Licence 3 Informatique
2019-2020

L*istes doublement chaînées*

KOUCHE Rabia
Encadré par : Veronique Jay

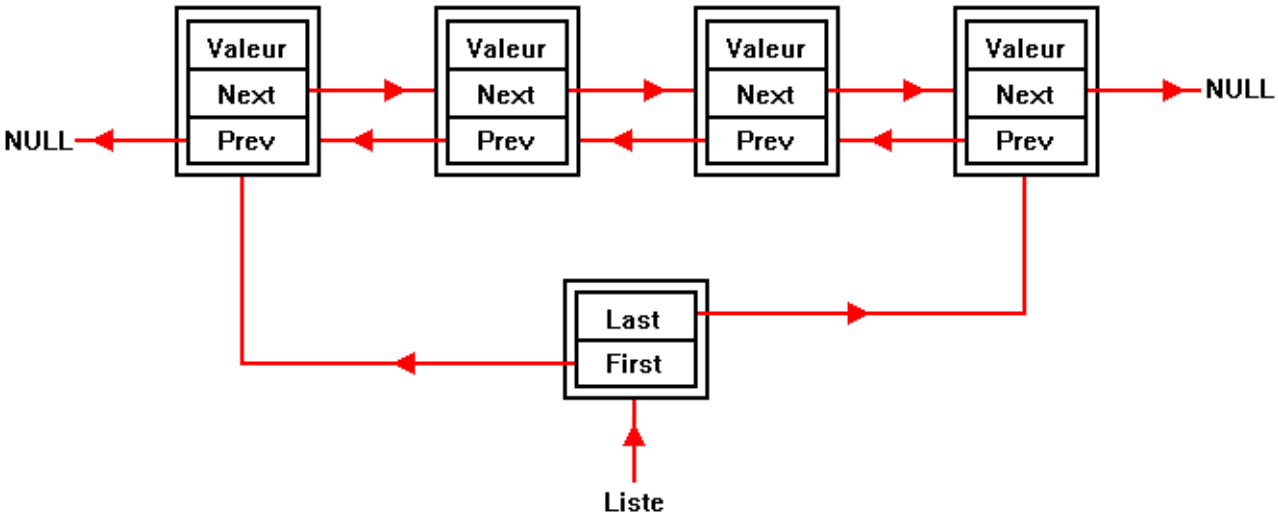


Figure1 : Liste chaînée en c++

Définition des listes doublement chaînées:

Une liste doublement chaînée est composée de nœuds, chacun ayant deux pointeurs vers le nœud suivant et le précédent de la liste. Chaque nœud contient une valeur.

Le précédent de premier élément et le suivant de dernier élément de la liste pointent sur NULL.

Pour accéder à un élément de la liste peut être parcourue dans les deux sens :

- *En commençant avec la tête, le pointeur suivant permettant le déplacement vers l'élément suivant.*

- *En commençant avec le dernier nœud, le pointeur précédent permettant le déplacement vers l'élément précédent.*

Contrairement aux tableaux une liste pour accéder à ses éléments il faut parcourir tout les éléments qui précédent l'élément rechercher.

Pour insérer ou supprimer un élément on est pas obligé de décalé les éléments suivants, chaque valeur est dans un bloc de mémoire différent.

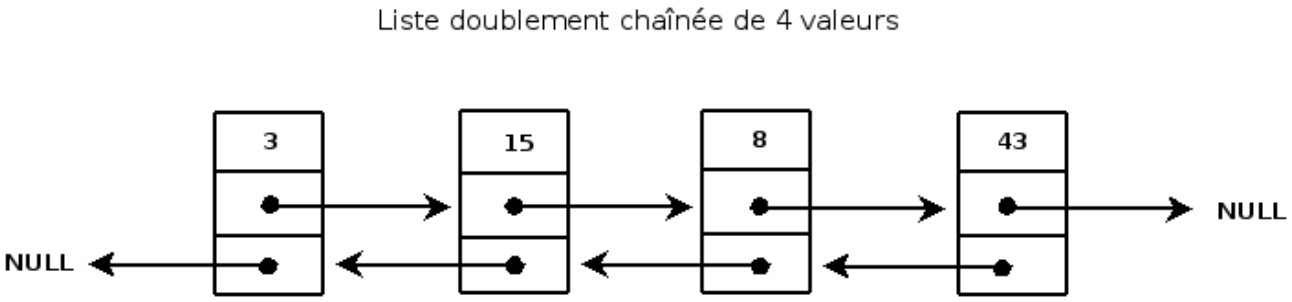
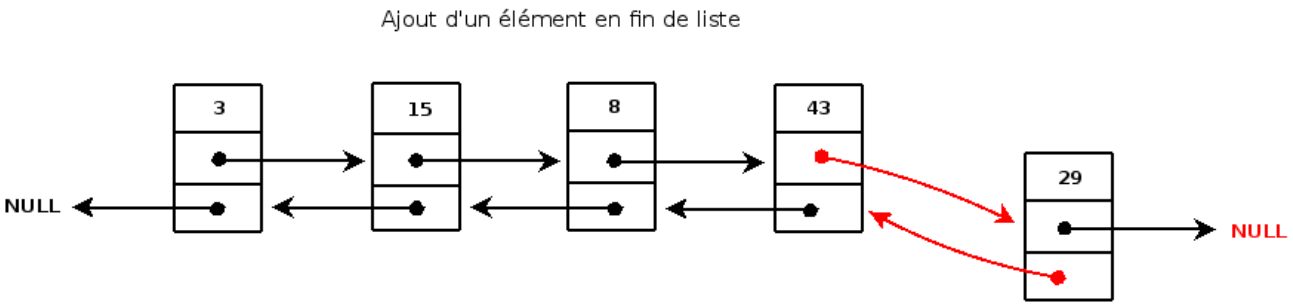


Figure2 : liste à 4 valeurs

Maintenant que nous avons un pointeur à la fois sur l'élément suivant et sur l'élément précédent, on peut supprimer, ajouter, et insérer un élément a une position donnée de la liste en refaisant les liens des éléments suivant et précédents pour qu'ils se connectent ensemble.

1-L’ajout d’un élément à la fin de la liste :



-figure 3-

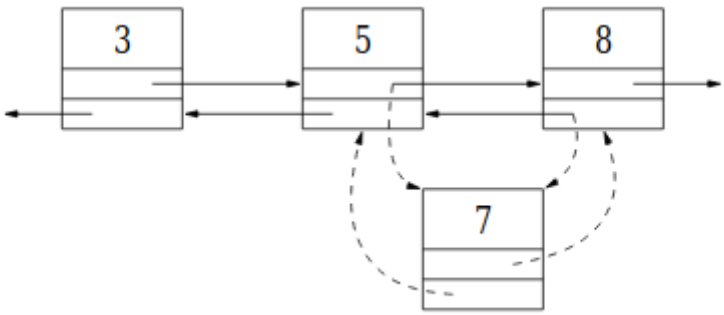
Code source de l’implémentation de la fonction ajouter :

```
// ajouter s a la fin de la liste
void Liste::ajouter(const std::string& s){
    Element* pe = new Element(s);
    if(premier == NULL){
      premier = dernier =pe;
    }
    else{
      pe->precedent = dernier;
      dernier ->suivant = pe;
      dernier = pe;
        }
}
```

L’ajout d’un élément à la fin de la liste se fait de la manière suivante:
si la liste est vide « if(premier == NULL) » on l’ajoute directement en attachant premier et le dernier vers l’élément pe.
Sinon :
- nous faisons pointer le précédent de l’élément pe vers le dernier
- nous faisons pointer le suivant de dernier vers «pe».
enfin nous attachons dernier vers pe.

2-Insertion d’un élément dans la liste :

Ainsi on peut facilement insérer un élément dans la liste comme le montre la figure ci-dessous.



Code source de l'implémentation de la fonction insérer :

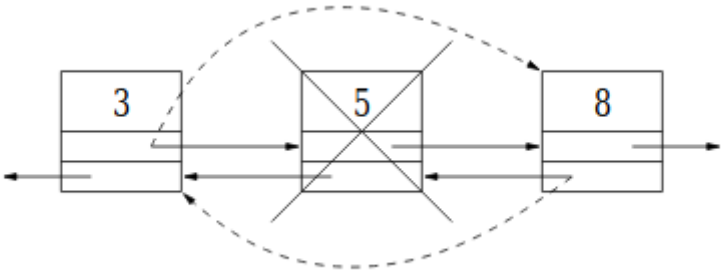
```
// ajouter s avant la position pos
void Liste::inserer(Iterateur& pos, const std::string& s)
{
    if (pos.position == NULL) this->ajouter(s); /* si la liste vide on l'insert directement */
      else {
            Element* element=new Element(s); /* création d'un nouveau element */
        if (pos.position == premier){ /* si la position est sur le premier element */
          element->suivant = pos.position;
          pos.position->precedent = element;
          premier = element;
        }else{ /* si la position est entre deux éléments de la liste */
          element->suivant = pos.position;
          pos.position->precedent->suivant = element;
          element->precedent = pos.position->precedent;
          pos.position->precedent = element;
        }
      }
  }
```

L'insertion d'un élément avant une position donnée se fait de la manière suivante :
Tout d'abord nous devons vérifier si notre liste est NULL, si oui on va l'ajouter directement avec notre fonction ajouter(s), si elle ne l'est pas nous allons crier un nouvel élément «element», à partir de là deux possibilités s'offre à nous :

1. *Si la position de notre élément est sur le premier élément de notre liste, alors nous faisons pointer le suivant de notre élément vers position « pos », et le précédent de notre position vers l'élément criée, enfin nous rattachons le premier de notre liste vers l'élément «element».*
2. *Si la position «pos» est entre deux éléments de la liste, alors nous faisons pointer le suivant de notre élément vers position, et le suivant de précédent de la position «pos» vers l'élément criée, et le précédent de «element» vers le précédent de «pos», enfin «pos»pointera sur «element».*

3-Suppression d'un élément dans la liste :

on peut aussi facilement supprimer un élément de la liste comme le montre la figure ci_dessous :



Code source de l'implémentation de la fonction supprimer :

```
// supprimer l'element a la position pos
void Liste::supprimer(Iterateur& pos) {
    if (pos.position != NULL){//la liste n'est pas vide !!
      if (premier == dernier) {//notre liste contient qu'un seul element
        premier = dernier = NULL;
      }else{
        if (pos.position->suivant == NULL) {//pos se trouve au dernier element dans la Liste
          dernier = pos.position->precedent;
          pos.position->precedent->suivant=NULL;


        }else{
          if (pos.position->precedent == NULL) {/*le cas ou pos se trouve dans le premier element */
            premier = pos.position->suivant;
            pos.position->suivant->precedent = NULL;
          }else{/* le cas ou element se trouve entre deux element de la liste */
            pos.position->precedent->suivant = pos.position->suivant;
            pos.position->suivant->precedent = pos.position->precedent;

          }
        }
      }
    }
    delete(pos.position);/*dans tout les cas on doit libérer la memoire*/

}
```

La suppression d'un élément dans la liste se fait de la manière suivante :

D'abord essayons de voir si la liste n'est pas vide, si oui on doit aussi vérifier si notre liste a qu'un seul élément (si premier = le dernier), si oui on le mit directement à NULL, et on libérera la mémoire qu'il occupe a la fin de notre fonction.

Sinon trois possibilités s'offrent à nous :

1. *Si «pos» est au dernier élément de la liste :*
   - *Nous faisons pointer le dernier élément vers le précédent de la position «pos».*
   - *Nous faisons pointer le suivant de précédent de «pos» à NULL;*

2. *Si «pos» est au premier élément de la liste :*
   - *Nous faisons pointer le premier élément vers le précédent de la position «pos».*
   - *Nous faisons pointer le suivant de précédent de «pos» à NULL.*

3. *Si «pos» est etre deux éléments de la liste :*
   - *Nous faisons pointer le suivant de précédent de la position «pos» vers le suivant de «pos».*
   - *En fin nous faisons pointer le précédent de suivant de la position «pos» vers le précédent de «pos».*

Et dans tout les cas nous devons libérer la mémoire avec ***delete.***

La surcharge des opérateurs :

Qu'est-ce que c'est ?

*C’est une technique qui nous permet de réaliser des opérations arithmétiques sur nos objets en criant des méthodes pour modifier le comportement des symboles +, -, *,==,!=, /, >=, etc.*

1-a) L’opérateur ++ (post fixé):

L'opérateur d'addition postfixe se définit comme ceci dans le ***.h****:*

```
Iterateur& operator++();
```

Une fois que nous aurons écrit cette fonction dans le ***.cc****, nous pourrons effectuer des additions d’itérateurs comme si de rien était.*

```
int main() {
Iterateur it = it.debut();
it++,it++,it++; /* ici c’est comme si on a fait «it = it.operator++()»*/
return 0;
}
```

1-b) L’opérateur ++ (infixe):

De même pour l'opérateur d'addition infixe il se définit comme ceci :

```
Iterateur operator++(int);
```

et dans le main()

```
int main(){
Iterateur it = it.debut();
++it;++it;++it;/*ici c'est comme si on a fait it=it.iterator++(1)*/

return 0;

}
```

La différence entre les deux (postfixe et infixe) c'est que dans le postefixe si on veut afficher (it++) il affiche d'abord l'itérateur it après il l'incrémente, en revanche le infixe il l'incrémente au premier lieu, puis il l'affiche.

Le postfixe on a retourner une référence parce que on modifier l'itérateur courant mais dans le infixe on crier un itérateur résultat on l'incrémente puis on le retourne.

«Idem pour la décrémentation»

2- L'opérateur *(get()) :

L'opérateur (*) se définit comme ceci dans le ***.h****:*

```
std::string& operator*()const;
```

- Elle nous permet de retourner la valeur de position courante.

- Le mot clé ***const*** *signifie que la valeur de l'élément courant peut être une valeur constante.*

- lors d'un passage par référence, la variable (ou l'objet) n'est pas copiée, utiliser une référence permet donc d'éviter la copie inutile d'objets.

3- Les opérateurs de comparaison (== et !=) :

voici leurs signatures :

```
bool operator==(const Iterateur&);
bool operator!=(const Iterateur&);
```

ici quand on fait la comparaison entre deux itérateurs (T1 == T2), le compilateur le traduit directement en T1.operator==(T2), un opérateur se transforme alors en appel de fonction.

Le compilateur appelle donc la fonction `operator==` *en passant en paramètres* `l'opérande` `T2`*. La fonction, elle renvoie un résultat de type* **`bool`***.*
l'itérateur passé en paramètre ne *doit pas être modifié. Le mot-clé* **`const`** *est donc très important dans ce cas.*

4- l'opérateur d'assignation = et le constructeur par copie :
On appel un constructeur que lorsque on réserve un espace mémoire pour un objet qui copie de l'objet déjà existant.
Par défaut il en existe un constructeur par copie, et l'opérateur d'assignation il en existe un aussi, si aucun constructeur n'est présent explicitement dans la classe, ils sont générés automatiquement.

Si la classe fait de la location dynamique il est impératif de crier le constructeur par copie.

Syntaxe constructeur par copie :

Liste::Liste(const Liste& l)

il prend un argument dont le type est celui de la classe, dans la plupart des cas l'argument est passé par référence constante, plus rarement il peut être passé par valeur ou référence non constante.

Syntaxe l'opérateur d'assignation = :

Pour surcharger l'opérateur= , il suffit de créer une méthode:

Liste& Liste::operator=(const Liste& l)
Elle prend en paramètre une référence constante, et on retourne une référence car on retourne une liste modifiée, et pour éviter la problématique de l'auto-affectation par exemple a = b = c,
on affecte les valeurs de l'objet c à b et puis celles de b à c.

Mise en place de la généricité

La liste qui vient d'être présenté ne peut contenir que des chaînes de caractères. Ainsi, s'il arrivait que l'on veuille utiliser une liste de double, d'entiers ou même d'objets, il faudrait recopier le code dans un autre fichier, remplacer les «string» par le type voulu et recompiler le tout. On se rend compte que cette pratique prendra du temps et peut être source d'erreur. C'est la raison pour laquelle on introduit les patrons de classe. Ceux-ci permettront de rendre une classe réutilisable. En effet, la classe sera maintenant en mesure de contenir n'importe quel type de donnée.

Pour pouvoir utiliser la généricité on a rajouté dans le fichier ***.cc*** *aux classes et leurs méthodes template <typename T>.*
et dans le ***.h*** *on a rajouter qu'aux classe le template <typename T>.*

Les ressources :

figure 1 :https://chgi.developpez.com/dblist/liste.gif
figure2 :
http://sdz.tdct.org/sdz/medias/uploads.siteduzero.com_files_99001_100000_99463.png
figure 3 :
http://sdz.tdct.org/sdz/medias/uploads.siteduzero.com_files_99001_100000_99520.png