IFT339
Structures de données

Thème 8 : Conteneurs linéaires non contigus

Aïda Ouangraoua
Département d'informatique

UNIVERSITÉ DE
SHERBROOKE

Liste, Pile, File

- Éléments non contigus en mémoire

- Liste : ajout, suppresion partout en O(1) (si une référence est donnée)
  - Pas besoin de recopie ou décalage des éléments

- Pile: ajout, suppresion au début en O(1)
  - Accès seulement au début

- File: ajout au début et suppression à la fin en O(1)
  - Accès seulement aux extrémités

- Occupe exactement la mémoire que réellement utilisée

Liste (List)

- Ajout, suppresion partout en O(1) (si une référence est donnée)
- Accès à tous les éléments en utilisant un itérateur
- Accès par position en O(n) (contrairement à O(1) pour vector et deque)

List : Prototypes des opérateurs

❑ **Constructeurs**

list : Ø → list& // par défaut
list : size_t → list & // avec paramètre (taille)
list : list& → list& // par copie

❑ **Destructeur**

~list : Ø → Ø // fait appel à la fonction clear()

❑ **Affectateur**

operator= : list& → Ø // copie le paramètre dans l'objet appelant

List : Prototypes des opérateurs

❑ **Modificateurs**

swap : list& → Ø // échange le paramètre avec l'objet appelant
push_front : type& → Ø // ajoute un élément du <type> à la fin
pop_front: Ø → Ø // retire le dernier élément
push_back : type& → Ø // ajoute un élément du <type> à la fin
pop_back: Ø → Ø // retire le dernier élément
insert: cell*, type& → Ø //ajoute un élément à une position (iterator)
erase: cell*, type& → Ø //retire un élément à une position (iterator)
remove: type& → Ø //retire toute les occurrences d'un élément
merge: list& → Ø // ajoute une liste à la fin
sort: Ø → Ø // trie les éléments de la liste

List : Prototypes des opérateurs

❑ **Accès**

operator[] : size_t → type& // accès par position
at : size_t → type& // accès par position en vérifiant la dimension
front : Ø → type& // accès au premier élément
back : Ø → type& // accès au dernier élément

Spécifications : Prototypes des opérateurs

❑ **Gestion dimension**

resize : size_t → Ø // change la dimension
size : Ø → size_t // retourne la dimension
empty : Ø → bool // True si la dimension est 0, False sinon
clear : Ø → Ø // libère toute la mémoire allouée dynamiquement

Spécifications : Sémantique des opérateurs (axiomes)

- **Constructeurs**

list().empty() == Vrai // par défaut
list(n).size() == n // avec paramètre (taille)
list(l).size() == l.size()
et pour tout i, 0 ≤ i < l.size(), list(l)[i] == l[i] // par copie

- **Affectateur**

l2 = l ~ l2.operator=(l) ; l2.size() == l.size()
et pour tout i, 0 ≤ i < l.size(), l2[i] == l[i] // affectateur

Spécifications : Sémantique des opérateurs (axiomes)

❑ Modificateurs

l11 = l1; l22 = l2; l1.swap(l2); l1 == l22 et l2 == l11 // échange
l.push_back(x)__.back() == x // ajoute un élément à la fin
l.push_back(x)__.pop_back()__ = l // retire le dernier élément
l.push_front(x)__.front() == x // ajoute un élément au début
l.push_front(x)__.pop_front()__ = l // retire le premier élément
it = &(l[i]);
l2 = l.insert(it,x)__ ➔ pour tout j, 0 ≤ j < i, l2[j] == l[j] ; l2[i] = x ;
et pour tout j, i ≤ j < l.size(), l2[j+1] == l[j] //ajoute à une position
l2 = l.erase(it)__ ➔ pour tout j, 0 ≤ j < i, l2[j] == l[j] ;
et pour tout j, i+1 ≤ j < l.size(), l2[j-1] == l[j] //retire à une position
l3 = l.merge(l2)__ ➔ pour tout i, 0 ≤ i < l.size(), l3[i] == l[i] ; et
pour tout i, 0 ≤ i < l2.size(), l3[i+l.size()] == l2[i] //ajoute une liste à la fin

Spécifications : Sémantique des opérateurs (axiomes)

❑ Accès

l.operator[](i) ~ l[i] == élément à la position i // accès par position
l.at(i) == élément à la position i si i < l.size() // accès par position en vérifiant la dimension
l.back() == l[l.size()-1] // accès au dernier élément
l.front() == l[0] // accès au premier élément

Spécifications : Sémantique des opérateurs (axiomes)

❑ **Gestion dimension**

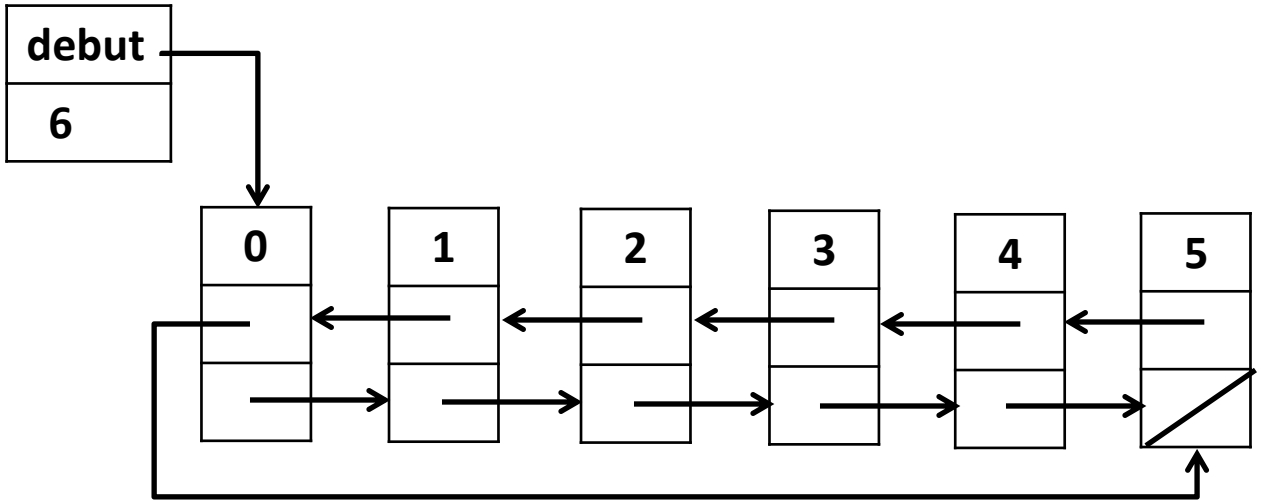
l.resize(n)__.size() == n // change la dimension
l.push_back(x)__.size() == l.size() + 1
l.pop_back()__.size() == l.size() – 1 si = l.size() > 0
l.push_front(x)__.size() == l.size() + 1
l.pop_front()__.size() == l.size() – 1 si = l.size() > 0
l.insert(it,x) __.size() == l.size() + 1
l.erase(it)__.size() == l.size() – 1
l.merge(l2) __.size() == l.size() + l2.size()
l.empty() = True si et seulement si l.size() == 0
l.clear()__ == list() // libère la mémoire allouée dynamiquement

```
#ifndef _list_h
#define _list_h

template <typename
TYPE>
class cell{
 private:
  type val;
  cell* prec, suiv;
 public:
   cell(type&);
}

template <typename
TYPE>
class list{
 private:
  cell *debut;
  size_t dim;
 public:
…
}
#endif
```

Représentation

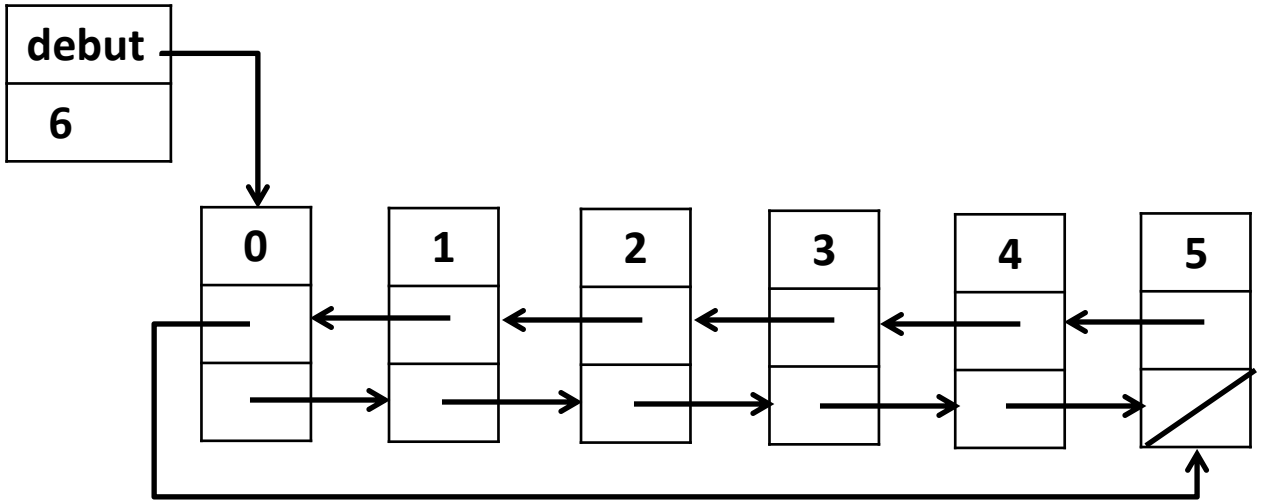


```
#ifndef _list_h
#define _list_h

template <typename
TYPE>
class cell{
 private:
  type val;
  cell* prec, suiv;
 public:
   cell(type&);
}

template <typename
TYPE>
class list{
 private:
  cell *debut;
  size_t dim;
 public:
  class iterator;
  class reverse_iterator;
  iterator begin();
  iterator end();
  reverse_iterator
rbegin();
  reverse_iterator rend();
```

Représentation



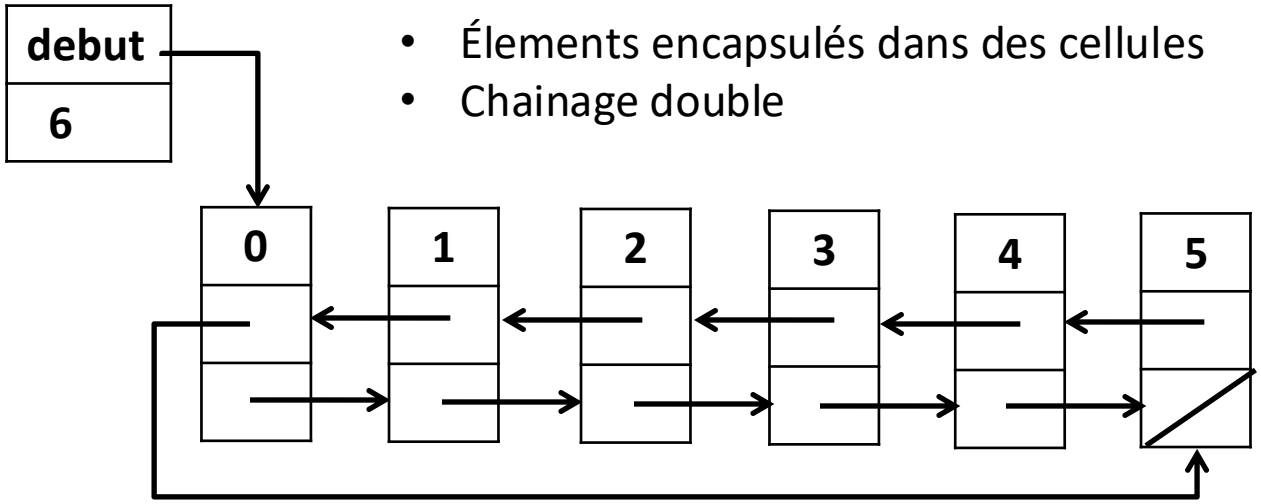
- **Itérateur**
- Appliquer le même traitement en séquence à plusieurs éléments
- Besoin d'une position de début (begin), une position de fin (end), un opérateur d'incrémentation (++), et de décrémentation (--)
- La fin est après le dernier élément
- La liste est vide si begin == end
- Classe iterator
  - définie dans la portée de la classe (pour avoir accès aux détails d'implémentation de la classe)
  - Représente une position dans le conteneur
  - Abstraction indépendante du type de conteneur

```
#ifndef _list_h
#define _list_h

template <typename
TYPE>
class cell{
 private:
  type val;
  cell* prec, suiv;
 public:
   cell(type&);
}

template <typename
TYPE>
class list{
 private:
  cell *debut;
  size_t dim;
 public:
  class iterator;
  class reverse_iterator;
  iterator begin();
  iterator end();
  reverse_iterator
rbegin();
  reverse_iterator rend();
```

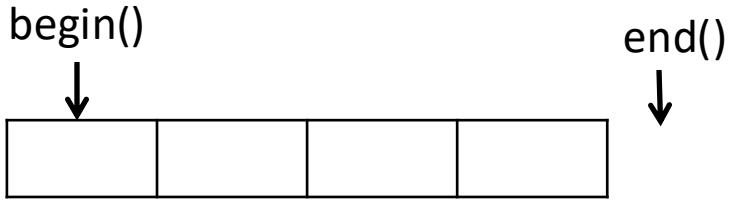
Représentation



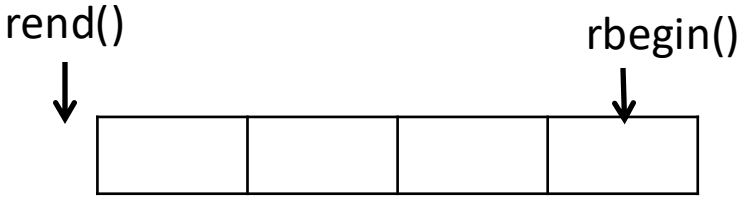
- Élements encapsulés dans des cellules
- Chainage double

❑ **Différents types d'itérateur**

- iterator
- const_iterator



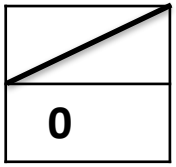
- reverse_iterator
- const_reverse_iterator



Algorithmes

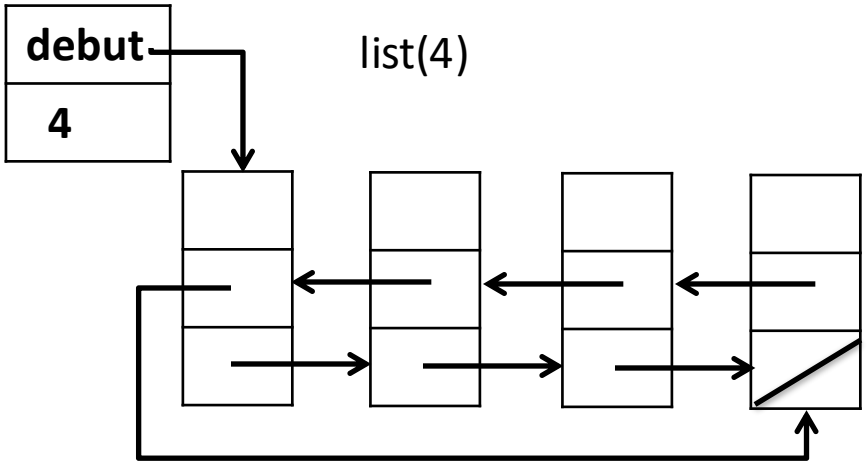
❑ Constructeurs

```
list::list(){
 debut= nullptr;
 dim = 0;
}
```



list()

```
list::list(size_t n):list(){
 for(int i = 0; i< n; i++)
    push_back(type());
}
```



list(4)

```
list::list(list& l) {
 *this = l;
}
```

Algorithmes

❑ Destructeur

```
list::~list(){
 clear();
}
```

```
list::clear(){
  while (dim !=0)
     pop_back()}
```

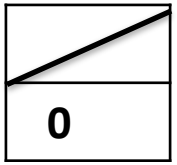
Algorithmes

❑ Affectateur

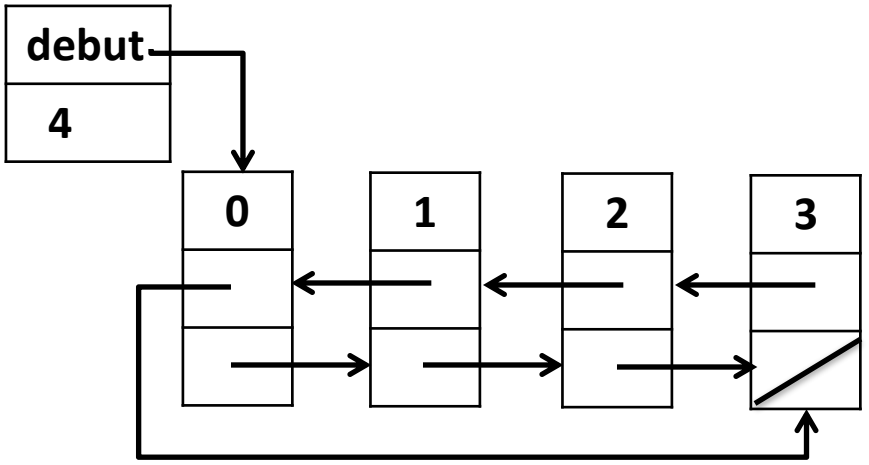
```
void list::operator=(list& l){
 clear();
 cell* c = l.debut;
 while (c != nullptr):
   push_back(c -> val);
   c = c -> suiv;
}
```

l = l2;

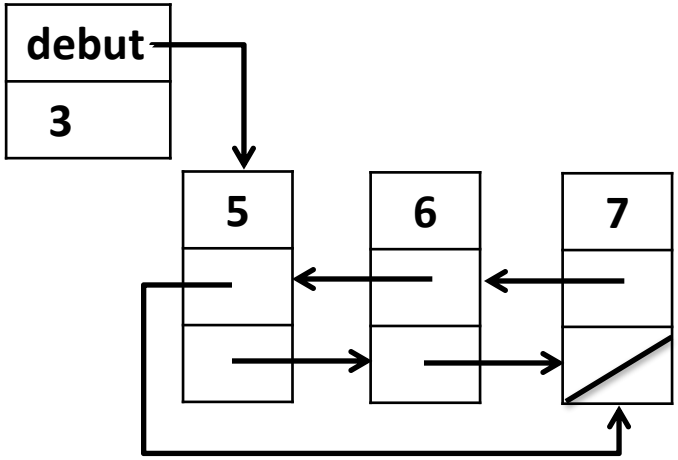
l



l



l2

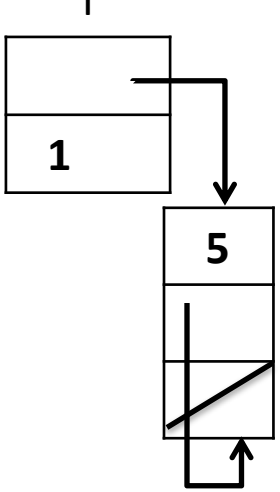


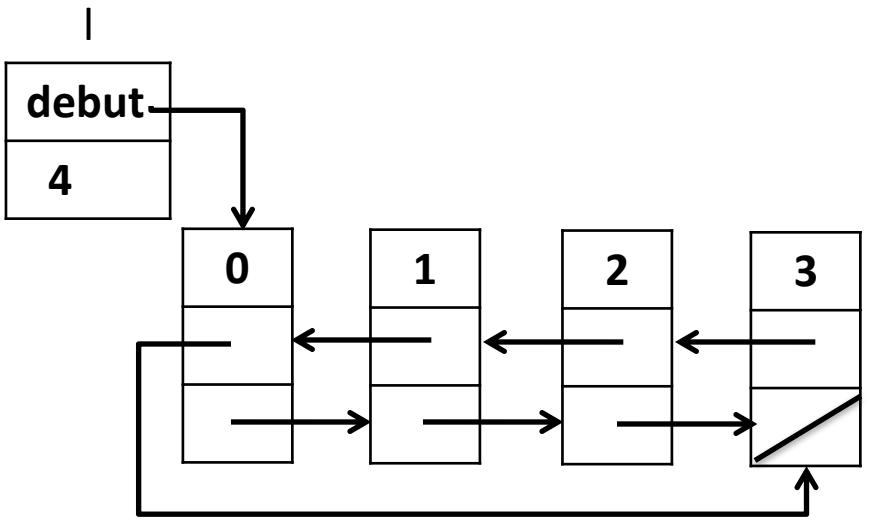
Algorithmes

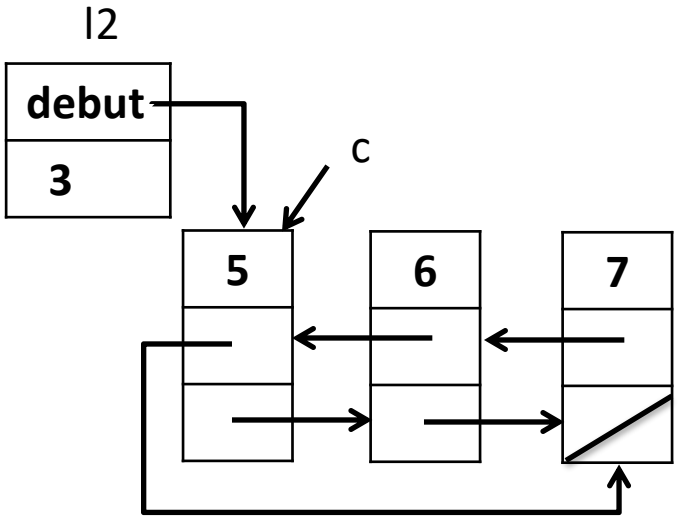
❑ Affectateur

```
void list::operator=(list& l){
 clear();
 cell* c = l.debut;
 while (c != nullptr):
   push_back(c -> val);
   c = c -> suiv;
}
```

l = l2;



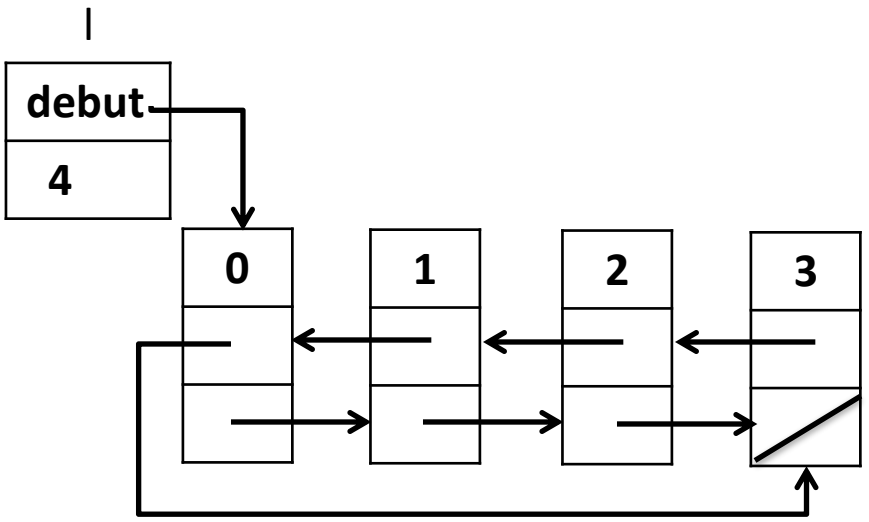




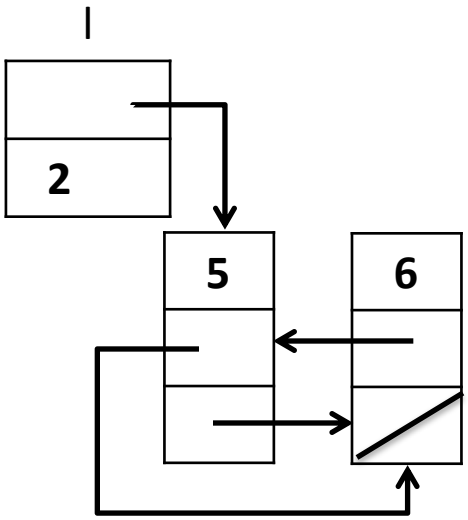
Algorithmes

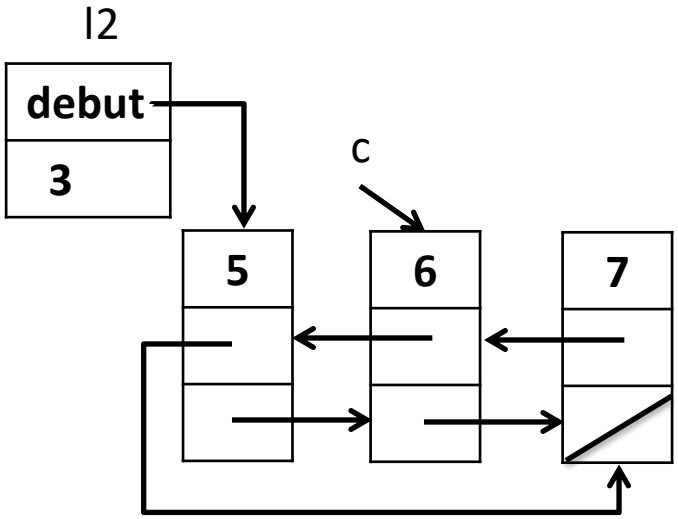
❑ Affectateur

```
void list::operator=(list& l){
 clear();
 cell* c = l.debut;
 while (c != nullptr):
   push_back(c -> val);
   c = c -> suiv;
}
```



l = l2;

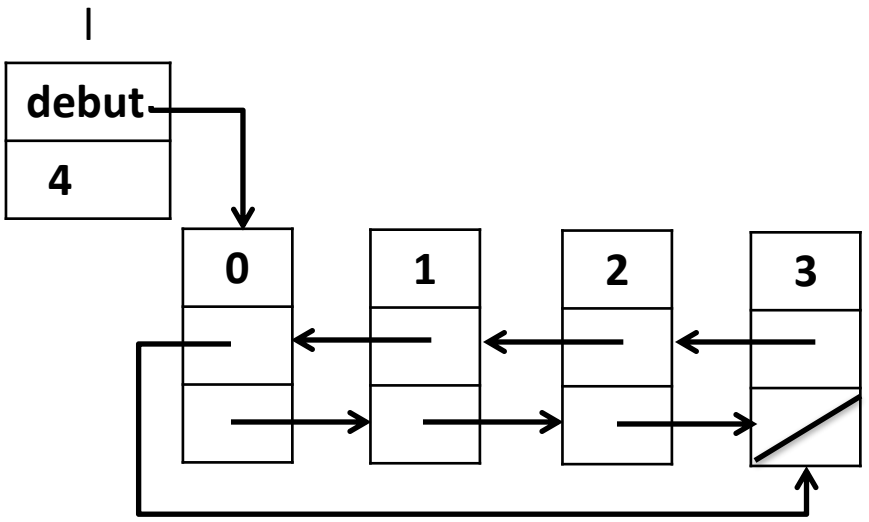




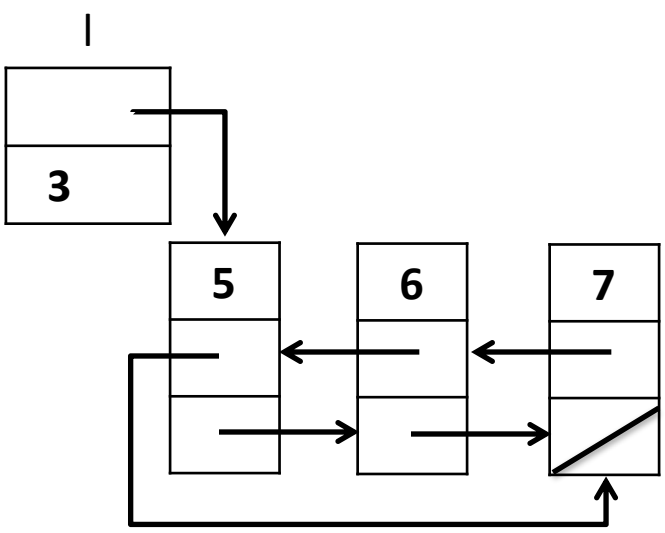
Algorithmes

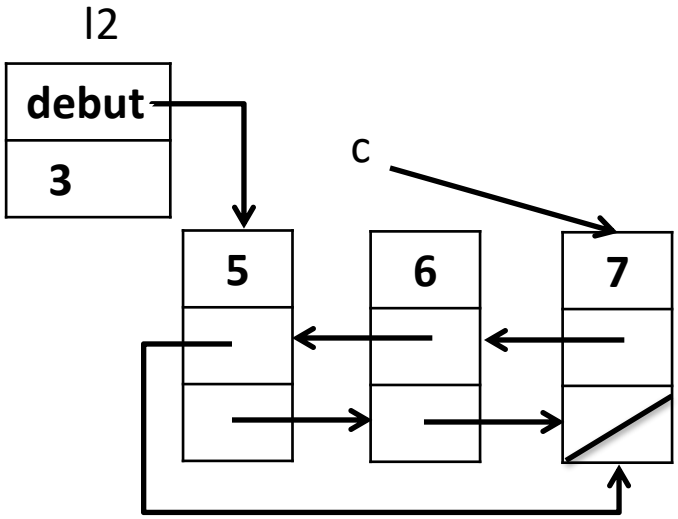
❑ Affectateur

```
void list::operator=(list& l){
 clear();
 cell* c = l.debut;
 while (c != nullptr):
   push_back(c -> val);
   c = c -> suiv;
}
```



l = l2;

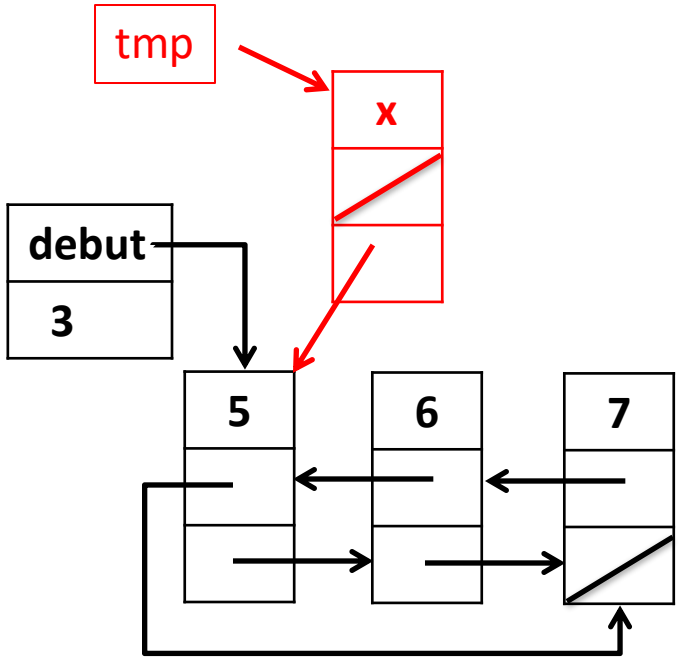




Algorithmes

❑ Modificateurs

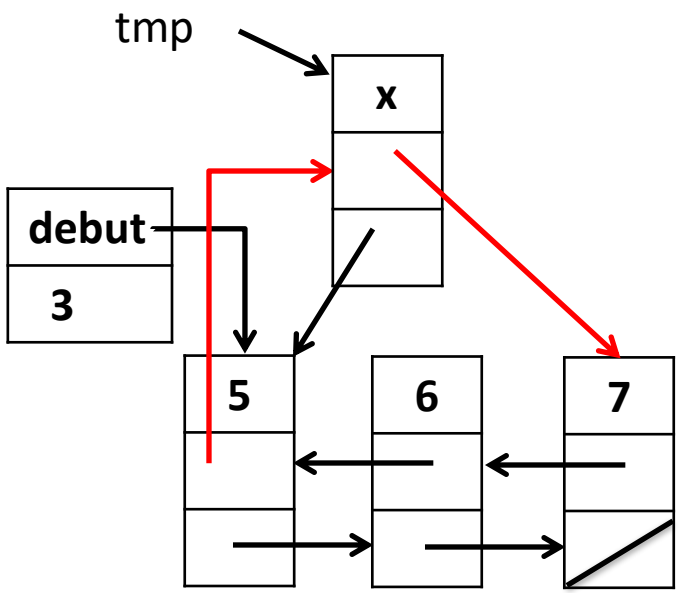
```
void list::push_front(type& x){
  cell* tmp = new cell(x);
  tmp -> suiv = debut;
  if(debut != nullptr){
    tmp -> prec = debut -> prec;
    debut -> prec = tmp;
   }
  debut = tmp;
  dim += 1;
}
```



Algorithmes

❑ Modificateurs

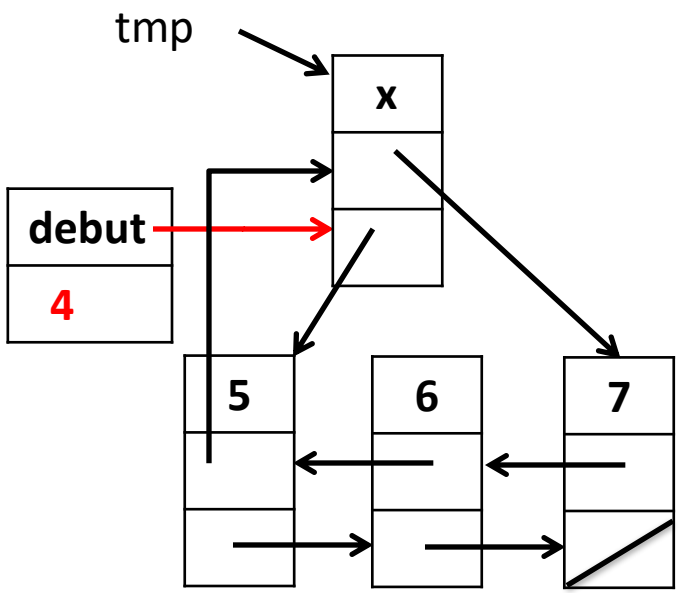
```
void list::push_front(type& x){
  cell* tmp = new cell(x);
  tmp -> suiv = debut;
  if(debut != nullptr){
    tmp -> prec = debut -> prec;
    debut -> prec = tmp;
   }
  debut = tmp;
  dim += 1;
}
```



Algorithmes

❑ Modificateurs

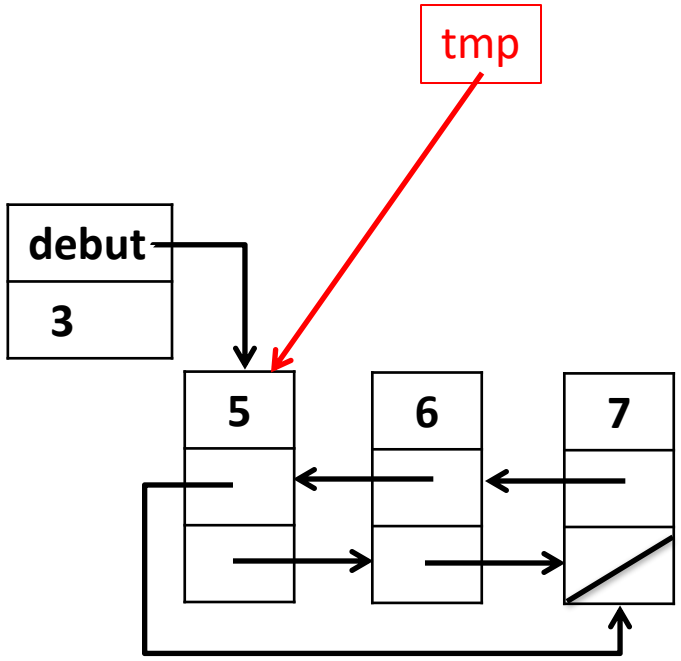
```
void list::push_front(type& x){
  cell* tmp = new cell(x);
  tmp -> suiv = debut;
  if(debut != nullptr){
    tmp -> prec = debut -> prec;
    debut -> prec = tmp;
   }
  debut = tmp;
  dim += 1;
}
```



Algorithmes

❑ Modificateurs

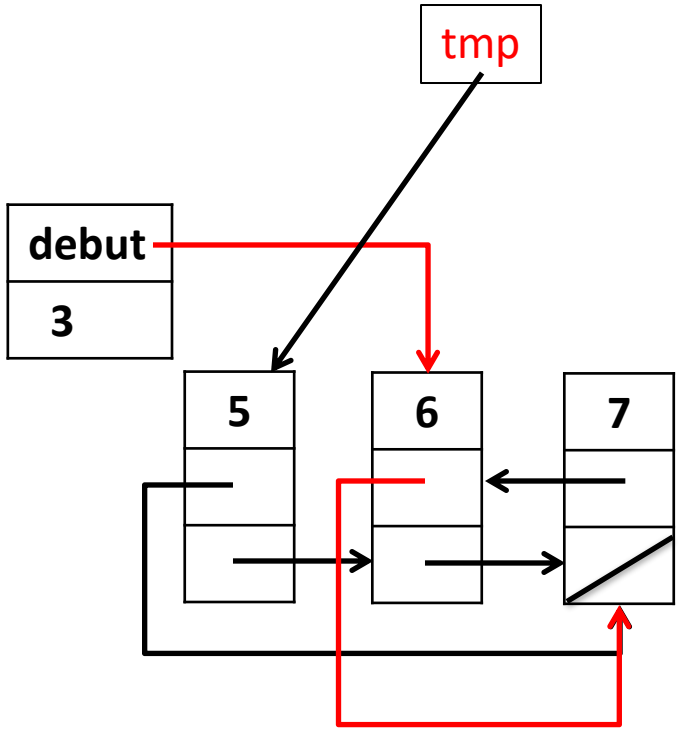
```
void list::pop_front(){
  cell* tmp = debut;
  if(debut != nullptr){
    debut = debut -> suiv;
    debut -> prec = tmp -> prec;
   delete tmp;
   dim -= 1;
}
}
```



Algorithmes

❑ Modificateurs

```
void list::pop_front(){
  cell* tmp = debut;
  if(debut != nullptr){
    debut = debut -> suiv;
    debut -> prec = tmp -> prec;
   delete tmp;
   dim -= 1;
}
}
```

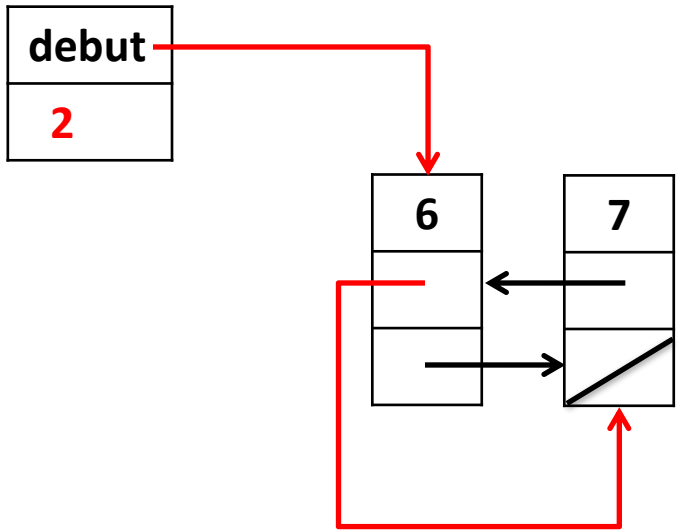


Algorithmes

tmp

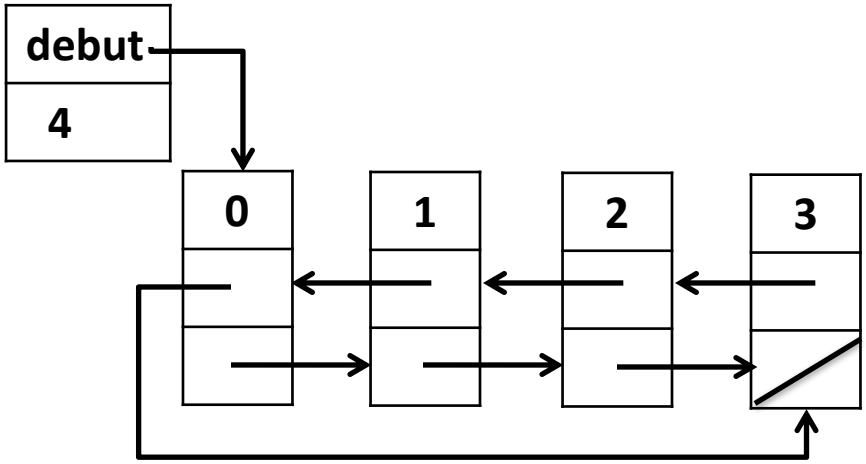
❑ Modificateurs

```
void list::pop_front(){
  cell* tmp = debut;
  if(debut != nullptr){
    debut = debut -> suiv;
    debut -> prec = tmp -> prec;
   delete tmp;
   dim -= 1;
}
}
```



Exercice 1

Écrire le code des fonctions push_back et pop_back pour cette représentation.

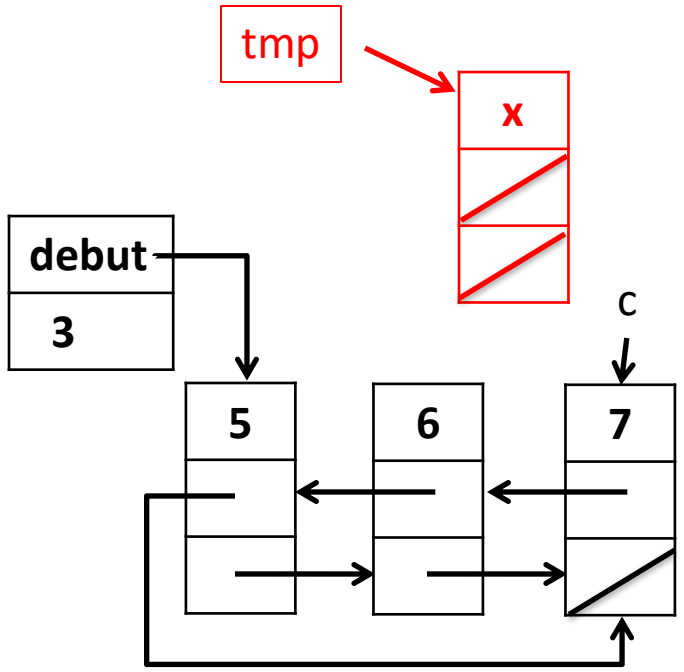


Algorithmes

❑ Modificateurs

```
void list::insert(cell* c, type& x){

  if(cell == debut)
    push_front(x);
  else{
     cell* tmp = new cell(x);
     tmp -> suiv = c;
     tmp -> prec = c -> prec;
     tmp -> suiv -> prec = tmp;
     tmp -> prec -> suiv = tmp;
     dim += 1;
   }
}
```

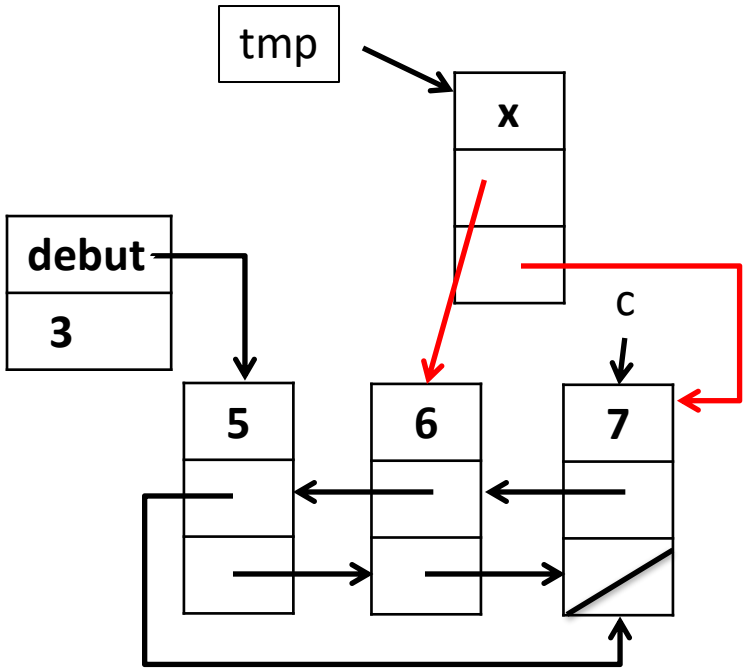


Algorithmes

❑ Modificateurs

```
void list::insert(cell* c, type& x){

  if(cell == debut)
    push_front(x);
  else{
     cell* tmp = new cell(x);
     tmp -> suiv = c;
     tmp -> prec = c -> prec;
     tmp -> suiv -> prec = tmp;
     tmp -> prec -> suiv = tmp;
     dim += 1;
   }
}
```

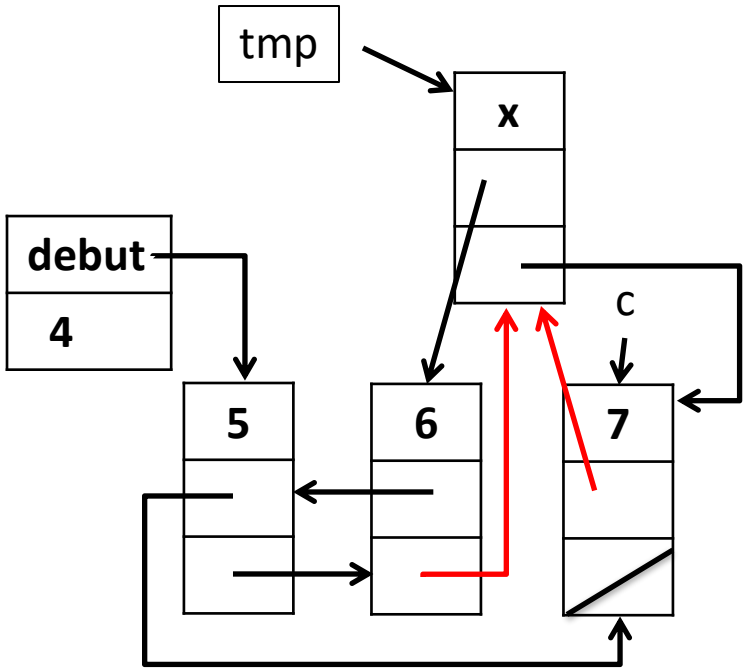


Algorithmes

❑ Modificateurs

```
void list::insert(cell* c, type& x){

  if(cell == debut)
    push_front(x);
  else{
     cell* tmp = new cell(x);
     tmp -> suiv = c;
     tmp -> prec = c -> prec;
     tmp -> suiv -> prec = tmp;
     tmp -> prec -> suiv = tmp;
     dim += 1;
   }
}
```



Algorithmes

❑ Modificateurs

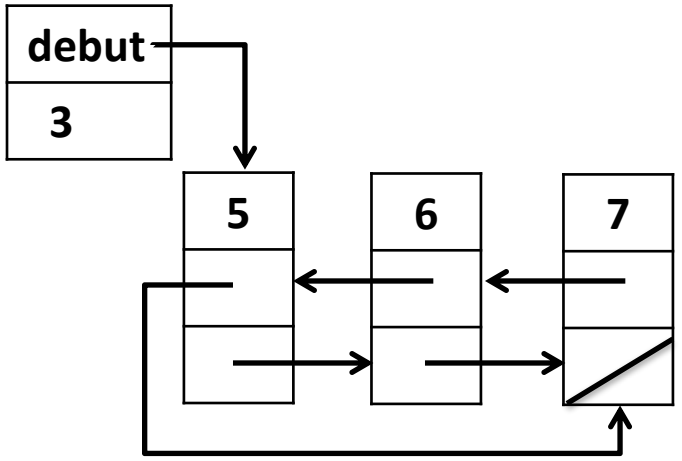
```
void list::insert(cell* c, type& x){

  if(cell == debut)
    push_front(x);
  else{
     cell* tmp = new cell(x);
     tmp -> suiv = c;
     tmp -> prec = c -> prec;
     tmp -> suiv -> prec = tmp;
     tmp -> prec -> suiv = tmp;
     dim += 1;
   }
}
```

1) Impossible de faire un insert si la list est vide.
2) Impossible de faire un insert après le dernier élément ?

Avec cette représentation:

- Le code n'est pas uniforme
- On ne peut pas insérer partout avec la fonction insert.

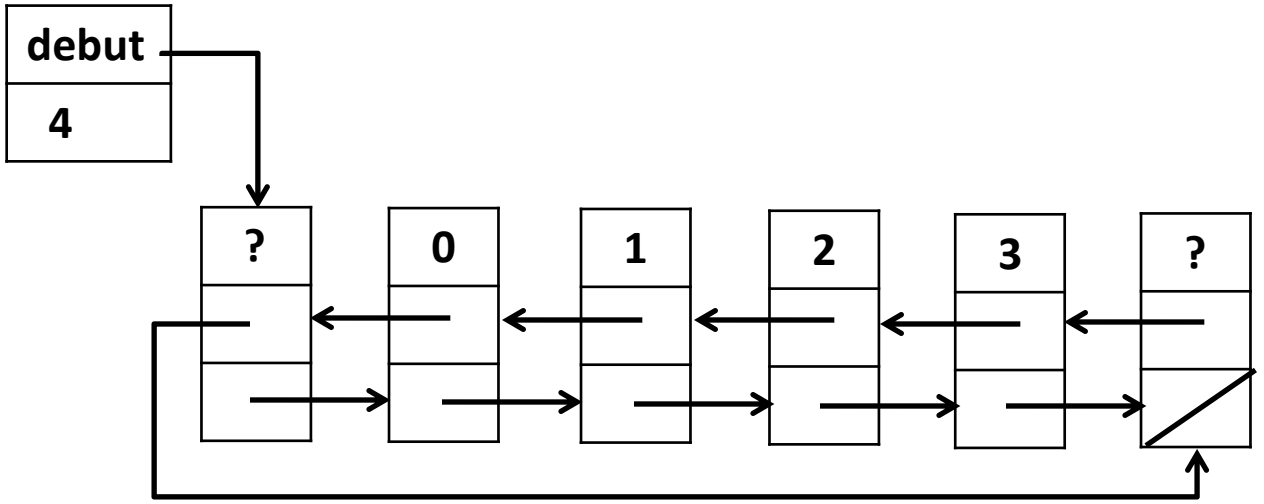


```
#ifndef _list2_h
#define _list2_h

template <typename
TYPE>
class cell{
 private:
  type val;
  cell* prec, suiv;
 public:
   cell(type&);
}

template <typename
TYPE>
class list2{
 private:
  cell *debut;
  size_t dim;
 public:
…
}
#endif
```

Représentation 2



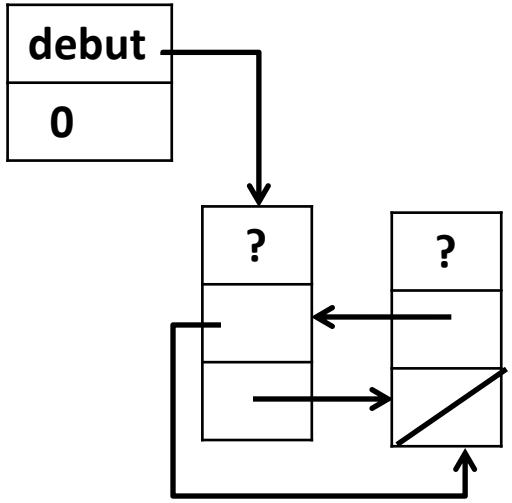
Cellule fictive au début

Cellule fictive à la fin

Algorithmes

Représentation 2

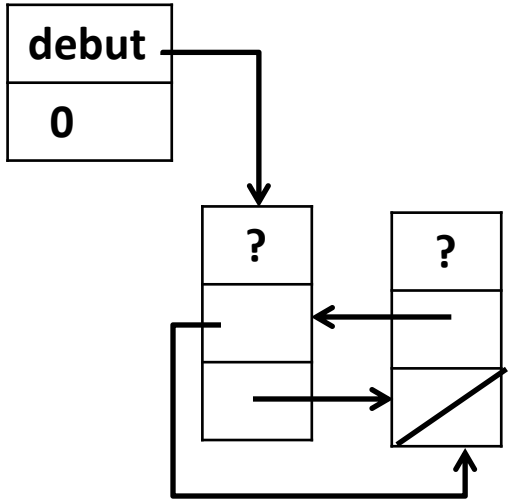
```
list2::list2(){
 debut= new cell();
 cell* fin = new cell();
 debut -> prec = debut -> suiv = fin;
 fin -> prec = debut;
 dim = 0;
}
```



Algorithmes Représentation 2

```
list2::list2(){
 debut= new cell();
 cell* fin = new cell();
 debut -> prec = debut -> suiv = fin;
 fin -> prec = debut;
 dim = 0;
}
```

```
list2::~list2(){
 clear();
 delete debut -> prec;
 delete debut;
 debut = nullptr;
}
```

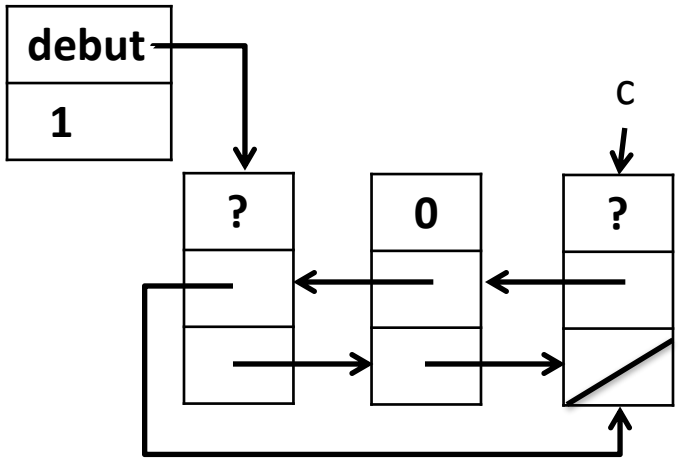


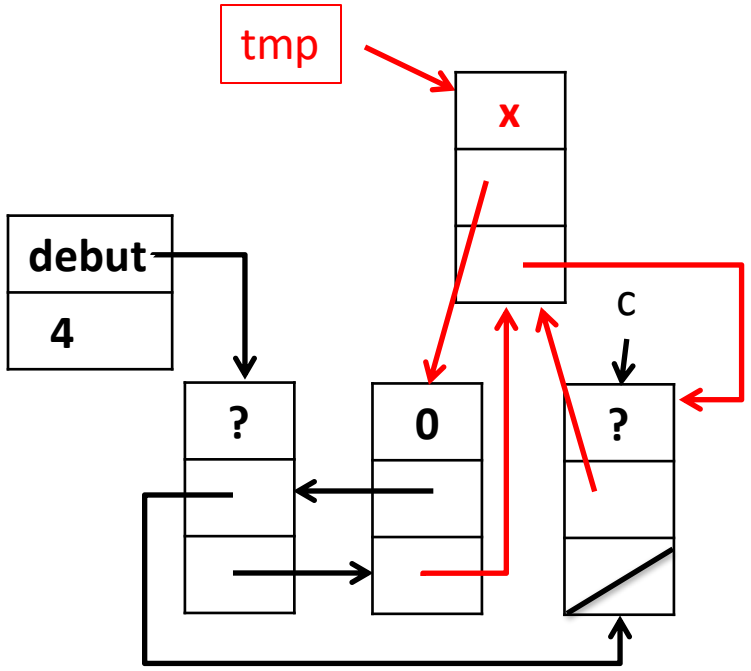
Algorithmes

Représentation 2

```
void list2::insert(cell* c, type& x){
    cell* tmp = new cell(x);
    tmp -> suiv = c;
    tmp -> prec = c -> prec;
    tmp -> suiv -> prec = tmp;
    tmp -> prec -> suiv = tmp;
    dim += 1;
}
```

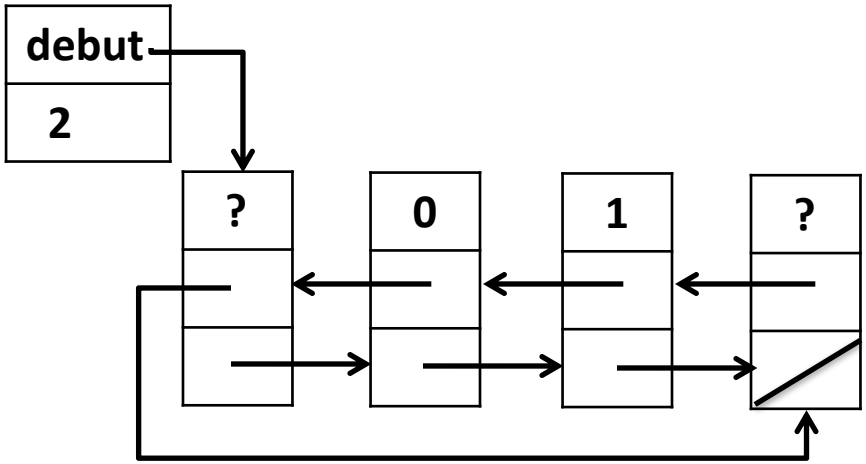
Le code est uniforme





Exercice 2

Écrire le code des fonctions erase, push_front, push_back, pop_front et pop_back pour cette représentation:
push_front et push_back utilisent insert.
pop_front et pop_back utilisent erase.



Pile (Stack) : Prototypes des opérateurs

- Principe: dernier inséré, premier retiré (LIFO)
- Ajout, suppression en O(1)
- Seul le dernier élément inséré est accessible
- Exemple d'utilisation:
  - gestion de la mémoire,
  - undo/redo,
  - évaluation d'expression mathématiques (deux piles: opérandes et opérateurs)
    Exemple : (((5+4)*2)/(2+1))

Pile (Stack) : Prototypes des opérateurs

❑ **Constructeurs**

stack : Ø → stack& // par défaut
stack : stack& → stack& // par copie

❑ **Destructeur**

~stack : Ø → Ø // fait appel à la fonction clear()

❑ **Affectateur**

operator= : stack& → Ø // copie le paramètre dans l'objet appelant

Pile (Stack) : Prototypes des opérateurs

Modificateurs, Accès et Gestion de dimension

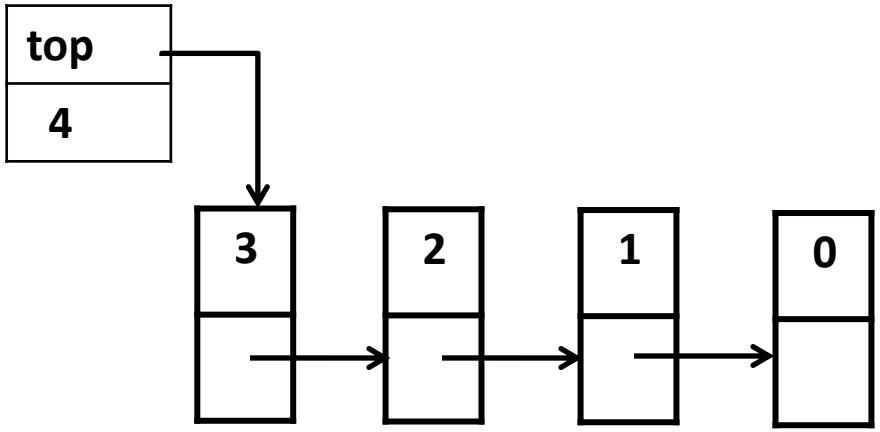
push : type& → Ø // ajoute un élément du <type> au début
pop: Ø → Ø // retire le premier élément
top: Ø → type& //retourne le premier élément
size : Ø → size_t // retourne la dimension
empty : Ø → bool // True si la dimension est 0, False sinon

```
#ifndef _stack_h
#define _stack_h

template <typename
TYPE>
class cell{
 private:
  type val;
  cell* next;
 public:
   cell(type&);
}

template <typename
TYPE>
class stack{
 private:
  cell *top;
  size_t dim;
 public:
 …
}
#endif
```

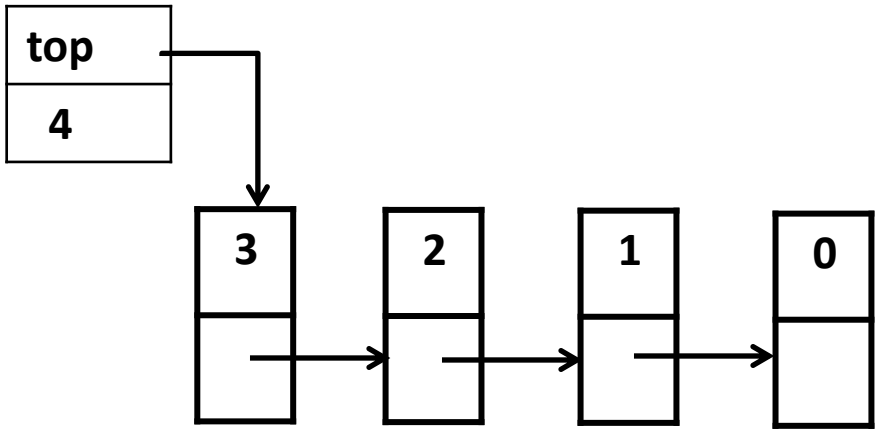
Représentation



- Élements encapsulés dans des cellules
- Chainage simple

Exercice 3

Écrire tous les operateurs de la pile pour cette représentation:
Ajout et retrait au top.



File (Queue) : Prototypes des opérateurs

- Principe: premier inséré, premier retiré (FIFO)
- Ajout, suppression en O(1)
- Seul le premier élément inséré est accessible

File (Queue) : Prototypes des opérateurs

❑ **Constructeurs**

queue : Ø → queue& // par défaut
queue : queue& → queue& // par copie

❑ **Destructeur**

~queue : Ø → Ø // fait appel à la fonction clear()

❑ **Affectateur**

operator= : queue& → Ø // copie le paramètre dans l'objet appelant

File (Queue) : Prototypes des opérateurs

Modificateurs, Accès et Gestion de dimension

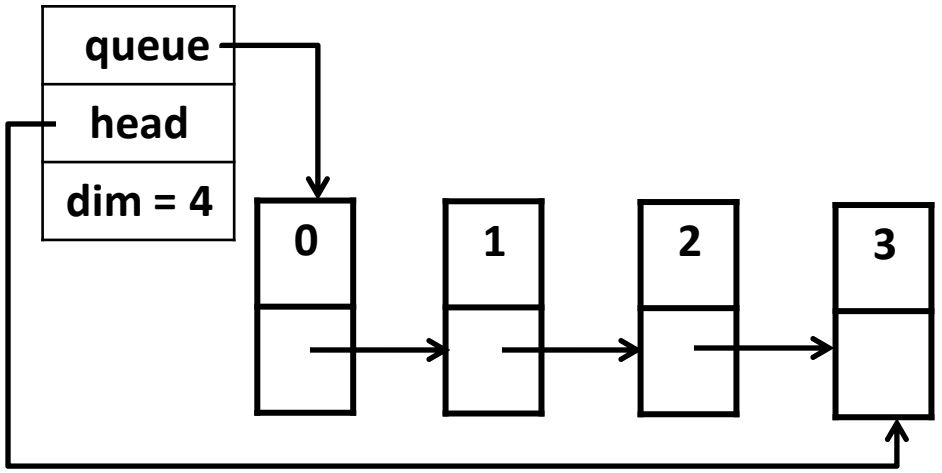
(push) enqueue : type& → Ø // ajoute un élément du <type>
(pop) dequeue: Ø → Ø // retire un élément
(top) queue: Ø → type& //retourne le premier élément inséré
size : Ø → size_t // retourne la dimension
empty : Ø → bool // True si la dimension est 0, False sinon

```
#ifndef _queue_h
#define _ queue _h

template <typename
TYPE>
class cell{
 private:
  type val;
  cell* next;
 public:
   cell(type&);
}

template <typename
TYPE>
class queue{
 private:
  cell *queue;
  cell *head;
  size_t dim;
 public:
 …
}
#endif
```

Représentation



- Élements encapsulés dans des cellules
- Chainage simple

Exercice 4

Écrire tous les operateurs de la file pour cette représentation:
Ajout à la tête; Retrait à la queue.