```
================
Files de priorite
================

Ce document presente, en MPD, la specification d'un type abstrait
FilePriorite, trois mises en oeuvre, et deux exemples d'utilisation.


- Interface
  * fp.mpd: Interface + mise en oeuvre partielle

- Mises en oeuvre:
  * fp1.mpd: Mise en oeuvre avec une sequence non-ordonnee
  * fp2.mpd: Mise en oeuvre avec une sequence ordonnee
  * fp3.mpd: Mise en oeuvre avec un monceau (heap)

- Exemples d'utilisation:
  * tri.mpd: tri avec file de priorite
  * sac-a-dos.mpd: sac a dos fractionnaire
```

```
=================================================
= Fichier fp.mpd: Interface + Mise en oeuvre (...) =
=================================================

#####################################################
# Type abstrait mutable pour des files de priorite.
#####################################################

resource FilePriorite
  # Type pour les elements de la file de priorite.
  type Element = int;

  # Type pour le niveau de priorite d'un element;
  type Priorite = int;  # Pour l'exemple du tri.


  #
  # Les trois operations de base exportees par le type.
  #
  # STATE: L'etat abstrait d'une pile est simplement le (multi−)ensemble
  #        des elements (et leurs priorites) contenus dans la file.
  #        L'ordre est associe au niveau de priorite,
  #        et non a la position dans la structure de donnees.
  # INITIALLY
  #   fp = {}
  #
  # Note: Dans une postcondition (operation avec modification
  #       a l'etat), fp' refere a l'etat de fp avant l'appel,
  #       alors que fp refere a l'etat apres l'appel.


  op inserer( Priorite prio, Element elem );
  # PRECONDITION
  #  |fp| < maxElements
  # POSTCONDITION
  #  fp = fp' U { (prio, elem) }

  op estVide() returns bool r;
  # POSTCONDITION
  #   r <=> fp = {}

  op retirerMin( res Priorite prio, res Element elem );
  # PRECONDITION
  #   ~estVide(fp)
  # POSTCONDITION
  #   (prio, elem) IN fp' & prio = MINIMUM( p :: (p, e) IN fp' )
  #   fp = fp' − { (prio, elem) }


  op insererN( Priorite prios[*], Element elems[*], int n );
  # PRECONDITION
  #  |fp|+n < maxElements
  # POSTCONDITION
  #  fp = fp' U {(prios[1], elems[1]), ..., (prios[n], elems[n]) }
```

```
body FilePriorite( int maxElements  )
  # maxElements = Nombre maximum d'elements permis (allocation statique).

  # Type pour les paires << (Element, Priorite) >>;
  type NoeudFp = rec( Priorite prio; Element elem );

  # Nombre effectif d'elements dans la file.
  int nbElements = 0;

  # La liste des elements presents et les priorites associees.
  NoeudFp elements[maxElements];

  #
  # Mise en oeuvre des operations.
  #
  # Voir fp1.mpd, fp2.mpd et fp3.mpd
  .
  .
  .
end
```

```
============================================================
= Fichier fp1.mpd: Mise en oeuvre avec sequence non−ordonnee =
============================================================
  #
  # Premiere mise en oeuvre: sequence non−ordonnee d'items.
  # On ajoute simplement l'element a la fin.
  #

  proc inserer( prio, elem )
  {
    # On insere simplement a la fin.
    nbElements += 1;
    elements[nbElements] = NoeudFp( prio, elem );
  }

  proc estVide() returns r
  { r = nbElements == 0; }

  proc retirerMin( prio, elem )
  {
    # Non−ordonnee => on doit trouver le bon element.
    int posMin = 1;
    for [i = 2 to nbElements] {
      if ( elements[i].prio < elements[posMin].prio ) {
        posMin = i;
      }
    }
    prio = elements[posMin].prio;
    elem = elements[posMin].elem;
    # On le remplace par le dernier.
    elements[posMin] = elements[nbElements];
    nbElements −= 1;
  }

  proc insererN( prios, elems, n )
  {
    for [i = 1 to n] {
      inserer( prios[i], elems[i] );
    }
  }
```

```
=============================================================
= Fichier fp2.mpd: Mise en oeuvre avec sequence ordonnee     =
=============================================================
  #
  # Deuxieme mise en oeuvre: sequence ordonnee d'items.
  # On ajoute l'element a la bonne position dans l'ordre.
  # Pour eviter de deplacer les elements, on
  # utilise une structure d'anneau (liste circulaire).
  #

  int premier;       # Position du premier, si present.
  int prochain = 1;  # Position d'insertion du prochain.

  procedure succ( int i ) returns int ip1
  { ip1 = (i % maxElements) + 1; }

  procedure pred( int i ) returns int im1
  { im1 = (i − 2 + maxElements) % maxElements + 1; }

  proc inserer( prio, elem )
  {
    int pos = prochain;
    if ( nbElements == 0 ) {
      premier = prochain;     # C'est le premier qu'on insere.
    } else {
      # On decale les elements pour creer un "trou" a la bonne place.
      while( pos != premier & prio < elements[pred(pos)].prio ) {
        elements[pos] = elements[pred(pos)];
        pos = pred(pos);
      }
    }
    # pos indique le "trou" approprie: on met elem.
    elements[pos] = NoeudFp( prio, elem );
    prochain = succ(prochain);
    nbElements += 1;
  }

  proc estVide() returns r
  { r = nbElements == 0; }

  proc retirerMin( prio, elem )
  {
    prio = elements[premier].prio;
    elem = elements[premier].elem;
    premier = succ(premier);
    nbElements −= 1;
  }

  proc insererN( prios, elems, n )
  {
    for [i = 1 to n] {
      inserer( prios[i], elems[i] );
    }
  }
```

```
============================================================
= Fichier fp3.mpd: Mise en oeuvre avec monceau             =
============================================================
  #
  # Troisieme mise en oeuvre : avec un monceau (tas, "heap").
  #

  # Les selecteurs de position (dans l'arbre binaire complet).
  procedure gauche( int i ) returns int g
  { g = 2 * i; }

  procedure droite( int i ) returns int g
  { g = 2 * i + 1; }

  procedure parent( int i ) returns int p
  { p = i / 2; }

  # Procedure auxiliaire: transporte un element vers le bas
  # a sa bonne position.
  procedure deplacerVersLeBas( int pos )
  {
    bool termine = false;
    while ( ~termine ) {
      int posMin = pos;
      int g = gauche(pos);
      if ( g <= nbElements & elements[g].prio < elements[posMin].prio ) {
        posMin = g;  # Le gauche est plus petit.
      }
      int d = droite(pos);
      if ( d <= nbElements & elements[d].prio < elements[posMin].prio ) {
        posMin = d;  # Le droite est encore plus petit.
      }

      # Note: Plus rapide (meme si egaux) qu'avec un if!
      termine = (pos == posMin);
      elements[pos]  :=: elements[posMin];
      pos = posMin;
    }
  }

  # Procedure auxiliaire: transporte un element vers le haut
  # a sa bonne position.
  procedure deplacerVersLeHaut( int pos )
  {
    while ( parent(pos) >= 1
            & elements[parent(pos)].prio > elements[pos].prio ) {
      elements[pos]  :=: elements[parent(pos)];
      pos = parent(pos);
    }
  }
```

```
  # Les operations publiques.
  proc inserer( prio, elem )
  {
    nbElements += 1;
    elements[nbElements] = NoeudFp( prio, elem );
    deplacerVersLeHaut( nbElements );
  }

  proc estVide() returns r
  { r = nbElements == 0; }

  proc retirerMin( prio, elem )
  {
    prio = elements[1].prio;
    elem = elements[1].elem;
    elements[1] = elements[nbElements];
    nbElements −= 1;
    deplacerVersLeBas( 1 );
  }

  proc insererN( prios, elems, n )
  # Particularite: Theta( nbElements + n )
  {
    for [i = 1 to n] {
      elements[nbElements+i] = NoeudFp( prios[i], elems[i] );
    }
    nbElements += n;
    for [i = nbElements/2 downto 1] {
      deplacerVersLeBas( i );
    }
  }
end
```

```
=====================================================
= Fichier tri.mpd: Tri avec file de priorite       =
=====================================================

  procedure trierAvecFilePriorite( ref int A[*] )
  {
    cap FilePriorite fp = create FilePriorite( ub(A) );

    # On ordonne partiellement les elements.
    for [k = 1 to ub(A)] {
      fp.inserer( A[k], A[k] );
    }

    # On les retire et on les insere dans le tableau.
    for [k = 1 to ub(A)] {
      fp.retirerMin( A[k], A[k] );
    }
  }




=====================================================
= Fichier sac−a−dos.mpd: Sac a dos fractionnaire   =
=====================================================

  procedure beneficeSac( int W, int poids[*], int benefs[*], int n,
                         res real S[*], res real benef )
  {
    cap FilePriorite ratios = create FilePriorite( n );

    # On veut la valeur maximum, d'ou le produit par "−1.0".
    for [i = 1 to n] {
      ratios.inserer( −1.0 * real(benefs[i]) / real(poids[i]), i );
    }

    benef = 0.0;
    S = ([n] 0.0);

    real w = 0.0;  # Poids restant

    while ( w < W ) {
      # Il reste de l'espace dans le sac.
      int i;
      real ratio;

      # On prend l'item avec le meilleur ratio benefice/unite de poids
      ratios.retirerMin( ratio, i );

      # On prend tout l'item...
      # a moins qu'il n'y ait plus assez d'assez d'espace.
      S[i] = min( poids[i], W−w );
      benef += S[i] * (−ratio);    # S[i] peut etre une fraction.
      w += S[i];
    }
  }
```