INF3105 – Listes chaînées

Jaël Champagne Gareau

Université du Québec à Montréal (UQAM)

Été 2024

http://cria2.uqam.ca/INF3105/

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

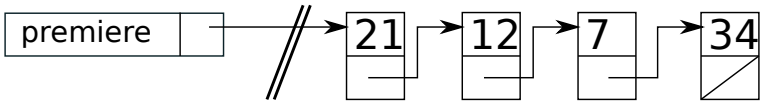
- Structure de données simple et linéaire.
- Plus générale que les piles et files.
- L'accès n'est pas limité à quelques positions, comme au sommet (pile) ou à la tête/queue (file).

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

```
1  template <class T>
2  class Liste {
3    public:
4      Liste();
5      ~Liste();
6      void vider();
7      struct Cellule { // temporairement public, sera private...
8          Cellule(const T& c, Cellule* s) : suivante(s) {contenu = c;}
9          T contenu;
10         Cellule* suivante;
11     };
12     void insererDebut(const T& e);
13     void inserer(const T& e, Cellule* c);
14   private:
15     Cellule* premiere;
16 };
```

Constructeur et destructeur

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

```
1 template <class T>
2 Liste<T>::~Liste() {
3     vider();
4 }
```

Insertion au début

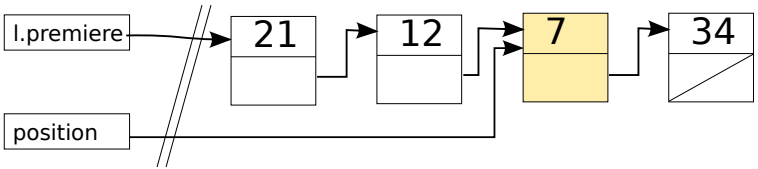
```
1  template <class T>
2  void Liste<T>::inserer_debut(const T& element) {
3      premiere = new Cellule(element, premiere);
4  }
```

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

```
1  template <class T>
2  Liste<T>::Cellule* Liste<T>::inserer_debut(const T& element) {
3      premiere = new Cellule(element, premiere);
4      return premiere;
5  }
```

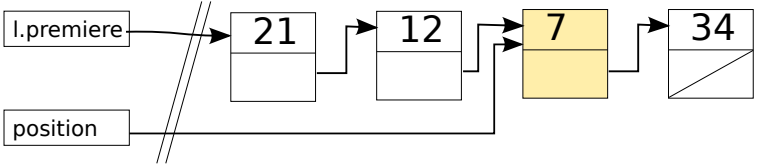
Concept de position

- 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`).

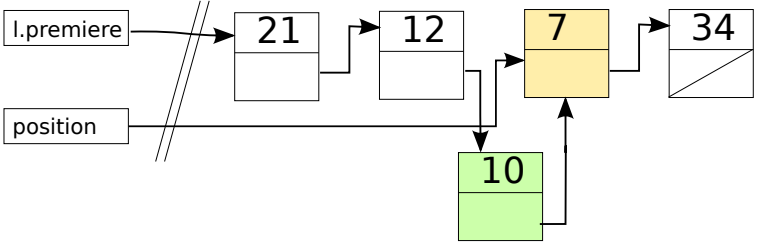


Insertion devant une cellule : Stratégie #1

Avant l'insertion :



Après l'insertion :



Insertion devant une cellule : Stratégie #1

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

```
1  template <class T>
2  void Liste<T>::inserer(Cellule* position, const T& element) {
3      Cellule* nouvCellule = new Cellule(element, position);
4      Cellule* c = premiere;
5      while(c->suivante != position) c = c->suivante;
6      c->suivante = nouvCellule;
7  }
```

Insertion devant une cellule : Stratégie #1

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

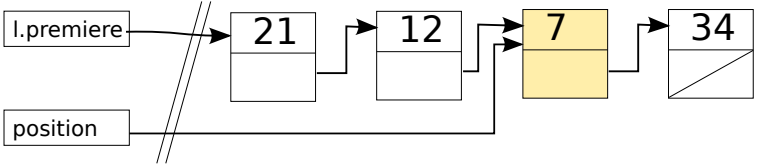
```
1  template <class T>
2  void Liste<T>::inserer(Cellule* position, const T& element) {
3      Cellule* nouvCellule = new Cellule(element, position);
4      Cellule* c = premiere;
5      while(c->suivante != position) c = c->suivante;
6      c->suivante = nouvCellule;
7  }
```

Problème

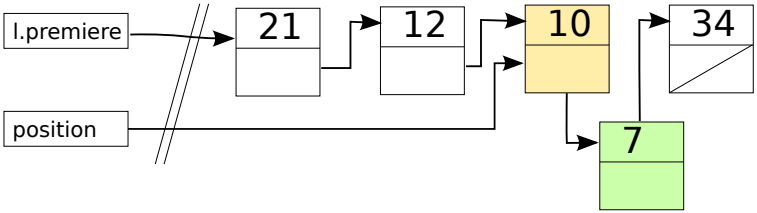
- Insertion en O(n).
- Pourquoi : parce qu'on ne peut pas remonter les pointeurs !
- La boucle while est inévitable avec cette représentation.

Insertion devant une cellule : Stratégie #2

Avant l'insertion :



Après l'insertion :



Insertion devant une cellule : Stratégie #2

Deuxième stratégie

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

```
1  template <class T>
2  void Liste<T>::inserer(Cellule* position, const T& element) {
3      Cellule* nouvCellule
4          = new Cellule(position->contenu, position->suivante);
5      position->suivante = nouvCellule;
6      position->contenu = element;
7  }
```

Problèmes ?

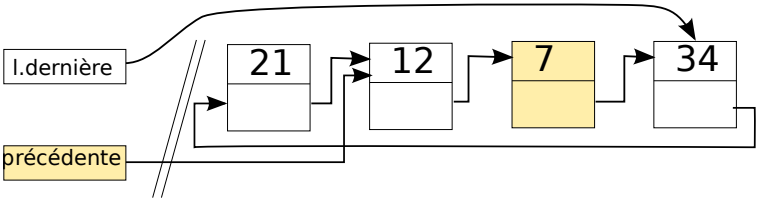
- Le problème d'insertion en O(n) est résolu.
- Insertion maintenant en O(1).
- Cependant, où pointe position ?
- Position ne pointe plus vers le même élément !
- Il faut tout de même déplacer un élément. S'il est petit, c'est ok. Mais, s'il est gros, pourrions-nous éviter cela ?
- Et l'enlèvement ?

Exercice : enlèvement d'une cellule

```
1  template <class T>
2  void Liste<T>::enlever(Cellule* position) {
3
4
5
6
7
8
9
10
11
12 }
```

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++

```
1  template <class T> class Liste {
2    public:
3      Liste();
4      ~Liste();
5      void vider();
6      struct Cellule { // temporairement public, sera private...
7          Cellule(const T& c, Cellule* s) : suivante(s) {contenu = c;}
8          T contenu;
9          Cellule* suivante;
10     };
11     void inserer(const T& e, Cellule* pos); // position précédente à celle d'intérêt
12   private:
13     Cellule* derniere;
14 };
```

Insertion

```
1  template <class T>
2  Cellule* Liste<T>::inserer(const T& element, Cellule* position) {
3      if (derniere == nullptr) {
4          derniere = new Cellule(element);
5          position = derniere->suivante = derniere;
6      } else if (position == nullptr) { // insertion à la fin
7          position = derniere;
8          derniere->suivante = new Cellule(element, derniere->suivante);
9          derniere = derniere->suivante;
10     } else
11         position->suivante = new Cellule(element, position->suivante);
12     return position;
13 }
```

Enlèvement

```
1 template <class T>
2 void Liste<T>::enlever(Cellule* position) {
3     assert(position != nullptr && derniere != nullptr);
4     Cellule* temp = position->suivante;
5     position->suivante = temp->suivante;
6     delete temp;
7     if(derniere == temp) derniere = position;
8     if(temp == position) derniere = position = nullptr;
9 }
```

Itérer sur une liste chaînée (rep. naïve)

```
1  template <class T> class Liste {
2    public: ...
3    private:
4      Cellule* premiere;
5  };
6  int main() {
7     Liste<int> liste;
8     for(int i = 0; i < 5; i++)
9        liste.inserer_debut(i);
10    for(Liste<int>::Cellule* c = liste.debut(); c != nullptr; c = c->suivante)
11       cout << c->contenu << endl;
12    return 0;
13 }
```

Pourquoi faut-il «cacher» les pointeurs ?

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

```
1  template <class T>
2  class Liste {
3    public: ...
4      Cellule* inserer(const T& e, Cellule* c);
5    private: ...
6  };
7  int main() {
8      Liste<int> liste;
9      Liste<int>::Cellule* c3 = liste.inserer(3, nullptr);
10     Liste<int>::Cellule* c7 = liste.inserer(7, nullptr);
11     Liste<int>::Cellule* c5 = liste.inserer(5, nullptr);
12     delete c3; // Cela devrait être interdit.
13     c5->contenu = 3; // Idem.
14 }
```

Itérateurs : encapsulation des pointeurs

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

```
1  template <class T>
2  class Liste {
3    public:
4      class Iterateur { Cellule* precedente; };
5    private: ...
6  };
```

- 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

```
1  int main() {
2      Liste<int> liste;
3      liste.inserer_debut(2);
4      liste.inserer_debut(1);
5      liste.inserer_fin(5);
6      Liste<int>::Iterateur iter5 = liste.trouver(5);
7      liste.inserer(3, iter5);
8      liste.inserer(4, iter5);
9      for(Liste<int>::Iterateur iter = liste.debut(); iter; iter++)
10         cout << liste[iter] << endl;
11 }
```

liste.h (1)

```
1  template <class T> class Liste {
2    public:
3      class Iterateur;
4      Liste(); ~Liste();
5      bool estVide() const;
6      void vider();
7      const Liste& operator = (const Liste&);
8      T& operator[] (const Iterateur&);
9      const T& operator[] (const Iterateur&) const;
10     Iterateur inserer(const T&, const Iterateur&);
11     Iterateur enlever(const Iterateur&);
12     Iterateur inserer_debut(const T&);
13     Iterateur inserer_fin(const T&);
14     Iterateur debut() const;
15     Iterateur fin() const;
16     Iterateur trouver(const T&) const;
17   [...]
18   }
```

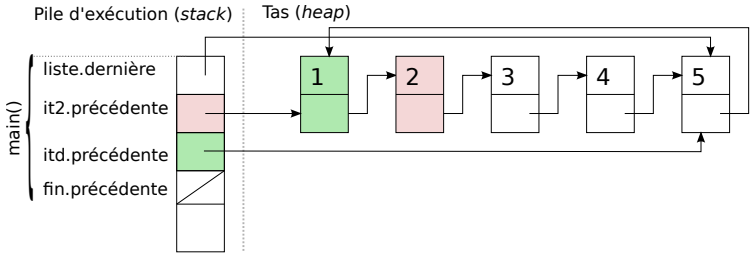
liste.h (2)

```
1  template <class T> class Liste {
2  [...]
3    private:
4      struct Cellule {
5          Cellule(const T& c, Cellule* s = nullptr) : suivante(s) { contenu = c; }
6          T contenu;
7          Cellule* suivante;
8      };
9      Cellule* derniere;
10 [...]
11 }
```

liste.h (3)

```
template <class T> class Liste {
[..]
  public:
    class Iterateur {
      public:
        Iterateur(const Iterateur&);
        Iterateur(const Liste&);
        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;
    };
};
```

```
1  int main() {
2      Liste<int> liste;
3      for(int i = 1; i <= 5; i++)
4          liste.inserer_fin(i);
5      Liste<int>::Iterateur it2 = liste.trouver(2);
6      Liste<int>::Iterateur itd = liste.debut();
7      Liste<int>::Iterateur fin = liste.fin();
8  }
```



Liste doublement chaînée