = \lfloor (x_d - d)/2 \rfloor$ et donc $pere_d(x_d) = (x_d - d/2 + d - 1 = (x_d - d + 2d - 2)/2 = (x_d + d - 2)/2$ qui est pair et donc $pere_d(x_d) = \lfloor (x_d + d - 2)/2 \rfloor = \lfloor (x_d + d - 1)/2 \rfloor$.
On a donc toujours
$$pere_d(x_d) = \lfloor (x_d + d - 1)/2 \rfloor$$
– De même
(a) $filsG_p(m, x_d) = filsG_1(m, x_1) + d - 1 = 2x_1 + d - 1 = 2(x_d - d + 1) + d - 1 = 2x_d + 1 - d$
(b) $filsD_p(m, x_d) = filsG_p(m, x_d) + 1 = 2x_d + 2 - d$
– on a donc en particulier :
$$pere_0(x_0) = \lfloor (x_0 - 1)/2 \rfloor \ , \ filsG_0(m, x_0) = 2x_0 + 1 \ , \ filsD_0(m, x_0) = 2x_0 + 2$$
– On obtient l'arbre des indices 0 1 2 3 4 5 et celui des indices 5 6 7 8 9 10.
2. – En ordre inverse on a : $succ^I(e) = e - 1$, $pred^I(e) = e + 1$, $plusGrand(e, f) = e < f$.
– En ordre inverse on a : $x^f = f - x_1 + 1$ (la position de la racine s'ecrit $rac^f = f$ et $succ^I(rac^f) = rac^f - 1$ donc $rac^f = f - rac^1 + 1 = f$ et $succ^I(rac^f) = f - succ(rac^1) + 1 = f - 1, etc...$).

- On a donc $pere^f(x^f) = f - pere_1(x_1) + 1 = f - \lfloor x_1/2 \rfloor + 1 = f - \lfloor (f - x^f + 1)/2 \rfloor + 1$.
  Si $f - x^f + 1$ est impair alors $\lfloor f - x^f + 1/2 \rfloor = f - x^f/2$ et donc $f - \lfloor (f - x^f + 1)/2 \rfloor + 1 = (2f - f + x^f + 2)/2 = (f + x^f + 2)/2$ et le numérateur étant pair il vient :
  $pere^f(x^f) = \lfloor (f + x^f + 2)/2 \rfloor$.
  Si $f - x^f + 1$ est pair alors $\lfloor f - x^f + 1/2 \rfloor = f - x^f + 1/2$ et donc $f - \lfloor (f - x^f + 1)/2 \rfloor + 1 = (2f - f + x^f - 1 + 2)/2 = (f + x^f + 1)/2$ et le numérateur étant pair, $f + x^f + 2$ est impair et on peut aussi écrire :
  $pere^f(x^f) = \lfloor (f + x^f + 2)/2 \rfloor$.
- On a également :
  $filsG^f(x^f) = f - filsG_1(x_1) + 1 = f - 2(f - x^f + 1) + 1 = 2x^f - 1 - f$ et :
  $filsD^f(x^f) = 2x^f - 2 - f$.
- On obtient l'arbre des indices 5 4 3 2 1 0.

fin Correction.

Exercice 3 (Insertion d'un élément et *plantation* d'un minimier)

Le but de cet exercice est de ranger un tableau t de n éléments dans un minimier à partir de rac. Il s'agit à la fois de ranger les éléments et de s'assurer de ce que la propriété de dominance soit respectée pour tous les noeuds de l'arbre. Il y a deux manières de procéder que nous allons expérimenter, mais d'abord nous introduisons la fonction d'insertion *d'un élément dans le minimier.*

1. *Une opération primitive est l'insertion d'un nouvel élément el dans un minimier. On la note* insere(m, dernier, el).*On opère de la manière suivante : le minimier est agrandi en modifiant* dernier*, l'élément el est placé à la position* dernier *dans le tableaum, enfin on fait remonter l'élément el en utilisant la fonction* maintientMinimierHaut. *Insérer dans le minimier* $cle(rac) = 0., 1., 3., 8., 4., 5., 4.$ *un élément de clé* 2 *et donner la suite d'instructions utilisée pour cela.*
2. *La première manière de planter un minimier est d'insérer successivement dans le minimier m vide les* taille *éléments du tableau t,. La fonction se note* plante(t,n, m) *et renvoie la position du dernier élément. Donner la suite d'instruction permettant de planter dans un minimier le tableau constitué des éléments 10. 9. 8. 7., décrire l'ensemble des échanges ayant lieu et donner le minimier obtenu.*
3. *La deuxième manière de planter un minimier est de recopier d'abord dans le minimier vide m les n éléments du tableau t, puis d'appliquer* maintientMinimierHaut *d'abord à* pere(dernier) *puis en allant vers l'arrière en l'appliquant à l'élément précédent* pred(pere(dernier)) *et ainsi de suite jusqu'à la racine. La fonction se note* planteBis(t, n,m) *et renvoie la position du dernier élément. Donner la suite d'instruction permettant de planter dans un minimier le tableau constitué des éléments 10. 9. 8. 7., décrire l'ensemble des échanges ayant lieu et donner le minimier obtenu.*
4. *Donner l'ordre de grandeur asymptotique des fonctions* maintientMinimierHaut(m,dernier) *et* maintientMinimierBas(m,rac) *si m est un minimier de n éléments.*
5. *Comparer les complexités de* plante(t,n, m) *et* planteBis(t, n, m)..

Correction.

On considère ici que les éléments sont réduits à leur clés.

1. *dernier=succ(dernier) ; m[dernier=2] ; maintientMinimierHaut(m,dernier)*
   (qui se réduit à *echange(dernier,pere(dernier))).* On obtient $cle(rac) = 0., 1., 3., 2., 4., 5., 4., 8.$
2. Au départ le minimier est vide et on a *dernier = pred(rac).* On exécute alors :
   - *dernier =insere (m,dernier, 10.)* ⇒ cle(rac)=10. (*dernier = rac*).
   - *dernier =insere (m,dernier, 9.)* ⇒ cle(rac)=9., 10. (après échange de 9. et 10.). (*dernier = succ(rac)*).
   - *dernier =insere (m,dernier, 8.)* ⇒ cle(rac)=8., 10., 9. (après échange de 8. et 9.).
   - *dernier=insere (m,dernier, 7.)* ⇒ cle(rac)=7., 8., 9. 10. (après échange de 7. et 10. puis de 7. avec 8.).
3. Au départ le minimier est vide et on a *dernier = pred(rac).*
   - On recopie t ce qui donne l'arbre $cle(rac) = 10.9.8.7.$ et *dernier* est la position de 7.
   - *maintientMinimierBas(m,pere(dernier))* ⇒ cle(rac)=10. 7. 8. 9. (après échange de 7. et 9.).

- *maintientMinimierBas(m,pred(pere(dernier)))* $\Rightarrow$ cle(rac)=7. 9. 8. 10. (après échange de 10. et 7., puis échange de 10. et 9.).

4. Si on considère la racine à hauteur $k = 0$, ses fils à hauteur $k = 1$ etc... on obtient à la hauteur k au plus $2^{k+1} = \sum_{i=0}^{i=k} 2^i$ éléments car chaque niveau i contient 2^i éléments (sauf peut-être le niveau k). La hauteur est donc de l'ordre de $\mathcal{O}(log_2(n))$ et c'est donc aussi l'ordre de grandeur du nombre d'échanges maximum résultant de *maintientMinimierHaut(m,dernier)* et *maintientMinimierBas(m,rac)*.
5. – Considérons la plantation par $plante(t, n, m)$ d'un minimier de hauteur k. Nous considérons ici que le minimier est complètement rempli. La fonction procède par n insertions successives et à chaque insertion on fait un appel à *maintientMinimierHaut*. Pour la racine on n'a pas d'échanges, Pour les 2 éléments du niveau 1 il y a au plus 1 échange, pour les 2^2 éléments au niveau 2, il y a au plus 2 échanges, et plus généralement pour les 2^i éléments au niveau i on a au plus i échanges. Ce qui donne en tout un nombre d'échanges $E(n) = \sum_{i=0}^{i=k} i2^i$ et on a une complexité en $\mathcal{O}(nlog_2(n))$.
   – Considérons la plantation par $planteBis(t, n, m)$ d'un minimier de hauteur k. Nous considérons ici encore que le minimier est complètement rempli. La fonction utilise *maintenanceMinimierBas* qui procède par échanges vers le bas. On commence par le niveau $k-1$ où pour les $n/2$ éléments de ce niveau on a au plus 1 échange, puis on passe au niveau $k-2$ où pour les $n/2^2$ éléments on a au plus 2 échanges, et on exécute ainsi i échanges pour les $n/2^i$ éléments du niveau $k-i$. Ce qui donne en tout un nombre d'échanges $E(n) = n\sum_{i=0}^{i=k-1} i/2^i$ et on a une complexité en $\mathcal{O}(n)$.

fin Correction.

Exercice 4 (tri par minimier)

Le tri en ordre croissant par minimier d'un tableau t de n éléments s'effectue de la manière suivante :

- *Les éléments du tableau à trier sont rangés dans un minimier m. L'indice i est initialisée au début du tableau t.*
- *La racine est rangée en t[i].*
- *La racine est supprimée du minimier, la structure du minimier, réduit d'un élément, étant maintenue.*
- *Si le minimier n'est pas vide, on recommence 4 en incrémentant i .*

On s'intéressera d'abord à la fonction supprimeRac(m, dernier) *qui renvoie* pred(dernier) *et qui maintient le minimier m.*

1. *Décrivez l'exécution de* supprimeRac(m, dernier) *sur le minimier* cle(rac) = 7., 8., 9. 10..
2. *Décrivez l'exécution de* triParMinimier(t,n) *sur le minimier* cle(rac) = 7., 8., 9. 10..
3. *Quelle est l'ordre de grandeur du nombre d'échanges effectués dans* triParMinimier(t,n) *?*

Correction.

1. On échange la racine 7. avec le dernier élément 10., et on réduit de 1 élément le minimier par *dernier=pred(dernier)*, ce qui donne *cle(rac)=10., 8., 9.*. On exécute alors *maintientMinimierBas(m, rac, dernier)* ce qui provoque l'échange de 10. avec 8. et aboutit au minimier *cle(rac)=8., 10., 9.*
2. On récolte et range successivement les racines des minimiers suivants :
   – *cle(rac)=7., 9., 8., 10.* $\rightarrow$ 7.
   – *cle(rac)=8., 9., 10.* $\rightarrow$ 8.
   – *cle(rac)=9., 10.* $\rightarrow$ 9.
   – *cle(rac)=10.* $\rightarrow$ 10.
3. On exécute *plante* ou *planteBis* sur n éléments. Puis on applique *maintientMinimierBas* successivement sur la racine de minimiers de taille allant de n à 1. Ces dernières opérations sont aussi celles exécutées (en ordre inverse) par l'insertion successive d'éléments dans un minimier. En conclusion cette deuxième partie s'effectue en $\mathcal{O}(nlog_2(n))$ échanges. Si on y ajoute le nombre d'échanges de *plante* ou *planteBis*, de l'ordre de $\mathcal{O}(nlog_2(n))$ ou $\mathcal{O}((n))$ échanges, on obtient un ordre de grandeur asymptotique du nombre d'échanges de *triParMinimier* de $\mathcal{O}(nlog_2(n))$.

fin Correction.

Exercice 5 (Implémentation d'un Minimier)
Soit un un type element_t *représenté par une structure contenant un champ* cle. *Un minimier rangé en ordre direct dans un tableau à partir d'une constante rac est représenté par le tableau m d'element_t et la position* dernier *du dernier élément dans le tableau. Le type élément utilisé et le type minimier seront déclarés comme suit :*

```
typedef struct Element {
        int cle;
        int pid;
        int priorite;
        int charge;
} element_t;
typedef element_t minimier[N];
```

1. *Implémenter les fonctions de gestion d'un minimier dont les en-têtes suivent. On utilisera pour cela les fonctions* pere(m,e), filsG(m,e), filsD(m,e), succ(e), pred(e), plusGrand(e,f).

   ```
   void maintientMinimierBas(minimier m, int r, int dernier);
   void maintientMinimierHaut ( minimier m,  int d);
   int insere (minimier m, int dernier, element_t el);
   int plante (element_t tab[], int n,  minimier m);
   int planteBis (element_t tab[], int n, minimier m);
   int supprimeRac(minimier m, int dernier);
   void triParMinimier(element_t t[], int n);
   void afficheMinimier(minimier m, int dernier);
   int tailleIntervalle(int e, int f); /* nombre d'elements entre les positions e et f dans le mini
   ```

2. *On supposera dans la suite que ces fonctions gèrent un minimier en ordre direct. On notera* pere_I(m,e)filsG_I(m,e), filsD_I(m,e) , succ_I(e), pred_I(e), plusGrand_I(e,f), maintientMinimier-Bas_I, ... , afficheMinimier_I *les fonctions correspondantes permettant de gérer un minimier en ordre inverse. On note* rac *la variable globale représentant la position de la racine dans un minimier en ordre direct et* rac_I *celle se rapportant à un minimier en ordre inverse. Suffit-il de transcrire vos fonctions de la question précédente en remplaçant* rac *par* rac_I, pere *par* pere_I,..., plusGrand *par* plusGrand_I *pour obtenir les fonctions correspondantes du cas inverse ?*

Correction.

1. voir listing complet à la fin de la correction
2. oui, si les algorithmes sont génériques, c'est à dire ne dépendent pas d'autre chose que des fonctions d'ordre pere, filsG, ..., pred, succ, plusGrand.

fin Correction.

Exercice 6 (Gestion d'un Biprocesseur)
Nous allons ici simuler la gestion d'un biprocesseur. Les processus arrivent alternativement aux files d'attente f_0 et f_1 des deux processeurs P_0 et P_1. Chaque processeur traite le processus à sa racine, ce qui diminue la charge de celui-ci d'une constante c. Si la charge du processus est nulle celui-ci est retiré de la file d'attente sinon son attente *augmente et il recule dans la file d'attente du processeur. Lorsque la différence du nombre de processus présents dans les deux processeurs est supérieure à 2 (la répartition des processus est dite* déséquilibrée*) le processus à la racine de la file d'attente du processeur le plus chargé est supprimé de cette file et inséré dans la file du processeur le moins chargé.*

Les deux files sont représentées par deux minimiers rangés dans le même tableau m de taille N. La file f_0 est rangée en ordre direct avec $rac = 0$ et la file f_1 en ordre inverse avec $rac_I = N - 1$. AInsi on peut avoir en tout N processus dans les deux files.

1. *Ecrire la fonction de prototype* int insertionProcessus(minimier m, int dernier, int valPid, int valPriorite, int valCharge) *qui insère un processus de pid* valPid*, de priorité* valPriorite *et de charge* valCharge *dans la file* f_0 *et renvoie la position du dernier élément de cette file : le dernier dans le minimier rangé dans* m *en ordre direct. (*f_0 *est représentée par* rac*,* m *et* dernier*).*
2. *Ecrire la fonction de prototype* int traiteProcessus(minimier m, int dernier) *qui traite le processus à la racine de la file* f_0 *et renvoie la position du dernier élément dans le minimier rangé dans* m *en ordre direct. Cette fonction diminue de 5. la charge du processus à la racine de* f_0*, le supprime si la charge est nulle, et sinon donne à sa clé la valeur* tailleIntervalle(rac,dernier) - m[rac].priorite*, puis fait redescendre le processus dans la file en accord avec sa nouvelle clé.*
3. *On suppose écrites les fonctions* insertionProcessus_I *et* traiteProcessus_I *correspondant à la gestion des processus de la file* f_1*. Ecrire la fonction d'en-tête* void gereFileBiprocesseur(void) *qui va effectuer les actions suivantes :*
   - *Entrer une série de processus alternativement dans les deux files. Le* i^{me} *processus entré a comme pid* $100 + i$*, comme priorité* $i mod 3$ *et comme charge* 20 *si* i *est pair et* 70 *s'il est impair.*
   - *Tant que tous les processus ne sont pas terminés, traiter les processus à la racine des files* $f f_0$ *et* f_1 *et éventuellement supprimer celui à la racine de* f_0 *et l'insérer dans* f_1 *(ou vice-versa) si la répartition des processus entre les deux processeurs n'est pas équilibrée.*

Correction.
voir le listing en fin de correction **fin Correction**.

```
#include <stdio.h>
#include <math.h>
# define N 40

typedef struct Element {
        int cle;
        int pid;
        int priorite;
        int charge;
} element_t;
typedef element_t minimier[N];



/* fonctions pour gerer l'ordre des elements d'un minimier commencant a la position "rac"*/
/* dans l'implementation de ces fonctions on suppose que l'ordre est l'ordre naturel et que la
 position de la racine est rac (0 en general en C (1er element d'un tableau)*/
int filsG( minimier m, int e);
int filsD( minimier m, int e);
int pere( minimier m, int e);
int plusGrand(int g, int p);/* g > p*/
int succ(int e);
int pred(int e);
int tailleIntervalle (int e, int f); /* nombre d'elements entre e et f dans le minimier y compris ceux-ci */



/* fonctions gerant le minimier de racine "rac" et dont le dernier element est "dernier"*/
/* ici rac (globale) et dernier representent des positions totalement ordonnees dans le minimier */
/* l'ordre des elements est gere' par les fonctions filsG, filsD, pere, plusGrand,succ, pred, tailleIntervalle */



/* pour maintenir la partie du minimier en dessous de r si r a diminue' */
void maintientMinimierBas(minimier m, int r, int dernier);

/*on ajoute un entier val au minimier apres le dernier element */
/* et on maintient la structure */
int insere (minimier m, int dernier, element_t el);

/* on supprime la racine et on maintient la structure (renvoie pred(dernier))*/
int supprimeRac(minimier m, int dernier);

/* construit le minimier a partir d'un tableau d'entier */
int plante (element_t tab[], int n, minimier m);
int planteBis (element_t tab[], int n, minimier m);
/* affiche le minimier */
void afficheMinimier(minimier m, int dernier);

/* pour maintenir la partie du minimier au dessus de r si en r la valeur a augmente'*/
void maintientMinimierHaut ( minimier m, int d);

/* pour maintenir le minimier quand l'element en position d prend la valeur val_e */
void maintientMinimierElement(minimier m, int e,int val_e, int dernier);

/* tri d'un tableau t de n elements */
void triParMinimier(element_t t[], int n);
```

```
/* echange les valeurs des elements en positions x et y dans le minimier t*/
void echange (minimier m, int x, int y);

/* fonctions pour gerer l'ordre des elements d'un minimier commencant a la position "rac_I"*/
/* dans l'implementation de ces fonctions on suppose que l'ordre est l'ordre inverse et que la
  position de la racine est racI ( N-1 en general en C dernier Ã©lÃ©ment d'un tableau)*/
int filsG_I ( minimier m, int e);
int filsD_I ( minimier m, int e);
int pere_I( minimier m, int e);
int plusGrand_I(int g, int p);/* g > p*/
int succ_I(int e);
int pred_I(int e);
int tailleIntervalle_I (int e, int f); /* nombre d'elements entre e et f dans le minimier y compris ceux-ci */

/* fonctions gerant le minimier de racine rac_I et dont le dernier element est "dernier"*/
/* ici rac (globale) et dernier representent des positions totalement ordonnees dans le minimier */
/* l'ordre des elements est gere' par les fonctions filsG_I, filsD_I, pere_I, plusGrand_I,succ_I, pred_I
, tailleIntervalle_I */



/* pour maintenir la partie du minimier en dessous de r si r a diminue' */
void maintientMinimierBas_I(minimier m, int r, int dernier);

/*on ajoute un entier val au minimier apres le dernier element */
/* et on maintient la structure */
int insere_I (minimier m, int dernier, element_t el);

/* on supprime la racine et on maintient la structure (renvoie pred(dernier))*/
int supprimeRac_I(minimier m, int dernier);

/* construit le minimier a partir d'un tableau d'entier */
int plante_I (element_t tab [], int n, minimier m);
int planteBis_I (element_t tab [], int n, minimier m);
/* affiche le minimier */
void afficheMinimier_I(minimier m, int dernier);

/* pour maintenir la partie du minimier au dessus de r si en r la valeur a augmente'*/
void maintientMinimierHaut_I ( minimier m, int d);

/* pour maintenir le minimier quand l'element en position d prend la valeur val_e */
void maintientMinimierElement_I(minimier m, int e,int val_e, int dernier);

/* tri d'un tableau t de n elements */
void triParMinimier_I(element_t t [], int n);

/* echange les valeurs des elements en positions x et y dans le minimier t*/
void echange_I (minimier m, int x, int y);

/* En utilisant les deux types de minimier on peut ainsi ranger dans un seul tableau deux minimiers */
/* (deux files d'anti-priorite'), un a partir de la gauche et l'autre a partir de la droite, en ne changeant*/
/* que les fonctions d'ordre selon que le minimier soit normal (a partir de la gauche) ou inverse (a partir de la droite) */
```

```
/* fonctions pour gerer des processus dont l'antipriorite (le processus execute' est celui de plus faible valeur)*/
/* un minimier represente la file */

/* traite le processus le plus prioritaire, renvoie la position du dernier element de m */
int traiteProcessus(minimier m, int dernier);
/* insertion d'un nouveau processus (il aura la plus faible priorite')*/
int insertionProcessus(minimier m, int dernier, int valPid, int valPriorite, int valCharge);

/* traite le processus le plus prioritaire, renvoie la position du dernier element de m (Inverse)*/
int traiteProcessus_I(minimier m, int dernier);
/* insertion d'un nouveau processus (il aura la plus faible priorite') (Inverse)*/
int insertionProcessus_I(minimier m, int dernier, int valPid, int valPriorite, int valCharge);

/* gestion des processus d'un Biprocesseur par deux minimiers direct et inverse
range's dans le meme tableau m */
void gereFileBiprocesseur( void);

int rac ; /* variable globale indiquant l'indice de la racine */
int rac_I; /* variable globale indiquant l'indice de la racine en ordre Inverse */

int main () {
        int i,dernier, dernier_I;
        element_t t[N], aux;
        minimier m;


        rac=0;
        rac_I=N-1;
//      for (rac=0; rac <=0 ;rac++){
                printf ("--------------------\n rac %d \n", rac);
                /* on remplit t */
                printf("on remplit t \n");
                t[0].cle=5;
                t[0].pid=100;
                t[0].priorite=3;
                t[i].charge=20;
                 printf("%d ",t[0].cle);
                for (i=1;i<10;i++) {
                        if (i<6) t[i].cle=i-1; else t[i].cle=i;
                        t[i].pid=100+i;
                        t[i].priorite=0;
                        t[i].charge=20;
                        printf("%d ",t[i].cle);
                }
                printf("\n");

                /* */

                printf(" on construit et on affiche le minimier a partir des elements de rang 0 a 9 de t\n");
                dernier_I=plante_I(t,10,m);
                for (i=N-1; i > N-11; i--) printf("%d ", m[i]);
                printf("\n");
                afficheMinimier_I(m, dernier_I);
                printf(" on construit et on affiche le minimier a partir des elements de rang 0 a 9 de t (methode 2)\n");
                dernier_I=planteBis_I(t,10,m);
                afficheMinimier_I(m, dernier_I);

                printf(" on construit et on affiche le minimier a partir des elements de rang 0 a 9 de t\n");
                dernier=plante(t,10,m);
                afficheMinimier(m, dernier);
                printf(" on construit et on affiche le minimier a partir des elements de rang 0 a 9 de t (methode 2)\n");
```

```
dernier=planteBis(t,10,m);
afficheMinimier(m, dernier);

/*  */
printf(" on modifie la racine, on affiche et on maintient en faisant descendre la racine\n");
m[rac_I].cle=100;
afficheMinimier_I(m, dernier_I);
maintientMinimierBas_I (m, rac_I, dernier_I);
afficheMinimier_I(m, dernier_I);

printf(" on modifie la racine, on affiche et on maintient en faisant descendre la racine\n");
m[rac].cle=100;
afficheMinimier(m, dernier);
maintientMinimierBas (m, rac, dernier);
afficheMinimier(m, dernier);



/* */
printf(" on reaffiche t et on le trie en ordre croissant par le tri par minimier\n");
for (i=0;i<10;i++) printf("%d ",t[i].cle);
printf("\n");
triParMinimier_I(t, 10);
for (i=0;i<10;i++) printf("%d ",t[i].cle);
printf("\n");

/*on reinitialise t*/
aux=t[5];
for(i=5; i>0;i--) t[i]=t[i-1];
t[0]=aux;


printf(" on reaffiche t et on le trie en ordre croissant par le tri par minimier\n");
for (i=0;i<10;i++) printf("%d ",t[i].cle);
printf("\n");
triParMinimier(t, 10);
for (i=0;i<10;i++) printf("%d ",t[i].cle);
printf("\n");

/*  */
printf("on modifie dans le minimier l'element en rac +3 ( -> 0)\n et on maintient (ici la valeur remonte)\n");
maintientMinimierElement_I(m,succ_I(succ_I(succ_I(rac_I))), 0, dernier_I);
afficheMinimier_I(m, dernier_I);

printf("on modifie dans le minimier l'element en rac +3 ( -> 0)\n et on maintient (ici la valeur remonte)\n");
maintientMinimierElement(m,rac+3, 0,dernier);
afficheMinimier(m, dernier);

/*  */
printf("on modifie dans le minimier l'element en rac +3 ( -> 10)\n et on maintient (ici la valeur descend)\n");
maintientMinimierElement_I(m,succ_I(succ_I(succ_I(rac_I))), 10, dernier_I);
afficheMinimier_I(m, dernier_I);

printf("on modifie dans le minimier l'element en rac +3 ( -> 10)\n et on maintient (ici la valeur descend)\n");
maintientMinimierElement(m,rac+3, 10,dernier);
afficheMinimier(m, dernier);


/* */
//          dernier_I=insertionProcessus_I( m, dernier_I, 111, 5, 30);
//          afficheMinimier_I(m, dernier_I);
```

```
//
//              while (! plusGrand_I(rac_I, dernier_I)){
//                      dernier_I=traiteProcessus_I(m,dernier_I);
//                      afficheMinimier_I(m, dernier_I);
//              }
//
//
//              dernier=insertionProcessus( m, dernier,  111, 5, 30);
//              afficheMinimier(m, dernier);
//
//              while (! plusGrand(rac, dernier)){
//                      dernier=traiteProcessus(m,dernier);
//                      afficheMinimier(m, dernier);
//              }

                /* on fait passer les processus de l'un Ã  l'autre*/

//      }
                gereFileBiprocesseur ();
    return 0;
}



int filsG ( minimier m, int e){
        /* renvoie le fils gauche de e dans le minimier de racine rac */
        return 2*e + 1 - rac;
}

int filsD ( minimier m, int e){
        /* renvoie le fils droit de e dans le minimier de racine rac */
        return 2*e + 2 - rac;
}

int pere( minimier m, int e){
        return (e-1+rac) / 2;
}

int plusGrand(int g, int p){
        return (g > p);
}

int succ(int e){
        return e+1;
}
int pred(int e){
        return e-1;
}

int  tailleIntervalle (int e, int f){
        return f -e +1 ;
}


void maintientMinimierBas (minimier m, int r, int dernier){
        /* le sous-minimier de racine r et de dernier noeud <="dernier" est correct sauf peut etre en r */
        /* on maintient le minimier en faisant descendre l'element racine a sa bonne place */
        /* on observe qu'il suffit pour cela d'echanger r avec le plus petit de ses deux fils */
        /*, notons le "fils ", puis de maintenir le sous-minimier de racine "fils "*/
        /* on suppose le minimier non vide */
```

```
        int fils ;

        if (plusGrand(filsG(m,r), dernier))
                return ; /* pas de fils */
        if (filsG(m,r) == dernier)
                fils =dernier; /* un seul fils */
        else /* deux fils */
        if (m[filsD(m,r)].cle <m[filsG(m,r)].cle)
                fils =filsD(m,r);
        else
                fils = filsG(m,r);
//      printf("%d %d %d \n",fils, m[r], m[fils ]);
        if (m[fils ].cle< m[r].cle) {
                echange(m, fils , r);
//              printf("apres %d %d %d \n",fils, m[r], m[fils ]);
        }
        /* le noeud en r est maintenant plus petit que ses deux fils */
        maintientMinimierBas ( m, fils , dernier);
        return;
}

int insere (minimier m, int dernier, element_t el){
        /*on ajoute un entier val au minimier apres le dernier element */
        /* et on maintient la structure */

        /* on agrandit le minimier et on place val en dernier */
        dernier=succ(dernier);
        m[dernier]=el;
        maintientMinimierHaut ( m, dernier);
//      afficheMinimier(m, dernier);
        return dernier;
}

int supprimeRac(minimier m, int dernier){
        /* on supprime la racine et on maintient la structure (renvoie pred(dernier))*/
        m[rac]=m[dernier];
        dernier=pred(dernier);
        maintientMinimierBas(m, rac, dernier);
        return dernier;
}

int plante (element_t tab[], int n, minimier m){
        /* construit le minimier a partir d'un tableau d'entier */
        /* par insertions successives en fin de minimier et maintenance */
        int dernier,i;
        dernier=pred(rac);
        for (i=0; i <n; i++){
                dernier =insere( m, dernier, tab[i]);
//              afficheMinimier(m,dernier);
        }
        return dernier;
}

int planteBis (element_t tab[], int n, minimier m){
        /* construit le minimier a partir d'un tableau d'entier */
        /* par recopie du tableau et maintenances du minimier en dessous de chaque noeud pere,
        du dernier noeud pere a la racine */
        int i,dernier,pos;
        /* recopie dans le minimier */
        pos=rac;
        for (i=0; i < n;i++) {
```

```
                m[pos]= tab[i];
                pos=succ(pos);
        }
        dernier = pred(pos);
        /* maintenance a parti du dernier pere (le pere du dernier element)*/

        for (pos=pere(m,dernier); ! plusGrand (rac, pos); pos=pred(pos)){
                maintientMinimierBas(m, pos, dernier);
//              afficheMinimier(m,dernier);
        }
        return dernier;
}


void afficheMinimier(minimier m, int dernier){
        int i,n,k,decal;
        int noeudsAuNiveauCourant;
        /* affiche le minimier entre rac et l'element dernier */
        i=rac;
        noeudsAuNiveauCourant=1;
        decal= 5*log( tailleIntervalle (rac, dernier));
//      printf("decal %d",decal);
        while (! plusGrand(i,dernier) ){
                for (k=1; k <decal ; k++) printf(" ");
                for (n=0; (n < noeudsAuNiveauCourant) && (! plusGrand(i,dernier) ); n++) {
                        printf("%d ", m[i].cle);
                        for (k=1; k <decal+decal/2; k++) printf(" ");
                        i=succ(i);
//                      printf("i %d",i);
                }
                printf("\n");
                noeudsAuNiveauCourant=2*noeudsAuNiveauCourant;
                decal=decal/2;
        }
        printf("\n");
        return;
}

void echange(minimier m, int x, int y){
        element_t aux;
        aux=m[x];
        m[x]=m[y];
        m[y]=aux;
}

void maintientMinimierHaut ( minimier m, int d){
        /* l'element en d est peut etre mal place' car peut etre
        plus petit que son pere */
        /* on doit alors retablir la structure au dessus*/
//      printf("entree\n");afficheMinimier(m,d);
        if (plusGrand( rac , pere(m, d) )    )
                return; /* d est la racine ou n'existe pas */

        if (m[d].cle < m[pere(m,d)].cle){
                /* on echange dernier et son pere et celui ci ayant change'*/
                /* on maintient la structure au dessus */
//              printf("echange(m,d,pere(m,d)) %d %d\n" ,d,pere(m,d) );
                echange(m,d,pere(m,d));
                maintientMinimierHaut ( m, pere(m,d));
        }
        /* il n'y a pas d'echange donc rien a faire au dessus */
```

```
        return;
}

void maintientMinimierElement(minimier m, int e, int val_e, int dernier){
        /* maintient le minimier quand l'element de position e entre rac et dernier */
        /* doit prendre la valeur val_e */
        if (val_e < m[e].cle){
                /* la valeur va diminuer, il faut maintenir la structure au dessus de e */
                m[e].cle=val_e;
                maintientMinimierHaut(m, e);
//              printf("affiche apres mainitentSousMinimierDernier\n");afficheMinimier(m,dernier);
                return;
        }
        if (val_e > m[e].cle){
                /* la valeur va augmenter, il faut maintenir la structure au dessous de e */
                m[e].cle=val_e;
                maintientMinimierBas(m, e, dernier);
                return;
        }
        return;
}

void triParMinimier(element_t t[], int n){
        /*  tri croissant d'un tableau */
        /* ici on on besoin d'un minimier mini range' dans un tableau auxiliaire */
        /* car il va etre detruit pendant le tri*/
        int dernier,i;
        minimier mini;
        dernier = plante (t, n, mini);
//      printf("plante dans tri \n");
//      afficheMinimier(mini,dernier);
        i=0;
//      printf("rac %d minni[rac] %d\n", rac,mini[rac]);
        while (! plusGrand(rac, dernier ) ){
                /* rangement de la racine, mise du dernier element
                        a la racine et diminution de l'arbre */
                t[i]=mini[rac];
                dernier=supprimeRac(mini,dernier);
//              afficheMinimier(t,dernier);
                i++;
        }
        return;
}

/* minimiers en ordre Inverse */

int filsG_I ( minimier m, int e){
        /* renvoie le fils gauche de e dans le minimier de racine rac_I */
        return 2*e - 1 - rac_I;
}

int filsD_I ( minimier m, int e){
        /* renvoie le fils droit de e dans le minimier de racine rac */
        return 2*e -2 - rac_I;
}

int pere_I( minimier m, int e){
        return (rac_I + e + 2 ) / 2;
}

int plusGrand_I(int g, int p){
```

```
        return (g < p);
}

int succ_I(int e){
        return e-1;
}
int pred_I(int e){
        return e+1;
}

int  tailleIntervalle_I (int e, int f){
        return e - f +1 ;
}


void maintientMinimierBas_I (minimier m, int r, int dernier){
        /* le sous-minimier de racine r et de dernier noeud <="dernier" est correct sauf peut etre en r */
        /* on maintient le minimier en faisant descendre l'element racine a sa bonne place */
        /* on observe qu'il suffit pour cela d'echanger r avec le plus petit de ses deux fils */
        /*, notons le "fils ", puis de maintenir le sous-minimier de racine "fils "*/
        /* on suppose le minimier non vide : r <= dernier */
        int fils ;

        if (plusGrand_I(filsG_I(m,r), dernier))
                return ; /* pas de fils */
        if ( filsG_I (m,r) == dernier)
                fils =dernier; /* un seul fils */
        else /* deux fils */
        if (m[filsD_I(m,r)]. cle <m[filsG_I(m,r)]. cle)
                fils =filsD_I(m,r);
        else
                fils = filsG_I(m,r);
//      printf("%d %d %d \n",fils, m[r], m[fils ]);
        if (m[ fils ]. cle< m[r].cle) {
                echange(m, fils , r);
//              printf("apres %d %d %d \n",fils, m[r], m[fils ]);
        }
        /* le noeud en r est maintenant plus petit que ses deux fils */
        maintientMinimierBas_I ( m, fils , dernier);
        return;
}

int insere_I (minimier m, int dernier, element_t el){
        /*on ajoute un entier val au minimier apres le dernier element */
        /* et on maintient la structure */

        /* on agrandit le minimier et on place val en dernier */
        dernier=succ_I(dernier);
        m[dernier]=el;
//      printf("m[dernier] %d\n", m[dernier]);
        maintientMinimierHaut_I ( m, dernier);
//      afficheMinimier(m, dernier);
        return dernier;
}

int supprimeRac_I(minimier m, int dernier){
        /* on supprime la racine et on maintient la structure (renvoie pred_I(dernier))*/
        m[rac_I]=m[dernier];
        dernier=pred_I(dernier);
        maintientMinimierBas_I(m, rac_I, dernier);
        return dernier;
```

```
}

int plante_I (element_t tab[], int n, minimier m){
        /* construit le minimier a partir d'un tableau d'entier */
        /* par insertions successives en fin de minimier et maintenance */
        int dernier,i;
        dernier=pred_I(rac_I);
//      printf("rac_I %d dernier %d\n", rac_I ,dernier);
        for (i=0; i <n; i++){
                dernier =insere_I( m, dernier, tab[i]);
//              afficheMinimier_I(m,dernier);
        }
        return dernier;
}

int planteBis_I (element_t tab[], int n, minimier m){
        /* construit le minimier a partir d'un tableau d'entier */
        /* par recopie du tableau et maintenances du minimier en dessous de chaque noeud pere,
        du dernier noeud pere a la racine */
        int i,dernier,pos;
        /* recopie dans le minimier */
        pos=rac_I;
        for (i=0; i < n;i++) {
                m[pos]= tab[i];
                pos=succ_I(pos);
        }
        dernier = pred_I(pos);
        /* maintenance a parti du dernier pere (le pere du dernier element)*/

        for (pos=pere_I(m,dernier); ! plusGrand_I (rac_I, pos); pos=pred_I(pos)){
                maintientMinimierBas_I(m, pos, dernier);
//              afficheMinimier(m,dernier);
        }
        return dernier;
}


void afficheMinimier_I(minimier m, int dernier){
        int i,n,k,decal;
        int noeudsAuNiveauCourant;
        /* affiche le minimier entre rac et l'element dernier */
        i=rac_I;
        noeudsAuNiveauCourant=1;
        decal= 5*log( tailleIntervalle_I (rac_I, dernier));
//      printf("decal %d",decal);
        while (! plusGrand_I(i,dernier) ){
                for (k=1; k <decal ; k++) printf(" ");
                for (n=0; (n < noeudsAuNiveauCourant) && (! plusGrand_I(i,dernier) ); n++) {
                        printf("%d ", m[i].cle);
                        for (k=1; k <decal+decal/2; k++) printf(" ");
                        i=succ_I(i);
//                      printf("i %d",i);
                }
                printf("\n");
                noeudsAuNiveauCourant=2*noeudsAuNiveauCourant;
                decal=decal/2;
        }
        printf("\n");
        return;
}
```

```
void echange_I(minimier m, int x, int y){
        element_t aux;
        aux=m[x];
        m[x]=m[y];
        m[y]=aux;
}

void maintientMinimierHaut_I ( minimier m, int d){
        /* l'element en d est peut etre mal place' car peut etre
        plus petit que son pere */
        /* on doit alors retablir la structure au dessus*/
//      printf("entree\n");afficheMinimier(m,d);
//      printf("rac_I %d pere_I(m, d) %d d %d\n",rac_I , pere_I(m, d), d);
        if (plusGrand_I( rac_I , pere_I(m, d) )    )
                return; /* d est la racine ou n'existe pas */

        if (m[d].cle < m[pere_I(m,d)].cle){
                /* on echange dernier et son pere et celui ci ayant change'*/
                /* on maintient la structure au dessus */
//              printf("echange(m,d,pere(m,d)) %d %d\n" ,d,pere(m,d) );
                echange_I(m,d,pere_I(m,d));
                maintientMinimierHaut_I ( m, pere_I(m,d));
        }
        /* il n'y a pas d'echange donc rien a faire au dessus */
        return;
}

void maintientMinimierElement_I(minimier m, int e, int val_e, int dernier){
        /* maintient le minimier quand l'element de position e entre rac et dernier */
        /* doit prendre la valeur val_e */
        if (val_e < m[e].cle){
                /* la valeur va diminuer, il faut maintenir la structure au dessus de e */
                m[e].cle=val_e;
                maintientMinimierHaut_I(m, e);
//              printf("affiche apres mainitentSousMinimierDernier\n");afficheMinimier(m,dernier);
                return;
        }
        if (val_e > m[e].cle){
                /* la valeur va augmenter, il faut maintenir la structure au dessous de e */
                m[e].cle=val_e;
                maintientMinimierBas_I(m, e, dernier);
                return;
        }
        return;
}

void triParMinimier_I(element_t t[], int n){
        /*  tri croissant d'un tableau */
        /* ici on on besoin d'un minimier mini range' dans un tableau auxiliaire */
        /* car il va etre detruit pendant le tri*/
        int dernier,i;
        minimier mini;
        dernier = plante_I (t, n, mini);
//      printf("plante dans tri \n");
//      afficheMinimier_I(mini,dernier);
        i=0;
//      printf("rac %d minni[rac] %d\n", rac,mini[rac]);
        while (! plusGrand_I(rac_I, dernier   ) ){
                /* rangement de la racine, mise du dernier element
                        a la racine et diminution de l'arbre */
                t[i]=mini[rac_I];
```

```
                dernier=supprimeRac_I(mini,dernier);
//              afficheMinimier_I(mini,dernier);
                i++;
        }
        return;
}


int traiteProcessus(minimier m, int dernier){
        /*traitement du processus a la racine */
        /* modification de l'antipriorite '  */
        if (dernier == pred(rac)) return dernier;
        m[rac].charge=m[rac].charge- 5;
        if (m[rac].charge > 0){
                /* le processus devient moins prioritaire (sa cle augmente) */
                m[rac].cle= tailleIntervalle (rac,dernier) - m[rac].priorite ;
                //              afficheMinimier(t,dernier);
        }
        else {
                /* suppression de la racine et maintenance du minimier */
                printf("fin processus %d\n",m[rac].pid);
                dernier=supprimeRac(m, dernier);
        }
        //              afficheMinimier(t,dernier);
        return dernier;
}

int traiteProcessus_I(minimier m, int dernier){
        /*traitement du processus a la racine */
        /* modification de l'antipriorite '  */
        if (dernier == pred_I(rac_I)) return dernier;
        m[rac_I].charge=m[rac_I].charge- 5;
        if (m[rac_I].charge > 0){
                /* le processus devient moins prioritaire (sa cle augmente) */
                m[rac_I].cle= tailleIntervalle_I (rac_I,dernier) - m[rac_I].priorite ;
                //              afficheMinimier(t,dernier);
        }
        else {
                /* suppression de la racine et maintenance du minimier */
                printf("fin processus %d\n",m[rac_I].pid);
                dernier=supprimeRac_I(m, dernier);
        }
        //              afficheMinimier(t,dernier);
        return dernier;
}


int insertionProcessus(minimier m, int dernier, int valPid, int valPriorite , int valCharge){
        /* insertion d'un nouveau processus (il aura a priori la plus faible priorite ')*/
        element_t el;
        el.pid=valPid;
        el.priorite =valPriorite;
        el.charge=valCharge;
        el.cle = tailleIntervalle (rac, dernier) - el.priorite ;
        printf(" taille % d priorite %d\n", tailleIntervalle (rac, dernier) , el.priorite );
        printf("debut processus %d\n",el.pid);
        dernier= insere ( m, dernier, el );
        return dernier;
}

int insertionProcessus_I(minimier m, int dernier, int valPid, int valPriorite , int valCharge){
```

```
        /* insertion d'un nouveau processus (il aura a priori la plus faible priorite ')*/
        element_t el;
        el.pid=valPid;
        el. priorite =valPriorite;
        el.charge=valCharge;
        el.cle = tailleIntervalle_I (rac_I, dernier) - el. priorite ;
        printf(" taille % d priorite %d\n", tailleIntervalle_I (rac_I, dernier) , el. priorite );
        printf("debut processus %d\n",el.pid);
        dernier= insere_I ( m, dernier, el);
        return dernier;
}

void gereFileBiprocesseur( void) {
        /* gestion des processus d'un Biprocesseur par deux minimiers direct et inverse
        range's dans le meme tableau m */
        element_t process;
        minimier m;
        int dernier, dernier_I,i;
        rac=0; rac_I=N-1;
        dernier=pred(rac);
        dernier_I=pred_I(rac_I);
        for (i=0; i<10; i++){
//              printf(" entrez un processus (pid, priorite, charge)\n");
//              scanf("%d%d%d", &process.pid,&process.priorite, &process.charge);
                process.pid=100+i;
                process. priorite =i%3;
                process.charge=20+(i%2)*50; /* les processus impairs sont plus lourds */
                printf(" processus %d priorite %d charge %d \n",process.pid, process. priorite , process.charge);
                if (i%2 == 0){
                        printf("insertion processus %d dans la file du processeur %d \n", process.pid, i%2 );
                        dernier=insertionProcessus(m, dernier, process.pid, process. priorite , process.charge);
                }
                else {
                        printf("insertion processus %d dans la file du processeur %d \n", process.pid, i%2 );
                        dernier_I=insertionProcessus_I(m, dernier_I, process.pid, process. priorite , process.charge);
                }
        }
        while (dernier !=pred(rac) || dernier_I!=pred_I(rac_I)){
                dernier=traiteProcessus( m, dernier);
                dernier_I=traiteProcessus_I( m, dernier_I );
                if ( tailleIntervalle (rac,dernier) - tailleIntervalle_I (rac_I,dernier_I) > 2){
                        printf("le processus %d passe du processeur %d au processeur %d\n",m[rac].pid, 0, 1);
                        process= m[rac];
                        dernier=supprimeRac(m, dernier);
                        dernier_I=insertionProcessus_I(m, dernier_I, process.pid, process. priorite , process.charge);
                }
                else if ( tailleIntervalle_I (rac_I,dernier_I)- tailleIntervalle (rac,dernier) > 2){
                        printf("le processus %d passe du processeur %d au processeur %d\n",m[rac_I].pid, 1, 0);
                        process= m[rac_I];
                        dernier_I=supprimeRac_I(m, dernier_I);
                        dernier=insertionProcessus(m, dernier, process.pid, process. priorite , process.charge);
                }
        }
}

//
//------------------
// rac 0
//on remplit t
//5 0 1 2 3 4 6 7 8 9
// on construit et on affiche le minimier a partir des elements de rang 0 a 9 de t
```

```
//0 2 1 5 3 4 6 7 8 9
//          0
//    2         1
// 5    3    4    6
//7 8 9
//
// on construit et on affiche le minimier a partir des elements de rang 0 a 9 de t (methode 2)
//          0
//    2         1
// 5    3    4    6
//7 8 9
//
// on construit et on affiche le minimier a partir des elements de rang 0 a 9 de t
//          0
//    2         1
// 5    3    4    6
//7 8 9
//
// on construit et on affiche le minimier a partir des elements de rang 0 a 9 de t (methode 2)
//          0
//    2         1
// 5    3    4    6
//7 8 9
//
// on modifie la racine, on affiche et on maintient en faisant descendre la racine
//          100
//    2         1
// 5    3    4    6
//7 8 9
//
//          1
//    2         4
// 5    3    100    6
//7 8 9
//
// on modifie la racine, on affiche et on maintient en faisant descendre la racine
//          100
//    2         1
// 5    3    4    6
//7 8 9
//
//          1
//    2         4
// 5    3    100    6
//7 8 9
//
// on reaffiche t et on le trie en ordre croissant par le tri par minimier
//5 0 1 2 3 4 6 7 8 9
//0 1 2 3 4 5 6 7 8 9
// on reaffiche t et on le trie en ordre croissant par le tri par minimier
//5 0 1 2 3 4 6 7 8 9
//0 1 2 3 4 5 6 7 8 9
//on modifie dans le minimier l'element en rac +3 ( -> 0)
// et on maintient (ici la valeur remonte)
//          0
//    1         4
// 2    3    100    6
//7 8 9
//
//on modifie dans le minimier l'element en rac +3 ( -> 0)
// et on maintient (ici la valeur remonte)
```

```
//            0
//     1         4
// 2    3    100    6
//7 8 9
//
//on modifie dans le minimier l'element en rac +3 ( -> 10)
// et on maintient (ici la valeur descend)
//            0
//     1         4
// 7    3    100    6
//10 8 9
//
//on modifie dans le minimier l'element en rac +3 ( -> 10)
// et on maintient (ici la valeur descend)
//            0
//     1         4
// 7    3    100    6
//10 8 9
//
//  processus 100 priorite 0 charge 20
//insertion processus 100 dans la file du processeur 0
// taille   0 priorite 0
//debut processus 100
//  processus 101 priorite 1 charge 70
//insertion processus 101 dans la file du processeur 1
// taille   0 priorite 1
//debut processus 101
//  processus 102 priorite 2 charge 20
//insertion processus 102 dans la file du processeur 0
// taille   1 priorite 2
//debut processus 102
//  processus 103 priorite 0 charge 70
//insertion processus 103 dans la file du processeur 1
// taille   1 priorite 0
//debut processus 103
//  processus 104 priorite 1 charge 20
//insertion processus 104 dans la file du processeur 0
// taille   2 priorite 1
//debut processus 104
//  processus 105 priorite 2 charge 70
//insertion processus 105 dans la file du processeur 1
// taille   2 priorite 2
//debut processus 105
//  processus 106 priorite 0 charge 20
//insertion processus 106 dans la file du processeur 0
// taille   3 priorite 0
//debut processus 106
//  processus 107 priorite 1 charge 70
//insertion processus 107 dans la file du processeur 1
// taille   3 priorite 1
//debut processus 107
//  processus 108 priorite 2 charge 20
//insertion processus 108 dans la file du processeur 0
// taille   4 priorite 2
//debut processus 108
//  processus 109 priorite 0 charge 70
//insertion processus 109 dans la file du processeur 1
// taille   4 priorite 0
//debut processus 109
//fin processus 102
//fin processus 100
```

```
//fin processus 104
//le processus 101 passe du processeur 1 au processeur 0
// taille   2 priorite 1
//debut processus 101
//fin processus 101
//fin processus 108
//le processus 105 passe du processeur 1 au processeur 0
// taille   1 priorite 2
//debut processus 105
//fin processus 105
//fin processus 106
//le processus 103 passe du processeur 1 au processeur 0
// taille   0 priorite 0
//debut processus 103
//fin processus 103
//fin processus 107
//fin processus 109
//
// td7_L2_filePriorite has exited with status 0.
```