INF3105 – Listes chaînées

Florent Avellaneda

Université du Québec à Montréal (UQAM)

Automne 2021

UQÀM

Sommaire

Rappel sur les tableaux

Avantages

- Accès aléatoire (indicé, direct, « random ») en temps O(1).
- Espace mémoire efficace quand la taille est connue à l'avance.

Limites et inconvénients

- Insertion ($\neq$ ajout à la fin) et enlèvement (sauf à la fin) en temps O(n) à cause du **déplacement** des éléments.
- Redimensionnement requis lorsque la taille est inconnue.
  - Politique de redimensionnement.
  - Perte de mémoire jusqu'à 50 % -1 éléments.
  - Compromis entre temps et mémoire.

La liste chaînée

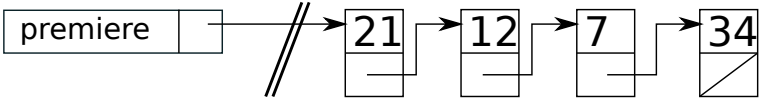
- Structure de données simple et linéaire.
- Plus générale que les piles et files.
- L'accès n'est pas limité à quelques positions, comme au sommet (pile) ou à la tête/queue (file).

Concept de position

- Limite des indices : non constant après une insertion.
- Nécessité d'avoir une abstraction du concept de position.
- Solution temporaire : position = pointeur (adresse mémoire) de cellule (ex : Cellule* position).

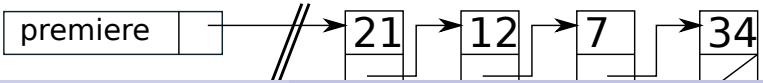
Représentation d'une liste naïve

- Une liste chaînée peut être représentée par un pointeur vers une «chaîne» de cellules.
- Une cellule contient un élément (contenu) et un pointeur vers la cellule suivante.
- La liste est terminée par un pointeur nul (`nullptr`).



Représentation C++ d'une liste naïve

```
template <class T>
class Liste{
  public:
    Liste();
    ~Liste();
    void vider();
    void insererDebut(const T& e);
    struct Cellule{ // temporairement public, sera private...
       Cellule(const T& c, Cellule* s) : suivante(s){contenu=c;}
       T contenu;
       Cellule* suivante;
    };
    void inserer(const T& e, Cellule* c);
  private:
    Cellule*  premiere;
};
```



Constructeur

```
template <class T>
Liste<T>::Liste()
 : premiere(nullptr)
{

}
```

Destructeur

```
template <class T>
Liste<T>::~Liste()
{
    vider();
}
```

Insertion au début

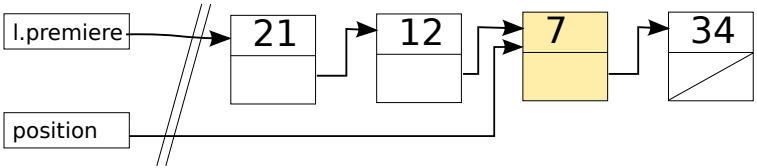
```
template <class T>
void Liste<T>::inserer_debut(const T& element){
    premiere = new Cellule(element, premiere);
}
```

Si on veut retourner la position de la nouvelle cellule...

```
template <class T>
Liste<T>::Cellule* Liste<T>::inserer_debut(const T& element){
    premiere = new Cellule(element, premiere);
    return premiere;
}
```

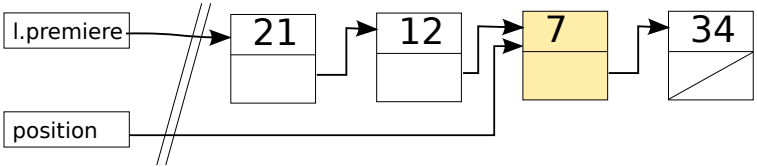
Concept de position

- Dans un tableau, une «position» est un indice représenté par un entier (ex. : `int i`).
- Problème dans une liste chaînée : les indices ne sont pas constants.
- Solution : une position peut être représentée par un pointeur vers la cellule contenant l'élément ciblé (ex. : `Liste<T>::Cellule i`).

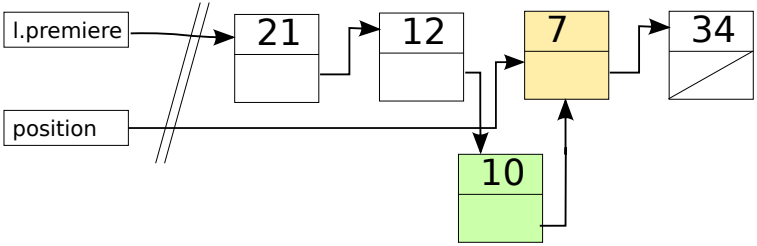


Insertion devant une cellule : Stratégie #1

Avant l'insertion :



Après l'insertion :



Insertion devant une cellule : Stratégie #1

Première stratégie

- Création d'une nouvelle cellule devant la cellule pointée.

```
template <class T>
void Liste<T>::inserer(Cellule* position, const T& element){
   Cellule* nouvellecellule = new Cellule(element, position);
   Cellule* c = premiere;
   while(c->suivante!=position)
      c = c->suivante;
   c->suivante = nouvellecellule;
}
```

Insertion devant une cellule : Stratégie #1

Première stratégie

- Création d'une nouvelle cellule devant la cellule pointée.

```
template <class T>
void Liste<T>::inserer(Cellule* position, const T& element){
   Cellule* nouvellecellule = new Cellule(element, position);
   Cellule* c = premiere;
   while(c->suivante!=position)
      c = c->suivante;
   c->suivante = nouvellecellule;
}
```

Problème

- Insertion en O(n).
- Pourquoi : parce qu'on ne peut pas remonter les pointeurs !
- La boucle while est inévitable avec cette représentation.

Insertion devant une cellule : Stratégie #1

Si on veut retourner la position de la nouvelle cellule...

```
template <class T>
Liste<T>::Cellule* Liste<T>::inserer(Cellule* position, const T& element){
   Cellule* nouvellecellule = new Cellule(element, position);
   Cellule* c = premiere;
   while(c->suivante!=position)
      c = c->suivante;
   c->suivante = nouvellecellule;
   return nouvellecellule;
}
```

Insertion devant une cellule : Stratégie #2

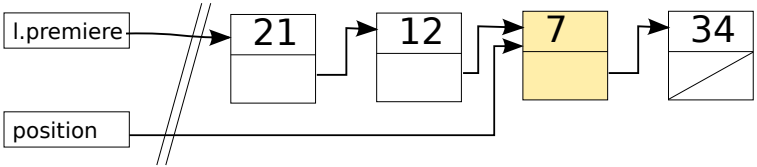
Deuxième stratégie

Créer une cellule et déplacer la valeur de la cellule pointée.

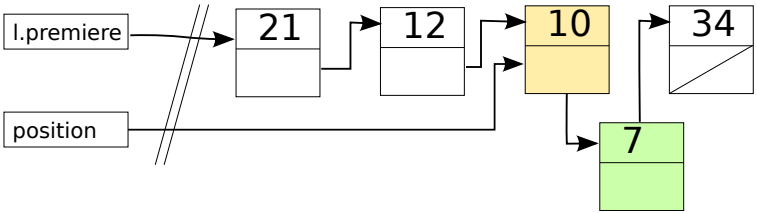
```
template <class T>
void Liste<T>::inserer(Cellule* position, const T& element){
   Cellule* nouvellecellule = new Cellule(position->contenu, position->suivante);
   position->suivante = nouvellecellule;
   position->contenu = element;
}
```

Insertion devant une cellule : Stratégie #2

Avant l'insertion :



Après l'insertion :



Problèmes ?

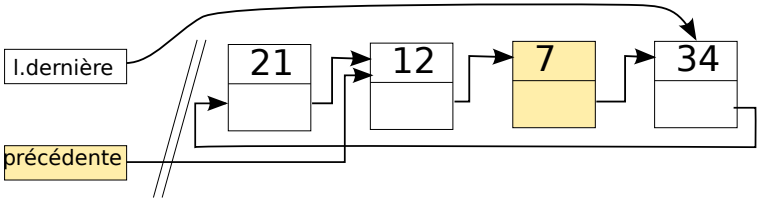
- Le problème d'insertion en O(n) est résolu.
- Insertion maintenant en O(1).
- Cependant, où pointe position ?
- Position ne pointe plus vers le même élément !
- Il faut tout de même déplacer un élément. S'il est petit, c'est ok. Mais, s'il est gros, pourrions-nous éviter cela ?
- Et l'enlèvement ?

Exercice : enlèvement d'une cellule

```
template <class T>
void Liste<T>::enlever(Cellule* position)
{




}
```

Décalage des pointeurs (vers cell. précédente)

- Pour définir une position, on pointe vers la cellule précédente.
- La dernière cellule pointe vers la première.
- Ci-dessous, l'objet «précédente» désigne la position de la cellule contenant 7.



Représentation en C++

Fichier d'entête partiel

```
template <class T> class Liste{
  public:
    Liste();
    ~Liste();
    void vider();
    struct Cellule{ // temporairement public, sera éventuellement private...
        Cellule(const T& c, Cellule* s) : suivante(s){contenu=c;}
        T contenu;
        Cellule* suivante;
    };
    void inserer(const T& e, Cellule* c);
  private:
    Cellule*  derniere;
};
```

Insertion

```
template <class T>
Cellule* Liste<T>::inserer(const T& element, Cellule* position){
 if(derniere==nullptr){
  derniere = new Cellule(element);
  position = derniere->suivante = derniere;
 }else
  if(position==nullptr){
   position=derniere;
   derniere->suivante = new Cellule(element, derniere->suivante);
   derniere = derniere->suivante;
  }else
   position->suivante = new Cellule(element, position->suivante);
 return position;
}
```

Enlèvement

```
template <class T>
Cellule* Liste<T>::enlever(Cellule* position){
  assert(position!=nullptr && derniere!=nullptr);
  Cellule* temp = position->suivante;
  position->suivante = temp->suivante;
  delete temp;
  Cellule* retour = temp==derniere ? nullptr : position;
  if(derniere==temp) derniere = position;
  if(temp==position)
    derniere = position = nullptr;
  return retour;
}
```

Itérer sur une liste chaînée (rep. naïve)

```
template <class T> class Liste{
  public:
    ...
    Cellule* premiere;
};
// ...
int main(){
   Liste<int> liste;
   for(int i=0;i<5;i++)
      liste.inserer_debut(i);
   for(Liste<int>::Cellule* c=liste.debut();c!=nullptr;c=c->suivante)
      cout << c->contenu << endl;
   return 0;
```

Pourquoi faut-il «cacher» les pointeurs ?

- Rappel : position représentée par pointeur de cellule.
- Donner un accès public aux pointeurs = mauvaise idée !

```
template <class T> class Liste{
 public:
  // ...
  Cellule* inserer(const T& e, Cellule* c);
  // ...
};
// ...
int main(){
  Liste<int> liste;
  Liste<int>::Cellule* c3 = liste.inserer(3, nullptr);
  Liste<int>::Cellule* c7 = liste.inserer(7, nullptr);
  Liste<int>::Cellule* c5 = liste.inserer(5, nullptr);
  delete c3; // Cela devrait être interdit.
  c5->contenu = 3; // Idem.
  return 0;
}
```

Itérateurs : encapsulation des pointeurs

- Solution : **encapsuler** la position (pointeur vers la cellule précédente) dans un objet de type **itérateur**.

```
template <class T> class Liste{
  class Iterateur{
    private:
      Cellule* precedente;
  };
};
```

- Fonctions d'un itérateur :
  - avancer dans la liste ;
  - reculer (si doublement chaînée) ;
  - accéder au contenu d'une cellule ;
  - tester si rendu à la fin.

Exemple d'utilisation d'une liste simplement chaînée

```
//...
int main(){
    Liste<int> liste;
    liste.inserer_debut(2);
    liste.inserer_debut(1);
    liste.inserer_fin(5);
    Liste<int>::Iterateur iter5 = liste.trouver(5);
    liste.inserer(3, iter5);
    liste.inserer(4, iter5);
    for(Liste<int>::Iterateur iter=liste.debut();iter;iter++)
        cout << liste[iter] << endl;
    return 0;
}
```

liste.h (1)

```
template <class T> class Liste{
  public:
    class Iterateur;
    Liste();
    ~Liste();

    bool estVide() const;
    void vider();

    const Liste& operator = (const Liste&);

    T& operator[] (const Iterateur&);
    const T& operator[] (const Iterateur&) const;

    Iterateur inserer(const T&, const Iterateur&);
    Iterateur enlever(const Iterateur&);

    Iterateur inserer_debut(const T&);
    Iterateur inserer_fin(const T&);

    Iterateur debut() const;
    Iterateur fin() const;

    Iterateur trouver(const T&) const;
[...]
```

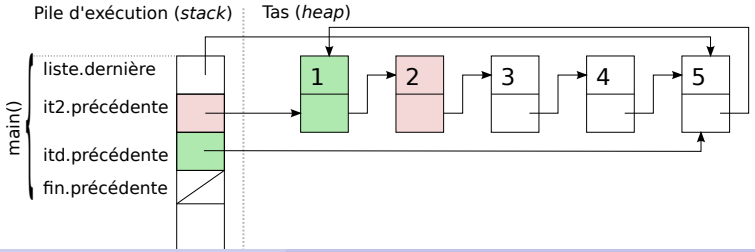
liste.h (2)

```
template <class T> class Liste{
[...]
  private:
    struct Cellule{
      Cellule(const T& c, Cellule* s=nullptr) : suivante(s) { contenu=c; }
      T contenu;
      Cellule* suivante;
    };
    Cellule* derniere;
[...]
```

liste.h (3)

```
template <class T> class Liste{
 //...
  public:
    class Iterateur{
      public:
        Iterateur(const Iterateur&);
        Iterateur(const Liste&);
        Iterateur& operator=(int);

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

        Iterateur operator++(int);
        Iterateur& operator++();

        //T& operator*(); // Bonne idee? Pourquoi? Qu'arrive-t-il si la liste est constante?
        const T& operator*() const;

        Iterateur operator+(int) const;
        Iterateur& operator+=(int);
        Iterateur& operator = (const Iterateur&);
      private:
        Cellule* precedente;
        const Liste& liste;
      friend class Liste;
    };
};
```

```
int main(){
  Liste<int> liste;
  for(int i=1;i<=5;i++)
    liste.inserer_fin(i);
  Liste<int>::Iterateur it2 = liste.trouver(2);
  Liste<int>::Iterateur itd = liste.debut();
  Liste<int>::Iterateur fin = liste.fin();
}
```



Liste doublement chaînée