École préparatoire aux sciences et techniques, Annaba
Année universitaire 2012/2013
Module : Informatique 2

Série de TD n°8 : Listes chainées Solution

Rappel sur les listes chainées

1. Écrire une structure de données qui permet de représenter une liste chainée d'entiers (Cette structure sera utilisée dans la suite de la série).

```
struct element {
  int val;
  element * suiv;
}
```

2. De quel autre manière plus lisible on pourrait écrire `(*p).val` ?

   `p->val`

3. Quel valeur constante indique qu'un pointeur est vide (ou ne pointe vers aucune adresse) ?

   `NULL ou 0`

4. Comment s'appelle le premier élément d'une liste chainée ?

   `La tête`

5. Comment s'appelle le dernier élément d'une liste chainée ?

   `La queue`

6. Sachant qu'une liste chainée est identifiée par un pointeur `tete` qui contient l'adresse de son premier élément, comment indiquer que cette liste est vide ?

   `tete = NULL; //Affecter NULL à la tête de la liste.`

7. Le dernier élément d'une liste chainée a une particularité, laquelle ?

   `Son champs suiv est égale à NULL.`

8. Le parcours d'une liste chainée se fait-t-il généralement en utilisant une boucle `while` ou une boucle `for` ? pourquoi ?

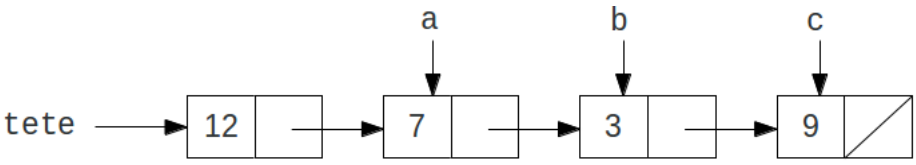
   `Une boucle while car on ignore combien d'éléments la liste contient.`

9. Remplir ce tableau de comparaison entre les listes chainées et les tableaux.

	Tableaux	**Listes chainées**
Accès aux éléments (direct ou sequentiel)	`direct`	`sequentiel`
Taille	`constante`	`variable`
Insertion (par décalage ou par chainage)	`décalage`	`chainage`

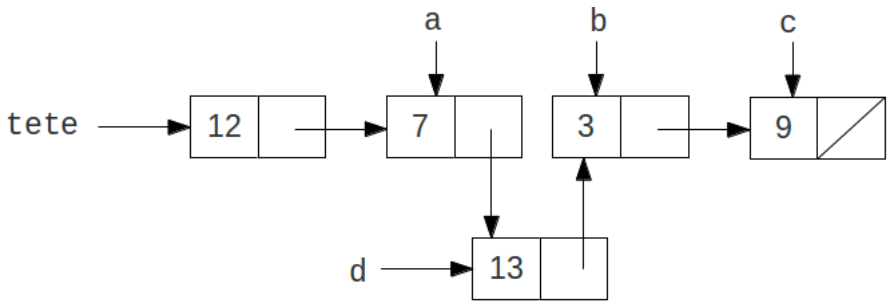
Exercice 1

1. Écrire un programme qui créé la liste chainée representée dans la figure suivante :




```
int main() {
  element * tete = new element;
  element * a = new element;
  element * b = new element;
  element * c = new element;
  tete->val = 12;
  tete->suiv = a;
  a->val = 7;
  a->suiv = b;
  b->val = 3;
  b->suiv = c;
  c->val = 9;
  c->suiv = NULL;
}
```

2. Ajouter au programme précédent les instructions qui permettent de créér et d'insérer l'élément pointé par `d`, entre les éléments `a` et `b` comme representé dans la figure suivante :




```
element * d = new element;
d->val = 13;
```

```
a->suiv = d;
d->suiv = b;
```

Exercice 2

1. Écrire une fonction `inserer` qui permet d'insérer un élémént au début d'une liste chainée d'entiers.

```
void inserer(element * & tete, int x) {
  //tete est passé par référence car elle va changer.
  //L'élement à insérer sera la nouvelle tête.
  element * e = new element;
  e->val = x;
  e->suiv = tete;
  tete=e;
}
```

2. Écrire une fonction `insererQueue` qui permet d'insérer un élémént à la fin d'une liste chainée d'entiers.

```
void insererQueue(element * & tete, int x) {
  element * e = new element;
  e->val = x;
  e->suiv = NULL;
  if (tete==NULL)
    tete = e;
  else {
    element * p = tete;
    while (p->suiv!=NULL)
      p=p->suiv;
    p->suiv = e;
  }
}
```

3. Écrire une fonction `afficher` qui permet d'afficher tous les éléments d'une liste chainée d'entiers.

```
void afficher(element * tete) {
  while (tete!=NULL) {
    cout << tete->val << "\t";
    tete = tete->suiv;
  }
}
```

4. Écrire une version récursive de la fonction `afficher`.

```
void afficherRecursive(element * tete) {
  if (tete!=NULL) {
    cout << tete->val << "\t";
    afficherRecursive(tete->suiv);
```

```
  }
}
```

5. Écrire une fonction `compter` qui retourne le nombre d'éléments d'une liste chainée d'entiers.

```
int compter(element * tete) {
  int c = 0;
  while (tete!=NULL) {
    c++;
    tete = tete->suiv;
  }
  return c;
}
```

6. Écrire une fonction `somme` qui retourne la somme des éléments d'une liste chainée d'entiers non vide.

```
int somme(element * tete) {
  int s = 0;
  while (tete!=NULL) {
    s += tete->val;
    tete = tete->suiv;
  }
  return s;
}
```

7. Écrire une fonction `min` qui retourne la valeur du plus petit élément d'une liste chainée d'entiers non vide.

```
int min(element * tete) {
  int m = tete->val;
  tete=tete->suiv;
  while (tete!=NULL) {
    if (tete->val<m) {
      m=tete->val;
    }
    tete = tete->suiv;
  }
  return m;
}
```

8. Écrire une fonction `existe` qui teste si un élémént donné existe dans une liste chainée d'entiers. La fonction doit retourner `true` si l'élément existe et `false` sinon.

```
bool existe(element * tete,int x) {
  while (tete!=NULL && tete->val!=x)
    tete = tete->suiv;
  if (tete==NULL)
    return false;
  else
```

```
    return true;
}
```

9. En utilisant les fonctions précédentes, écrire un programme principale qui
   (a) Déclare une liste chainée d'entiers vide `li`.
   (b) Insére des éléments arbitraires à la liste chainée `li`.
   (c) Affiche tous les éléments de la liste chainée `li`.
   (d) Affiche le nombre d'éléments de la liste chainée `li`.
   (e) Affiche la somme des éléments de la liste chainée `li`.
   (f) Affiche la valeur du plus petit élément de la liste chainée `li`.

```
int main() {
  //(a)
  element * li = NULL;
  //(b)
  inserer(li,12);
  inserer(li,7);
  inserer(li,18);
  //(c)
  afficher(li); // affichera 18 7 12
  //(d)
  cout << compter(li); affichera 3
  //(e)
  cout << somme(li); // affichera 47
  //(f)
  cout << min(li); // affichera 7
}
```

Exercice 3

Écrire une fonction `link` qui crée un lien entre deux listes chainées d'entiers `li1` et `li2` non vides. Cette fonction chaine le dernier élément de la liste `li1` au premier élément de la liste `li2`.

```
void link(element * li1, element * li2) {
  while (li1->suiv!=NULL)
    li1=li1->suiv;
  li1->suiv = li2;
}
```

Exercice 4

1. Écrire une fonction `tabToList` qui transforme un tableau d'entiers en liste chainée. La fonction doit retourner un pointeur qui indique la tête de la liste.

```
element * tabToList(int tab[], int taille) {
  element * tete = NULL;
  for (int i=taille-1;i>=0;i--) {
    element * e = new element;
    e->val = tab[i];
    e->suiv = tete;
    tete = e;
  }
  return tete;
}
```

2. Écrire une fonction `listToTab` qui transforme une liste chainée d'entiers en tableau. La fonction doit retourner un tableau qui contient tous les éléménts de la liste chainée.

```
int * listToTab(element * tete) {
  int taille = compter(tete);
  int * t = new int[taille];
  for (int i=0;i<taille;i++) {
    t[i] = tete->val;
    tete = tete->suiv;
  }
  return t;
}
```

Exercice 5

1. Écrire une fonction `supprimerListe` qui supprime de la mémoire tous les éléments d'une liste chainée d'entiers.

```
void supprimerListe(element * & tete) {
  element * p = tete;
  while (p!=NULL) {
    element * tmp = p->suiv;
    delete p;
    p = tmp;
  }
  tete = NULL;
}
```

2. Écrire une version récursive de la fonction `supprimerListe`.

```
void supprimerListeRecursive(element * & tete) {
  if (tete!=NULL) {
    element * listeRestante = tete->suiv;
    delete tete;
    tete = NULL;
    supprimerListeRecursive(listeRestante);
  }
```

```
}
```

3. Écrire une fonction `supprimerElement` qui supprime la première occurence d'un élément donnée d'une liste chainée d'entiers. Si l'élément n'existe pas, la liste reste inchangée.

```
void supprimerElement(element * & tete, int x) {
  if (tete!=NULL) {
    if (tete->val==x) {
      element * p = tete->suiv;
      delete tete;
      tete = p;
    } else {
        element * p = tete;
        while (p->suiv!=NULL) {
          if (p->suiv->val==x) {
            element * tmp = p->suiv;
            p->suiv = tmp->suiv;
            delete tmp;
            break;
          }
          p=p->suiv;
        }
      }
  }
}
```

4. Écrire une version récursive de la fonction `supprimerElement`.

```
void supprimerRecursif(element * & tete, int x) {
  if (tete!=NULL) {
    if (tete->val==x) {
      element * tmp = tete;
      tete = tete->suiv;
      delete tmp;
    }
    else
      supprimerRecursif(tete->suiv,x);
  }
}
```

Exercice 6

Nous voulons implémenter une pile en utilisant une liste chainée. Expliquer comment implémenter les opérations suivantes sans donner de code source.

1. `int sommet(element * tete) //on admet que la liste chainée est non vide`

   `Retourner la valeur de l'élement tete (tete->val).`

2. `void empiler(element * & tete, int val)`

   `Insérer un élément en tête de la liste.`
   `La nouvelle tête de la liste sera cet élément.`

3. `int depiler(element * & tete) //on admet que la liste chainée est non vide`

   `Supprimer la tête de la liste et renvoyer sa valeur.`
   `La nouvelle tête de la liste sera l'élément qui suit l'ancienne tête.`

Exercice 7

Nous voulons implémenter une file en utilisant une liste chainée. Expliquer comment implémenter les opérations suivantes sans donner de code source.

1. `void enfiler(element * & tete, int val)`

   `Insérer un élément en queue de la liste.`
   `Si la liste est vide la nouvelle tête sera cet élément.`

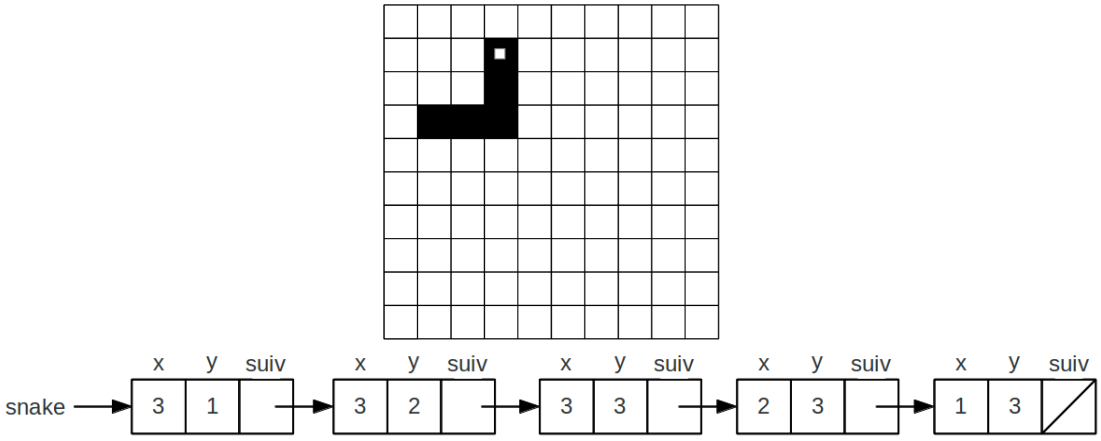
2. `int defiler(element * & tete) //on admet que la liste chainée est non vide`

   `Supprimer la tête de la liste et renvoyer sa valeur.`
   `La nouvelle tête de la liste sera l'élément qui suit l'ancienne tête.`

Problème

Dans le cadre du développement du célébre jeu de serpent *snake*, l'équipe de développement a décidé d'implémenter le snake en utilisant une liste chainée. Chaque element de cette liste correspond à un maillon du snake et contient les coordonnées x et y de ce maillon. Le snake évolue dans une grille de 10x10 cases et peut posséder de 2 à 9 maillons.

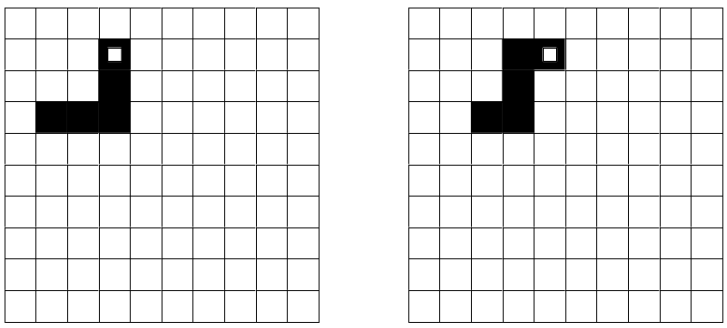
La figure suivante montre l'état du snake à un moment donnée ainsi que la liste chainée correspondante.



1. Donner une structure de données qui permet de representer le snake.

```
struct element {
  int x;
  int y;
  element * suiv;
}
```

2. Le snake se déplace d'une seule case à chaque étape vers une direction qui peut être nord sud est ou ouest. L'équipe de développement a choisi d'implementer le mouvement du snake en faisant un ajout en tête et une suppression en queue. Les coordonnées de la nouvelle tete du snake son calculées en fonction de la direction voulue. La figure suivante donne un exemple de mouvement du snake vers l'est.



– Ecrire une fonction `mouvement` qui permet de faire avancer d'une seule case sur la grille un snake donné en paramètre vers une direction donnée (nord, sud, est ou ouest). Il faut prendre en compte le fait que le snake peut sortir d'une des extremités de la grille et apparaitre à l'extermité opposée.

```
void mouvement(element * & snake, char direction) {
  //Ajout en tête
  element * nouvelleTete = new element;
  switch (direction) {
    case 'n': nouvelleTete->x = snake->x;
              nouvelleTete->y = (10 + snake->y - 1) % 10;
              break;
    case 's': nouvelleTete->x = snake->x;
              nouvelleTete->y = (snake->y + 1) % 10;
              break;
    case 'e': nouvelleTete->x = (10 + snake->x - 1) % 10;
              nouvelleTete->y = snake->y;
              break;
    case 'o': nouvelleTete->x = (snake->x + 1) % 10;
              nouvelleTete->y = snake->y;
              break;
  }
  nouvelleTete->suiv = snake;
  snake = nouvelleTete;

  //Suppression en queue
  element * p = snake;
  while (p->suiv->suiv!=NULL)
    p=p->suiv;
  delete p->suiv;
  p->suiv = NULL;
}
```