Université de Nantes
M2 Bioinformatique

Support pédagogique

Listes chaînées

Christine Sinoquet

SOMMAIRE

```
typedef struct {
        int val;
        struct TCell* next;
} TCell;
typedef  TCell* TPointer;Affichage d'une liste chaînée
Version itérative en C

/*********************************************************/

void parcours_it(TPointer tete){
   /* Pointeur initialisé en tête de liste, qui va permettre de la parcourir */
   TPointer p = tete;
   /* Tant que l'on n'est pas au bout de la liste */
   while (p != NULL){
      /* On affiche la valeur de la brique */
      afficher (p → val);
      /* On avance d'une brique */
      p = p →  next;
   }
}

/*********************************************************/

void main(){
 TPointer liste = creer_liste_chainee() ; /* primitive non développée ici */
 parcours_it(liste) ;
}
```

Ajout en tête d'une liste chaînée
Version itérative en C

```
void ajouterTete (TPointer* adr_tete_liste, int val){
 TPointer new_p = (TPointer) malloc (sizeof(TCell));
 new_p → val   = val;
 new_p → next = *adr_tete_liste;
 *adr_tete_liste = new_p;
}
```

Clônage d'une liste chaînée
Version itérative en C

```
//*******************************************************

void clone_iteratif (TPointer tete, TPointer* adr_tete_clone) {
 TPointer p = tete;
 TPointer p_clone;
 Tpointer prec_clone = NULL;  // prec_clone correspond au predecesseur de p. Il permet le chainage des
                              // elements de la liste
 while (p != NULL) {
  p_clone = (TPointer) malloc (sizeof (TCell));
  if (p_clone == NULL) { exit(0); } // Si l'allocation dynamique de la memoire echoue, le programme est
                                    // abandonne
  p_clone → val = p → val ;
  if (prec_clone == NULL) { //Au premier tour de boucle, la tete de la liste clonee est initialisee
   *adr_tete_clone = p_clone ;
  } else { // Aux tours suivants, le chainage est effectue avec la brique precedente de la liste clonee
   prec_clone → next = p_clone;
  }
  p = p → next;
 }

 if (tete != NULL) {
  p_clone → next = NULL; // Si la liste n'est pas vide, le dernier element de la liste clonee pointe vers NULL
 }
 else {
  *adr_tete_clone = NULL; // Si la liste a cloner est vide, alors la liste clonee est egalement vide
 }
} // fin clone_iteratif

//*******************************************************

void main {
 TPointer tete = ...; // tete pointe vers le debut d'une liste chainee.
 TPointer tete_clone;
 clone_iteratif (tete, &tete_clone);
}
```

Accès à la position k d'une liste chaînée
Version itérative en C

```
/*******************************************************/

void accesPos(TPointer tete, int k, TPointer* adr_ptr_cell, boolean* adr_trouve) {
 int            i  = 1 ;
 TPointer* p = tete ;
 while(p != NULL) {  // parcours de liste
  if(i == k) {  // la position k existe
   *adr_trouve = TRUE ;
   *adr_ptr_cell = p;
   return;
  }
  p = p → next ;
  i++ ;
 }
 *adr_trouve = FALSE ;  // La liste a moins de k éléments.
} // fin  accesPos

/*******************************************************/

void main(){
 TPointeur liste = creer_liste_chainee_4_briques(); /* primitive non développée ici */
 int          pos = 3;
 Tpointer ptr_cell;
 boolean   trouve;
 accesPos(liste, pos, &ptr_cell, &trouve) ;
}
```

Accès par valeur dans une liste chaînée
Version itérative en C

```
/*******************************************************/

 void access_val (TPoiter tete, int val, TPointer* adr_ptr_cell, bool* adr_trouve)
 {
  TPointer p     = tete;
  *adr_trouve   = FALSE;
  *adr_ptr_cell = NULL;

  while (p != NULL)
  {
   if (p → val == val)
   {
    *adr_ptr_cell = p;
    *adr_trouve = TRUE;
    return;
   }
   p = p->next;
  }
 } // end  access_val

/*******************************************************/

 int main()
 {
  TPointeur liste = creer_liste_chainee_4_briques(); /* primitive non développée ici */
  int val = 13;
  TPointer ptr_cell;

  access_val(liste, val, &prt_cell, &trouve);
 } // end main

/*******************************************************/
```

Ajout à la position k dans une liste chaînée
Version itérative en C

```
/*********************************************************/

void ajoutPos(TPointer* adr_tete, int k, boolean* adr_possible, int val){
        int i;
        TPointer p        = *adr_tete;
        TPointer prec_p = null;
        *adr_possible     = false;

        for(i=1; i<k; i++){
          if (p == null) return; // The list is too short.
           prec_p = p;
           p =  p→next;
        }
        // postcondition :
        // p correspond à  la position k <> 1 et la kième brique existe (insertion en milieu de liste),
        // ou bien p correspond à la position k <> 1 mais la kième brique n'existe pas (insertion en fin de
        // liste),
        // ou bien k = 1 et l'insertion en tête de liste est toujours possible
        *adr_possible = true;
        TPointer new_p = (TPointer) malloc (sizeof(TCell));
        new_p→val = val;
        new_p→next = p;
        if(prec_p == null){ //
          *adr_tete = new_p; // insertion en tête de liste
        }else{
          prec_p→next = new_p;
|       }
} // fin  ajoutPos


/*********************************************************/
void main(){
 TPointer liste = creer_liste_chainee_6_briques(); /* primitive non développée ici */
 int          pos  = 3;
 boolean    possible;
 int          val   = 999;

 ajoutPos(&liste, pos, &possible, val);
}


/*********************************************************/
```

Ajout dans une liste chaînée triée

Version itérative en C

```
/*********************************************************/

void ajouterTri(TPointer *adr_tete_liste, int val){
// precondition :
// La liste *adr_tete_liste est triée par ordre croissant.

        TPointer  p      = NULL;
        TPointer prec = NULL;
        p = *adr__tete_liste;

        //Préparation de la brique à insérer
        Tpointer new_p = (Tpointer) malloc (sizeof(Tcell));
        new_p → next = NULL // Précondition (*) prise en compte ultérieurement
        new_p → val   = val;

        // On vérifie que la liste n'est pas vide
        // Si elle est vide, la brique en cours d'insertion devient la tête de liste
        if (*adr_tete_liste == NULL) {
                *adr_tete_liste = new_p;
                // (*) prise en compte
                return;
        }

        //On vérifie si la valeur à insérer est inférieure ou égale à la valeur de la tête de liste,
        // Si oui, la nouvelle brique devient la nouvelle tête de liste.
        if (p → val >= val){
                new_p → next = *adr_tete_liste;
                *adr_tete_liste = new_p;
                return;
        }

        //La brique n'est pas insérée en tête de liste, on cherche sa place dans la liste
        while (p != NULL){
                if (p → val  >=  val){ // Il faut insérer la nouvelle brique entre les briques d'adresses
                                       // physiques respectives prec et p.
                        new_p → next = p;
                        prec → next     = new_p;
                        return;
                 }
                prec = p;
                p      = p → next;
```

```
        }
        // Il reste à traiter le cas de l'insertion en fin d'une liste non vide.
        prec → next = new_p;
        // (*) prise en compte
} // fin  ajouterTri


/*******************************************************/
```

Génération de la liste chaînée inverse d'une liste chaînée
Version itérative en C

```
/*********************************************************/

void miroir(TPointer tete, TPointer* adr_tete_inv){
        if (tete == NULL){
                *adr_tete_inv = NULL;
                return;
        }

        TPointer p = tete;
        TPointer p_inv;
        TPointer prec_inv = NULL;

        while (p != NULL){
                p_inv = (TPointer) malloc(sizeof(TCell));
                p_inv->info = p->info;

                if (prec_inv == NULL){
                        p_inv->next = NULL;
                } else {
                        p_inv->next = prec_inv;
                }

                prec_inv = p_inv;
                p = p → next;
        }

        *adr_tete_inv = p_inv;
} // fin  miroir

/*********************************************************/

void main(){

 Tpointer   liste;
 Tpointer   liste_miroir;
  liste = creer_liste_chainee(…) // non devéloppé ici

miroir(liste, &liste_miroir);
} // fin main


/*********************************************************/
```

Gestion d'une matrice creuse à l'aide d'une liste chaînée de listes chaînées
Version en C

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

typedef struct TCellCol {
 int      col;
 double val;
 struct TCellCol* suiv;
} TCellCol;

typedef struct TCellLig {
 int lig;
 struct TCellCol* tete_cols;
 struct TCellLig* suiv;
} TCellLig;

typedef TCellLig* TPtrCellLig;
typedef TCellCol* TPtrCellCol;

typedef struct TCellMat {
 int             nb_max_ligs;
 int             nb_max_cols;
 double        val_def;
 TPtrCellLig tete_ligs;
} TCellMat;

typedef TCellMat* TMat;

/*********************************************************/

void init(TMat        pmat,
          int           nbMaxLignes,
          int           nbMaxColonnes,
          double       val_def,
          TPtrCellLig tete_ligs)
{
 pmat → nb_max_ligs  = nbMaxLignes;
 pmat → nb_max_cols = nbMaxColonnes;
 pmat → val_def         = val_def;
 pmat → tete_ligs       = tete_ligs;
} // fin init

/*********************************************************/

void recherche_i(TPtrCellLig     tete_ligs,
                 int                 i,
                 TPtrCellLig*   adr_p_prec_i,
                 TPtrCellLig*   adr_p_i)
{
// precondition :
// tete_ligs est trié par ordre croissant.
//
// postcondition :
```

```
// Soit *adr_p_i == NULL et *adr_p_prec_i est valué significativement
// Soit *adr_p_i != NULL (et est valué significativement) et *adr_p_prec_i est valué significativement.

 TPtrCellLig p, prec;
 p = tete_ligs; prec = NULL;
 while( p!= NULL )
 {
  if ( p → lig == i )
  {
   *adr_p_i      = p;
   *adr_p_prec_i = prec;
   return;
  }

  if ( p → lig > i )
  {
   *adr_p_i      = NULL;
   *adr_p_prec_i = prec;
   return;
  }
  prec = p;
  p    = p → suiv;
 }
// precondition :
// i n'est pas present dans la liste tete_ligs (éventuellement parce que la liste est vide).
 *adr_p_i = NULL;
 if (tete_ligs == NULL) { *adr_p_prec_i = NULL; } else { *adr_p_prec_i = prec; }
} // fin recherche_i

/*********************************************************/

void recherche_j(TPtrCellCol   tete_cols,
                 int           j,
                 TPtrCellCol*  adr_p_prec_j,
                 TPtrCellCol*  adr_p_j)
{
// precondition :
// tete_cols est trié par ordre croissant.
//
// postcondition :
// Soit *adr_p_j == NULL et *adr_p_prec_j est valué significativement
// Soit *adr_p_j != NULL (et est valué significativement), et *adr_p_prec_j est valué significativement.

 TPtrCellCol p, prec;
  p = tete_cols; prec = NULL;
 while( p!= NULL )
 {
  if( p → col == j )
  {
   *adr_p_j      = p;
   *adr_p_prec_j = prec;
   return;
  }

  if( p → col > j )
  {
```

```
   *adr_p_j      = NULL;
   *adr_p_prec_j = prec;
   return;
  }
  prec = p;
  p    = p → suiv;
 }
 // precondition :
// j n'est pas present dans la liste tete_cols (éventuellement parce que la liste est vide).
 *adr_p_j= NULL;
 if (tete_cols == NULL) { *adr_p_prec_j = NULL; } else { *adr_p_prec_j = prec; }
} // fin  recherche_j

/*******************************************************/

bool recherche_i_j(TMat          pmat,
                   int           i,
                   int           j,
                   TPtrCellLig*  adr_p_prec_i,
                   TPtrCellLig*  adr_p_i,
                   TPtrCellCol*  adr_p_prec_j,
                   TPtrCellCol*  adr_p_j)
{
 recherche_i(pmat→tete_ligs, i, adr_p_prec_i, adr_p_i);
 if (*adr_p_i == NULL) { return false; }
 // postcondition:
// La cellule i existe.

 recherche_j((*adr_p_i)→tete_cols, j, adr_p_prec_j, adr_p_j);
 return (adr_p_j != NULL);
} // end  recherche_i_j

/*******************************************************/

void supprimer_i_j(TMat          pmat,
                   TPtrCellLig   p_prec_i,
                   TPtrCellLig*  adr_p_i,
                   TPtrCellCol   p_prec_j,
                   TPtrCellCol*  adr_p_j)
{
// precondition:
// *adr_p_i et *adr_p_j sont non nuls et valués significativement.
/ /p_prec_i et p_prec_j sont valués significativement (à NULL si i ou j sont en tête de liste)

 if(p_prec_j == NULL) { (*adr_p_i)→tete_cols = (*adr_p_j)→suiv; } // chaînage particulier pour la
                                                                  // tête de liste
 else { p_prec_j→suiv = (*adr_p_j)→suiv; } // chaînage en milieu de liste (y compris chaînage en fin de
                                           //  liste)
 free(*adr_p_j); *adr_p_j=NULL;

 if ((*adr_p_i)->tete_cols == NULL) // On vient de supprimer la dernière brique de la liste
                                    // (*adr_p_i)→tete_cols.
 {
  if (p_prec_i == NULL) { pmat→tete_ligs = (*adr_p_i)→suiv; } // chaînage particulier pour la
                                                              // tête de liste
```

```
  else { p_prec_i→suiv = (*adr_p_i)→suiv; } // chaînage en milieu de liste (y compris chaînage en fin de
                                                //  liste)
  free(*adr_p_i); *adr_p_i = NULL;
 }

} // fin supprimer_i_j

/*********************************************************/

void inserer_i_j(TMat           pmat,
                 TPtrCellLig    p_prec_i,
                 TPtrCellLig*   adr_p_i,
                 TPtrCellCol    p_prec_j,
                 TPtrCellCol*   adr_p_j,
                 int            i,
                 int            j,
                 double         val)
{
// precondition
// Soit *adr_p_i != NULL, et *adr_p_j == NULL et p_prec_j est valué significativement (éventuellement à
// NULL)
// Soit *adr_p_i == NULL et p_prec_i est valué significativement (éventuellement à NULL).

 if(*adr_p_i == NULL) { // pas de brique i
  *adr_p_i = (TCellLig*) malloc(sizeof(TCellLig));
  if (*adr_p_i == NULL) {printf("Allocation mémoire impossible."); exit(1);}
  (*adr_p_i)→lig=i;
  if(p_prec_i == NULL) // insertion en tête de liste pmat→tete_ligs
  {
   (*adr_p_i)→suiv = pmat→tete_ligs;
   pmat→tete_ligs   = *adr_p_i;
  } else { // insertion en milieu (y compris fin) de liste pmat→tete_ligs
   (*adr_p_i)→suiv = p_prec_i→ suiv;
   p_prec_i→suiv    = *adr_p_i;
  }
 }
//  postcondition :
//  La brique i existe, qu'on vienne de la créer ou non.

 *adr_p_j = (TCellCol*) malloc(sizeof(TCellCol));
 if(*adr_p_j == NULL) { printf("Allocation mémoire impossible."); exit(1); }
 (*adr_p_j)→ col=j; (*adr_p_j)→val=val;

 if(p_prec_j == NULL) // insertion en tête de liste (*adr_p_i)→tete_cols
 {
  (*adr_p_j)→suiv        = (*adr_p_i)→tete_cols;
  (*adr_p_i)→tete_cols = *adr_p_j;
 } else { // insertion en milieu (y compris fin) de liste (*adr_p_i)→tete_cols
 {
  (*adr_p_j)→suiv = p_prec_j→suiv;
  p_prec_j→suiv    = *adr_p_j;
 }
} // fin  inserer_i_j

/*********************************************************/
```

```
void set(TMat pmat, int i, int j, double val)
{
 bool exist_dans_struct;
 TPtrCellLig p_i      = NULL;
 TPtrCellLig p_prec_i = NULL;
 TPtrCellCol p_j      = NULL;
 TPtrCellCol p_prec_j = NULL;

 exist_dans_struct = recherche_i_j(pmat, i, j, &p_prec_i, &p_i, &p_prec_j, &p_j);
 if(exist_dans_struct) // La valeur à remplacer n'est pas la valeur par défaut.
 {
  if (val == pmat→val_def) { // La valeur remplaçante est la valeur par défaut.
   supprimer_i_j(pmat, p_prec_i, &p_i, p_prec_j, &p_j);
  } else {
   p_j→val = val;
  }
 } else { // La valeur à remplacer est la valeur par défaut.
  if (val == pmat→val_def) { // // La valeur remplaçante est la valeur par défaut.
   // action vide
  } else {
    inserer_i_j(pmat, p_prec_i, &p_i, p_prec_j, &p_j, i, j, val);
  }
 }
} // fin  set

/*********************************************************/
double get(TMat pmat, int i, int j)
{
 TPtrCellLig p_prec_i = NULL;
 TPtrCellLig p_i      = NULL;
 TPtrCellCol p_prec_j = NULL;
 TPtrCellCol p_j      = NULL;

 if (recherche_i_j(pmat, i, j, &p_prec_i, &p_i, &p_prec_j, &p_j))
 {
  return p_j→val; // La «case» i_j existe bien dans la structure.
 }
 // postcondition:
 // La «case» i_j n'existe pas dans la structure, donc la valeur par défaut est associée à la «case» i_j .
 return pmat->val_def;
} // fin  get

/*********************************************************/


void main()
{
 TMat matrice = (TCellMat*)malloc(sizeof(TCellMat));
 init(matrice, 5, 5, 0, NULL); // Pour cette matrice creuse, la valeur par défaut est 0.
 set(matrice, 1, 1, 18);  // mat[1][1] ← 18
 set(matrice, 4, 4, 0);    // mat[4][4] ← 0

 printf("%f\n", get(matrice, 1, 1)); // afficher(mat[1][1])
 printf("%f\n", get(matrice, 4, 4)); // afficher(mat[4][4])
```

```
 set(matrice, 1, 1, 0);     // mat[1][1] ← 0
 set(matrice, 4, 4, 43);    // mat[4][4] ← 43

 printf("%f\n", get(matrice, 1, 1));
 printf("%f\n", get(matrice, 4, 4));
} // fin main
```