Algorithmes et structures de données
GLO-2100/IFT-2008

Mondher Bouden

Les structures de données de base

Édition ÉTÉ 2018

Département d'informatique
et de génie logiciel

Introduction

- L'objectif de ce module est de vous présenter les structures de données de base couramment utilisées en informatique (tels que les vecteurs, listes, files, piles) et qui servent également de support à d'autres structures de données plus complexes que nous verrons plus loin dans le cours (tels que les arbres, graphes, tables de dispersions).

Plan

- Introduction et les types de données abstraites (TDA)
- Les vecteurs
- Les listes
- Les files
- Les piles

Les types de données abstraites (TDA)

- Un type de données abstraites (TDA) est, en principe, un ensemble d'objets muni d'un ensemble d'opérations.
  - Sans mentionner comment ces opérations sont implémentés.

- Les vecteurs, listes, piles, files, arbres etc sont des exemples de TDA.

- Dans ce cours, nous verrons comment construire en C++ des TDA **en masquant les détails d'implémentation aux utilisateurs.**
  - **Seul l'interface et les spécifications sont visibles.**
  - Chaque TDA sera constitué d'une classe (ou un patron de classe) qui sera, possiblement, constituée de sous classes.
  - Cette construction constituera une version plausible (mais simplifiée) des structures de données (conteneurs pour données) offertes par la **Standard Template Library (STL).**

Interface, implémentation et spécifications

- L'**interface** du TDA est définie par le prototype de ses méthodes publiques
  - Se retrouve donc dans la section publique du fichier .h de la classe

- L'**implémentation** est définie par la définition (en C++) de toutes ses méthodes et l'ensemble de ses attributs (membres). Cela est caché à l'utilisateur.

- Les **spécifications** sont les descriptions (en langage courant) des fonctionnalités des méthodes, de leurs pré-conditions et post-conditions, des exceptions lancées et des ressources qu'elles utilisent.
  - Elles devraient donc donner l'ordre de croissance du temps d'exécution des méthodes en fonction de la taille de l'instance à traiter
  - Elles devraient également fournir l'info sur l'utilisation d'autre ressources (comme la mémoire utilisée) lorsque cela est pertinent
  - Elles sont habituellement données dans la documentation qui est générée (à l'aide d'un outil comme Doxygen) par les commentaires se trouvant dans l'implémentation (fichiers .cpp et .hpp) et l'interface.
    - En principe, seule cette documentation est fournie à l'utilisateur.

Le TDA vecteur

- Une **vecteur** est une **séquence ordonnée d'éléments** d'un certain type.
  - L'ordre signifie que chaque élément possède une position dans le vecteur: il y a un 1er élément, un 2^{e}, etc...

- Opérations devant être supportées:
  - Accès (direct) au k-ième élément
  - Suppression du k-ième élément
  - Insertion d'un élément avant celui situé à la position k

- Spécifications souhaitées :
  - L'accès au k-ième élément se fait en temps $O(1)$
  - Les 2 autres opérations se font en temps $O(n)$
    - où n = nombre d'éléments stockés dans le vecteur

Utilité du TDA vecteur

- Vient du fait que l'accès au k-ième élément se fait en $O(1)$

- C'est très rapide en comparaison du temps d'insertion/suppression qui sont sont en $O(n)$

- Les vecteurs sont implémentés à l'aide de tableaux dynamiques
  - Les éléments occupent donc des cases contigües en mémoire
  - Le temps d'accès a k-ième élément est donc constant
  - Mais pour insérer ou supprimer un élément au milieu du tableau, il faudra déplacer la moitié des éléments puisque ceux-ci doivent occuper des positions contigües en mémoire: ce qui nécessite un temps en $\Theta(n)$.

- C'est ce qui est fait pour le conteneur vector de la STL.
- Ici, voyons comment utiliser ce conteneur.

Conteneur vector de la STL

- **#include <vector> //directive pour avoir accès au conteneur**
- Ce conteneur utilise les tableaux dynamiques pour le stockage des éléments dans un espace mémoire contigüe. Mais, contrairement aux tableaux, il est possible de modifier dynamiquement la taille.
  - Cela implique une réallocation de la mémoire, mais pour éviter que cela soit fait trop souvent, un vector a généralement plus d'espace mémoire de disponible que celui occupé par les éléments stockés. Donc on a:
    - accès aléatoire par index en O(1)
    - insertions et retraits lents au milieu et au début (en O(n))
    - suppressions à la fin en O(1)
    - insertions à la fin en O(1) lorsque cela ne requiert pas une ré-allocation du tableau
- **Constructeurs**

  vector(); //construit un vecteur de zéro élément

  vector(size_type n); //construit un vecteur de n éléments

  vector(size_type n, const T& a); //chaque élément est une copie de a
- **Opérateur indexation**

  T& operator[](size_type); //retourne un élément de type T modifiable

  const T& operator[](size_type) const; //non modifiable par l'appelant

std::vector

- Les **méthodes** les plus utilisées:

```
void clear(); //détruit tous les éléments
size_type size(); //donne le nombre d'éléments présents
void resize( size_type n ); //change la taille (nb d'éléments) à n

size_type capacity(); //donne la taille de l'espace mémoire allouée
void reserve( size_type n ); //demande une capacité >= n

void push_back(const T&); //ajoute un élément à la fin
void pop_back(); //détruit le dernier élément
T& front(); //retourne le premier élément
T& back();  //retourne le dernier élément
T& at ( size_type n ); //retourne l'élément en position n et lance une
                       //exception out_of_range si l'élément n n'existe pas
iterator begin(); //retourne un itérateur au début (voir + loin)
iterator end();  //retourne un itérateur à la fin (voir + loin)
...
```

- Les **opérateurs** les plus utilisées sont : `=` , `==`, `<`, `[ ]`.

std::vector

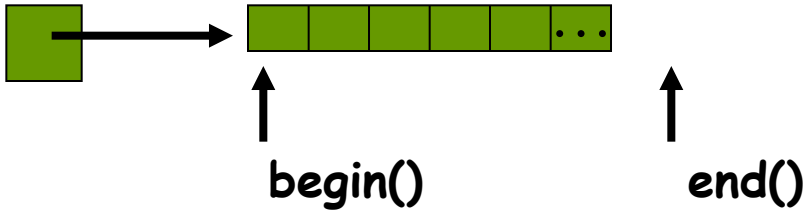
- **Exemple:**

```
using namespace std;
vector<int> x, y(10), z(10,-1);

x.resize(100); //x contient 100 entiers (initialisés par défaut)
for(unsigned int i = 0; i < x.size(); ++i)
   x[i]=i+1;
x.clear(); //maintenant x.size() == 0

for(unsigned int i = 0; i < 100; ++i)
   x.push_back(i+1);
```

Attention: aucun «**bound checking**» dans STL ! …sauf la méthode at() qui lance l'exception *out_of_range* si débordement.

Pour plus d'information: http://www.cplusplus.com

Itérateur



- Chaque conteneur de la STL fourni un type d'**itérateur**. Ils offrent les mêmes fonctions de base permettant aux itérateurs d'accéder séquentiellement aux éléments peu importe leurs types
  - **begin()**: retourne un itérateur pointant sur le 1er élément
  - **end()**: retourne un itérateur pointant sur l'élément passé le dernier

- **Exemple pour <vector>**
  - Le type vector<int> donne un itérateur de type : **vector<int>::iterator**

```
int lSum = 0;
for(vector<int>::iterator i = x.begin(); i != x.end(); ++i)
        lSum += *i; //déréférenciation de l'itérateur donne l'élément pointé
```

- Tous les conteneurs définissent deux types d'itérateur
  - *container*::**iterator** //permet un accès en mode lecture/écriture
  - *container*::**const_iterator** //mode lecture seulement

- Les principaux opérateurs sont * donnant accès à la valeur, ++ et -- pour aller à l'élément suivant et aller à l'élément précédent.

Le TDA liste

- Une **liste** est une **séquence ordonnée d'éléments** d'un certain type.
  - L'ordre signifie que chaque élément possède une position dans la liste: il y a un 1er élément, un 2e, etc...
  - Il y a aussi le concept d'une **position courante** qui est l'endroit où l'on peut insérer ou supprimer un élément.

- Opérations devant être supportées:
  - Suppression d'un élément situé à la position courante
  - Insertion d'un élément avant celui situé à la position courante
  - Accéder à l'élément suivant (opérateur ++)
  - Accéder à l'élément précédent (si la liste est doublement chaînée)

- Spécifications souhaitées :
  - Ces 4 opérations se font en temps $O(1)$

- **Remarque**: absence d'opération «se rendre au nième élément de la liste».
  - Pour se rendre au nième élément, il faut «accéder à l'élément suivant» n-1 fois à partir du 1er élément.

Utilité du TDA liste

- Vient du fait que les insertions et suppressions à la position courante se font en $O(1)$

- C'est très rapide en comparaison du temps d'insertion/suppression dans les vecteurs et les tableaux qui sont en $O(n)$
  - Car pour insérer ou supprimer un élément situé au milieu d'un vecteur il faut déplacer la moitié de tous les éléments puisque ceux-ci doivent occuper des positions contigües en mémoire

- Pour avoir un temps d'insertion/suppression en $O(1)$ il ne faut pas contraindre les éléments à occuper des positions contigües en mémoire
  - Le temps d'accès à une position arbitraire ne pourra donc pas être en $O(1)$ comme c'est le cas pour les vectors et tableaux
  - C'est possiblement pour cela que le conteneur **list** de la STL ne surcharge pas l'opérateur indexation []

Comparaison entre liste et vecteur

- Pour liste:
  - Temps d'insertion/suppression (à la position courante) en $O(1)$
  - temps d'accès à une position arbitraire en $O(n)$

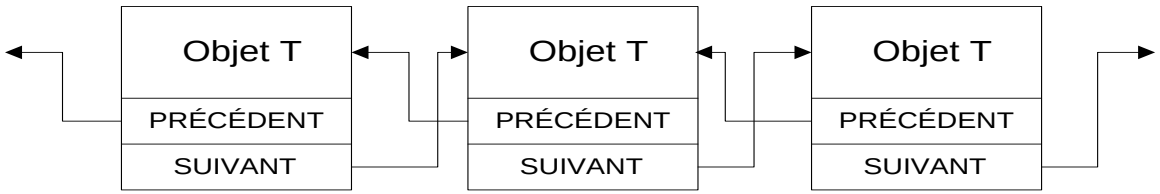
- Pour vecteur:
  - Temps d'accès à une position arbiraire en $O(1)$
  - Temps de d'insertion/suppression en $O(n)$

- Donc on utilisera préférablement une liste si la majorité des opérations sont des suppressions/insertion (à la position courante).
- Dons le cas contraire, on utilisera préférablement un vecteur

- Une liste est une structure de données de base qui est utilisée pour construire d'autres structures comme les files, les piles et plusieurs autres…

Classe List

- Nous présentons une implémentation plausible (mais simplifiée) du conteneur **list** de la STL
  - que nous nommerons **List** (pour éviter toute ambiguïté)
  - Il s'agit d'une liste doublement chaînée
  - La liste simplement chaînée est un cas plus simple
  - Adapté de Mark Allen Weiss, Data structures and Algorithm Analysis in C++, 4e édition, 2014, Pearson.
- Une liste est une chaîne constituée de **nœuds**
- Chaque **nœud**, contient trois parties :
  1. une partie contenant l'**élément** stocké
  2. une partie contenant un **pointeur** pointant au nœud suivant.
  3. une partie contenant un **pointeur** pointant au nœud précédent



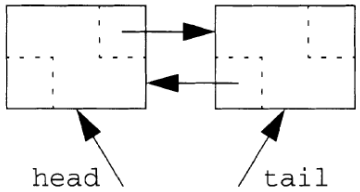
Une classe interne de List pour les noeuds

```
template <typename Object>
class List
{
public:
    //….
private:
    struct Node
    {
        Object data; //la donnée stockée
        Node * prev; //adresse du nœud précédent
        Node * next; //adresse du nœud suivant

        Node( const Object & d = Object( ), Node *p = 0, Node *n = 0 )
          : data( d ), prev( p ), next( n ) { } //constructeur
    };
    int   theSize; //nb d'éléments contenus dans la liste
    Node * head; //pointeur au nœud du début de la liste
    Node * tail;  //pointeur au nœud de fin de la liste
   //..
};
```

Les noeuds

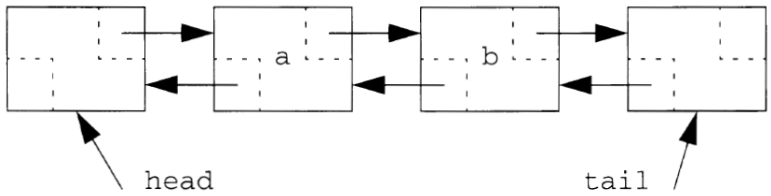
- **La classe Node est une classe interne et privée de List**
  - Seules les méthodes de List peuvent manipuler les objets Node
- On a facilement accès au nœud suivant et au nœud précédent
  - en utilisant les pointeurs next et prev de Node.
- **Les nœuds de tête et de fin servent de sentinelle**
  - Il n'y a pas de données de l'utilisateur stockée dans ces nœuds

Liste vide:



[source Weiss: DATA STRUCTURES AND ALGORITHM ANALYSIS IN C++, 4th ed., Pearson 1014.]

Liste contenant 2 éléments:

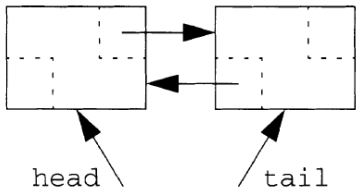


Utilité des sentinelles

- **Simplifie les méthodes d'insertion et de suppression**
  - Cela enlève la nécessité de mettre à jour les pointeurs head et tail de List pour ces opérations car il suffit de mettre à jour les pointeurs next et prev des nœud pertinents peu importe où se fait l'opération
  - Il n'y a donc pas de cas particulier à traiter
- Comme tous les conteneurs de la STL, nous utiliserons un **iterator** pour se déplacer dans la liste.
  - L'itérateur retourné par begin() devra donc pointer sur le premier élément contenant une donnée
  - L'itérateur retourné par end() devra alors pointé sur le noeud sentinelle à la fin (un élément passé le dernier)

Constructeurs et destructeur de List

```
template <typename Object>
class List
{
public:
      List() { init(); }
      List(const List & rhs) {  //constructeur de copie
         init();
         *this = rhs;  } //utilise la surcharge de l'operateur = (définie plus loin)
      ~List() {
         clear(); //methode à définir: efface tous les nœuds non sentinelle
         delete head;
         delete tail;  }
      //…
private:
     //…
    void init()
    {  //création des sentinelles
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail;
        tail->prev = head;
        head->prev = tail->next = 0;
    }
};
```



La classe interne iterator

- Les objets de type **iterator** sont utilisés pour se déplacer par itération le long de la liste et accéder aux éléments
  - On surchargera l'opérateur * pour «déréférencer» un itérateur et obtenir l'élément «pointé»
  - Surcharge des opérateurs ++ et -- pour se déplacer
  - Surcharge de == et != pour comparer les itérateurs

- Cette classe interne est un **membre publique de List**
  - Pour permettre à l'utilisateur de List d'obtenir un itérateur afin de parcourir la liste
    - à l'aide des méthodes begin() et end() de List

iterator

```
class iterator
{
public:
    iterator() : current(0) {}; //constructeur sans paramètre

     Object & operator* ( ) //permet à l'appelant de modifier current->data
      { return current->data; }

    iterator & operator++ ( ) //version prefix
      { current = current->next; return *this; }

    iterator & operator-- ( ) //version prefix
      { current = current->prev; return *this; }

    //…(suite page suivante)

private: //on ne veut pas permettre aux utilisateurs de List d'accéder à ça
    Node* current;
    iterator(Node* p) : current(p) { } //constructeur appelé par des méthodes de List
    friend class List<Object>; //rend accessible cette section aux membres de List
};
```

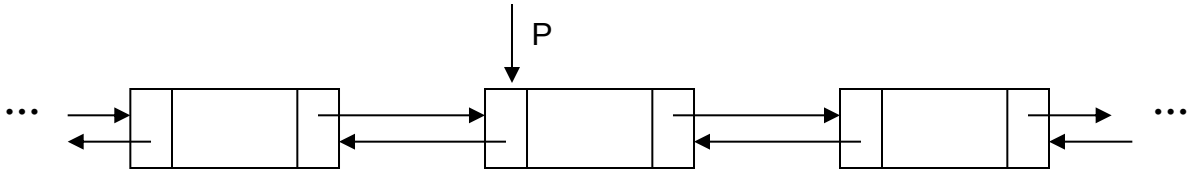
iterator (suite)

```
class iterator
{
public:
     //…
    iterator operator++ (int) { //version postfix
       iterator old = *this;
       ++(*this);  //version prefix de ++ utilisée ici
       return old;
    }
    iterator operator-- (int) { //version postfix
       iterator old = *this;
       --(*this);  //version prefix de -- utilisée ici
       return old;
    }
    bool operator==(const iterator & rhs) const
       { return current == rhs.current; }

    bool operator!=(const iterator & rhs) const
       { return current != rhs.current; }

private:
     //…
};
```

Utilisation des itérateurs pour autres méthodes de List

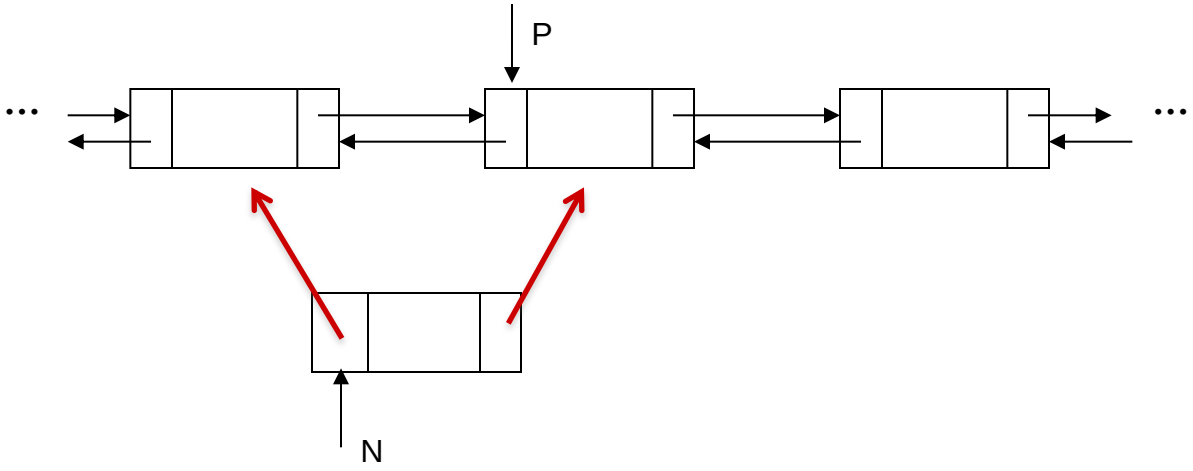
- Les itérateurs nous permettent d'écrire les autres méthodes de List:
  - La surcharge de **operator=** (utilisée par le constructeur de copie)
  - La méthode **clear()** utilisée par le destructeur
  - La méthode **erase(const iterator & itr)** pour détruire le nœud pointé par itr
  - La méthode **insert(const iterator & itr, const Object & x)** pour insérer l'objet x à la position précédent le nœud pointé par itr
  - Les méthode **begin()** et **end()** permettant aux utilisateurs de List d'obtenir un itérateur (pour parcourir la liste)
  - etc..

Insertion d'un nouveau nœud entre p et p->prev



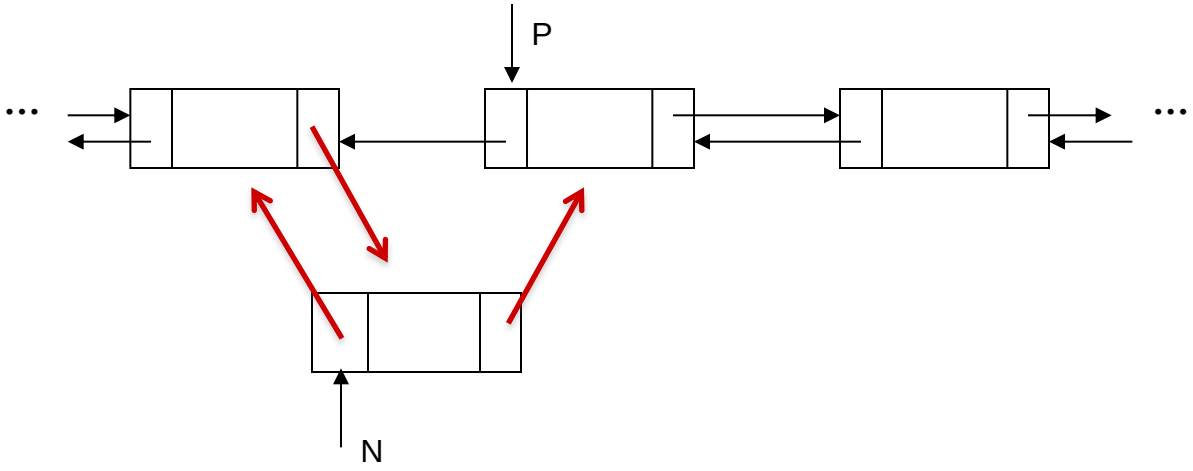
Insertion d'un nouveau nœud entre p et p->prev

```
Node* N = new Node(x, p->prev, p);
```



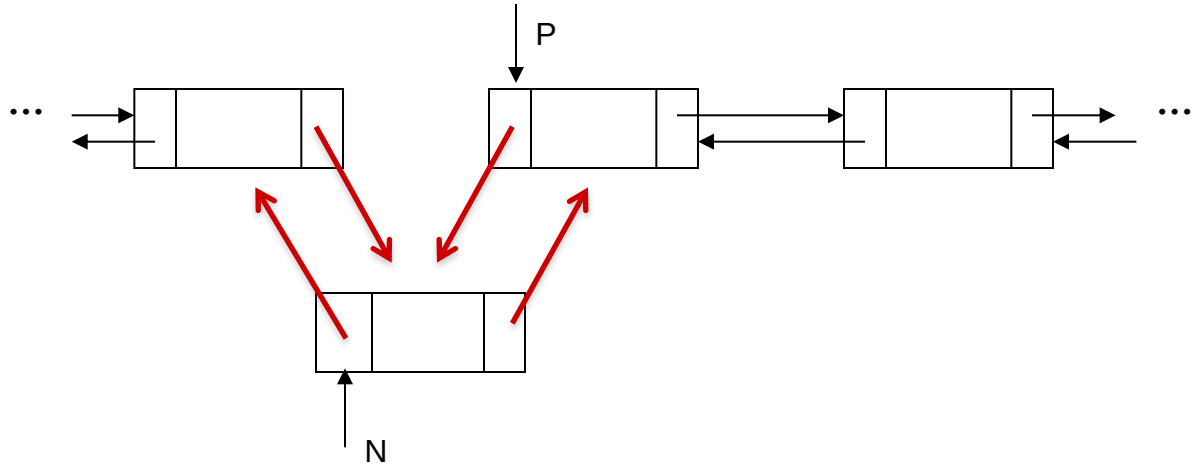
Insertion d'un nouveau nœud entre p et p->prev

```
Node* N = new Node(x, p->prev, p);
p->prev->next = N;
```



Insertion d'un nouveau nœud entre p et p->prev

```
Node* N = new Node(x, p->prev, p);
p->prev->next = N;
p->prev = N;
```



Insertion d'un nouveau nœud entre p et p->prev

```
Node* N = new Node(x, p->prev, p);
p->prev->next = N;
p->prev = N;
```

Est équivalent à:

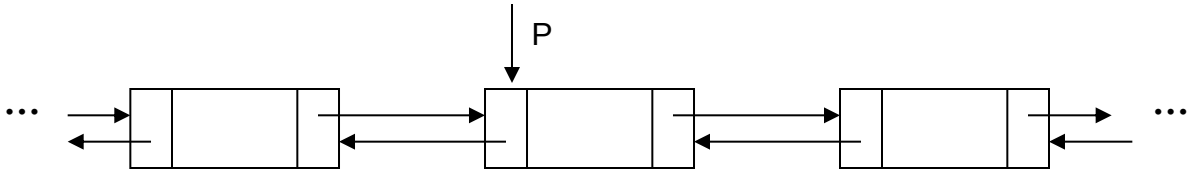
```
Node* N = new Node(x, p->prev, p);
p->prev = p->prev->next = N;
```

Est équivalent à:

```
p->prev = p->prev->next = new Node(x, p->prev, p);
```

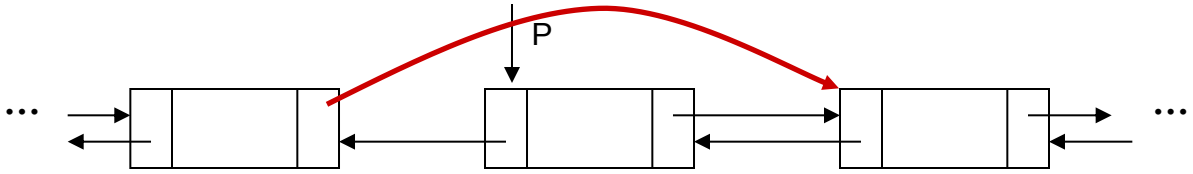
```
//insertion de x avant itr
//l'iterator retourné pointe sur le nœud contenant l'objet inséré
iterator insert( const iterator & itr, const Object & x)
{
   Node* p = itr.current;
   theSize++;
   return iterator( p->prev = p->prev->next = new Node(x, p->prev, p) );
}
```

Suppression nœud pointé par p



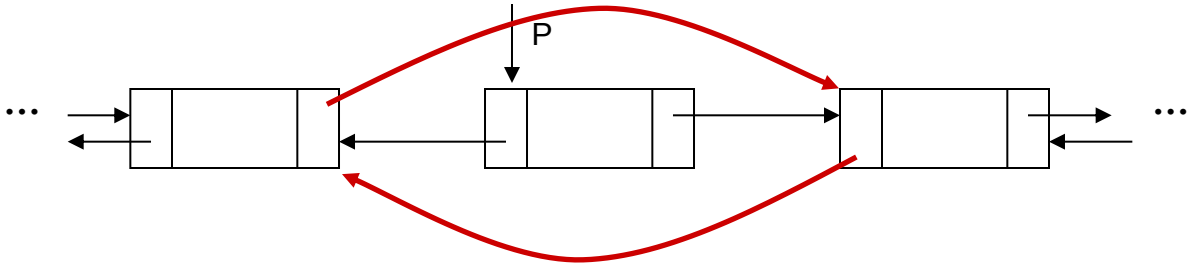
Suppression nœud pointé par p

```
p->prev->next = p->next;
```



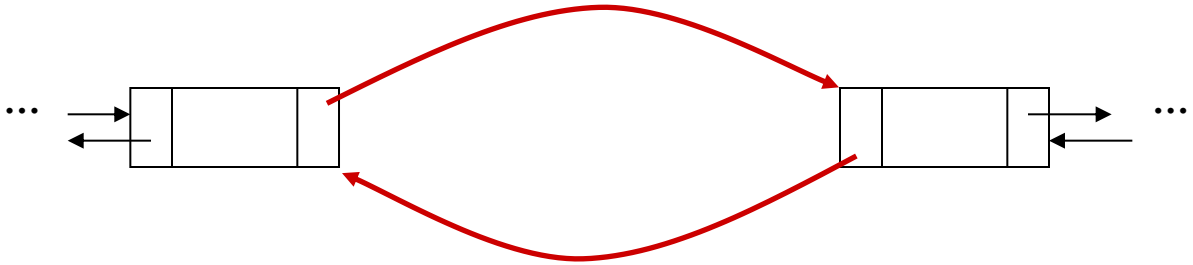
Suppression nœud pointé par p

```
p->prev->next = p->next;
p->next->prev = p->prev;
```



Suppression nœud pointé par p

```
p->prev->next = p->next;
p->next->prev = p->prev;
delete p;
```



Suppression nœud pointé par p

```
p->prev->next = p->next;
p->next->prev = p->prev;
delete p;
```

```
//suppression du noeud pointé par itr
//l'iterator retourné pointe sur le noeud suivant le noeud supprimé
iterator erase( const iterator & itr)
{
   if (theSize<=0) throw logic_error("La liste est vide");
   Node* p = itr.current;
   iterator retIt(p->next);
   p->prev->next = p->next;
   p->next->prev = p->prev;
   delete p;
   theSize--;
   return retIt;
}
```

Méthodes de List utilisant les itérateurs

```
//donne un itérateur pointant sur le premier nœud contenant un Object
iterator begin() { return iterator(head->next); }

//donne un itérateur pointant sur le nœud sentinelle tail
iterator end() { return iterator(tail); }

Object & front() { return *begin(); } //retourne le 1er élément
Object & back() { return *--end(); } //retourne le dernier élément

//enlève le premier nœud non sentinelle
void pop_front()
{ erase( begin() ); }
//enlève le dernier nœud non sentinelle
void pop_back()
{ erase( --end() ); }

void clear() //détruit tous les nœuds non sentinelle
{
    while( !empty() )
        pop_front();
}

bool empty() const
{ return theSize==0; }
```

Méthodes utilisant les itérateurs

```
//insère x dans le premier nœud non sentinelle
void push_front(const Object & x)
{
    insert(begin(), x);
}
//insère x dans le dernier nœud non sentinelle
void push_back(const Object & x)
{
   insert(end(), x);
}

//surcharge de l'opérateur = pour List
//on doit passer par référence car itr peut modifier rhs…
const List & operator= (List & rhs)
{
    if(this!=&rhs)
    {
        clear();
        for( iterator itr = rhs.begin(); itr != rhs.end(); ++itr )
            push_back(*itr);
    }
    return *this;
}
```

List.h

```
template<typename Object>
class List
{
public:
//classe interne iterator . . .
     List( );
     List(const List & rhs); //constructeur de copie
    ~List( );
     const List & operator= (List & rhs)
    iterator begin( );
    iterator end( );
    bool empty( ) const;
    void clear( );
    Object & front( );
    Object & back( );
    void push_front( const Object & x );
    void push_back( const Object & x );
    void pop_front( );
    void pop_back( );
    iterator insert( const iterator & itr, const Object & x );
    iterator erase( const iterator & itr );
private:
//classe interne Node . . .
    int   theSize;
    Node * head;
    Node * tail;
    void init();
};
```

Remarques

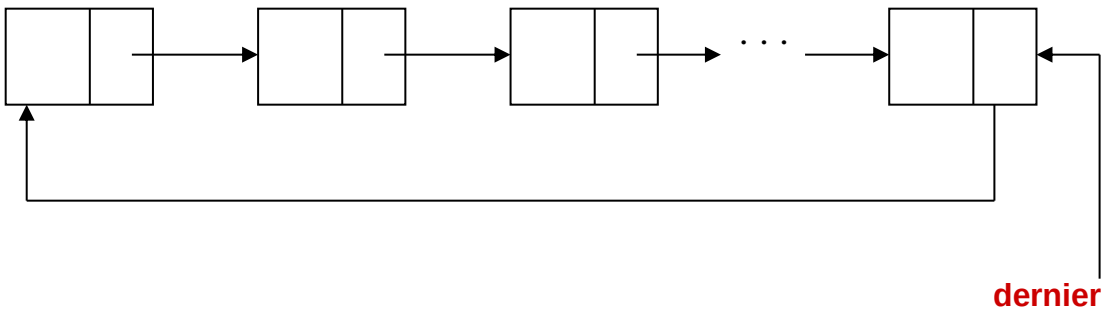
- **List** est une implémentation plausible (mais simplifiée) du conteneur **list** de la STL
  - En plus, **list** fourni un itérateur **const_iterator** qui permet d'accéder aux éléments **en lecture seulement**
  - D'autres méthodes sont fournies. Ces méthodes sont facilement construites à partir des méthodes de **List**
  - Mais on ne fournit pas de méthodes (et l'opérateur [ ]) nous permettant d'accéder au nième élément.
    - Pour cela il faut faire itr = begin() et faire ++itr (n-1) fois.

- Il est possible d'implémenter une liste dans un tableau dynamique.
  - Cela permet alors de surcharger l'operateur **[ ]** et ainsi accéder au nième élément en O(1).
  - Dans ce cas, les insertions et suppressions à la position courante devront se faire en O(n)
  - On ne rencontre donc pas les spécifications souhaitées du TDA liste
    - Bien que l'interface pourrait être identique à List
  - Ce n'est donc pas vraiment une liste

Autres types de liste

- **Liste simplement chainée**:
  - Chaque nœud possède un seul pointeur: sur le nœud suivant
  - On peut donc parcourir la liste seulement dans une seule direction

- **Liste circulaire**:
  - C'est une liste simplement chainé où le pointeur next du dernier nœud pointe sur le premier nœud.
  - La liste n'a ni début ni fin: prendre garde aux boucles infinies!!
  - Doit faire les mises à jour du pointeur dernier lors de la suppression du dernier nœud où l'ajout d'un nœud à cette position



Les TDA pile et file

Structures de données auxiliaires

- car utilisées par d'autres structures de données
- Support aux algorithmes

Autre Utilité :

- modélisation de la réalité
- en informatique : système d'exploitation
  évaluation d 'expressions
  etc.

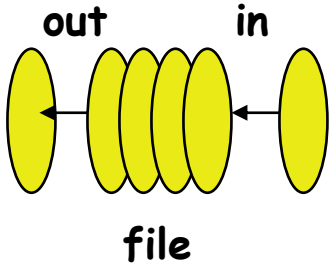
Le TDA file

Files :

FIFO : first in, first out
PAPS : premier arrivé, premier sorti

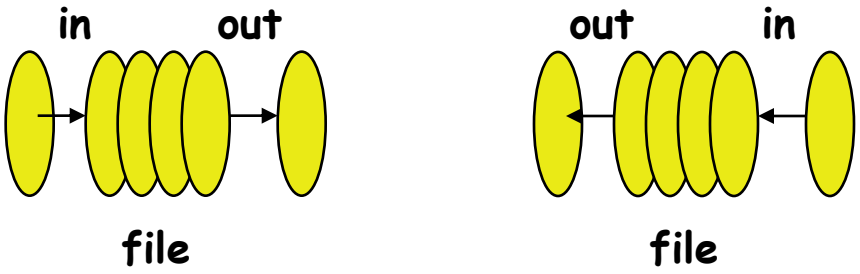
ex. : Files d'attentes pour entrer au cinéma
File d'attente pour imprimante
Processus à exécuter dans un système d'exploitation

Traitement de transactions
ex: réservation de billets d'avion ou de spectacles



File = liste + gestion adaptée

manipulations (enfiler et défiler) par des points d'accès opposés



Le TDA file

Sert à permettre la mise en attente d'informations dans le but de les récupérer plus tard.
La première information à être récupérée est celle qui a été mise en attente en premier.

Opérations devant être supportées:

- ajouter un nouvel élément dans la file (enfiler, push);
- ôter l'élément le plus ancien de la file (défiler, pop);

Spécification désirées:

- Ces opérations doivent se faire en $O(1)$

Implémentation choisie:

Puisqu'il s'agit d'ajout et de suppression d'éléments à des positions définies, nous allons **implémenter la file à l'aide d'une liste**.

Implémentation de File

```
#include <list>
template<typename Objet>
class File
{
public:
        File() { } //Équivalent à celui par défaut
        ~File() { laFile.clear(); } //Équivalent à celui par défaut

        void enfiler(const Objet & x)
        {
                laFile.push_front(x); //enfilement au début de la liste
        }

        Objet defiler() //defilement à la fin de la liste
        {
                if (taille()==0) throw logic_error("La file est vide");
                Objet ret = laFile.back(); //retourne le dernier élément
                laFile.pop_back(); //enlève cet élément de la liste
                return ret;
        }
        unsigned int taille() const { return laFile.size(); }
private:
        list<Objet> laFile;
};
```

Remarques sur l'implémentation de File

- Avant de défiler on doit tester si la file est vide

- Par contre, il n'est pas nécessaire de tester si la file est pleine avant d'enfiler
  - Ceci vient du fait que nous utilisons **list** et, conséquemment, l'ajout d'un nouvel élément se fait dynamiquement sur le heap (tas/monceau) avec l'opérateur new utilisé dans **list** de la STL
    - s'il ne reste plus de mémoire de disponible, push_front() lancera une exception de type std::bad_alloc
- Cela ne serait pas le cas si l'implémentation était faite dans un tableau dynamique.
  - celui-ci possède une taille fixe au moment de son allocation. La file aurait donc une capacité limitée et il faudrait alors tester si la file est pleine avant d'enfiler.

Utilisation de File

```
#include "File.h"
using namespace std;

int main()
{
        File<int> maFile;

        for(int i=1; i<=10; i++)
                maFile.enfiler(i);
        cout << "taille = "<< maFile.taille() << endl;

        unsigned int n = maFile.taille();
        for(unsigned int i=0; i<n; i++)
                cout << maFile.defiler() << " ";
        cout << endl;
        cout << "taille = " << maFile.taille() << endl;

        return 0;
}
```

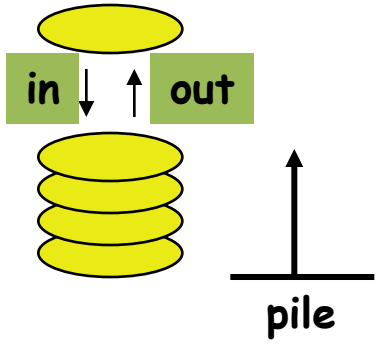
Le TDA pile

Piles :

LIFO : last in, first out

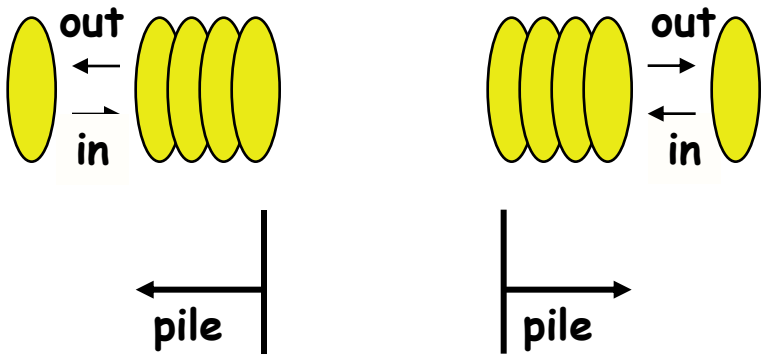
DAPS : dernier arrivé, premier sorti

ex. :assiettes
livres
factures

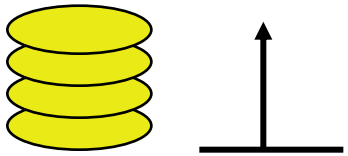


Pile = liste + gestion adaptée

manipulations (empiler et dépiler) par le même point d'accès



Le TDA pile



Sert à permettre la mise en attente d'informations dans le but de les récupérer plus tard.
La première information à être récupérée est celle qui a été mise en attente en dernier.

Opérations devant être supportées:

- ajouter un nouvel élément dans la Pile (empiler, push);
- ôter l'élément se trouvant au sommet de la Pile (dépiler, pop);

Spécification désirées:

- Ces opérations doivent se faire en O(1)

Implémentation choisie:

Puisqu'il s'agit d'ajout et de suppression d'éléments à une position définie, nous allons **implémenter la file à l'aide d'une liste**.

Implémentation de Pile

```
#include <list>
template<typename Objet>
class Pile
{
public:
        Pile() { } //Équivalent à celui par défaut
        ~Pile() { laPile.clear(); } //Équivalent à celui par défaut

        void empiler(const Objet & x)
        {
                laPile.push_front(x); //empilement au début de la liste
        }

        Objet depiler() //defilement à la fin de la liste
        {
                if (taille()==0) throw logic_error("La pile est vide");
                Objet ret = laPile.front(); //retourne le premier
élément
                laPile.pop_front(); //enlève cet élément de la liste
                return ret;
        }
        unsigned int taille() const { return laPile.size(); }
private:
        list<Objet> laPile;
}
```

Remarques sur l'implémentation de Pile

- Avant de dépiler on doit tester si la pile est vide

- Par contre, il n'est pas nécessaire de tester si la pile est pleine avant d'enpiler
  - Ceci vient du fait que nous utilisons **list** et, conséquemment, l'ajout d'un nouvel élément se fait dynamiquement sur le heap (tas/monceau) avec l'opérateur new utilisé dans **list** de la STL
    - s'il ne reste plus de mémoire de disponible, push_front() lancera une exception de type std::bad_alloc
- Cela ne serait pas le cas si l'implémentation serait faite dans un tableau dynamique.
  - celui-ci possède une taille fixe au moment de son allocation. La pile aurait donc une capacité limitée et il faudrait alors tester si la pile est pleine avant d'empiler.

Utilisation de Pile

```
#include "Pile.h"
using namespace std;

int main()
{
        Pile<int> maPile;

        for(int i=1; i<=10; i++)
                maPile.empiler(i);
        cout << "taille = "<< maPile.taille() << endl;

        unsigned int n = maPile.taille();
        for(unsigned int i=0; i<n; i++)
                cout << maPile.depiler() << " ";
        cout << endl;
        cout << "taille = " << maPile.taille() << endl;

        return 0;
}
```

Conteneurs STL pour séquences de données

- Les structures de données de base présentées dans ce chapitre sont fournies par la STL à l'aide de ces conteneurs pour séquences de données:
  - **<vector>** pour les vecteurs
  - **<list>** pour les listes doublement chainées
  - **<forward_list>** pour les listes simplement chainées
  - **<queue>** pour les files (FIFO)
  - **<stack>** pour les piles (LIFO)
  - **<deque>** pour les files extensibles « dans les deux directions »
- Le conteneur <deque> permet l'ajout et la suppression d'éléments en temps constant en autant qu'ils sont effectués au début ou à la fin du conteneur.
  - Notez que l'ajout et la suppression se fait en temps constant pour un <vector> seulement lorsque cela est effectué à la fin du conteneur (et qu'il reste de la place dans le tableau dynamique sous-jacent).
  - L'insertion/suppression dans un <deque> à une position arbitraire se fait en O(n).
- Tous les éléments de <deque> sont accessibles à temps constant à l'aide de l'opérateur [] (comme c'est le cas pour <vector>).
  - Mais, contrairement à <vector>, les élément de <deque> occupent de l'espace mémoire contigüe seulement par morceaux.
- Voir www.cplusplus.com pour plus d'information.