IFT339
Structures de données

Thème 7 : Les tableaux dynamiques
Vector, Deque

Aïda Ouangraoua
Département d'informatique

UNIVERSITÉ DE
SHERBROOKE

Vector

- Éléments contigus en mémoire
- Recopie en cas d’augmentation de capacité
- Occupe plus de mémoire que réellement utilisée
- Accès en O(1) à tout élément à partir de sa position i
- Ajout d’élément à la fin en O(1)

Vector

- Dimension : nombre d'éléments contenus dans le vector
- Capacité : taille de la mémoire allouée
- En tout temps : Dimension <= Capacité

Spécifications : Prototypes des opérateurs

- **Constructeurs**

vector : Ø → vector& // par défaut
vector : size_t → vector& // avec paramètre (dimension)
vector : vector& → vector& // par copie

- **Destructeur**

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

- **Affectateur**

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

Spécifications : Prototypes des opérateurs

- **Modificateurs**

swap : vector& → Ø // échange le paramètre avec l'objet appelant
push_back : type& → Ø // ajoute un élément du <type> à la fin
pop_back: Ø → Ø // retire le dernier élément

- **Accès**

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

Spécifications : Prototypes des opérateurs

- **Gestion capacité/dimension**

resize : size_t → Ø   // change la dimension
size : Ø → size_t   // retourne la dimension

reserve : size_t → Ø   // augmente la capacité, mais ne la réduit pas
capacity : Ø → size_t   // retourne la capacité

empty : Ø → bool   // True si la dimension est 0, False sinon
shrink_to_fit : Ø → Ø   // ramène la capacité à la dimension
clear : Ø → Ø   // libère toute la mémoire allouée dynamiquement

Spécifications : Sémantique des opérateurs (axiomes)

- **Constructeurs**

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

- **Affectateur**

v2 = v ~ v2.operator=(v) ; v2.size() == v.size()
et pour tout i, 0 ≤ i < v.size(), v2[i] == v[i] // affectateur

Spécifications : Sémantique des opérateurs (axiomes)

❑ Modificateurs

v11 = v1; v22 = v2; v1.swap(v2); v1 == v22 et v2 == v11 // échange
v.push_back(x)__.back() == x // ajoute un élément à la fin
v.push_back(x)__.pop_back()__ = v   // retire le dernier élément

❑ Accès

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

Spécifications : Sémantique des opérateurs (axiomes)

- **Gestion capacité/dimension**

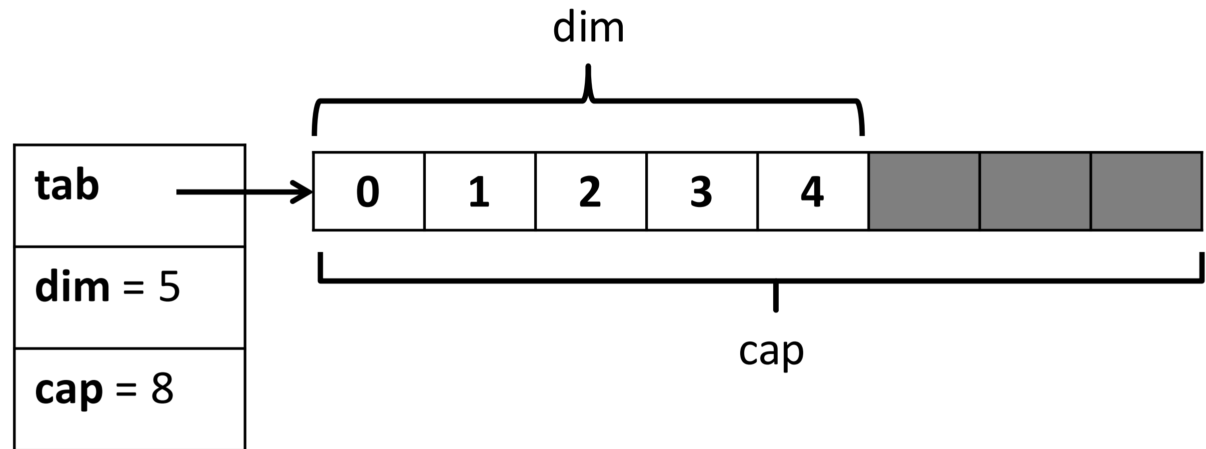
v.resize(n)__.size() == n // change la dimension
v.push_back(x)__.size() == v.size() + 1
v.pop_back()__.size() == v.size() – 1 si = v.size() > 0 // retourne la dimension

v.reserve(m)__.capacity() == m si m ≥ v.capacity(), // augmente la capacité, mais ne la réduit pas

v.empty() = True si et seulement si v.size() == 0
v.shrink_to_fit()__.capacity() == v.size() : Ø → Ø // ramène la capacité à la dimension
v.clear()__ == vector() // libère la mémoire allouée dynamiquement

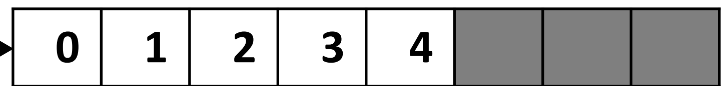
Représentation

```
#ifndef _vector_h
#define _vector_h

template <typename TYPE>
class vector
{
 private:
  TYPE *tab;
  size_t dim;
  size_t cap;

 public:
  ...
}
#endif
```



v

tab
dim = 5
cap = 8



Algorithmes

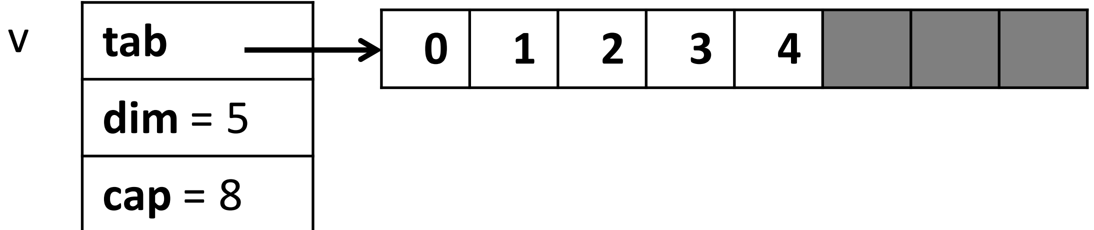
❑ Constructeurs

```
vector::vector(){
 tab= nullptr;
 dim = cap = 0;
}
```

tab = nullptr
dim = 0
cap = 0

vector();

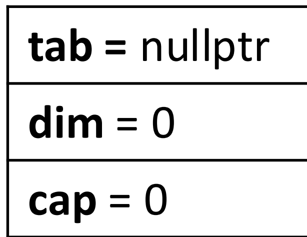
v



Algorithmes

❑ Constructeurs

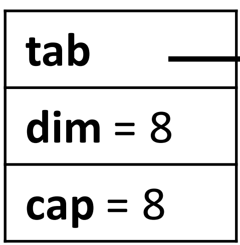
```
vector::vector(){
 tab= nullptr;
 dim = cap = 0;
}
```

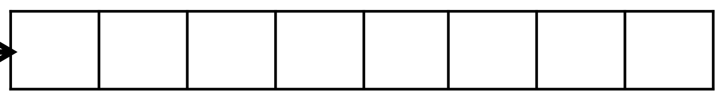


vector();

```
vector::vector(size_t n){
 tab= new type[n];
 dim = cap = n;
}
```

vector(8);

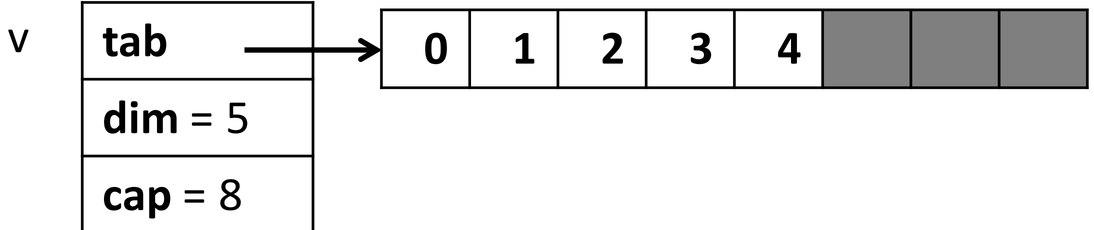




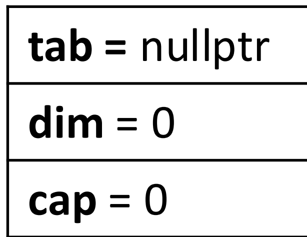
Algorithmes

❑ Constructeurs

v



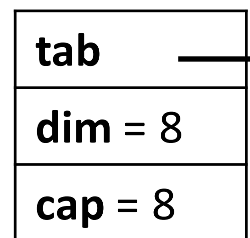

```
vector::vector(){
 tab= nullptr;
 dim = cap = 0;
}
```



vector();

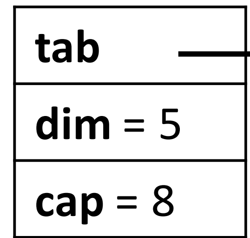
```
vector::vector(size_t n){
 tab= new type[n];
 dim = cap = n;
}
```

vector(8);




```
vector::vector(vector& v){
 tab= new type[v.capacity()];
 dim = v.size();
 cap = v.capacity();
 for(i=0; i< dim;i++)
    tab[i] = v.tab[i];
}
```

vector(v);



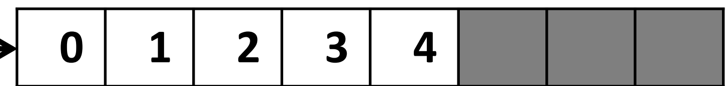
0 1 2 3 4

Algorithmes

❑ Destructeur

v

tab
dim = 5
cap = 8




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

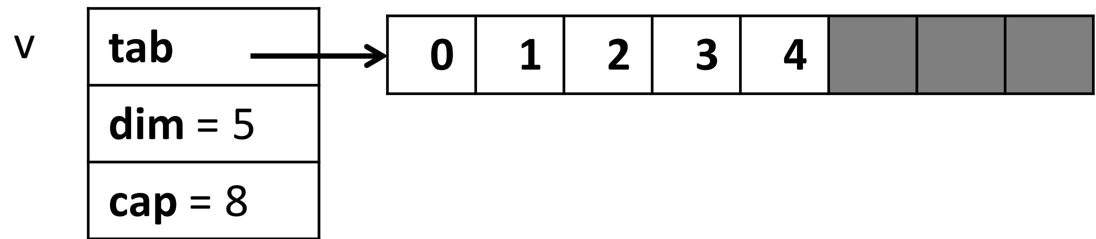
v.clear();

```
vector::clear(){
 delete [] tab;
 tab = nullptr;
 dim = cap = 0;
}
```

Algorithmes

❑ Destructeur

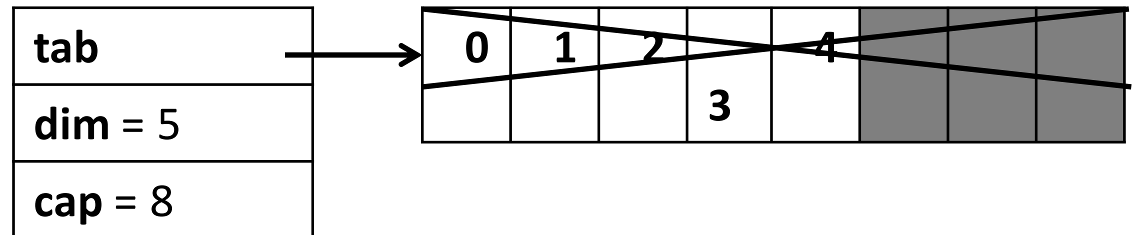
v



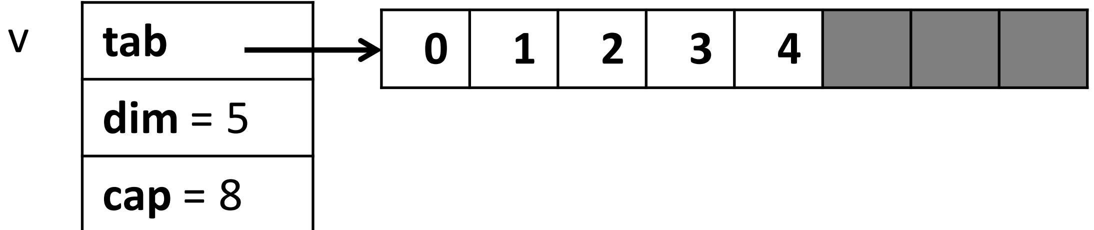

```
vector::~vector(){
 clear();
}
```

```
void vector::clear(){
 delete [] tab;
 tab = nullptr;
 dim = cap = 0;
}
```

v.clear();
delete [] tab;



Algorithmes

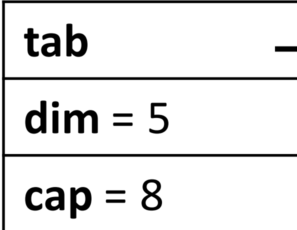


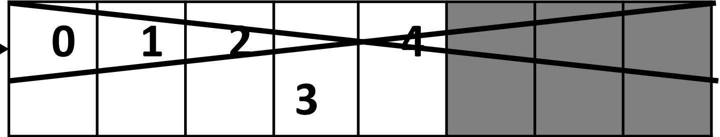
❑ Destructeur

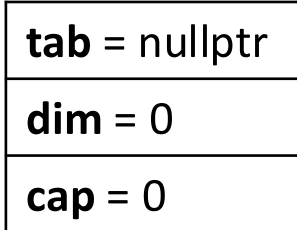
```
vector::~vector(){
 clear();
}
```

```
void vector::clear(){
 delete [] tab;
 tab = nullptr;
 dim = cap = 0;
}
```

v.clear();
delete [] tab;



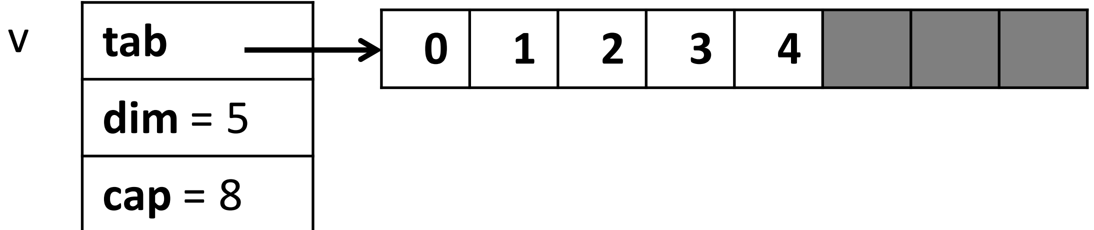




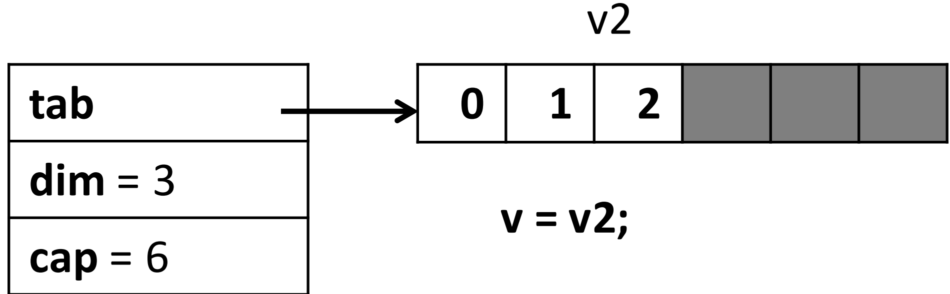
tab = nullptr;
dim = cap =
0;

Algorithmes

❑ Affectateur




```
void
vector::operator=(vector& v){
 reserve(v.capacity());
 dim = v.dim;
 for(i=0; i< dim;i++)
    tab[i] = v.tab[i];

}
```



v = v2;

Algorithmes

❑ Affectateur

```
void
vector::operator=(vector& v){
 reserve(v.capacity());
 dim = v.dim;
 for(i=0; i< dim;i++)
    tab[i] = v.tab[i];

}
```

v

tab
dim = 5
cap = 8

0	1	2	3	4			

v2

tab
dim = 3
cap = 6

0	1	2			

v = v2;

reserve(v.capacity());

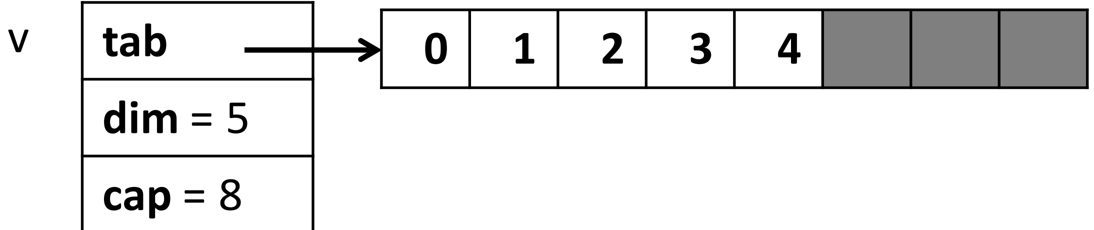
tab
dim = 3
cap = 8

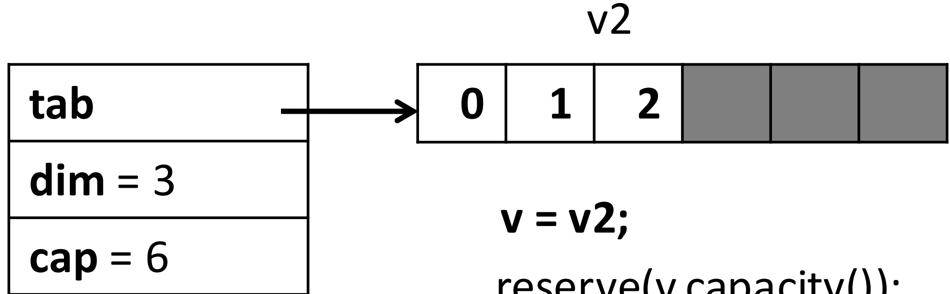
0	1	2					

Algorithmes

❑ Affectateur

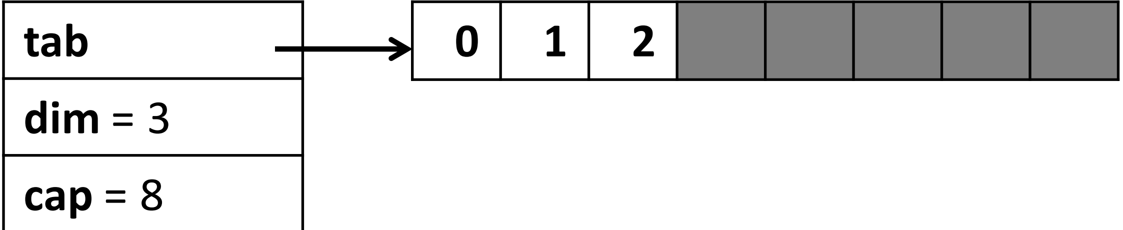
```
void
vector::operator=(vector& v){
 reserve(v.capacity());
 dim = v.dim;
 for(i=0; i< dim;i++)
    tab[i] = v.tab[i];
}
```

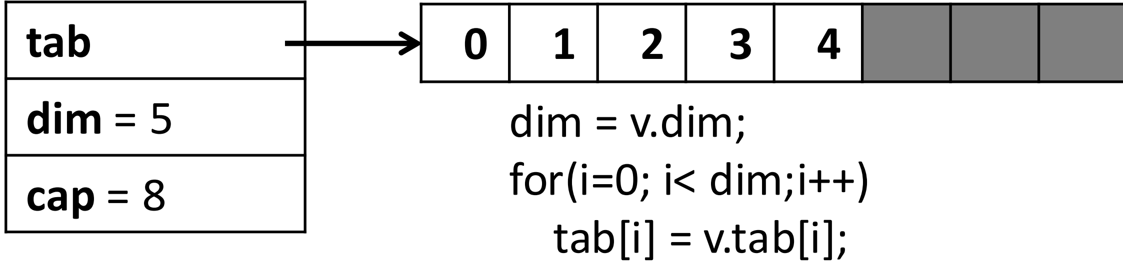




v = v2;

reserve(v.capacity());



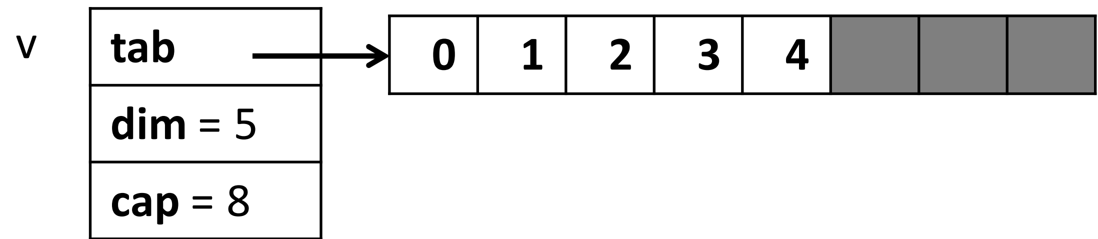


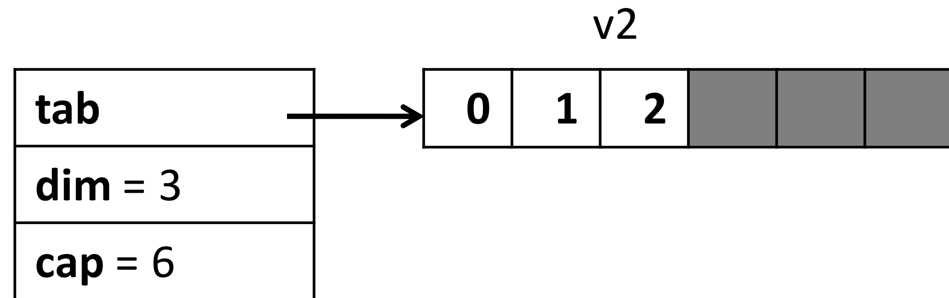

```
dim = v.dim;
for(i=0; i< dim;i++)
   tab[i] = v.tab[i];
```

Algorithmes

❑ Modificateurs

```
void vector::swap(vector& v){
 tab.swap(v.tab);
 dim.swap(v.dim);
 cap.swap(v.cap);
}
```

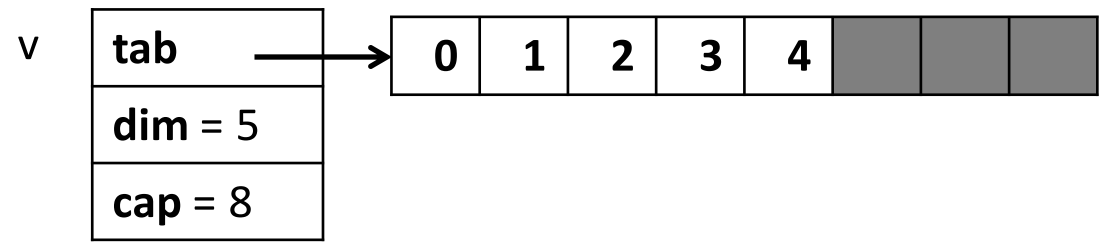


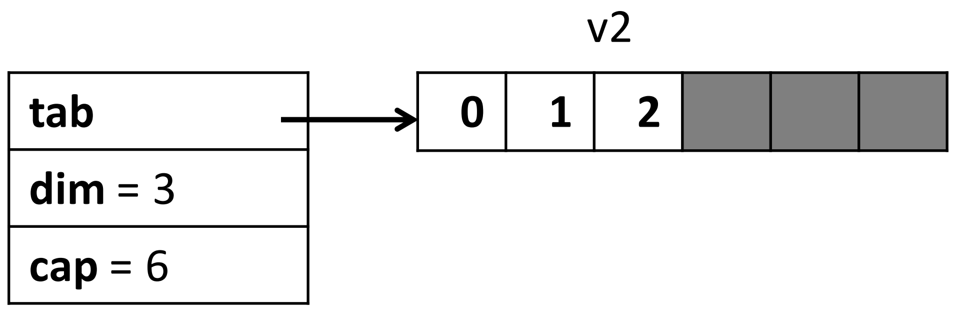


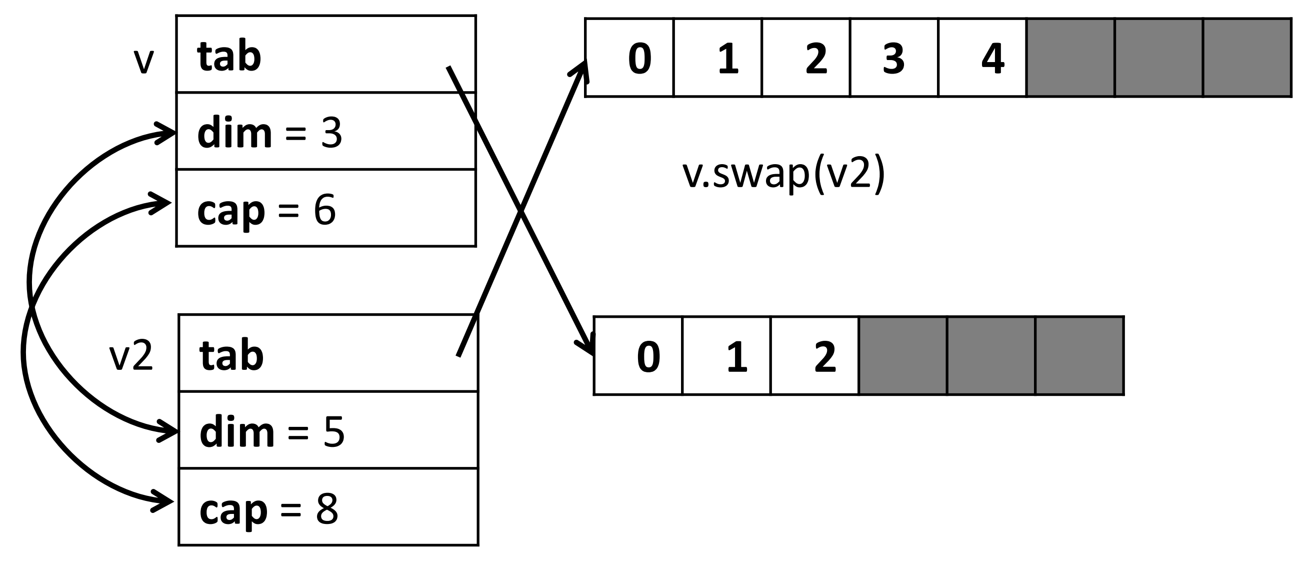
Algorithmes

❑ Modificateurs

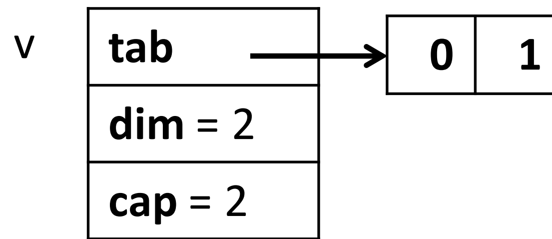
```
void vector::swap(vector& v){
 tab.swap(v.tab);
 dim.swap(v.dim);
 cap.swap(v.cap);
}
```







Algorithmes



❑ Modificateurs

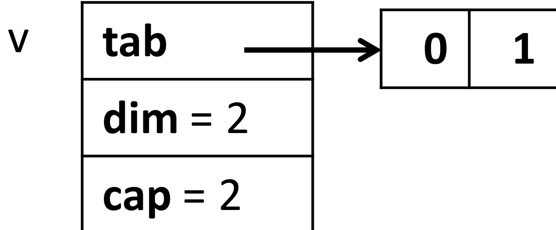
```
void vector::push_back(type&
x){
 if(dim +1 > cap)
   reserve(2*(dim+1));
 dim += 1;
 tab[dim-1] = x ;
}
```

v.push_back(2)

reserve(2*(dim+1));

Algorithmes

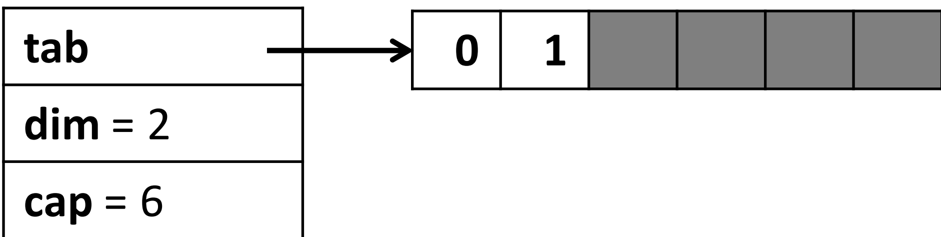
❑ Modificateurs




```
void vector::push_back(type& x){
 if(dim +1 > cap)
   reserve(2*(dim+1));
 dim += 1;
 tab[dim-1] = x ;
}
```

v.push_back(2)

reserve(2*(dim+1));

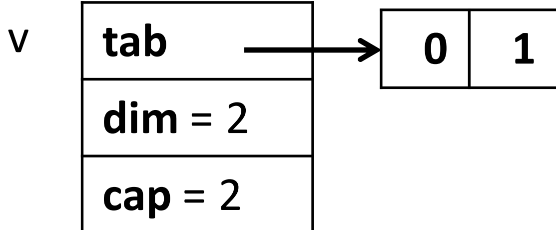


Algorithmes

❑ Modificateurs

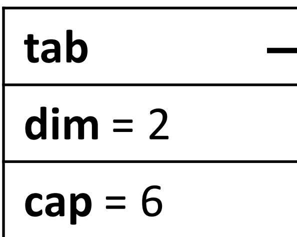
```
void vector::push_back(type& x){
 if(dim + 1 > cap)
   reserve(2*(dim+1));
 dim += 1;
 tab[dim-1] = x ;
}
```

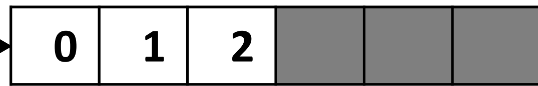
v



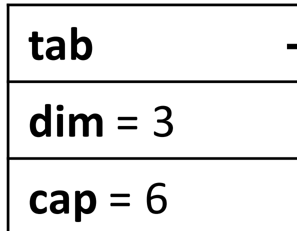
v.push_back(2)

reserve(2*(dim+1));





dim += 1;
tab[dim-1] = x

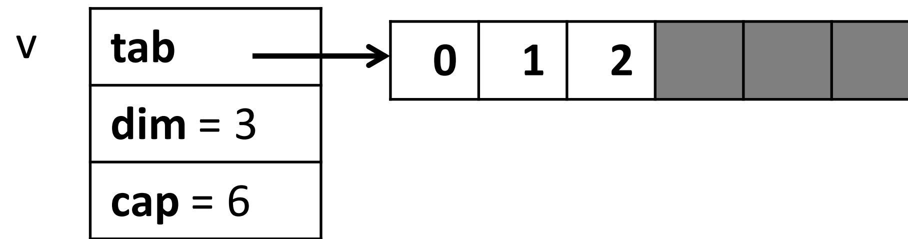


0 1 2

Algorithmes

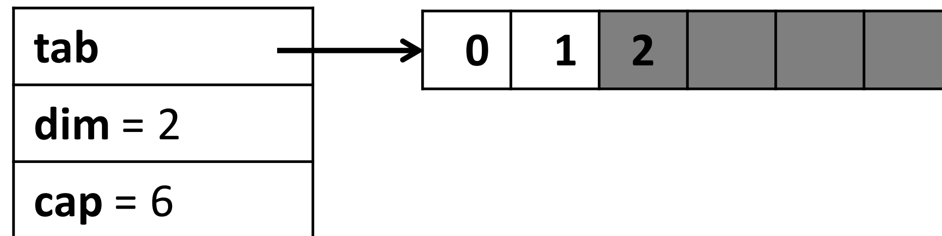
❑ Modificateurs

```
void vector::pop_back(){
 if(dim > 0)
   dim -= 1;
}
```



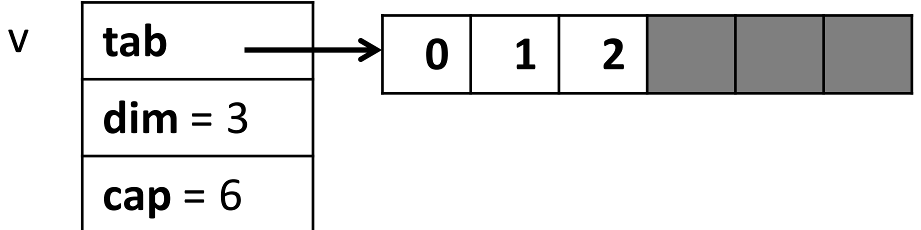
v.pop_back()

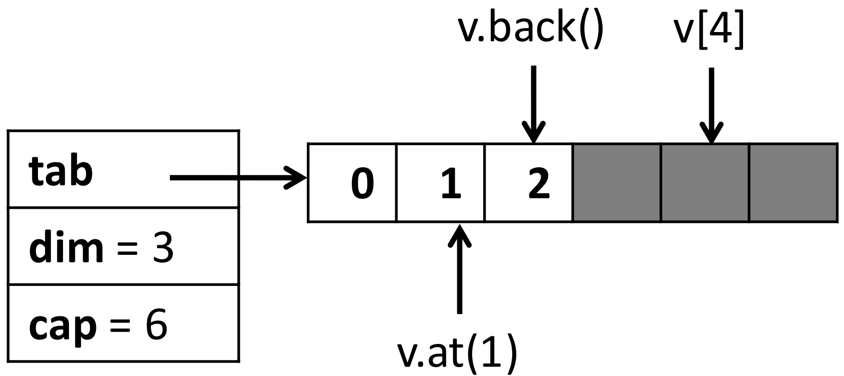
dim -= 1



Algorithmes

❑ Accès






```
type&
vector::operator[](size_t i){
 return tab[i];
}
```

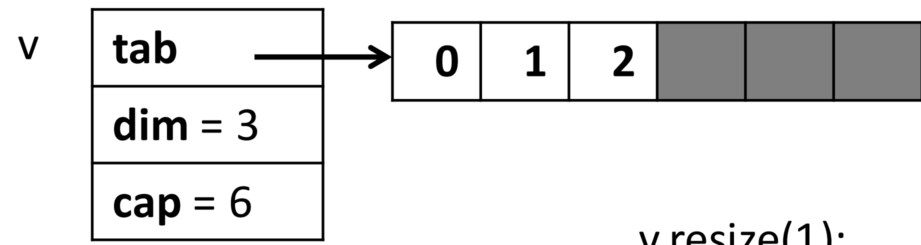
```
type& vector::at(size_t i){
if (i >= size())
   exception;
  else
   return tab [i];
}
```

```
type& vector::back(){
  return tab[size()-1];
}
```

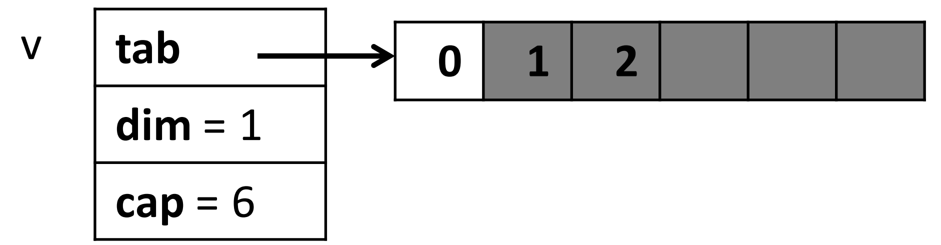
Algorithmes

- **Gestion dimension/capacité**

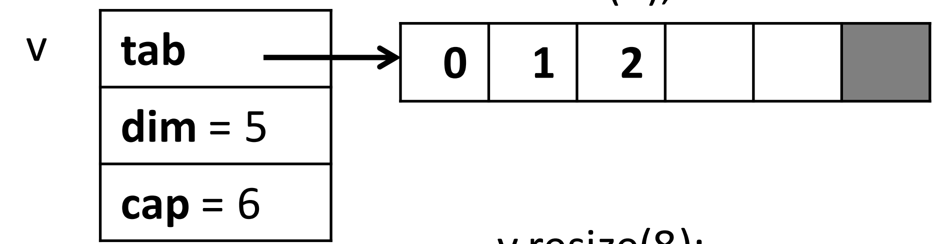
```
void vector::resize(size_t n){
  if(n > cap)
     reserve(n);
  dim = n;
}


size_t vector::size(){return dim;}
```



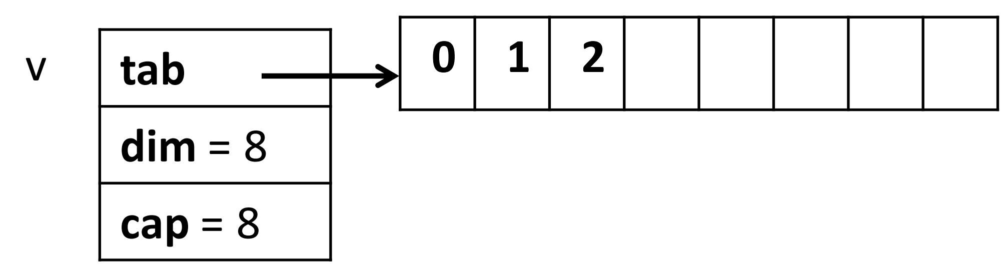
v.resize(1);



v.resize(5);



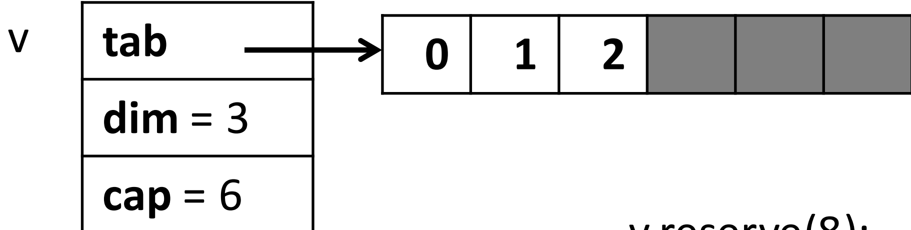
v.resize(8);



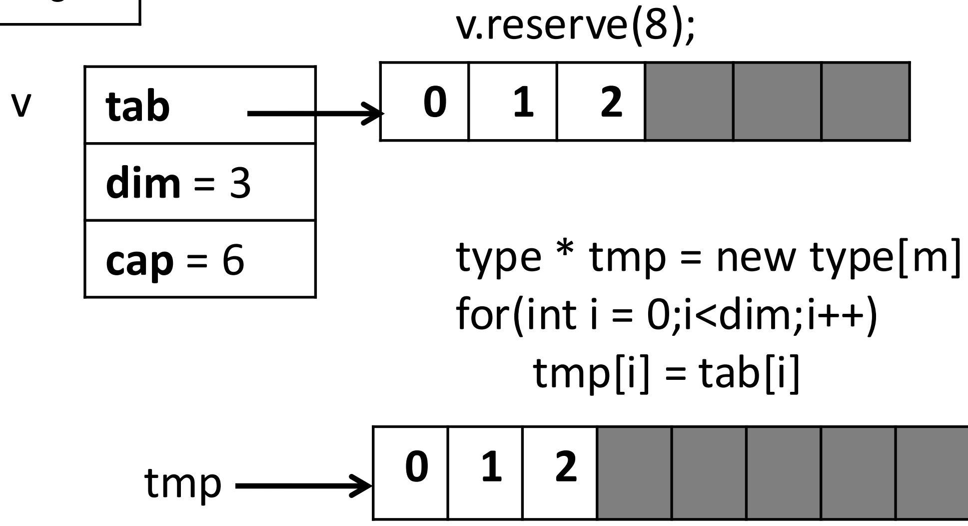
Algorithmes

- **Gestion dimension/capacité**

```
void vector::reserve(size_t m){
 if(m > cap){
  type * tmp = new type[m];
  for(int i=0;i<dim;i++)
     tmp[i] = tab[i]
  delete [] tab;
  tab = tmp;
}

size_t vector::capacity(){return
cap;}
```



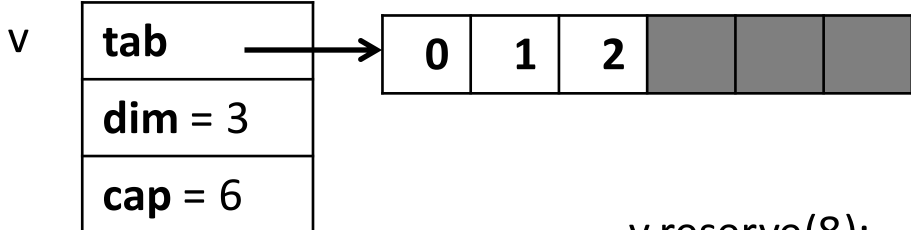
v.reserve(8);

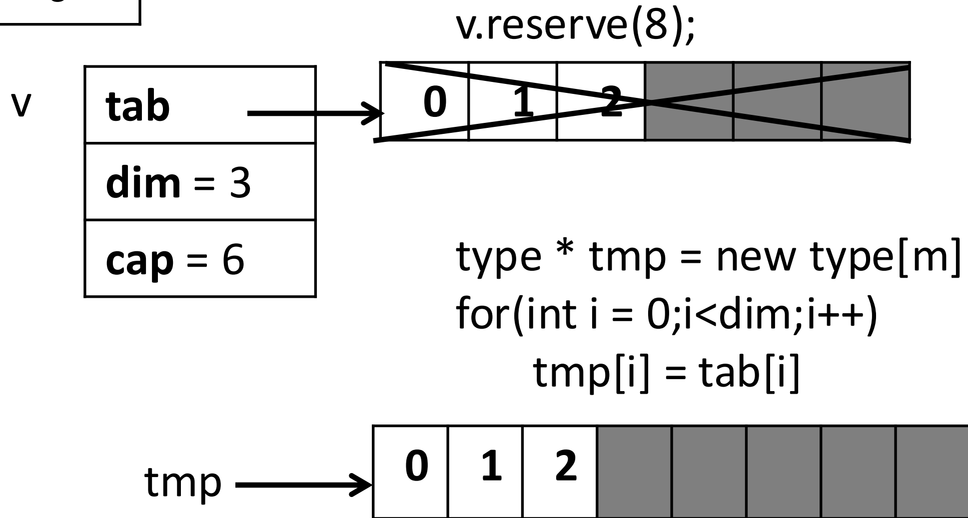


Algorithmes

❑ Gestion dimension/capacité

```
void vector::reserve(size_t m){
 if(m > cap){
   type * tmp = new type[m];
   for(int i = 0;i<dim;i++)
      tmp[i] = tab[i]
   delete [] tab;
   tab = tmp;
}


size_t vector::capacity(){return
cap;}
```

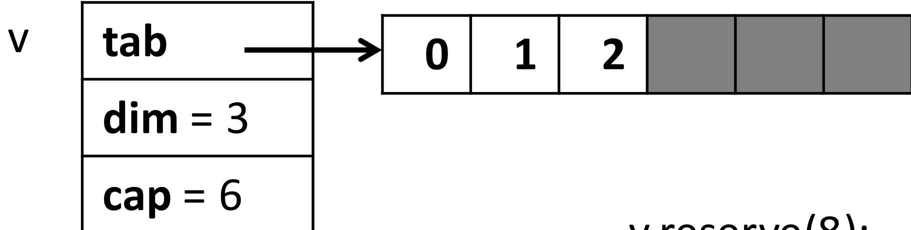


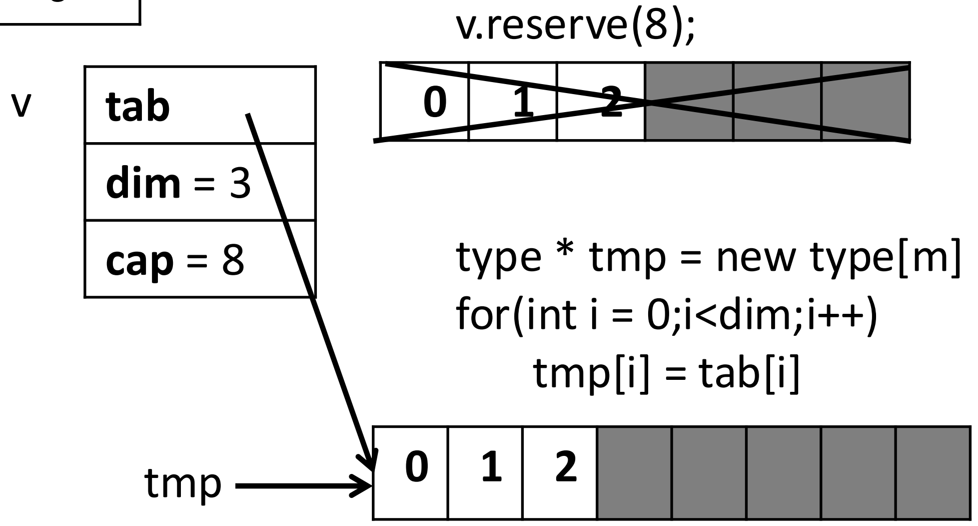


Algorithmes

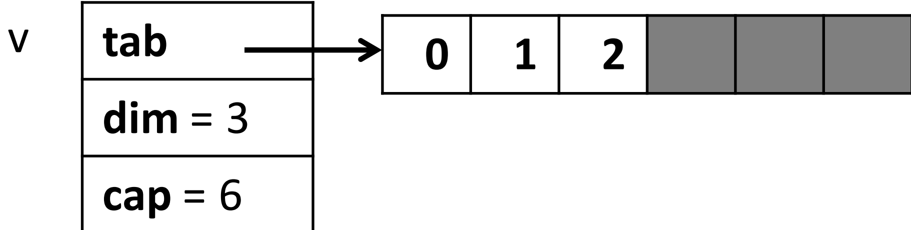
❑ Gestion dimension/capacité

```
void vector::reserve(size_t m){
 if(m > cap){
  type * tmp = new type[m];
  for(int i = 0;i<dim;i++)
     tmp[i] = tab[i]
  delete [] tab;
  tab = tmp;
  cap = m;
}


size_t vector::capacity(){return
cap;}
```



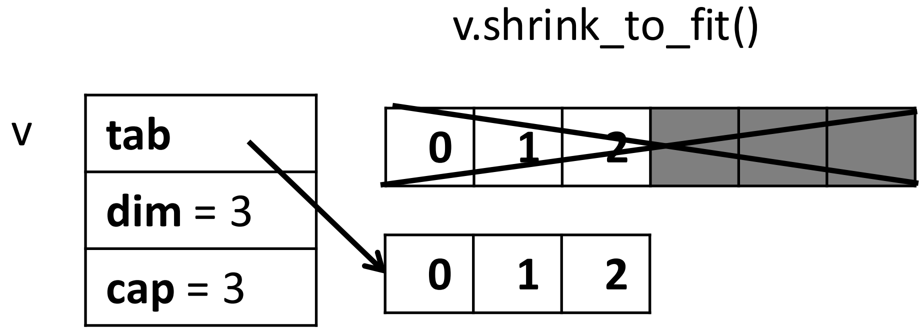


Algorithmes



- **Gestion dimension/capacité**

v.shrink_to_fit()




```
void vector::shrink_to_fit(){
 type * tmp = new type[dim];
 for(int i = 0;i<dim;i++)
      tmp[i] = tab[i]
   delete [] tab;
   tab = tmp;
   cap = dim;
}

bool vector::empty(){return dim ==
0;}
```

Deque

- Éléments semi-contigus en mémoire
- Pas de recopie en cas d'augmentation de capacité
- Occupe plus de mémoire que réellement utilisée
- Accès en O(1) à tout élément à partir de sa position i
- Ajout d'élément à la fin en O(1)
- Ajout d'élément au début en O(1)

Deque

- Éléments semi-contigus en mémoire
- Pas de recopie en cas d'augmentation de capacité
- Occupe plus de mémoire que réellement utilisée
- Accès en O(1) à tout élément à partir de sa position i
- Ajout d'élément à la fin en O(1)
- Ajout d'élément au début en O(1)

Améliorations par rapport au vector

Deque

- ❑ Dimension : nombre d’éléments contenus dans le deque
- ❑ Capacité : non définie explicitement
- ❑ En tout temps : Dimension <= Capacité

Spécifications : Prototypes des opérateurs

- **Constructeurs**

deque : Ø → deque& // par défaut
deque : size_t → deque& // avec paramètre (dimension)
deque : deque& → deque& // par copie

- **Destructeur**

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

- **Affectateur**

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

Spécifications : Prototypes des opérateurs

- **Modificateurs**

swap : deque& → Ø // échange le paramètre avec l'objet appelant
push_back : deque& → Ø // ajoute un élément du <type> à la fin
pop_back: Ø → Ø // retire le dernier élément
push_font : deque& → Ø // ajoute un élément du <type> au début
pop_front: Ø → Ø // retire le premier élément

- **Accès**

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

Spécifications : Prototypes des opérateurs

❑ **Gestion capacité/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
shrink_to_fit : Ø → Ø // libère la mémoire non-utilisée (pas toujours entièrement)
clear : Ø → Ø // libère toute la mémoire allouée dynamiquement

Spécifications : Sémantique des opérateurs (axiomes)

❑ Constructeurs

deque().empty() == Vrai // par défaut

deque(n).size() == n // avec paramètre (dimension)

deque(d).size() == d.size()

et pour tout i, 0 ≤ i < d.size(), deque(d)[i] == d[i] // par copie

❑ Affectateur

d2 = d ~ d2.operator=(d) ; d2.size() == d.size()

et pour tout i, 0 ≤ i < d.size(), d2[i] == d[i] // affectateur

Spécifications : Sémantique des opérateurs (axiomes)

❑ Modificateurs

d11 = d1; d22 = d2; d1.swap(d2); d1 == d22 et d2 == d11 // échange

d.push_back(x)__.back() == x // ajoute un élément à la fin

d.push_back(x)__.pop_back()__ = d // retire le dernier élément

d.push_front(x)__.front() == x // ajoute un élément au début

d.push_front(x)__.pop_front()__ = d // retire le premier élément

❑ Accès

d.operator[](i) ~ d[i] == élément à la position i // accès par position

d.at(i) == élément à la position i si i < d.size() // accès par position en vérifiant la dimension

d.back() == d[d.size()-1] // accès au dernier élément

d.front() == d[0] // accès au premier élément

Spécifications : Sémantique des opérateurs (axiomes)

❑ **Gestion capacité/dimension**

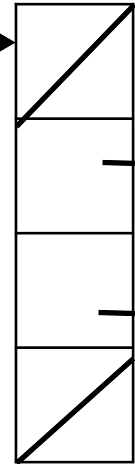
d.resize(n)__.size() == n // change la dimension
d.push_back(x)__.size() == d.size() + 1
d.pop_back()__.size() == d.size() – 1 si = d.size() > 0
d.push_front(x)__.size() == d.size() + 1
d.pop_front()__.size() == d.size() – 1 si = d.size() > 0 // retourne la dimension

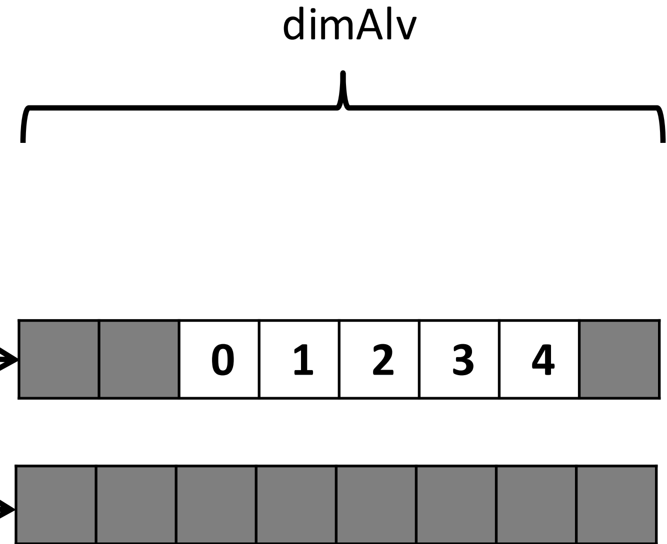
d.empty() = True si et seulement si d.size() == 0
d.shrink_to_fit() // libère la mémoire non-utilisée (pas toujours)
d.clear()__ == deque() // libère la mémoire allouée dynamiquement

Représentation

```
#ifndef _deque_h
#define _deque_h

template <typename TYPE>
class deque
{
 private:
  TYPE **tab; //handle
  size_t dimAlv; //taille
alveole
  size_t nbAlv; // nb alveole
  size_t debut;//index
premier
  size_t dim; //nb elements
 public:
  ...
}
```

tab
dimAlv = 8
nbAlv = 4
debut = 10
dim = 5

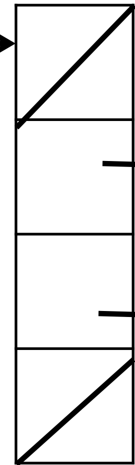


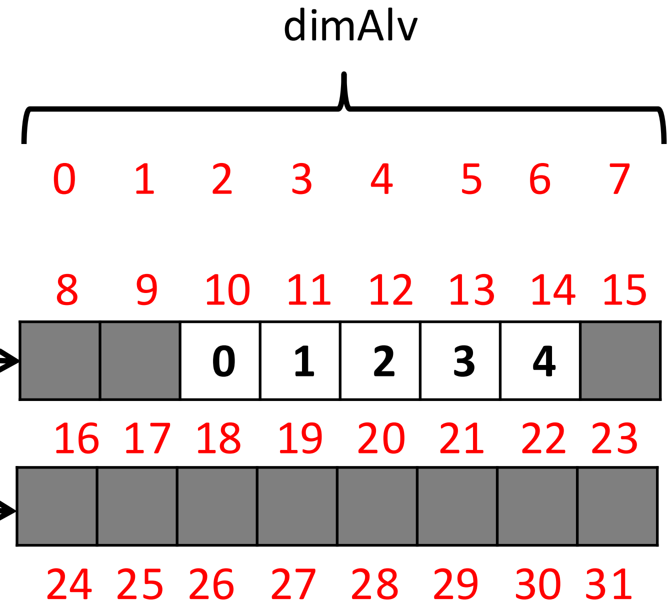


Représentation

```
#ifndef _deque_h
#define _deque_h

template <typename TYPE>
class deque
{
 private:
  TYPE **tab; //handle
  size_t dimAlv; //taille
alveole
  size_t nbAlv; // nb alveole
  size_t debut;//index
premier
  size_t dim; //nb elements
 public:
  ...
}
```

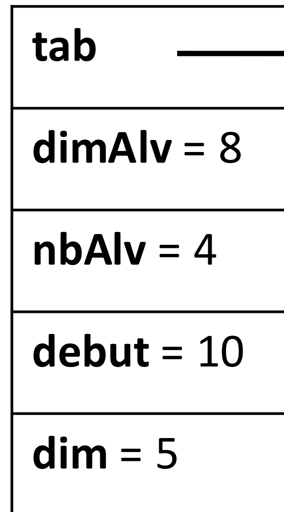
tab
dimAlv = 8
nbAlv = 4
debut = 10
dim = 5

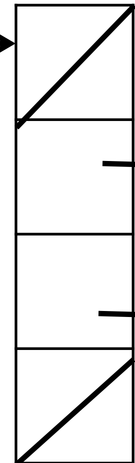


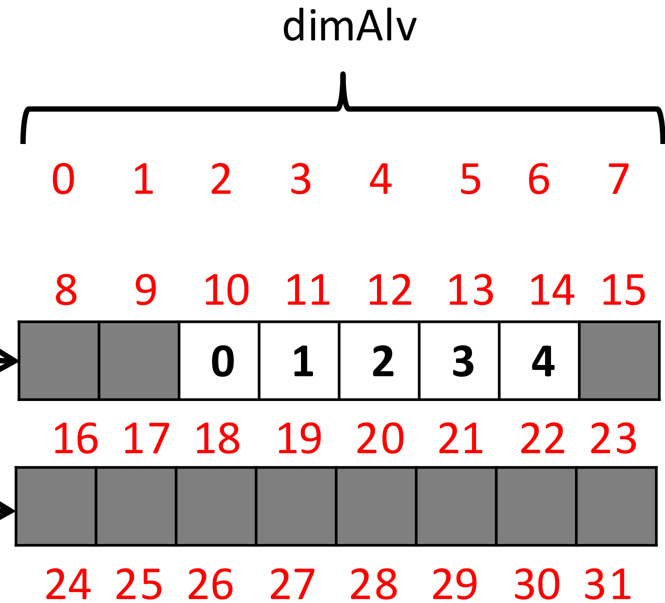


Représentation

```
#ifndef _deque_h
#define _deque_h

template <typename TYPE>
class deque
{
 private:
  TYPE **tab; //handle
  size_t dimAlv; //taille
alveole
  size_t nbAlv; // nb alveole
  size_t debut;//index
premier
  size_t dim; //nb elements
 public:
  ...
}
```







Capacité implicite = 32

Algorithmes

❑ Constructeurs

deque();

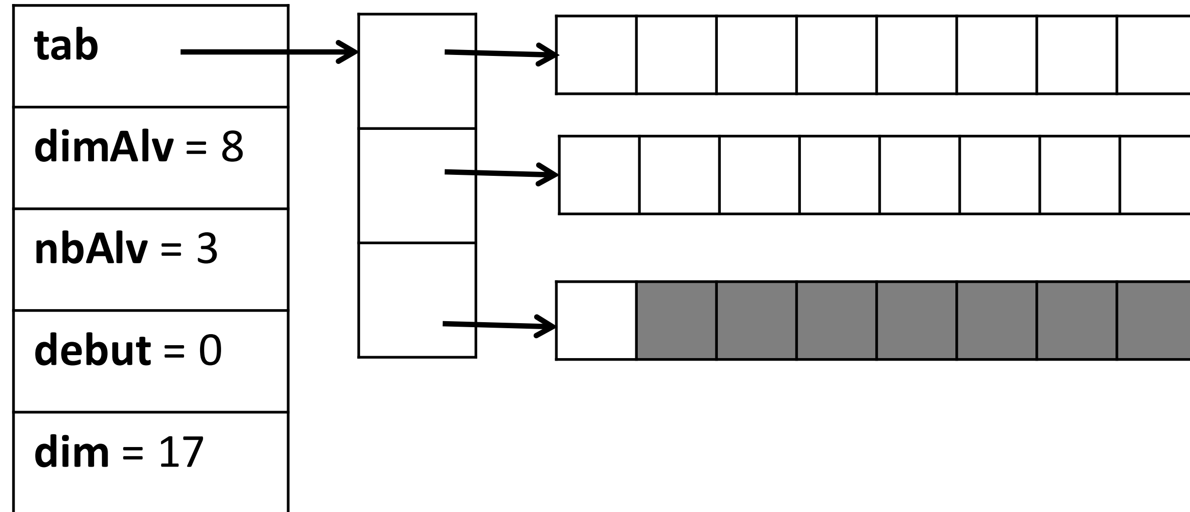
```
deque::deque(){
 tab= nullptr;
 dimAlv = 8;//valeur par
défaut
 nbAlv = debut = dim = 0;
}
```

Tab = nullptr
dimAlv = 8
nbAlv = 0
debut = 0
dim = 0

Algorithmes

❑ Constructeurs

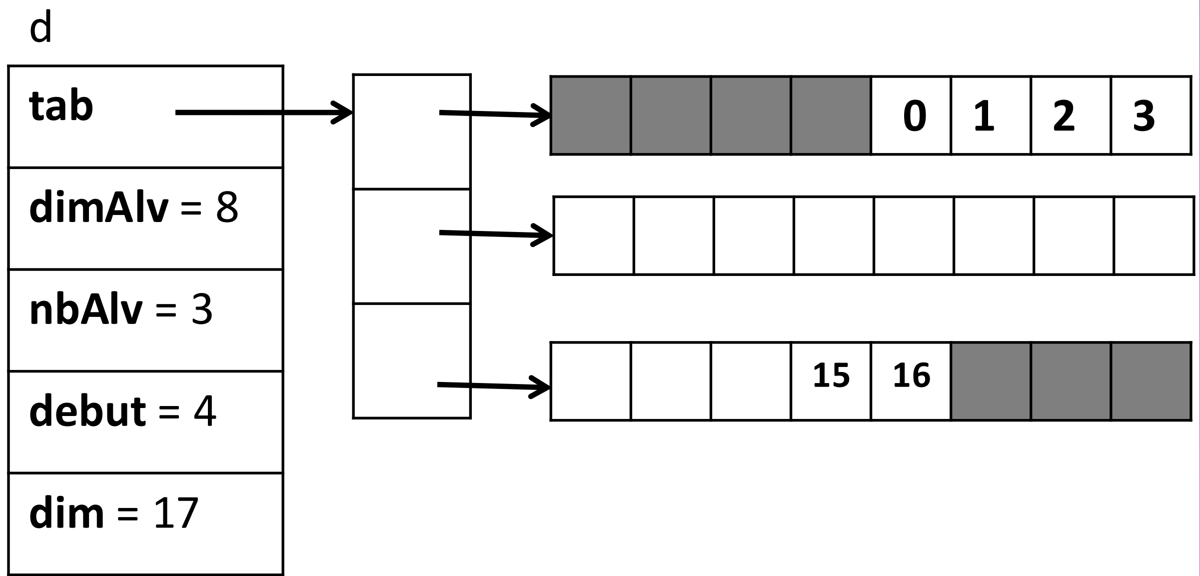
```
deque::deque(size_t n){
 dim = n;
 nbAlv = dim/dimAlv +1;
 tab = new type*[nbAlv];
 for(int i = 0; i< nbAlv;i++)
    tab[i] = new type[dimAlv];
}
```



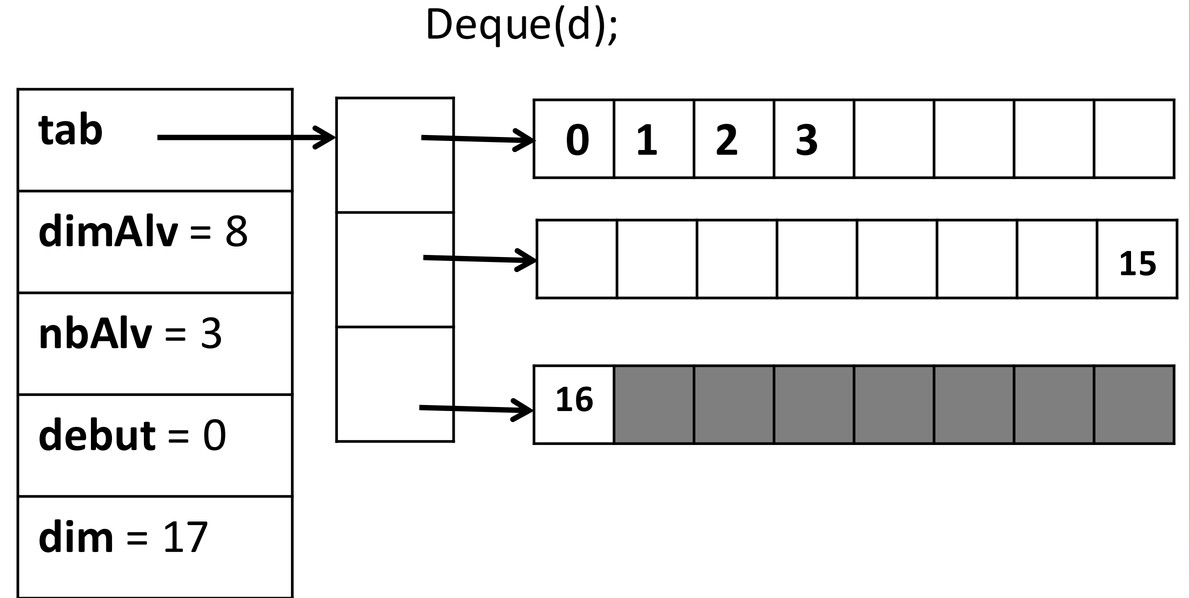
Algorithmes

❑ Constructeurs

```
deque::deque(deque& d){
 dim = d.dim;
 dimAlv = d.dimAlv;
 nbAlv = d.nbAlv;
 debut = 0
 tab = new type*[nbAlv];
 for(int i = 0; i< nbAlv;i++)
   tab[i] = new type[dimAlv];
 for(int i = 0; i< dim;i++)
   at(i) = d.at(i);

}
```



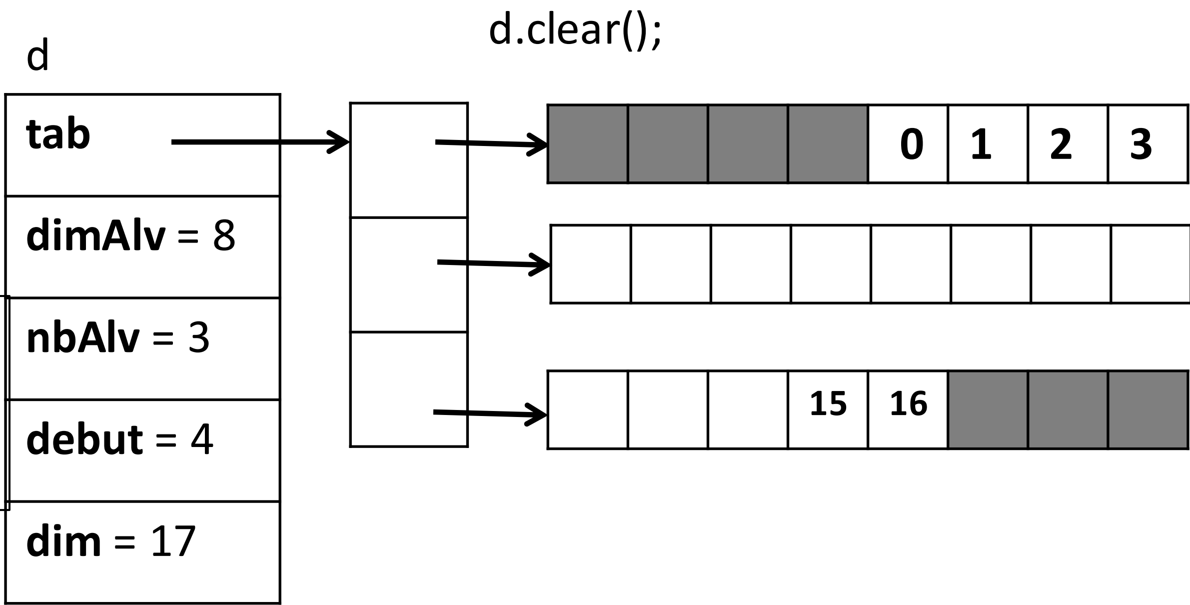
Deque(d);



Algorithmes

❑ Destructeur

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

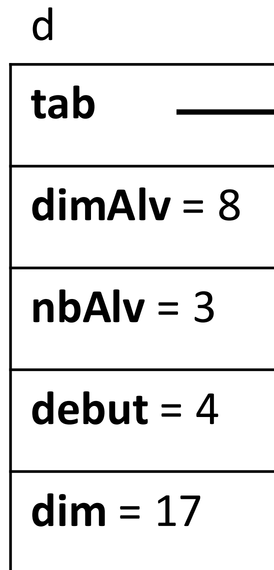


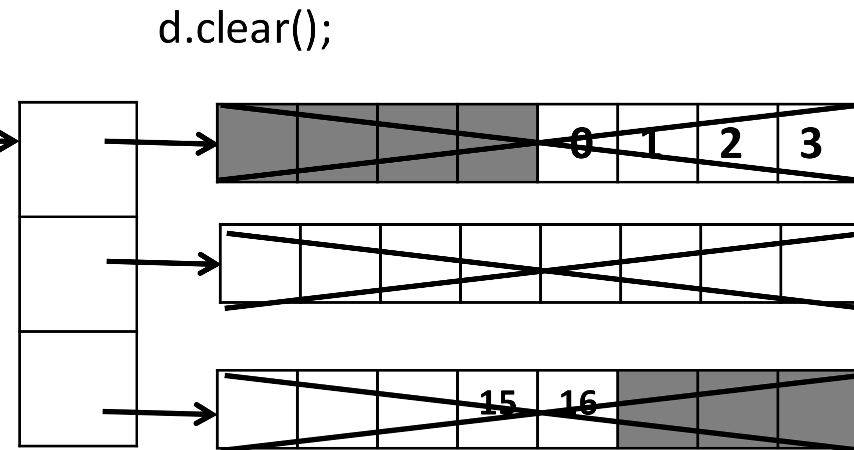
Algorithmes

❑ Destructeur

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

```
deque::clear(){
  for(int i = 0; i< nbAlv;i++)
    delete [] tab[i];
 delete [] tab;
 tab = nullptr;
 nbAlv = debut = dim = 0;
}
```






```
for(int i = 0; i< nbAlv;i++)
  delete [] tab[i];
```

Algorithmes

❑ Destructeur

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

d

tab =nullptr

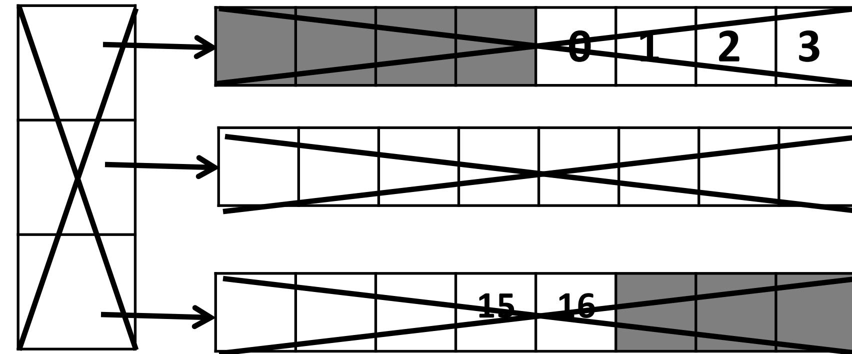
dimAlv = 8

nbAlv = 0

debut = 0

dim = 0

d.clear();



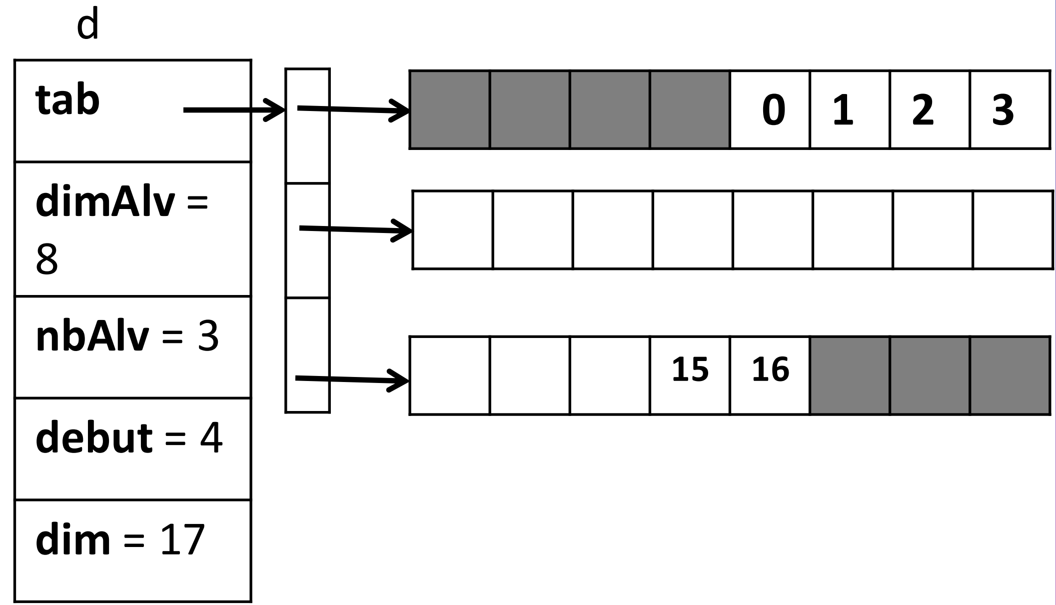

```
deque::clear(){
  for(int i = 0; i< nbAlv;i++)
    delete [] tab[i];
 delete [] tab;
 tab = nullptr;
 dim = nbAlv = 0;
}
```

```
delete [] tab;
tab = nullptr;
dim = cap = 0;
```

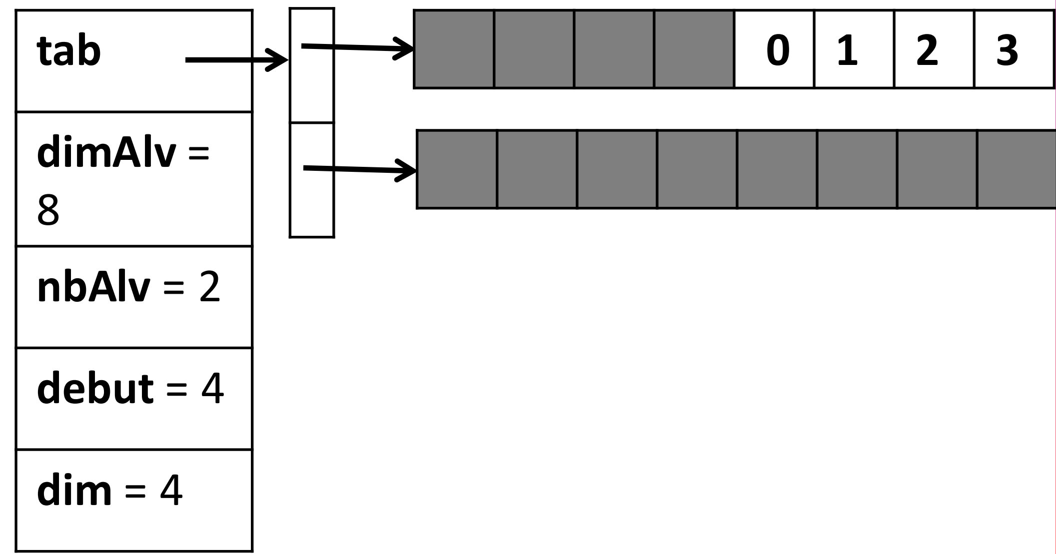
Algorithmes

❑ Affectateur

```
void deque::operator=(deque& d){
 dim = d.dim
 if(nbAlv*dimAlv < dim){
   nvnbAlv = dim/dimAlv +1;
   type **tmp = new
type*[nvnbAlv];
   for(int i = 0; i< nvnbAlv;i++)
      tmp[i] = new type[dimAlv];
   clear();
   tab = tmp;
   nbAlv = nvnbAlv;
 }
 debut = 0;
 for(int i = 0; i< dim;i++){
   if(tab[i/dimAlv] == nullptr)
      tab[i/dimAlv] = new
type[dimAlv];
   at(i) = d.at(i);
```



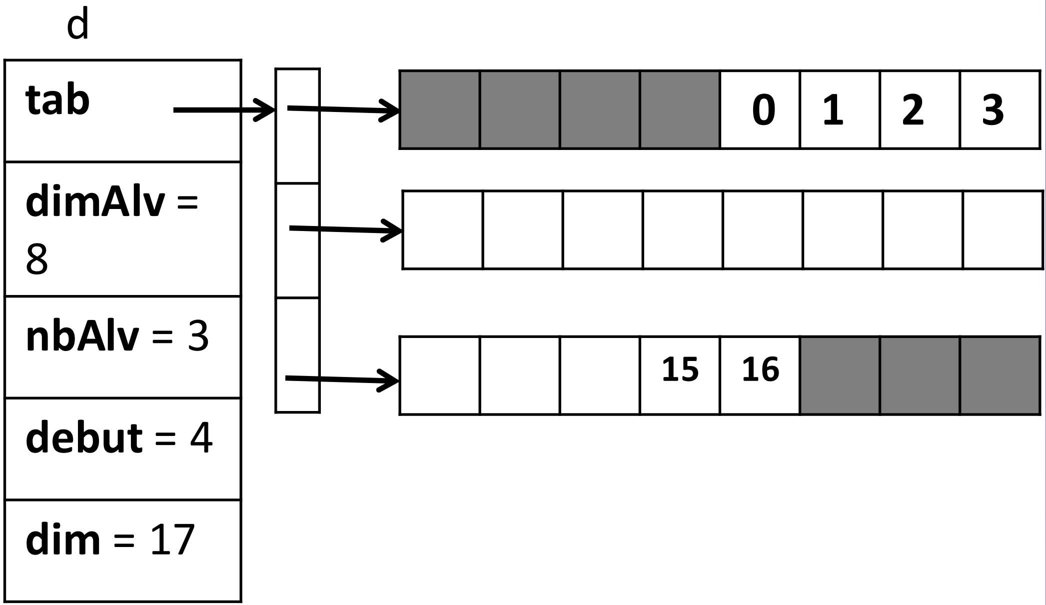
d2 = d;



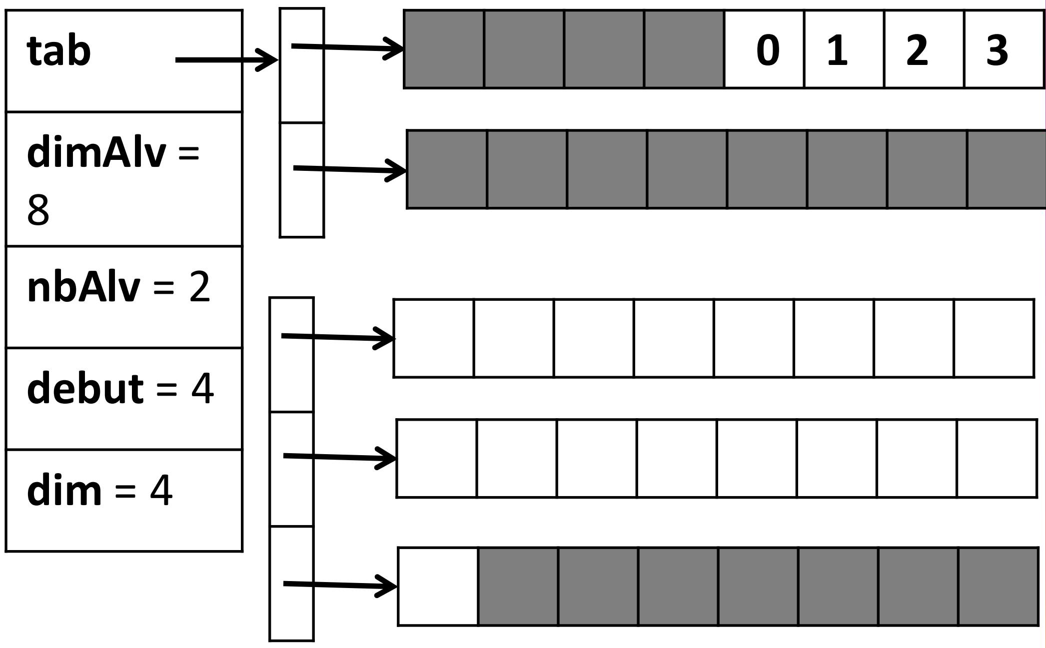
Algorithmes

❑ Affectateur

```
void deque::operator=(deque& d){
 dim = d.dim
 if(nbAlv*dimAlv < dim){
   nvnbAlv = dim/dimAlv +1;
   type **tmp = new
type*[nvnbAlv];
   for(int i = 0; i< nvnbAlv;i++)
      tmp[i] = new type[dimAlv];
   clear();
   tab = tmp;
   nbAlv = nvnbAlv;
 }
 debut = 0;
 for(int i = 0; i< dim;i++){
   if(tab[i/dimAlv] == nullptr)
       tab[i/dimAlv] = new
type[dimAlv];
   at(i) = d.at(i);
```



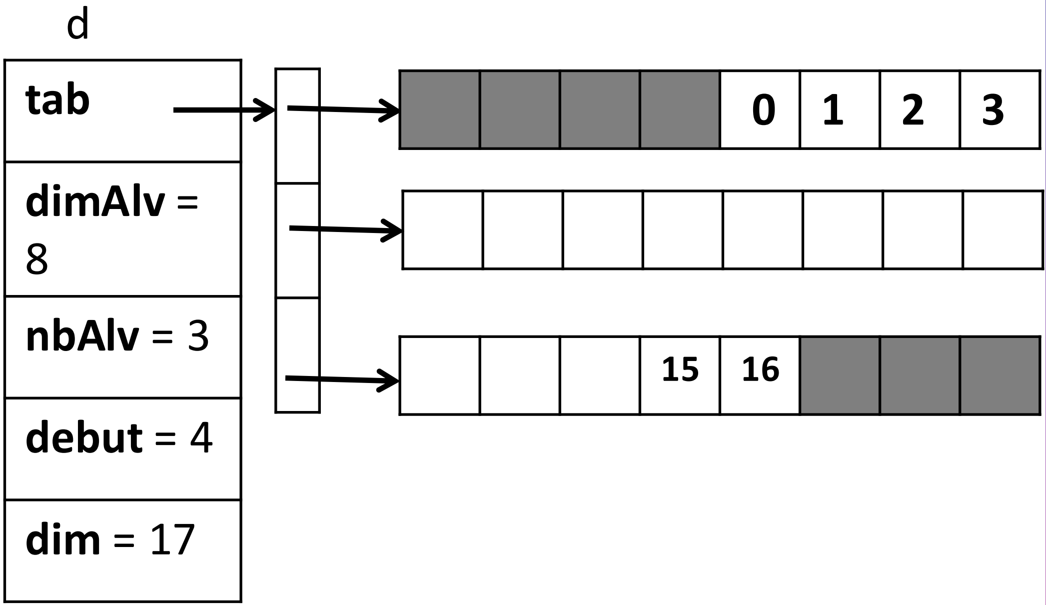
d2 = d;



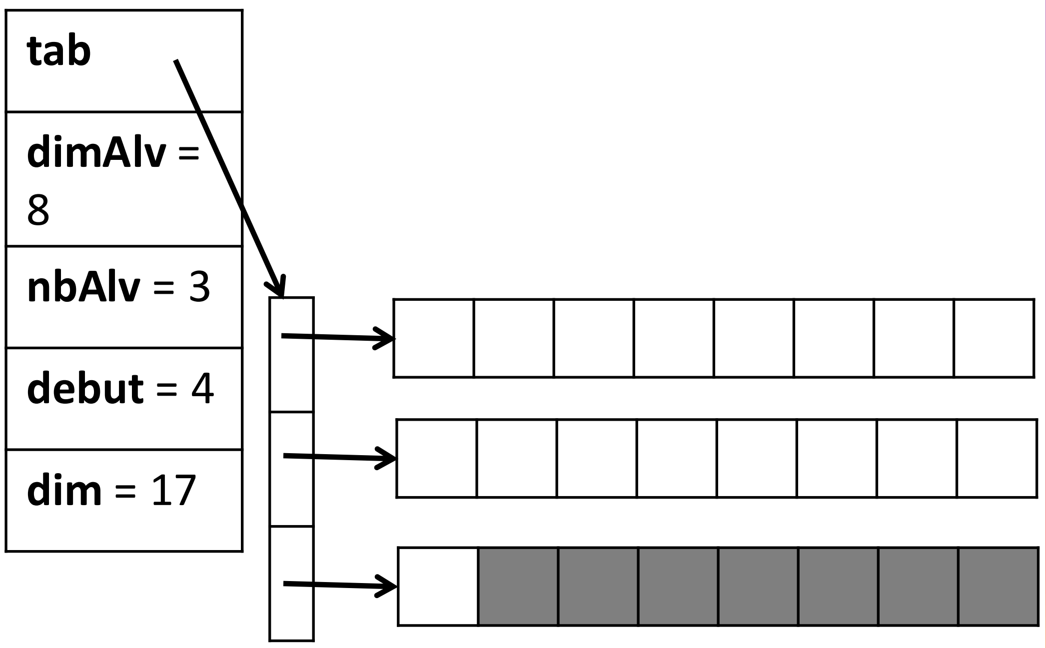
Algorithmes

❑ Affectateur

```
void deque::operator=(deque& d){
 dim = d.dim
 if(nbAlv*dimAlv < dim){
   nvnbAlv = dim/dimAlv +1;
   type **tmp = new
type*[nvnbAlv];
   for(int i = 0; i< nvnbAlv;i++)
      tmp[i] = new type[dimAlv];
   clear();
   tab = tmp;
   nbAlv = nvnbAlv;
 }
 debut = 0;
 for(int i = 0; i< dim;i++){
   if(tab[i/dimAlv] == nullptr)
       tab[i/dimAlv] = new
type[dimAlv];
    at(i) = d.at(i);
```



d2 = d;



Algorithmes

❑ Affectateur

```
void deque::operator=(deque& d){
 dim = d.dim
 if(nbAlv*dimAlv < dim){
   nvnbAlv = dim/dimAlv +1;
   type **tmp = new
type*[nvnbAlv];
   for(int i = 0; i< nvnbAlv;i++)
      tmp[i] = new type[dimAlv];
   clear();
   tab = tmp;
   nbAlv = nvnbAlv;
 }
 debut = 0;
 for(int i = 0; i< dim;i++){
   if(tab[i/dimAlv] == nullptr)
       tab[i/dimAlv] = new
type[dimAlv];
    at(i) = d.at(i);
```

d

tab
dimAlv = 8
nbAlv = 3
debut = 4
dim = 17

				0	1	2	3
			15	16			

d2 = d;

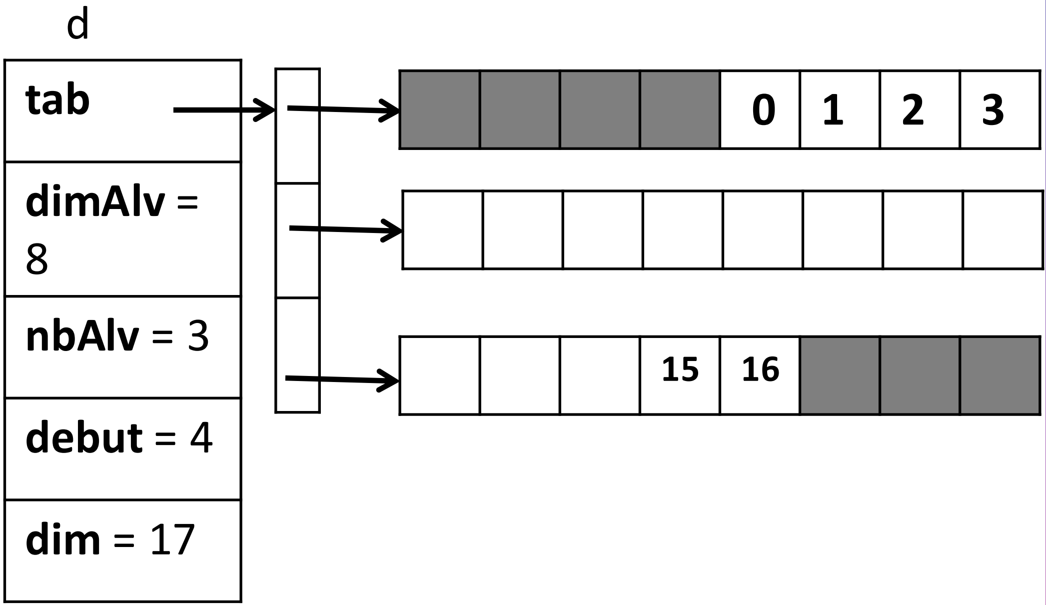
tab
dimAlv = 8
nbAlv = 3
debut = 0
dim = 4

0	1	2	3				
16							

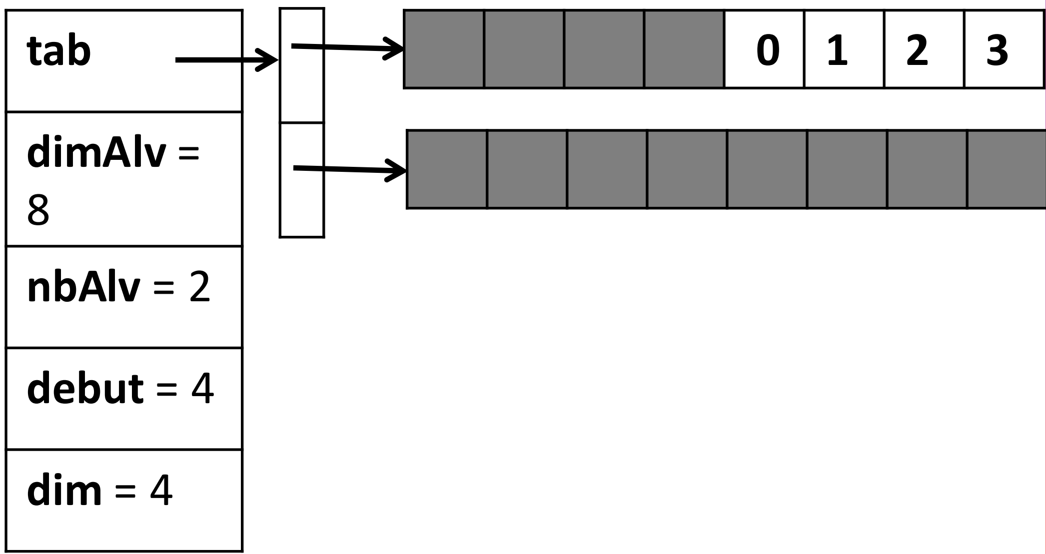
Algorithmes

❑ Modificateurs

```
boid deque::swap(deque&
vd){
 tab.swap(d.tab);
 dimAlv.swap(d.dimAlv);
 nbAlv.swap(d.nbAlv);
 debut.swap(d.debut);
 dim.swap(d.dim);
}
```



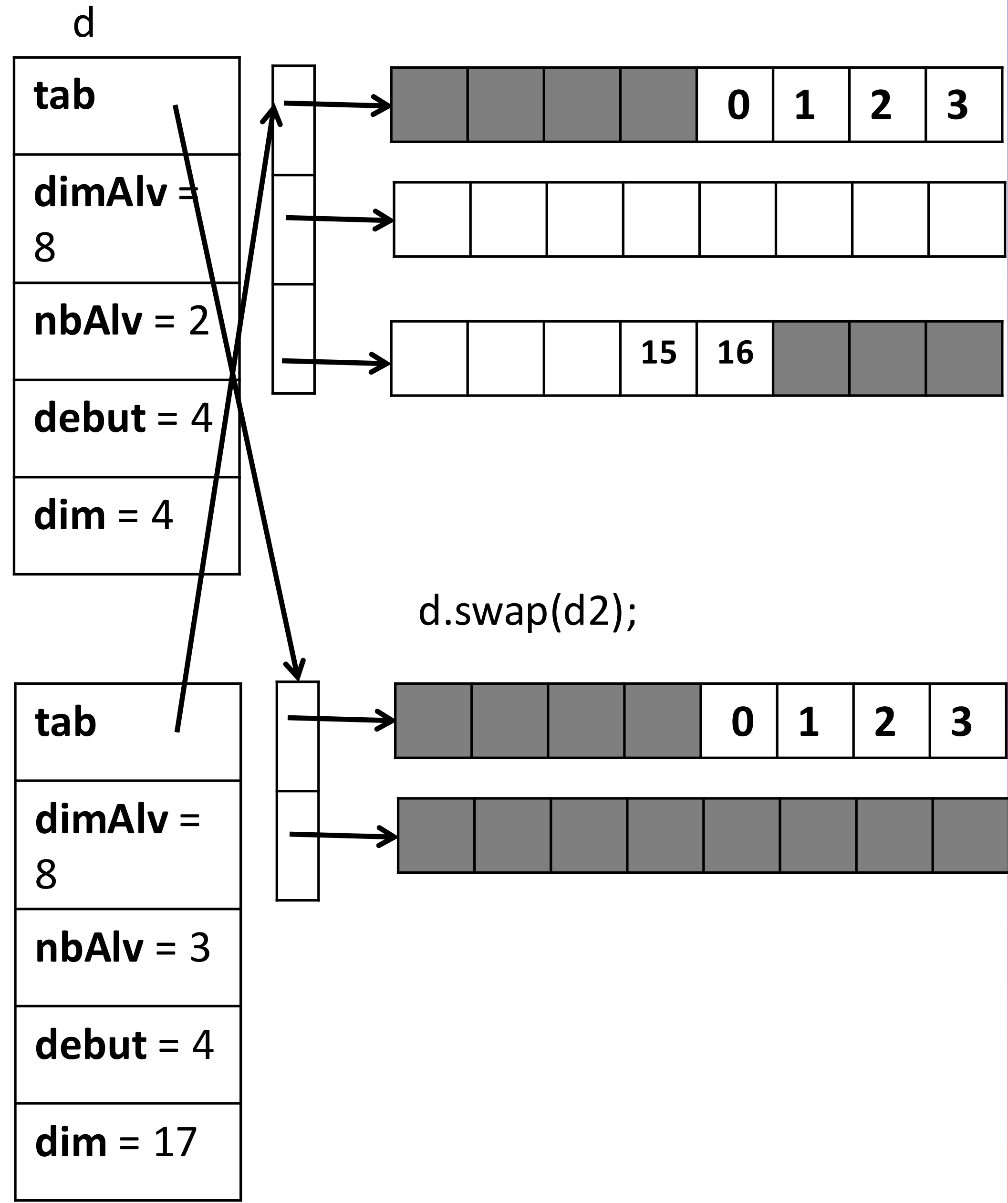
d.swap(d2);



Algorithmes

❑ Modificateurs

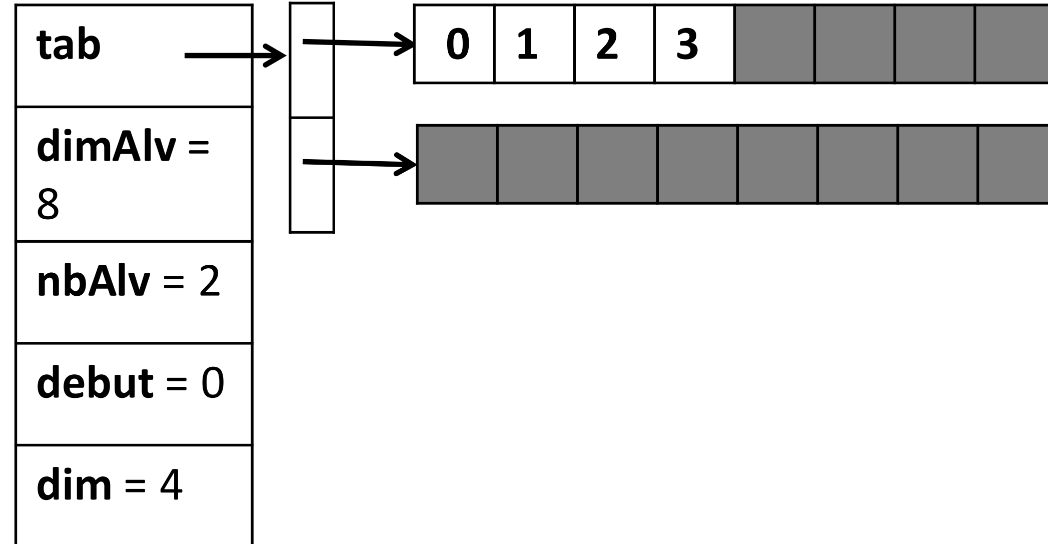
```
boid deque::swap(deque&
vd){
 tab.swap(d.tab);
 dimAlv.swap(d.dimAlv);
 nbAlv.swap(d.nbAlv);
 debut.swap(d.debut);
 dim.swap(d.dim);
}
```



Algorithmes

❑ Modificateurs

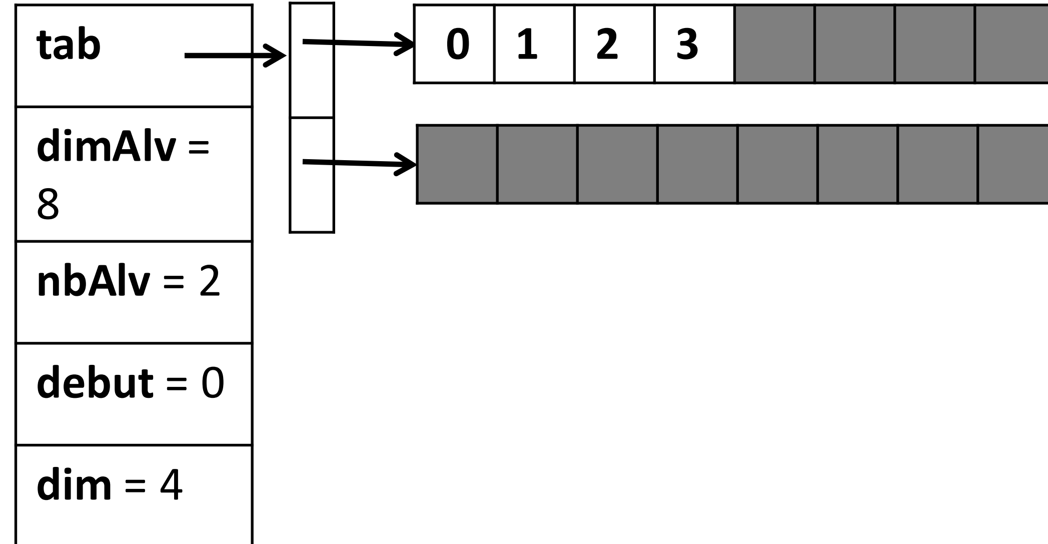
```
void deque::push_front(type& x){
if(debut ==0){
  type **tmp = new
type*[2*nbAlv];
  for(int i = 0; i< nbAlv;i++)
      tmp[nbAlv+i] = tab[i];
  delete [] tab;
  tab = tmp;
  debut = debut + nbAlv*dimAlv;
  nbAlv = nbAlv*2;}
debut = debut -1;
alv = debut/dimAlv
cell = debut%dimAlv;
If(tab[alv] == nullptr)
  tab[alv] = new type[dimAlv]
tab[alv][cell] = x;
dim = dim +1
}
```



Algorithmes

❑ Modificateurs

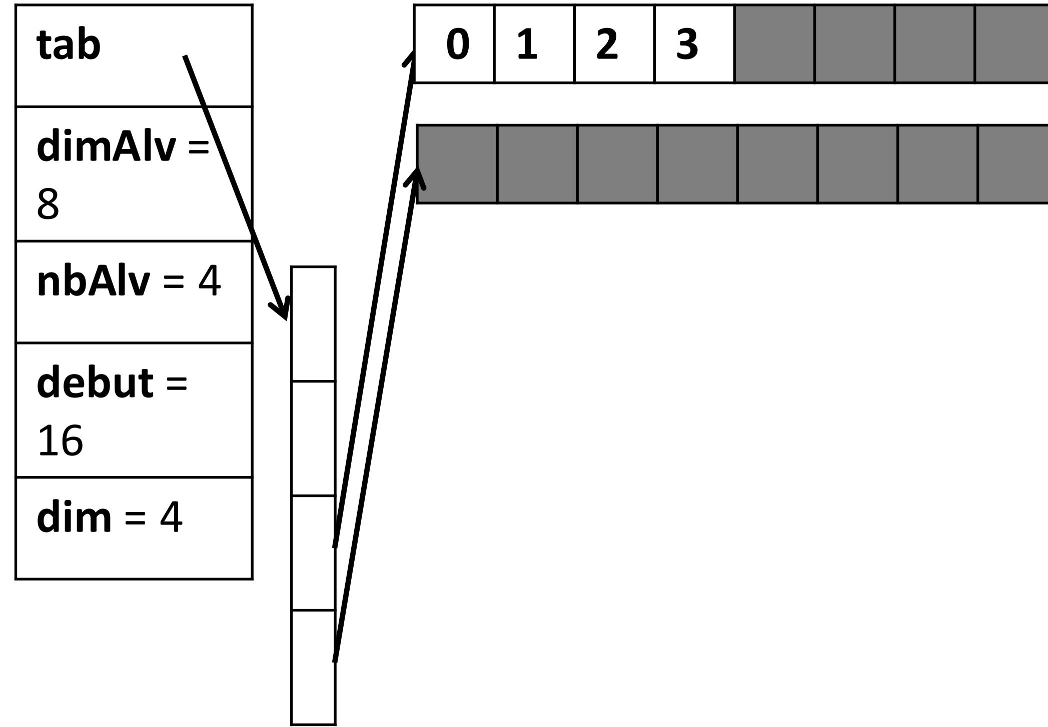
```
void deque::push_front(type& x){
if(debut ==0){
  type **tmp = new
type*[2*nbAlv];
  for(int i = 0; i< nbAlv;i++)
     tmp[nbAlv+i] = tab[i];
  delete [] tab;
  tab = tmp;
  debut = debut + nbAlv*dimAlv;
  nbAlv = nbAlv*2;}
debut = debut -1;
alv = debut/dimAlv
cell = debut%dimAlv;
If(tab[alv] == nullptr)
  tab[alv] = new type[dimAlv]
tab[alv][cell] = x;
dim = dim +1
}
```



Algorithmes

❑ Modificateurs

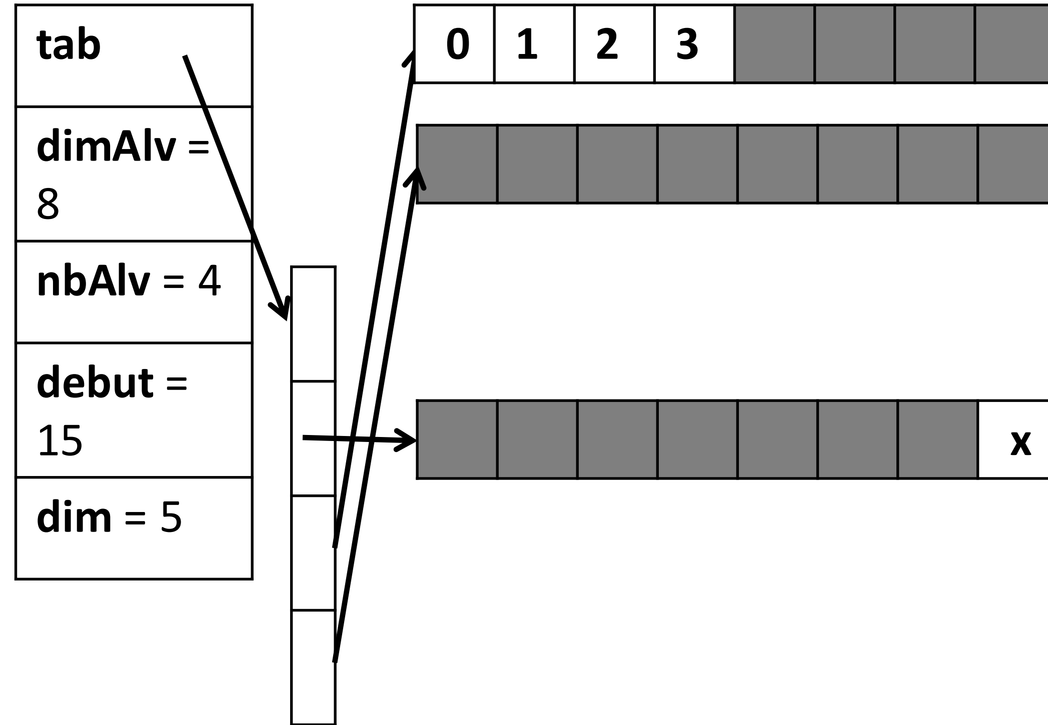
```
void deque::push_front(type& x){
if(debut ==0){
  type **tmp = new
type*[2*nbAlv];
  for(int i = 0; i< nbAlv;i++)
     tmp[nbAlv+i] = tab[i];
  delete [] tab;
  tab = tmp;
  debut = debut + nbAlv*dimAlv;
  nbAlv = nbAlv*2;}
debut = debut -1;
alv = debut/dimAlv
cell = debut%dimAlv;
If(tab[alv] == nullptr)
  tab[alv] = new type[dimAlv]
tab[alv][cell] = x;
dim = dim +1
}
```



Algorithmes

❑ Modificateurs

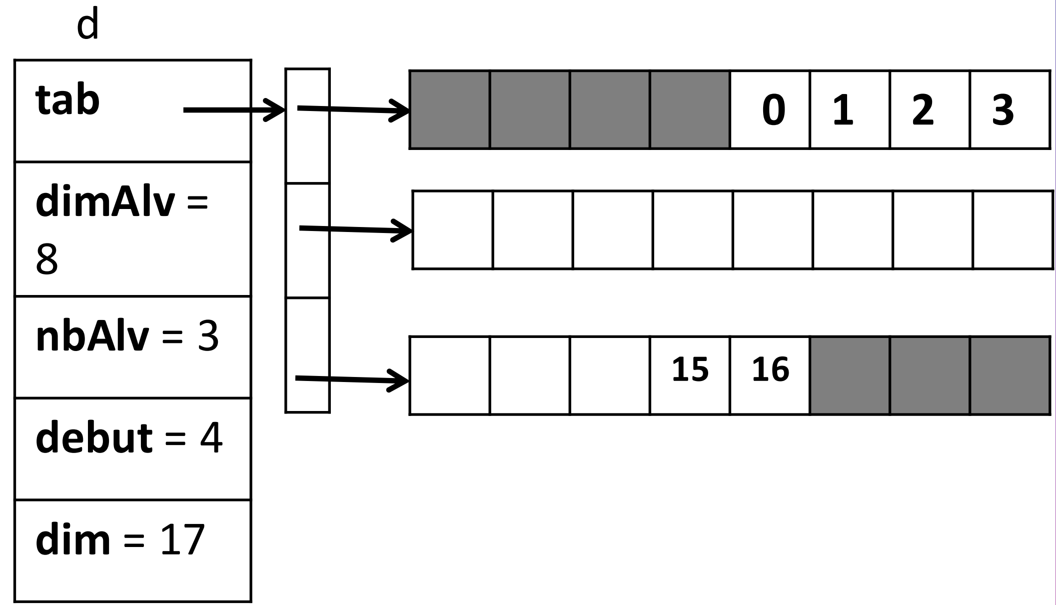
```
void deque::push_front(type& x){
if(debut ==0){
  type **tmp = new
type*[2*nbAlv];
  for(int i = 0; i< nbAlv;i++)
     tmp[nbAlv+i] = tab[i];
  delete [] tab;
  tab = tmp;
  debut = debut + nbAlv*dimAlv;
  nbAlv = nbAlv*2;}
debut = debut -1;
alv = debut/dimAlv
cell = debut%dimAlv;
If(tab[alv] == nullptr)
  tab[alv] = new type[dimAlv]
tab[alv][cell] = x;
dim = dim +1
}
```

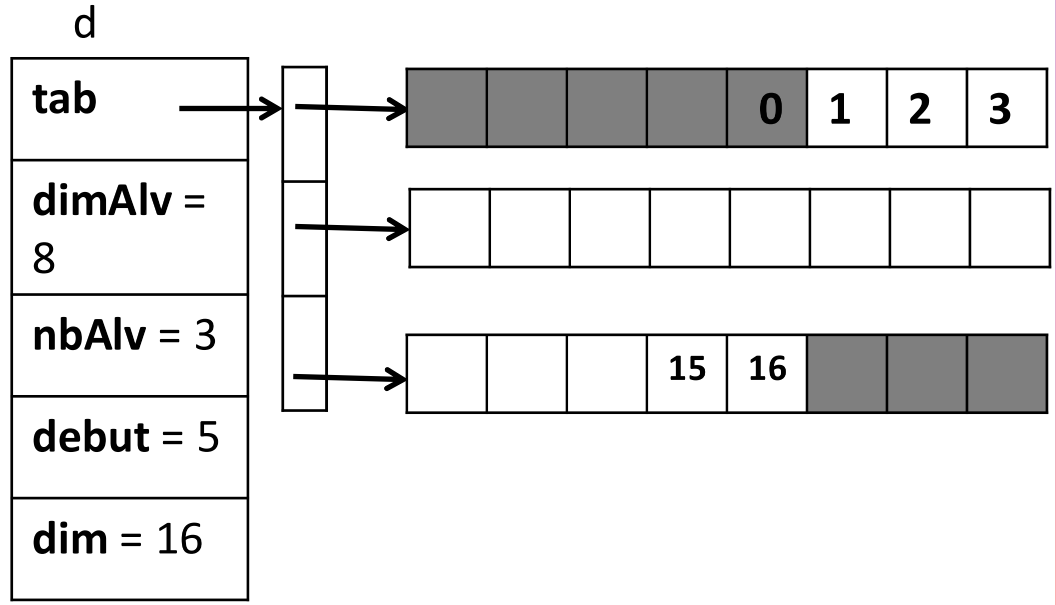


Algorithmes

❑ Modificateurs

```
void deque::pop_front(){
 if(dim > 0)
   debut += 1
   dim -= 1;
}
```

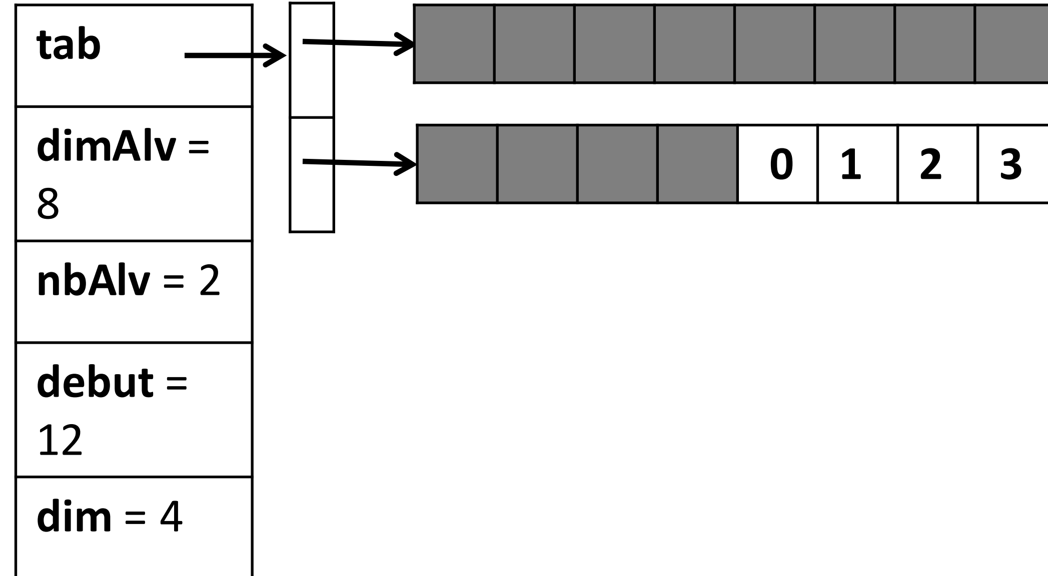




Algorithmes

❑ Modificateurs

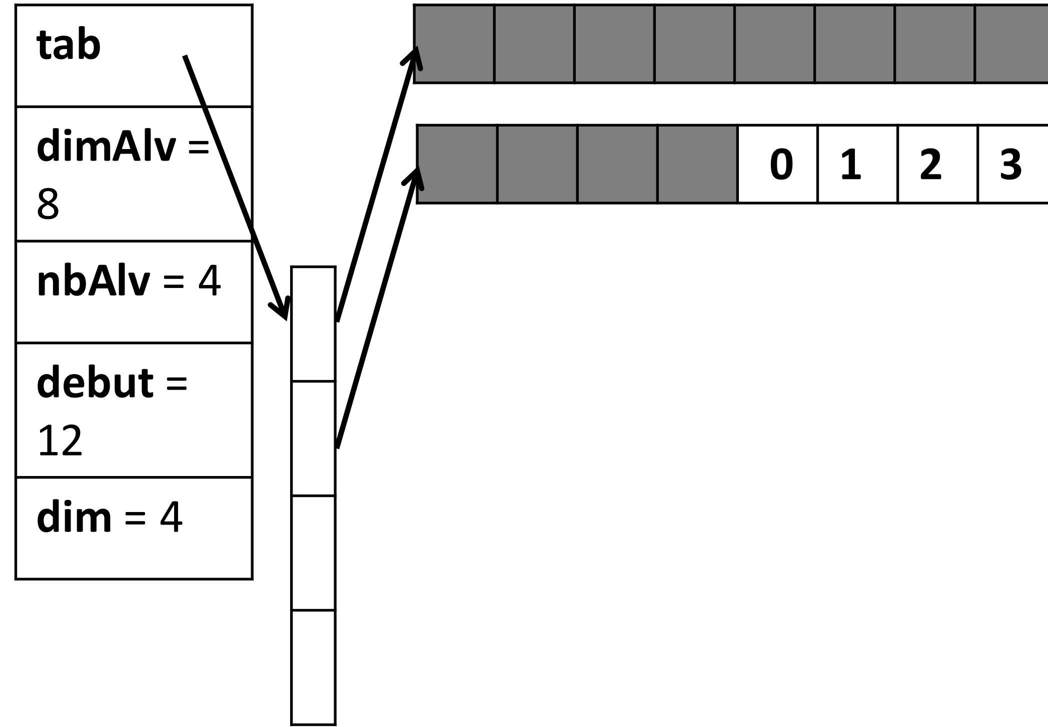
```
void deque::push_back(type& x){
if(debut +dim-1==nbAlv*dimAlv-1){
  type **tmp = new
type*[2*nbAlv];
  for(int i = 0; i< nbAlv;i++)
     tmp[i] = tab[i];
  delete [] tab;
  tab = tmp;
  nbAlv = nbAlv*2;}
size_t back = debut+dim;
alv = back/dimAlv
cell = back%dimAlv;
If(tab[alv] == nullptr)
  tab[alv] = new type[dimAlv]
tab[alv][cell] = x;
dim = dim +1
}
```



Algorithmes

❑ Modificateurs

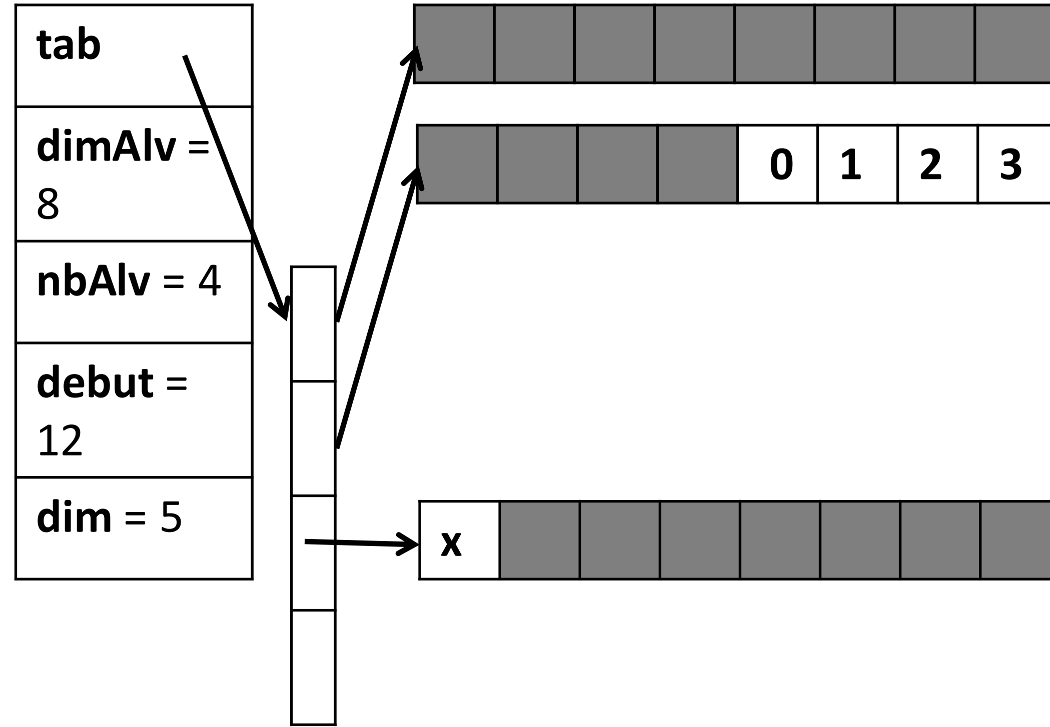
```
void deque::push_back(type& x){
if(debut +dim-1==nbAlv*dimAlv-1){
  type **tmp = new
type*[2*nbAlv];
  for(int i = 0; i< nbAlv;i++)
     tmp[i] = tab[i];
  delete [] tab;
  tab = tmp;
  nbAlv = nbAlv*2;}
size_t back = debut+dim;
alv = back/dimAlv
cell = back%dimAlv;
if(tab[alv] == nullptr)
  tab[alv] = new type[dimAlv]
tab[alv][cell] = x;
dim = dim +1
}
```



Algorithmes

❑ Modificateurs

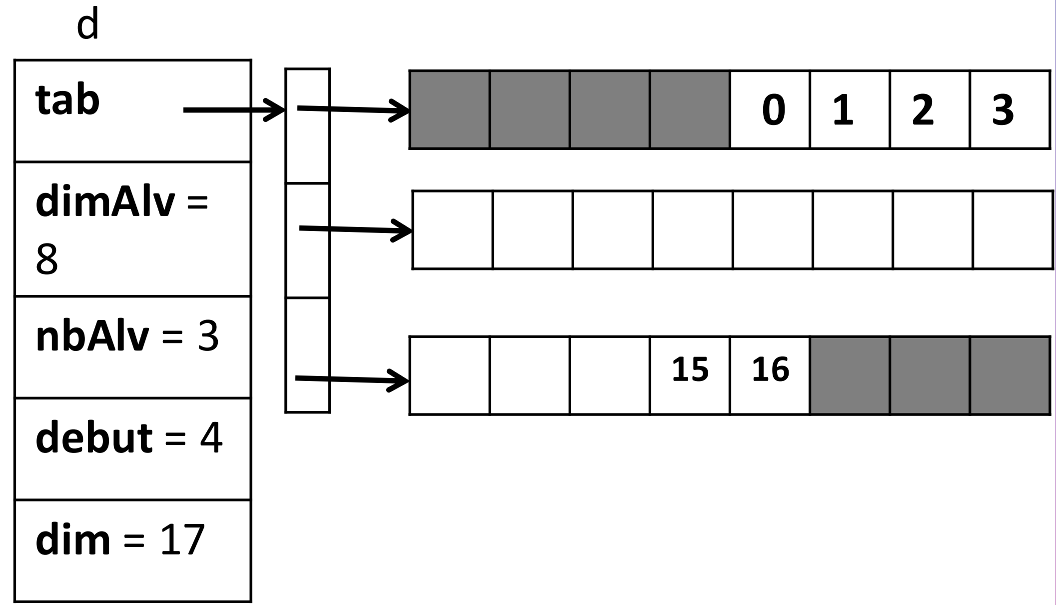
```
void deque::push_back(type& x){
if(debut +dim-1==nbAlv*dimAlv-1){
  type **tmp = new
type*[2*nbAlv];
  for(int i = 0; i< nbAlv;i++)
     tmp[i] = tab[i];
  delete [] tab;
  tab = tmp;
  nbAlv = nbAlv*2;}
size_t back = debut+dim;
alv = back/dimAlv
cell = back%dimAlv;
If(tab[alv] == nullptr)
  tab[alv] = new type[dimAlv]
tab[alv][cell] = x;
dim = dim +1
}
```

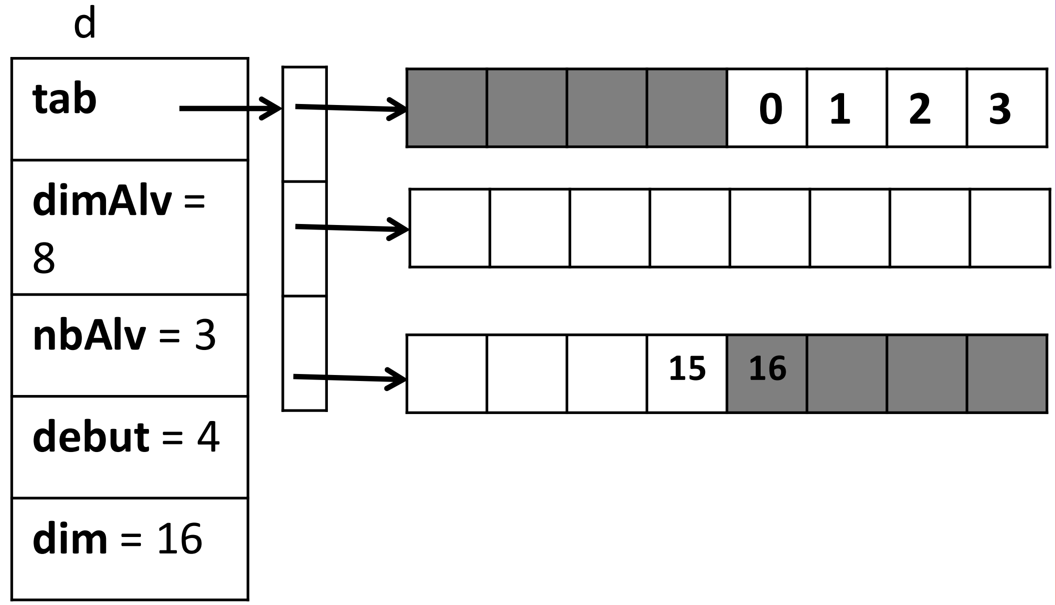


Algorithmes

❑ Modificateurs

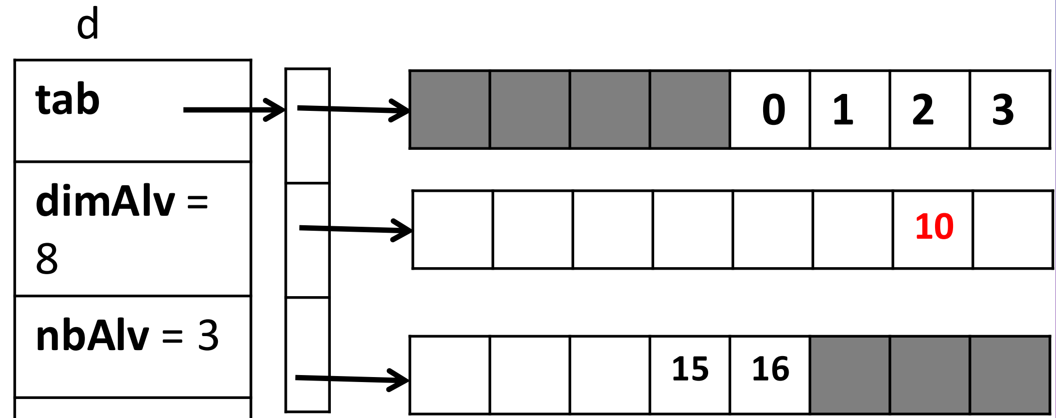
```
void deque::pop_back(){
 if(dim > 0)
   dim -= 1;
}
```





Algorithmes

- **Accès**

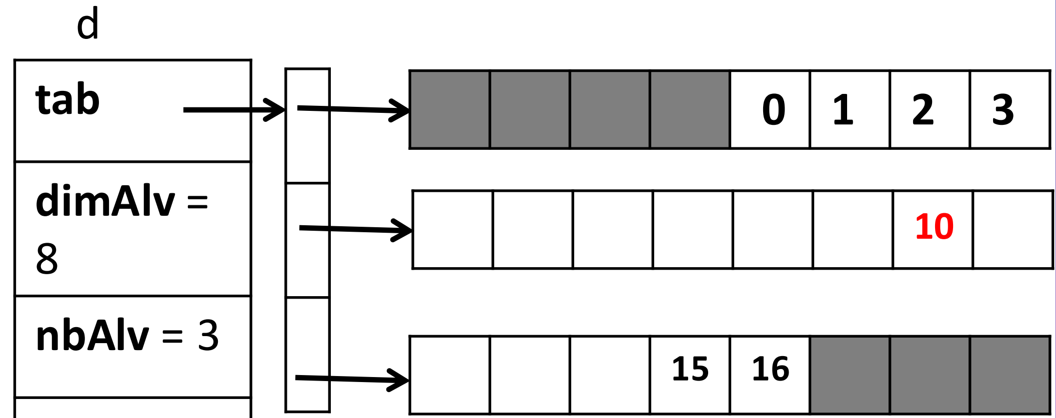



```
type&
deque::operator[](size_t i){
  pos = debut+i;
  alv = pos/dimAlv
  cell = pos%dimAlv;
  return tab[alv][cell];
}
```

D[10]
i+debut = 14
alv = 14/8 =1
cell = 14%8 = 6

Algorithmes

❑ Accès



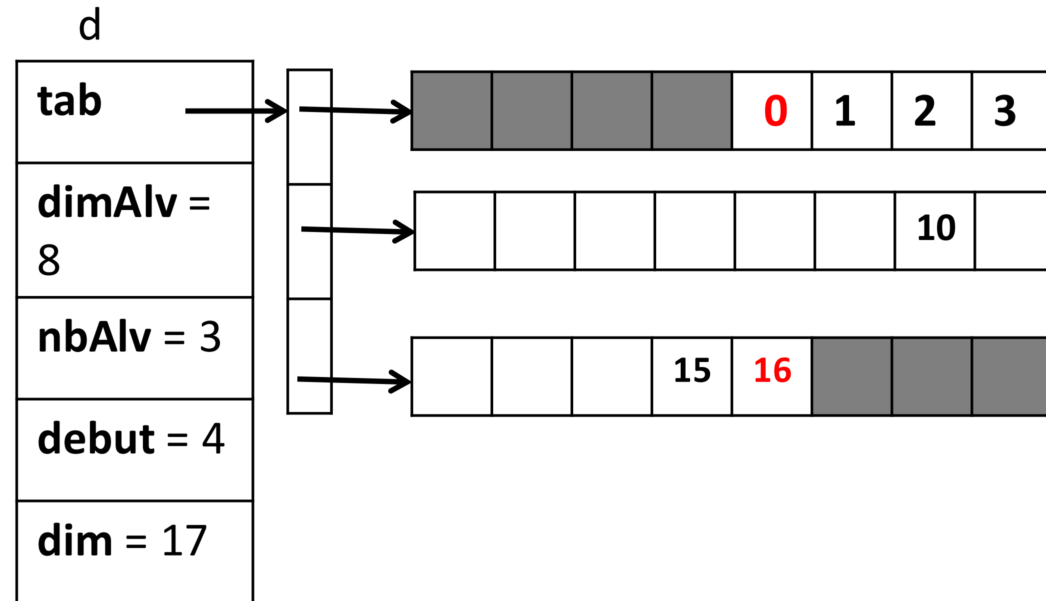
D[10]
i+debut = 14
alv = 14/8 =1
cell = 14%8 = 6

```
type&
deque::operator[](size_t i){
  pos = debut+i;
  alv = pos/dimAlv
  cell = pos%dimAlv;
  return tab[alv][cell];
}
```

```
type& deque::at(size_t i){
if (i >= size())
    exception;
  else
    return operator[](i);
}
```

Algorithmes

❑ Accès



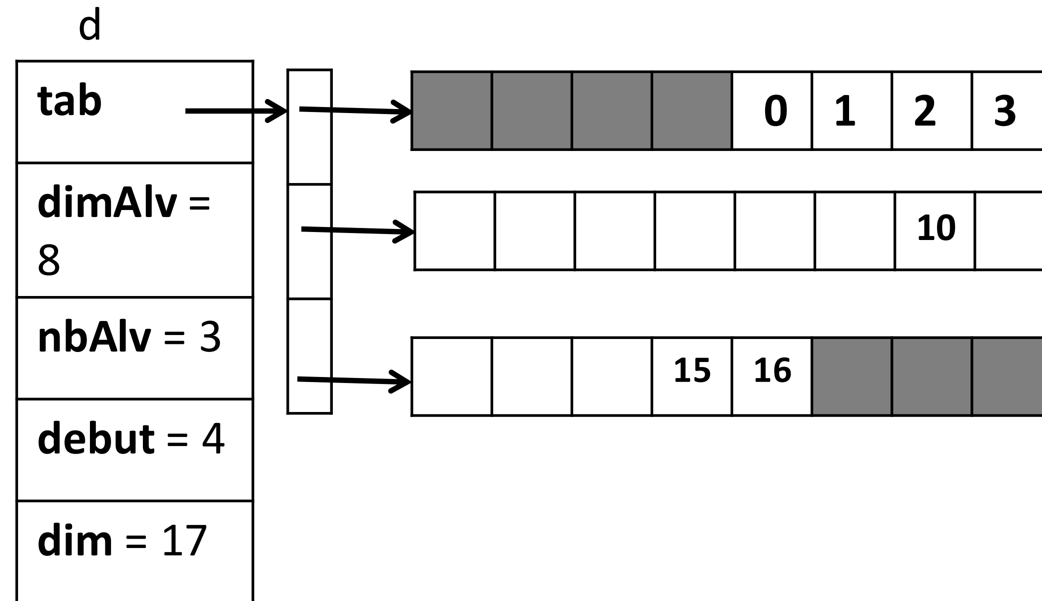

```
type& deque::front(){
   return operator[](0);
}
```

```
type& deque::back(){
   return operator[](size()-1);
}
```

Algorithmes

- **Gestion dimension/capacité**

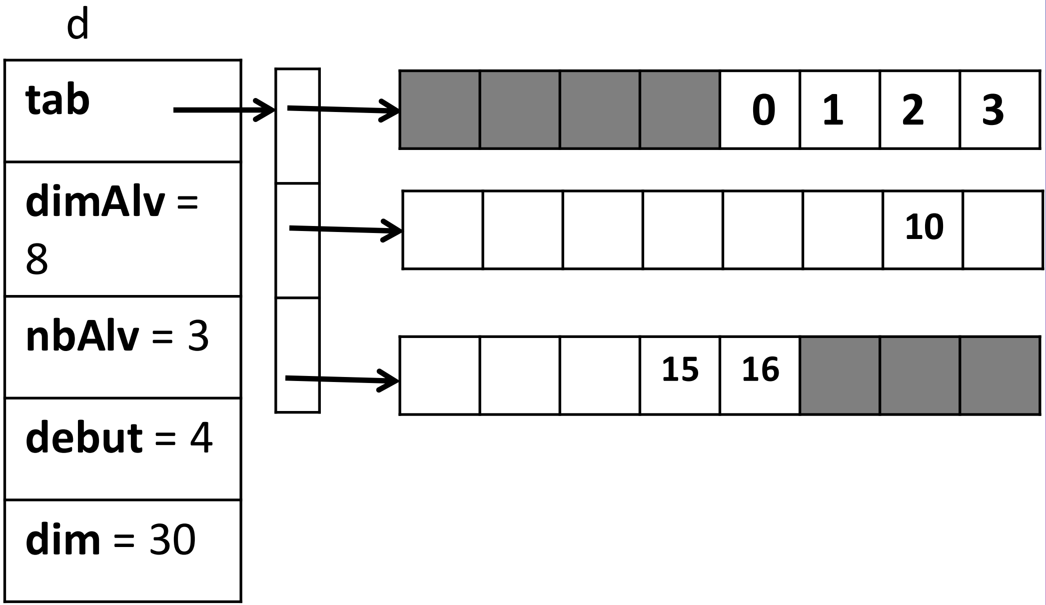
```
void deque::resize(size_t n){
 dim = n
 if(nbAlv*dimAlv-1 < debut+dim-1){
  back = debut+dim-1;
  alv = back/dimAlv;
  nvnbAlv = alv+1;
  type **tmp = new
type*[nvnbAlv];
  for(int i = 0; i< nbAlv;i++)
    tmp[i] = tab[i];
  for(int i = nbAlv; i< nvnbAlv;i++)
    tmp[i] = new type[dimAlv];
  delete [] tab;
  tab = tmp;
  nbAlv = nvnbAlv;}
}
```



Algorithmes

- **Gestion dimension/capacité**

```
void deque::resize(size_t n){
 dim = n
 if(nbAlv*dimAlv-1 < debut+dim-1){
  back = debut+dim-1;
  alv = back/dimAlv;
  nvnbAlv = alv+1;
  type **tmp = new
type*[nvnbAlv];
  for(int i = 0; i< nbAlv;i++)
    tmp[i] = tab[i];
  for(int i = nbAlv; i< nvnbAlv;i++)
    tmp[i] = new type[dimAlv];
  delete [] tab;
  tab = tmp;
  nbAlv = nvnbAlv;}
}
```

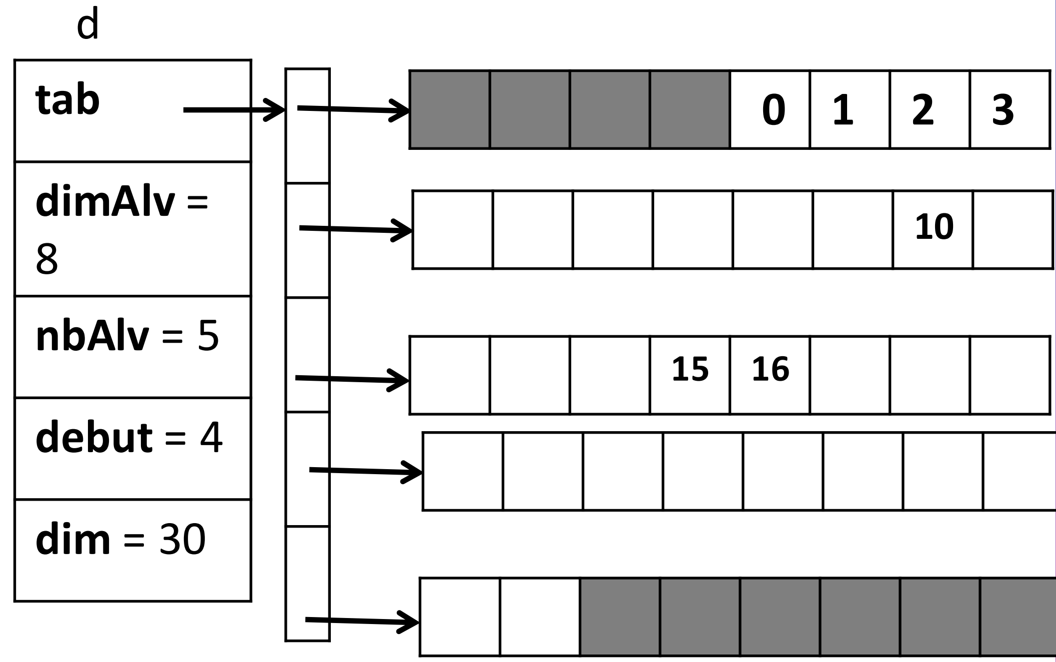


d.resize(30);
back = 4+29=33
alv = 4
nvnbAlv = 5

Algorithmes

❑ Gestion dimension/capacité

```
void deque::resize(size_t n){
 dim = n
 if(nbAlv*dimAlv-1 < debut+dim-1){
  back = debut+dim-1;
  alv = back/dimAlv;
  nvnbAlv = alv+1;
  type **tmp = new
type*[nvnbAlv];
  for(int i = 0; i< nbAlv;i++)
    tmp[i] = tab[i];
  for(int i = nbAlv; i< nvnbAlv;i++)
    tmp[i] = new type[dimAlv];
  delete [] tab;
  tab = tmp;
  nbAlv = nvnbAlv;}
}
```



d.resize(30);
back = 4+29=33
alv = 4
nvnbAlv = 5