La librairie standard **Rust** définit plusieurs collections, c'est-à-dire des types composites génériques servant à stocker des données en mémoire. Nous en avons déjà rencontré à plusieurs reprises, notamment les vecteurs et les tables de hachage (dictionnaires), **Vec** et **HashMap**. Nous allons découvrir l'ensemble des méthodes de ces deux types en détail lors de cette étude, et décrirons les quelques autres collections apparentées.

*Rappelons les différences fondamentales entre les collections **Rust** et celles des autres langages. Il existe d'abord le mécanisme de transfert de **possession** et **d'emprunt** qui sont en œuvre partout. **Rust** se sert de ces **transferts** pour éviter les copies complètes des valeurs pénalisantes. Par exemple, la méthode **Vec<T>::push(item)** prend son paramètre d'entrée par valeur et **non par référence**, le vecteur devenant **possesseur** de la valeur.*

*Ensuite, **Rust** ne connaît par les erreurs d'invalidation liées aux bogues de pointeurs errants lors de la retaille d'une collection ou d'une modification quelconque alors que le programme utilise encore un pointeur sur les données qu'elle contient. Ce genre d'erreur est une importante source de comportements incertains du **C++**. Le contrôle **d'emprunt** de **Rust** bloque ces soucis dès la compilation.*

Aperçu global

Les trois types de collections les plus utilisés sont **Vec<T>**, **HashMap<K, V>** et **HashSet<T>**, les autres sont destinés à des cas particuliers. Nous découvrirons tous les types plus ou moins en détail lors de cette étude

Collections	Définition	Description
Vec<T>	**Tableau dynamique**	**Vec<T>** est un tableau de valeurs du type **T** redimensionnable.
VecDeque<T>	**Double queue (tampon circulaire dynamique)**	**VecDeque<T>** ressemble à **Vec<T>** en étant spécialisé en tant que queue **FIFO**, premier entré-premier sorti. Elle est très efficace pour ajouter et enlever des valeurs en début et en fin de liste. Toutes les autres opérations sont de ce fait moins rapides.
LinkedList<T>	**Liste liée double**	**LinkedList<T>** propose un accès rapide à l'avant et à l'arrière de la liste comme **VecDeque<T>**, en y ajoutant une concaténation rapide. En général, cette liste liée est plus lente que **Vec<T>** et **VecDeque<T>**.
BinaryHeap<T> where T : Ord	**Liste avec plus grand devant dans le tas**	**BinaryHeap<T>** est une queue à priorité qui organise les valeurs de sorte qu'il soit toujours rapide de trouver et de supprimer la plus grande valeur.
HashMap<K, V> where K : Eq+Hash	**Table de hachage clés/valeurs**	**HashMap<K, V>** est une table de **paires clé-valeur**. La recherche d'une valeur par **clé** est rapide, mais les entrées sont stockées sans ordre.
BtreeMap<K, V> where K : Ord	**Table clés/valeurs trié**	**BtreeMap<K, V>** ressemble à **HashMap<K, V>** mais les entrées sont triées dans l'ordre des **clés**, ce qui est moins rapide. Le type **BtreeMap<String, i32>** par exemple stocke les entrées dans l'ordre de comparaison du type **String**.
HashSet<T> where T : Eq+Hash	**Ensemble de valeurs**	**HashSet<T>** représente un ensemble de valeurs uniques du type **T** et permet d'ajouter et d'enlever rapidement des valeurs. Vous pouvez aussi vite savoir si une valeur se trouve dans l'ensemble ou pas.
BtreeSet<T> where T : Ord	**Ensemble trié**	**BtreeSet<T>** ressemble à **HashSet<T>** en y ajoutant un stockage trié par valeurs. À moins d'avoir besoin de ce tri, **HashSet** est beaucoup plus rapide.

Vect<T>

Rappelons qu'un vecteur est constitué de trois champs : la **longueur**, la **capacité maximale** et le **pointeur** vers la zone dans le tas où sont stockées les éléments. Nous pouvons maintenant nous intéresser aux différentes méthodes et aux mécanismes internes de ce type.

L'exemple ci-dessous montre un exemple d'implantation mémoire de trois vecteurs. Celui qui est vide possède une capacité égale à zéro. Il ne consomme aucun espace dans la zone du tas jusqu'à ce qu'un premier élément soit alloué

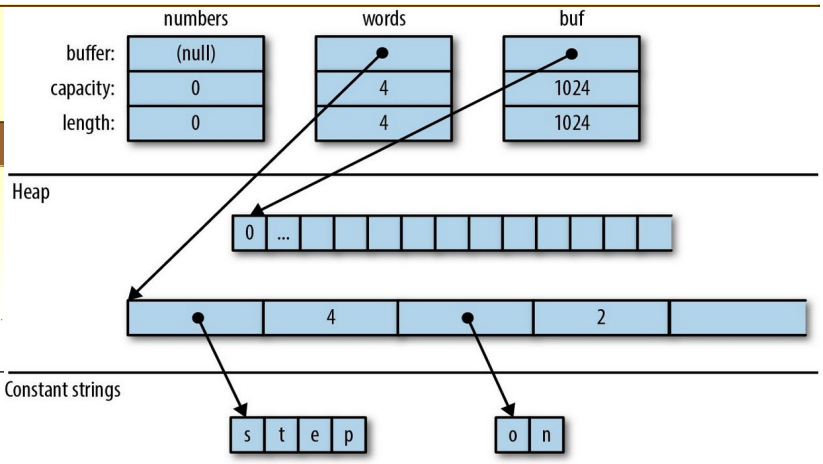
Adaptateur de transformation et de filtre

```
fn main() {
  // Création d'un vecteur vide
  let mut numbers: Vec<i32>= vec![];
```

```rust
// Création d'un vecteur avec contenu
let words = vec!["step", "on", "no", "pets"];
let mut tampon = vec![0u8; 1024];
}
```

Résultat

*Puisqu'il s'agit d'une collection, **Vec** implémente **std::iter::FromIterator**, ce qui permet de créer un vecteur depuis un itérateur au moyen de la méthode **collect()**, comme nous l'avons vu dans l'étude précédente sur **les itérateurs**.*



Accès aux éléments d'un vecteur

L'accès à un élément pour un tableau, une tranche ou un vecteur au moyen d'un indice est très simple, comme le montre l'exemple ci-dessous. Dans les quatre syntaxes, une panique est déclenchée si l'indice dépasse les limites.

Adaptateur filter_map()

```rust
fn main() {
  let nombres = vec![1, 2, 3, 5, 7, 11];

  // Obtient une référence sur un élément
  let premier = &nombres[0];

  // Obtient une copie d'un élément
  let deuxieme = nombres[1];         // Requiert Copy
  let troisime = nombres[2].clone();  // Requiert Clone

  // Obtient une référence sur une tranche
  let plusieurs = &nombres[3..];

  // Obtient une copie d'une tranche
  let vecteur = nombres[3..].to_vec();

  println!("{} {} {} {:?} {:?}", premier, deuxieme, troisime, plusieurs, vecteur);
}
```

Résultat

```
1 2 3 [5, 7, 11] [5, 7, 11]
```

*Vous savez que **Rust** est sans pitié au niveau du mélange entre types numériques, et il ne fait pas d'exception pour les vecteurs. Les longueurs et les indices des vecteurs sont du type **usize**. Vous obtenez une erreur si vous essayez d'utiliser un **u32**, **u64** ou **isize** comme indice. Vous pouvez faire un transtypage avec **n as usize** si nécessaire.*

Vous disposez de plusieurs méthodes pour accéder facilement à des éléments en particulier dans un vecteur ou une tranche. Rappelons que toutes les méthodes des tranches (slices) sont applicables aux tableaux et aux vecteurs.

- ***slice.first()** renvoie une référence au premier élément de la tranche, s'il existe. Le type renvoyé est **Option<T>**, et la valeur renvoyée est **None** si la tranche est vide ou **Some(&slice[0])** sinon.*
- ***slice.last()** renvoie une référence au dernier élément.*
- ***slice.get(index)** renvoie une référence **Some** sur **slice[index]**, si elle existe, ou bien **None** si la tranche possède au moins de **index+1** éléments.*
- ***slice.first_mut(), slice.last_mut(), slice.get_mu()** sont des variations des précédentes qui empruntent des références mutables.*
- ***slice.to_vec()**. En renvoyant un élément de type **T** par valeur, cela provoque le transfert de la possession. Pour éviter cela, les méthodes qui accèdent aux éléments renvoient donc systématiquement les éléments par référence. Il existe une exception avec la méthode **to_vec()** qui effectue des copies. Cette méthode n'est utilisable que si les éléments peuvent être clonés, c'est-à-dire avec **where T : Clone**.*

Accès aux éléments d'un vecteur

```rust
fn main() {
  let nombres = vec![1, 2, 3, 5, 7, 11];

  if let Some(nombre) = nombres.first() {
    println!("Premier nombre : {}", nombre);
  }
  if let Some(nombre) = nombres.last() {
    println!("Premier nombre : {}", nombre);
  }
```

```
  println!("Cinquième élément : {:?}", nombres.get(4));
  println!("Septième élément : {:?}", nombres.get(6));

  if let Some(inconnu) = nombres.get(6) {
    println!("Inconnu : {}", inconnu);
  }
  match nombres.get(4) {
    Some(nombre) => println!("Cinquième élément : {}", nombre),
    None => println!("Cinquième élément inconnu")
  }
  match nombres.get(6) {
    Some(nombre) => println!("Septième élément : {}", nombre),
    None => println!("Septième élément inconnu")
  }
  let mut tranche = [0, 1, 2, 3];
  {
    let dernier = tranche.last_mut().unwrap();
    *dernier += 10;
  }
  println!("Tranche : {:?}", tranche);
  println!("Nombres : {:?}", nombres.to_vec());
  println!("Nombres : {:?}", nombres[3..].to_vec());
}
```

Résultat

```
Premier nombre : 1
Premier nombre : 11
Cinquième élément : Some(7)
Septième élément : None
Cinquième élément : 7
Septième élément inconnu
Tranche : [0, 1, 2, 13]
Nombres : [1, 2, 3, 5, 7, 11]
Nombres : [5, 7, 11]
```

Itérations

Comme nous pouvons nous y attendre, les vecteurs et les tranches sont itérables, aussi bien par valeur que par référence. Le mécanisme suivi est celui impliquant l'implémentation de **IntoIterator**.

- *En itérant sur un **Vec<T>**, nous produisons des éléments de type **T** qui sont consommés en étant extraits du vecteur un par un.*
- *En itérant sur une valeur du type **&[T ; N], &[T]** ou bien **&Vec<T>**, c'est-à-dire une référence à un tableau, une tranche ou un vecteur, cela produit des éléments du type **&T** dont le possédant ne change pas, car ce sont des références aux éléments.*

*Les tableaux, les tranches et les vecteurs disposent en outre des deux méthodes **iter()** et **iter_mut()** décrites dans l'étude précédente. Elles permettent de créer des itérateurs produisant des références sur leurs éléments.*

Nous verrons quelques moyens plus sophistiqués pour balayer une tranche dans la section des méthodes de distribution (famille split) un peu plus loin.

Références partagées, références mutables ou possession

```
fn main() {
  let mut nombres = vec![5, 18, -3, 12, 23, 55, 9];

  for nombre in &nombres { print!("..{}.", nombre); }
  println!();
  for nombre in &mut nombres {
    *nombre += 10;
    print!("..{}.", nombre);
  }
  println!();
  for nombre in nombres { print!("..{}.", nombre); }
  println!("{:?}", nombres);                          // Impossible, nombres n'existe plus
}
```

Résultat

```
..5...18...-3...12...23...55...9.
..15...28...7...22...33...65...19.
..15...28...7...22...33...65...19
```

Contrôle de la taille des vecteurs

La **longueur** d'un tableau, d'une tranche ou d'un vecteur correspond au nombre d'éléments qu'il contient. La charge utile d'un vecteur, ses éléments, est stockée sous forme d'un bloc mémoire contigu dans la zone du tas. La capacité d'un vecteur est sa **jauge**, c'est-à-dire le nombre maximal d'éléments qu'il peut accueillir.

*Cette valeur de **jauge** est normalement gérée automatiquement, en réservant un tampon plus grand et en déplaçant les éléments lorsque vous avez besoin de plus d'espace que la **capacité** actuelle. Vous disposez de quelques méthodes pour intervenir manuellement sur cette **capacité**.*

Toutes les méthodes de cette section se chargent d'augmenter ou de diminuer la taille d'un vecteur. Elles ne s'appliquent pas aux tableaux et aux tranches dont la longueur est figées dès le départ.

- ***slice.len() :** renvoie la **longueur** d'une tranche en tant que valeur de type **usize**.*
- ***slice.is_empty()** : vaut **true** si la tranche ne contient aucun élément (tranche vide).*
- ***Vec::with_capacity(n) :** crée un nouveau vecteur vide de capacité **n**.*
- ***vec.capacity() :** renvoie la capacité du vecteur de type **usize**. L'expression **vec.capacity() ≥ vec.len()** est toujours vraie.*
- ***vec.reserve(n) :** vérifie que le vecteur dispose encore d'assez de place pour ajouter **n** éléments, c'est-à-dire que **vec.capacity()** est au moins égal à **vec.len()+n**. S'il n'y a pas assez de place, un tampon plus vaste est réservé puis le contenu actuel y est déplacé.*
- ***vec.reserve_exact(n) :** ressemble à **vec.reserve(n)**, mais n'ajoute pas de marge quant à la nouvelle taille du vecteur. Une fois l'opération réalisée, **vec.capacity()** coïncide exactement avec **vec.len()+n**.*
- ***vec.shrink_to_fit() :** essaye de restituer au système la mémoire inutilisée lorsque **vec.capacity()** est supérieur à **vec.len()**.*

Gestion de la taille et de la capacité d'un vecteur

```
fn main() {
  let mut nombres : Vec<i32> = Vec::with_capacity(3);
  println!("{:?}, taille={}, capacité={}", nombres, nombres.len(), nombres.capacity());
  nombres.reserve(3);
  nombres.push(1);
  println!("{:?}, taille={}, capacité={}", nombres, nombres.len(), nombres.capacity());
  nombres.reserve_exact(3);
  nombres.push(3);
  println!("{:?}, taille={}, capacité={}", nombres, nombres.len(), nombres.capacity());
}
```

Résultat

```
[], taille=0, capacité=3
[1], taille=1, capacité=3
[1, 3], taille=2, capacité=4
```

***Vec<T>** possède un ensemble de méthodes pour ajouter et supprimer des éléments, ce qui a un effet sur la longueur du vecteur. Toutes ces méthodes reçoivent leur paramètre **self** sous forme d'une référence mutable.*

- ***vec.push(valeur) :** ajoute la valeur indiquée à la fin du vecteur.*
- ***vec.pop()** : récupère et enlève le dernier élément du vecteur.*

Attention : la méthode push() reçoit son paramètre d'entrée par valeur et non par référence. De même pop() retourne une valeur et non une référence. Cette remarque s'applique à la plupart des autres méthodes de cette section et injectent des valeurs et non des références.

- ***vec.insert(index, valeur) :** insère la valeur à l'indice **vec[index]** en repoussant d'une position à droite toutes les autres valeurs dans **vec[index..]**. la méthode panique si **index > vec.len()**.*
- ***vec.remove(index)** : récupère et enlève **vec[index]** en ramenant les valeurs existantes d'une position vers la gauche pour combler le trou. Panique si **index ≥ vec.len()** puisque dans ce cas il n'existe aucun élément **vec[index]** à enlever. Plus le vecteur est grand, plus l'opération prend du temps. Si vous avez besoin de faire ce type d'opération souvent, choisissez plutôt à la variante **VecDeque** décrite un peu plus loin au lieu de **Vec**.*

Ces deux méthodes insert() et remove() sont plus lentes s'il existe beaucoup de positions à modifier. Trois méthodes sont disponibles pour définir la nouvelle longueur d'un vecteur.

- ***vec.resize(nouvelle, valeur) :** choisit la longueur de **vec** comme égale à **nouvelle**. Si cela augmente sa longueur, le nouvel espace est peuplé avec des copies de **valeur**. Le type de l'élément doit implémenter le trait **Clone**.*
- ***vec.truncate(nouvelle)** : réduit la longueur du vecteur en abandonnant les éléments correspondant à la plage **vec[nouvelle..]**. Si **vec.len()** est déjà inférieure à **nouvelle**, il ne se passe rien.*
- ***vec.clear()** : supprime tous les éléments du vecteur, ce qui revient à faire **vec.truncate(0)**.*
- ***vec.extend(itérable) :** ajoute tous les éléments issus de la valeur **itérable** à la fin du vecteur, dans l'ordre. C'est une version collective de **push()**. Le paramètre **itérable** doit implémenter le trait **IntoIterator<Item=T>**. cette méthode sert tellement souvent qu'un trait lui a été dédié, **Extend**. Ce trait est implanté par toutes les collections standard.*
- ***vec.split_off(index)** : ressemble à **vec.truncate(index)**, sauf qu'il renvoie un **Vec<T>** qui contient les valeurs supprimées de la fin du vecteur. C'est une version collective de **pop()**.*

- ***vec.append(&mut vec2)*** *: déplace tous les éléments de* ***vec2*** *(un autre vecteur de type* ***Vec<T>*** *vers* ***vec****, laissant* ***vec2*** *vide. Ressemble à* ***vec.extends(vec2)*** *sauf que* ***vec2*** *existe toujours après l'opération, avec la même capacité.*
- ***vec.drain(plage)*** *: avec* ***plage*** *une valeur de plage comme* ***..*** *ou bien* ***0..4****, enlève la plage* ***vec[plage]*** *du vecteur et renvoie un itérateur sur les éléments enlevés.*
- ***vec.retain(test) :*** *supprime tous les éléments qui ne satisfont pas un test. Le paramètre* ***test*** *doit être une fonction ou une clôture qui implémente* ***FnMut(&T) → bool****. La méthode appelle pour chaque élément du vecteur* ***test(&élément)****, et enlève l'élément si l'appel renvoie* ***false****. L'effet est le même que l'écriture suivante, avec des performances différentes :* ***vec = vec.into_iter().filter(test).collect() ;***
- ***vec.dedup()*** *: élimine les éléments en double, ce qui ressemble à l'outil* ***shell d'Unix*** *nommé* ***uniq****. La méthode recherche les voisins de chaque élément et supprime les doublons pour n'en conserver qu'un de chaque.*
- ***vec.dedub_by(&mut elem1, &mut elem2)*** *: équivaux à* ***vec.dedup()*** *mais se sert de la fonction ou de la clôture* ***testidem(&mut elem1, &mut elem2)*** *à la place de l'opérateur* ***==*** *pour tester si deux éléments sont égaux.*
- ***vec.dedub_by_key(&mut elem1, &mut elem2)*** *: est identique avec* ***vec.dedup()*** *mais considère que les deux éléments sont égaux si* ***clé(&mut elem1) == clé(&mut elem2)****.*

Ajout et suppression d'éléments dans un vecteur

```
fn main() {
  let mut nombres : Vec<u32> = (1..50).filter(|n| est_premier(*n)).collect();
  println!("Original = {:?}", nombres);
  let dernier = nombres.pop().unwrap();
  println!("Pop = {:?}, ({})", nombres, dernier);
  nombres.insert(3, 4);
  println!("Insertion = {:?}", nombres);
  nombres.remove(3);
  println!("Suppression = {:?}", nombres);
  nombres.resize(20, 0);
  println!("Retailler = {:?}", nombres);
  nombres.truncate(10);
  println!("Tronquée = {:?}", nombres);
  nombres.extend((29..50).filter(|n| est_premier(*n)));
  println!("Extension = {:?}", nombres);
  let mut derniers = nombres.split_off(10);
  println!("Nombres = {:?}, Derniers = {:?}", nombres, derniers);
  nombres.append(&mut derniers);
  println!("Nombres = {:?}, Derniers = {:?}", nombres, derniers);
  nombres.drain(10..);
  println!("Nombres = {:?}", nombres);
  nombres.retain(|n| *n > 10);
  println!("Nombres = {:?}", nombres);
}

fn est_premier(n: u32) -> bool {
  let limite = (n as f32).sqrt() as u32;
  for i in 2..=limite {
    if n%i == 0 { return false }
  }
  true
}
```

Résultat

```
Original = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
Pop = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43], (47)
Insertion = [1, 2, 3, 4, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43]
Suppression = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43]
Retailler = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 0, 0, 0, 0, 0]
Tronquée = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23]
Extension = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
Nombres = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23], Derniers = [29, 31, 37, 41, 43, 47]
Nombres = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47], Derniers = []
Nombres = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23]
Nombres = [11, 13, 17, 19, 23]
```

De toutes les méthodes que nous venons de voir dans cette section, la seule qui clone les valeurs est resize(). Toutes les autres transfèrent la possession des valeurs d'un endroit à l'autre.

Jointure

Vous disposez de deux méthodes dédiées aux tableaux de tableaux, c'est-à-dire des tableaux, des tranches ou des vecteurs dont les éléments sont eux-mêmes des tableaux, des tranches ou des vecteurs.

- ***slices.concat() :*** *renvoie un vecteur après avoir mis bout à bout toutes les tranches d'entrée.*
- ***slices.join(&separateur)*** *: fait le même travail en ajoutant une copie de la valeur* ***séparateur*** *entre deux tranches.*

Joindre plusieurs tableaux

```
fn main() {
  let nombres = [[1, 2], [3, 4], [5, 6]].concat();
  println!("Nombres = {:?}", nombres);
  let nombres = [[1, 2], [3, 4], [5, 6]].join(&0);
  println!("Nombres = {:?}", nombres);
}
```

Résultat

```
Nombres = [1, 2, 3, 4, 5, 6]
Nombres = [1, 2, 0, 3, 4, 0, 5, 6]
```

Distribution (split)

Rust prévoit plusieurs méthodes pour emprunter des références mutables vers deux ou plus parties d'un tableau, d'une tranche ou d'un vecteur, en une fois. Ces méthodes sont sûres parce que dès la conception, elles répartissent les données en plusieurs régions qui ne se chevauchent pas.

La plupart de ces méthodes sont également intéressantes pour travailler sur des tranches non mutables, et c'est pourquoi il existe une variante mutable ou une autre non mutable pour chacune d'elles. Dans l'exemple ci-dessous, nous remarquons que nous pouvons facilement récupérer des références non mutables dans un tableau, une tranche ou un vecteur.

Références non mutables

```
fn main() {
  let i = 2;
  let j = 2;
  let nombres = vec![5, 6, 7, 8];
  let a= &nombres[i];
  let b= &nombres[j];
  let milieu = nombres.len() / 2;
  let debut = &nombres[..milieu];
  let fin = &nombres[milieu..];

  println!("Nombres={:?}, a={}, b={}, debut={:?}, fin={:?}", nombres, a, b, debut, fin);
}
```

Résultat

```
Nombres=[5, 6, 7, 8], a=7, b=7, debut=[5, 6], fin=[7, 8]
```

L'opération est beaucoup moins facile pour récupérer plusieurs références mutables comme ci-dessous :

Références mutables

```
fn main() {
  let i = 2;
  let j = 3;
  let mut nombres = vec![5, 6, 7, 8];
  let a= &mut nombres[i];
  let b= &mut nombres[j];
  let milieu = nombres.len() / 2;
  let debut = &nombres[..milieu];
  let fin = &nombres[milieu..];

  println!("Nombres={:?}, a={}, b={}, debut={:?}, fin={:?}", nombres, a, b, debut, fin);
}
```

Résultat

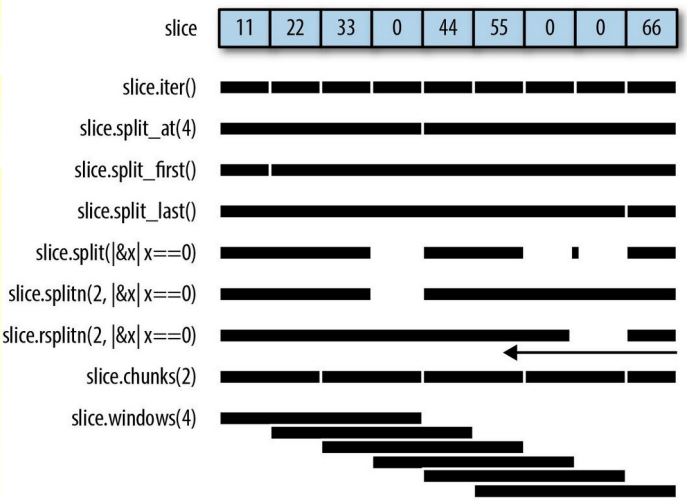
```
error[E0499]: cannot borrow `nombres` as mutable more than once at a time
   --> src/main.rs:6:15
    |
5   |  let a= &mut nombres[i];
    |           ------- first mutable borrow occurs here
6   |  let b= &mut nombres[j];
    |           ^^^^^^^ second mutable borrow occurs here
...
11  |  println!("Nombres={:?}, a={}, b={}, debut={:?}, fin={:?}", nombres, a, b, debut, fin);
    |                                                                 - first borrow later used here
```

*Le compilateur refuse parce que dans le cas où **i==j**, **a** et **b** deviendraient deux références mutables vers la même valeur entière, ce qui est contraire aux règles de sécurité de **Rust** (revoir la discussion sur le partage et les mutations).*

Les méthodes disponibles sont illustrées ci-dessous. Aucune d'elles ne modifie directement la structure concernée. Elles renvoient de nouvelles références à des portions de données du contenu.

- ***slice.iter()** et **slice.iter_mut() :** produisent une référence sur chaque élément de la tranche **slice**. Nous les avons vues dans l'étude sur les itérations.*

- ***slice.split_at(index)*** *et* ***slice.split_at_mut(index)*** *: coupent une tranche en deux en renvoyant une paire. La première des deux équivaut à (**&slice[..index]**, **&slice[index..]**). Les deux méthodes paniquent si l'index est hors limite.*
- ***slice.split_first()*** *et* ***slice.split_first_mut() :*** *renvoient aussi une paire : une référence au premier élément (**slice[0]**) et une référence au reste (**slice[1..]**).*
- ***slice.split_last(index)*** *et* ***slice.split_last_mut(index)*** *: sont complémentaires des précédentes puisqu'elles extraient à part le dernier élément au lieu du premier.*
- ***slice.split(is_sep)*** *et* ***slice.split_mut(is_sep) :*** *découpent la tranche en une ou plusieurs sous-tranches en se servant de la fonction ou de la clôture **is_sep** pour choisir quel endroit couper et renvoient un itérateur sur les sous-tranches résultantes. Lorsque vous consommez l'itérateur, celui-ci appelle **is_sep(&element)** pour chaque élément de la tranche. Si cet appel renvoie **true**, c'est que l'élément est un séparateur . Les séparateurs ne sont pas insérés dans les sous-tranches résultantes. Le résultat est toujours au moins une sous-tranche plus une par séparateur. Vous obtenez une sous-tranche vide dès que deux séparateurs sont adjacents, ou pour un séparateur trouvé en dernier dans la tranche d'entrée.*



- ***slice.splitn(n, is_sep)*** *et* ***slice.splitn_mut(n, is_sep) :*** *sont similaires tout en limitant le nombre de sous-tranches à la valeur **n**. dès que **n-1** tranches ont été trouvées**. is_sep** n'est plus appelé et la dernière sous-tranche contient ce qui restait dans l'entrée.*
- ***slice.rsplitn(n, is_sep)*** *et* ***slice.rsplitn_mut(n, is_sep) :*** *ressemblent à **splitn()** et **sliptn_mut()** sauf que la tranche d'entrée est traitée en sens inverse. Les méthodes découpent à partir des derniers **n-1** séparateurs au lieu des premiers et les sous-tranches sont produites en commençant par la fin.*
- ***slice.chunks(n)*** *et* ***slice.chunks_mut(n) :*** *renvoient un itérateur sur les sous-tranches qui ne se chevauchent pas de longueur **n**. lorsque **slice.len()** n'est pas un multiple de **n**, la dernière sous-tranche aura une longueur inférieure à celle demandée.*
- ***slice.windows(n) :*** *renvoient un itérateur qui se comporte comme une « **fenêtre mobile** » sur les données de la tranche en produisant des sous-tranches constituées de n éléments consécutifs. La première valeur correspond à **&slice[0,,n]**, la seconde à **&slice[1,,n+1]**, etc. Si la valeur n est supérieure à la longueur totale de la tranche d'entrée, aucune tranche n'est produite. Si **n** vaut **0**, vous déclenchez une panique. Cette méthode n'offre pas de variante pour référence mutable, car les sous-tranches se chevauchent.*

Découpe de vecteurs

```
fn main() {
  let nombres = vec![11, 22, 33, 0, 44, 55, 0, 0, 66];
  println!("Nombres={:?}", nombres);
  let (debut, fin) = nombres.split_at(4);
  println!("Début={:?}, Fin={:?}", debut, fin);

  if let Some((element, tableau)) = nombres.split_first() {
    println!("Élément={:?}, Tableau={:?}", element, tableau);
  }
  if let Some((element, tableau)) = nombres.split_last() {
    println!("Élément={:?}, Tableau={:?}", element, tableau);
  }
  for (tranche, numero) in nombres.split(|&n| n==0).zip(1..) {
    println!("(split) --> {} : {:?}", numero, tranche);
  }
  for (tranche, numero) in nombres.splitn(2, |&n| n==0).zip(1..) {
    println!("(splitn) --> {} : {:?}", numero, tranche);
  }
  for (tranche, numero) in nombres.rsplitn(2, |&n| n==0).zip(1..) {
    println!("(rsplitn) --> {} : {:?}", numero, tranche);
  }
  for (tranche, numero) in nombres.chunks(2).zip(1..) {
    println!("(chunks) --> {} : {:?}", numero, tranche);
  }
  for (tranche, numero) in nombres.windows(7).zip(1..) {
    println!("(windows) --> {} : {:?}", numero, tranche);
  }
}
```

Résultat

```
Nombres=[11, 22, 33, 0, 44, 55, 0, 0, 66]
Début=[11, 22, 33, 0], Fin=[44, 55, 0, 0, 66]
Élément=11, Tableau=[22, 33, 0, 44, 55, 0, 0, 66]
```

```
Élément=66, Tableau=[11, 22, 33, 0, 44, 55, 0, 0]
(split) --> 1 : [11, 22, 33]
(split) --> 2 : [44, 55]
(split) --> 3 : []
(split) --> 4 : [66]
(splitn) --> 1 : [11, 22, 33]
(splitn) --> 2 : [44, 55, 0, 0, 66]
(rsplitn) --> 1 : [66]
(rsplitn) --> 2 : [11, 22, 33, 0, 44, 55, 0]
(chunks) --> 1 : [11, 22]
(chunks) --> 2 : [33, 0]
(chunks) --> 3 : [44, 55]
(chunks) --> 4 : [0, 0]
(chunks) --> 5 : [66]
(windows) --> 1 : [11, 22, 33, 0, 44, 55, 0]
(windows) --> 2 : [22, 33, 0, 44, 55, 0, 0]
(windows) --> 3 : [33, 0, 44, 55, 0, 0, 66]
```

Permutation (swap)

La méthode utilitaire **swap()** est prévue pour permuter deux éléments, Les vecteurs disposent également d'une méthode apparentée **swap_remove()** pour enlever n'importe quel élément de façon efficace.

- ***slice.swap(i, j) :*** *permute les deux éléments* ***slice[i]*** *et* ***slice[j]****.*
- ***slice.swap_remove(i) :*** *enlève et renvoie* ***vec[i]****. Cela équivaut à* ***vec.remove(i)*** *sauf qu'au lieu de décaler tous les autres éléments pour boucher le trou, nous déplaçons le dernier élément de* ***vec*** *pour combler. La méthode est utile lorsque vous n'avez pas besoin de conserver l'ordre des éléments dans le vecteur.*

Permutation de valeurs dans un vecteur

```rust
fn main() {
  let mut nombres = vec![11, 22, 33, 0, 44, 55, 0, 0, 66];
  println!("Nombres={:?}", nombres);
  nombres.swap(1, 7);
  println!("Nombres={:?}", nombres);
  println!("Élément={}, Nombres={:?}", nombres.swap_remove(6), nombres);
}
```

Résultat

```
Nombres=[11, 22, 33, 0, 44, 55, 0, 0, 66]
Nombres=[11, 0, 33, 0, 44, 55, 0, 22, 66]
Élément=0, Nombres=[11, 0, 33, 0, 44, 55, 66, 22]
```

Tris

Trois méthodes de tri sont disponibles pour les tranches. Vous pouvez également trier selon un champ particulier en vous servant d'un second champ comme critère en comparant des tuples.

- ***slice.sort() :*** *trie les éléments dans l'ordre croissant, mais la méthode n'est disponible que si le type de l'élément implémente* ***Ord****.*
- ***slice.sort_by(cmp) :*** *trie les éléments en s'appuyant sur une fonction ou une clôture* ***cmp****. Celle-ci doit implémenter* ***Fn(&T, &T) → std::cmp::>Ordering****. La définition de* ***cmp*** *à la main est fastidieuse, sauf à la déléguer à une méthode* ***cmp()****.*
- ***slice.sort_by_key(clé) :*** *trie les éléments de la tranche dans l'ordre croissant de la clé qui est incarné par la fonction ou la clôture* ***clé****. Le type de celle-ci doit implémenter* ***Fn(&T) → K*** *avec* ***K : Ord****. Cette méthode est pratique lorsque* ***T*** *comporte un ou plusieurs champs triés, ce qui permet de profiter de plusieurs ordres de tri. Notez que les valeurs de clés de tri ne sont pas mises en cache pendant l'opération. La fonction ou la clôture* ***clé*** *risque donc d'être appelée un nombre de fois plus grand que* ***n****. Des raisons techniques interdisent à* ***clé(element)*** *de renvoyer une référence empruntée à partir de l'élément.*
- ***slice.reverse() :*** *pour trier dans le sens inverse, servez-vous de* ***sort_by()*** *avec une clôture* ***cmp*** *qui permute les deux paramètres. En travaillant avec* ***|b, a|*** *plutôt que* ***|a, b|*** *vous obtenez un tri inverse. Vous pouvez également appeler* ***reverse()*** *après le tri.*

Permutation de valeurs dans un vecteur

```rust
#[derive(Debug)]
struct Personne<'a> {
  nom: &'a str,
  prenom: &'a str
}

fn main() {
  let mut nombres = vec![7, 4, 15, 3, 22, 8, 17, 20, 19, 14, 1];
  println!("Nombres={:?}", nombres);
```

```
  nombres.sort();
  println!("Nombres={:?}", nombres);
  nombres.sort_by(|b, a| a.cmp(b));
  println!("Nombres={:?}", nombres);
  nombres.reverse();
  println!("Nombres={:?}", nombres);
  let mut personnes = vec![
    Personne{nom: "BROSSE", prenom: "Adam"},
    Personne{nom: "BORÉALE", prenom: "Aurore"},
    Personne{nom: "TÉRIEUR", prenom: "Alex"},
    Personne{nom: "TÉRIEUR", prenom: "Alain"},
  ];
  personnes.sort_by_key(|personne| personne.prenom);
  for personne in personnes {
    println!("{:?}", personne);
  }
}
```

Résultat

```
Nombres=[7, 4, 15, 3, 22, 8, 17, 20, 19, 14, 1]
Nombres=[1, 3, 4, 7, 8, 14, 15, 17, 19, 20, 22]
Nombres=[22, 20, 19, 17, 15, 14, 8, 7, 4, 3, 1]
Nombres=[1, 3, 4, 7, 8, 14, 15, 17, 19, 20, 22]
Personne { nom: "BROSSE", prenom: "Adam" }
Personne { nom: "TÉRIEUR", prenom: "Alain" }
Personne { nom: "TÉRIEUR", prenom: "Alex" }
Personne { nom: "BORÉALE", prenom: "Aurore" }
```

Recherches

Une fois qu'une tranche est bien triée, elle peut être efficacement recherchée. Effectivement, une recherche binaire ne travaille que sur une tranche triée dans l'ordre demandé. Dans le cas contraire, le résultat sera sans valeur : flou entrant donne flou sortant.

*Les flottants **f32** et **f64** peuvent prendre la valeur **NaN** et ne peuvent donc pas implémenter le tri **Ord**. Vous ne pouvez pas vous en servir comme clés dans les méthodes de tri et de recherche binaire. Pour obtenir l'équivalent, vous devez recourir à la caisse standard nommée **ord_subset**.*

- ***slice.binary_search(&val), slice.binary_search_by(&val, cmp) et slice.binary_search_by_key(&val, clé) :** recherche une valeur dans la tranche demandée. La valeur est transmise par référence. Ces méthodes renvoient un type **Result<usize, usize>** qui est **Ok(index)** si **slice[index]** est égale à val dans l'ordre de tri mentionné. Si l'index n'existe pas, elle renvoie **Err(point_insertion)**, ce qui permet de maintenir l'ordre en insérant la valeur à ce point d'insertion.*
- ***slice.contains(&val) :** renvoie **true** dès qu'un élément de la tranche possède la valeur demandée. La méthode scrute tous les éléments jusqu'à trouver un positif. Ici aussi la valeur est transmise par référence.*

Permutation de valeurs dans un vecteur

```
fn main() {
  let mut nombres = vec![7, 4, 15, 3, 22, 8, 17, 20, 19, 14, 1];
  println!("Nombres={:?}", nombres);
  nombres.sort();
  println!("Nombres={:?}", nombres);
  match nombres.binary_search(&7) {
    Ok(position) => println!("C'est le {}ème élément", position+1),
    Err(_) => println!("Cet élément n'existe pas")
  }
  if let Ok(position) = nombres.binary_search(&15) {
    println!("C'est le {}ème élément", position+1);
  }
  if nombres.contains(&14) {
    println!("L'élément 14 existe bien dans la collection");
  }
  else { println!("L'élément 14 n'est pas présent dans la collection"); }

  if nombres.contains(&18) {
    println!("L'élément 18 existe bien dans la collection");
  }
  else { println!("L'élément 18 n'est pas présent dans la collection"); }
}
```

Résultat

```
}Nombres=[7, 4, 15, 3, 22, 8, 17, 20, 19, 14, 1]
Nombres=[1, 3, 4, 7, 8, 14, 15, 17, 19, 20, 22]
C'est le 4ème élément
C'est le 7ème élément
L'élément 14 existe bien dans la collection
L'élément 18 n'est pas présent dans la collection
```

Comparaison de tranches

Lorsqu'un type T supporte les deux opérateurs d'identité **==** et **!=** (trait **PartialEq**), alors les tableaux **[T ; N]**, les tranches **[T]** et les vecteurs **Vec<T>** les supportent aussi. Deux tranches sont considérées comme égales si elles ont la même longueur et si les éléments des mêmes portions ont égaux. Il en va de même pour les tableaux et les vecteurs.

*Lorsqu'un type **T** supporte les opérateurs relationnels **<**, **≤**, **>** et **≥** (trait **PartialOrd**), alors les tableaux, les tranches et les vecteurs de type **T** les supportent aussi. Les comparaisons entre les tranches sont de style lexicographique. Deux méthodes utilitaires permettent de comparer les tranches :*

- ***slice.starts_with(autre) :** renvoie **true** si la tranche commence par une séquence de valeurs égale à celle d'une autre tranche.*
- ***slice.ends_width(autre) :** est complémentaire, car elle teste la fin de la tranche.*

Comparaison de collections

```
fn main() {
  let mut nombres = [7, 4, 15, 3, 22, 8, 17, 20, 19, 14, 1];
  nombres.sort();
  let debut = [1, 3, 4 , 7];
  let fin = [19, 20, 22];
  println!("{:?}, {:?} => début similaire : {}", nombres, debut, nombres.starts_with(&debut));
  println!("{:?}, {:?} => fin similaire : {}", nombres, fin, nombres.ends_with(&fin));
  println!("{:?}, [17, 19, 22] => fin similaire : {}", nombres, nombres.ends_with(&[17, 19, 22]));
}
```

Résultat

```
[1, 3, 4, 7, 8, 14, 15, 17, 19, 20, 22], [1, 3, 4, 7] => début similaire : true
[1, 3, 4, 7, 8, 14, 15, 17, 19, 20, 22], [19, 20, 22] => fin similaire : true
[1, 3, 4, 7, 8, 14, 15, 17, 19, 20, 22], [17, 19, 22] => fin similaire : false
```

Traitements aléatoires

Les valeurs numériques ne font pas partie de la librairie standard de **Rust**, mais la caisse nommée **rand** offre deux méthodes pour prélever au hasard depuis un tableau, une tranche ou un vecteur :

- ***slice.choose(rng) :** renvoie une référence à un élément de la tranche choisie au hasard. Comme **slice_first()** et **slice.last()**, la valeur renvoyée est **Option<&T>** qui vaut None si la tranche est vide.*
- ***slice.shuffle(rng) :** dé-trie les éléments d'une tranche en place au hasard. Notez que la tranche doit être transmise par référence **mut**, fort logiquement.*

Cargo.toml

```
[package]
name = "test-rust"
version = "0.1.0"
authors = ["manu"]
edition = "2018"

[dependencies]
rand = "0.8.4"
```

Nombres aléatoires

```
fn main() {
  use rand::prelude::*;

  let mut rng = thread_rng();
  let mut nombres: Vec<i32> = (1..20).collect();
  println!("Avant : {:?}", nombres);
  nombres.shuffle(&mut rng);
  println!("Après : {:?}", nombres);
  if let Some(valeur) = nombres.choose(&mut rng) {
    println!("Valeur au hazard : {}", valeur);
  }
  if let Some(valeur) = nombres.choose(&mut rng) {
    println!("Valeur au hasard : {}", valeur);
  }
}
```

Résultat

```
Avant : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
Après : [3, 13, 2, 8, 11, 7, 5, 4, 19, 14, 1, 10, 6, 16, 18, 12, 15, 17, 9]
Valeur au hasard : 6
Valeur au hasard : 14
```

VecDeque<T>

Les opérations d'ajout et d'enlèvement d'un élément dans une vecteur **Vec** ne sont efficaces que pour le dernier élément. Les performances sont moins bonnes dès qu'il faut intervenir sur une autre position que la dernière.

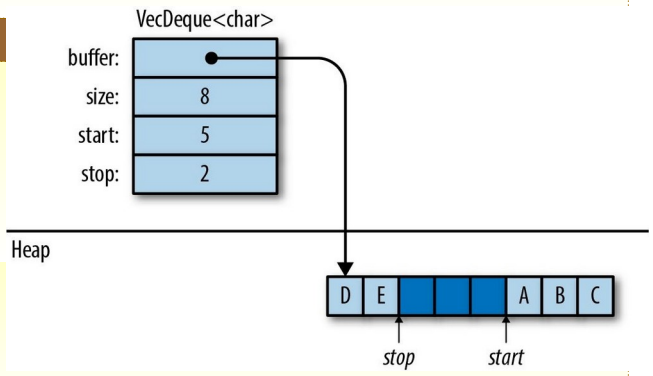
***Rust** propose la collection **std::collections::VecDeque<T>** qui est une queue à double extrémité, que nous appelons une **dequeue**, à prononcer « deux-queue ». Elle permet d'ajouter et d'enlever efficacement des éléments à l'avant comme à l'arriére.*

- ***deque.push_front(valeur) :** rajoute une valeur en début de **queue**.*
- ***deque.push_back(valeur) :** ajoute une valeur en fin de queue. Cette méthode sert beaucoup plus souvent que la précédente parce que les queues sont souvent utilisées en mode **PEPS** – premier entré, premier sortie (**FIFO**), c'est-à-dire que **les valeurs sont ajoutées à la fin mais enlevées au début** comme dans une vrai file d'attente.*
- ***deque.pop_front() :** enlève la valeur de début de **queue** et la renvoie en tant que **Option<T>** qui vaut **None** si la **queue** est vide, de la même manière que **vec.pop()**.*
- ***deque.pop_back() :** enlève et renvoie la valeur trouvée à la fin de la **queue** en renvoyant un **Option<T>**.*
- ***deque.front()** et **deque.back() :** s'apparentent à **vec.first()** et **vec.last()** en renvoyant une référence de devant ou de derrière. La valeur renvoyée est **Option<T>** qui vaut **None** si la **queue** est vide.*
- ***deque.front_mut()** et **deque.back_mut() :** sont similaires à **vec.first_mut()** et **vec.last_mut()** en renvoyant un **Option<&mut T>**.*

*Comme le montre la figure ci-dessous, une collection **VecDeque** est implémenté en tant que tampon circulaire. Comme un vecteur normal, la double-queue réserve une zone dans le tas pour le contenu, mais à la différence de **Vec**, les données ne sont pas nécessairement stockées en commençant au début de la région, et un rebouclage peut se produire.*

*L'ordre des éléments dans cet exemple est **['A', 'B', 'C', 'D', 'E']**. La collection **VecDeque** possède des champs privés qui portent les noms **start** et **stop** dans la figure et permettent de savoir où les données commencent et se finissent dans le tampon. Pour ajouter une valeur dans la **queue**, il faut réclamer une case libre (en bleu foncé) ou bien reboucler ou alors réserver un tampon mémoire plus spacieux.*

Implantation mémoire d'une collection VecDeque



*Les deux méthodes dont vous aurez presque toujours besoin dans une **double-queue** sont **push_back()** et **pop_front()** pour gérer de façon classique une **FIFO**. Les méthodes statiques pour créer une queue **VecDeque::new()** et **VecDeque::with_capacity()** s'apparentent à celle de **Vec**. La plupart des méthodes de **Vec** sont également disponibles pour **VecDeque : len()**, **is_empty()**, **insert(index, valeur)**, **remove(index)**, **extend(itérable)**, etc.*

*Comme un vecteur, vous pouvez scruter une double-queue par valeurs, par référence partagées ou par références exclusives mutables. Vous disposez de trois méthodes d'itérateur : **into_iter()**, **Iter(()** et **iter_mut()**. L'accès par indice est classique : **deque[index]**.*

En revanche, les double-queue ne peuvent pas hériter de toutes les méthodes des tranches parce qu'elles ne stockent pas les éléments de façon contiguë en mémoire. Pour réaliser des opérations habituelles sur les vecteurs et les tranches avec les données d'une double-queue, vous devez convertir VecDeque en Vec, faire l'opération puis reconvertir dans l'autre sens :

- ***Vec<T>)** implémente **From<VecDeque<T>> :** ce qui permet à **Vec::from(deque)** de convertir une deux-queue en vecteur, ce qui dure **O(n)** temps, car les éléments devront peut-être être triés de nouveau.*
- ***VecDeque<T>** implémente **From<Vec<T>> :** ce qui permet à **VecDeque::from(vec)** de convertir un vecteur en deux-queue. Cela prend également **O(n)** mais l'opération est normalement rapide même avec un grand vecteur parce que la zone réservée dans le tas pour le vecteur peut être directement transférée à la deux-queue.*

Nombres stockés dans une pile FIFO

```
fn main() {
  use std::collections::VecDeque;
  let mut nombres = VecDeque::with_capacity(7);
  nombres.push_back(5);
  nombres.push_back(7);
  nombres.push_back(11);
  println!("{:?}, taille={}, capacité={}", nombres, nombres.len(), nombres.capacity());
  while let Some(nombre) = nombres.pop_front() {
    println!("{:?} => enlève = {} ", nombres, nombre);
  }
}
```

Résultat

```
[5, 7, 11], taille=3, capacité=7
[7, 11] => enlève = 5
[11] => enlève = 7
[] => enlève = 11
```

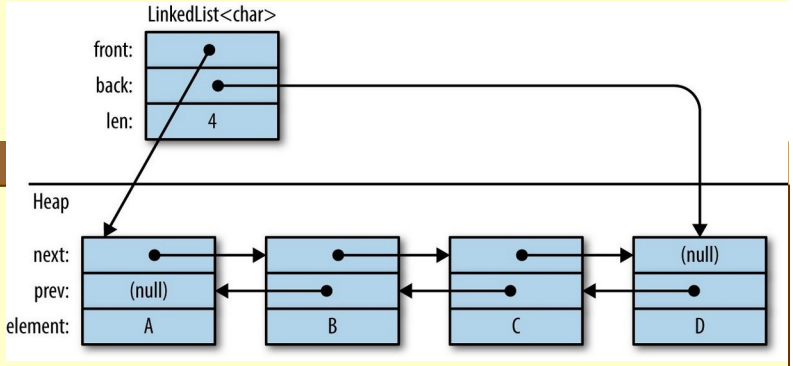
LinkedList<T>

La liste liée constitue une autre technique classique pour stocker une séquence de valeurs, chaque valeur étant placée dans une zone distincte du tas comme la figure ci-dessous. La double liste liée de **Rust** correspond à **std::collections::LinkedList<T>**. Elle dispose d'une sélection de méthodes de **VecDeque**.

*Toutes les méthodes qui travaillent avec **front()** et **back()** sont disponibles ainsi que les méthodes d'itérateurs, **LinkedList::new()**, et quelques autres. En revanche, les méthodes qui accèdent aux éléments par un indice sont absentes, car une liste liée n'est en général pas efficace dans ce mode d'accès indicé.*

Le principal avantage de LinkedList par rapport à VecDeque pour le moment est que la combinaison de deux listes est très performante. La méthode qui déplace tous les éléments d'une liste dans une autre liste.append(&mut liste2) se limite à la modification de quelques pointeurs, ce qui prend un temps constant.

Au contraire, les méthodes append() de Vec et VecDeque doivent parfois prendre beaucoup de temps pour déplacer de nombreuses valeurs d'un tableau vers un autre dans la zone du tas.



Liste doublement chaînée

```
fn main() {
  use std::collections::LinkedList;

  let mut nombres = LinkedList::new();
  nombres.push_back(5);
  nombres.push_back(7);
  nombres.push_back(11);
  println!("{:?}, taille={}", nombres, nombres.len());
  while let Some(nombre) = nombres.pop_front() {
    println!("{:?} => enlève = {} ", nombres, nombre);
  }
}
```

Résultat

```
[5, 7, 11], taille=3
[7, 11] => enlève = 5
[11] => enlève = 7
[] => enlève = 11
```

BinaryHeap<T>

Un tas binaire **BinaryHeap** contient des éléments plus ou moins triés, en garantissant que la plus grande valeur remonte toujours en début de queue. Bien sûr, vous pouvez stocker autre chose que des valeurs numériques. Tous les types de valeurs qui disposent du trait standard **Ord** sont acceptables. Voici les trois méthodes pour **BinaryHeap** :

- ***tas.push(valeur) :** ajoute une valeur à la collection.*
- ***tas.pop() :** enlève et renvoie la plus grande valeur de la collection, avec le type **Option<T>** qui vaut **None** si la collection est vide.*
- ***tas.peek() :** renvoie une référence à la plus grande valeur dans la collection, avec le type **Option<T>**.*

***BinaryHeap** dispose d'un sous-ensemble des méthode de Vec, parmi lesquelles : **new(), len(), is_empty() capacity()** et **append(&mut tas2)**.*

*Vous pouvez ainsi vous servir de **BinaryHeap** pour une queue de tâche à réaliser. Vous définissez une structure implémentant **Ord** en fonction des priorités pour que les tâches les plus urgentes soient supérieures aux autres. Vous créez ensuite votre collection **BinaryHeap** pour stocker toutes les tâches en attente.*

*La méthode **pop()** renverra toujours la prochaine tâche la plus importante à réaliser. Comme dans les exemples précédents, pour consommer les valeurs de **BinaryHeap** dans l'ordre des priorités décroissantes, utilisez systématiquement la boucle **while**.*

Notes les plus hautes

```
fn main() {
  use std::collections::BinaryHeap;
  let mut notes = BinaryHeap::from(vec![8, 12, 18, 5, 14]);
  println!("{:?}, taille={}", notes, notes.len());
  while let Some(note) = notes.pop() {
    println!("{:?} => plus forte = {} ", notes, note);
  }
}
```

Résultat

```
[18, 14, 8, 5, 12], taille=5
[14, 12, 8, 5] => plus forte = 18
[12, 5, 8] => plus forte = 14
```

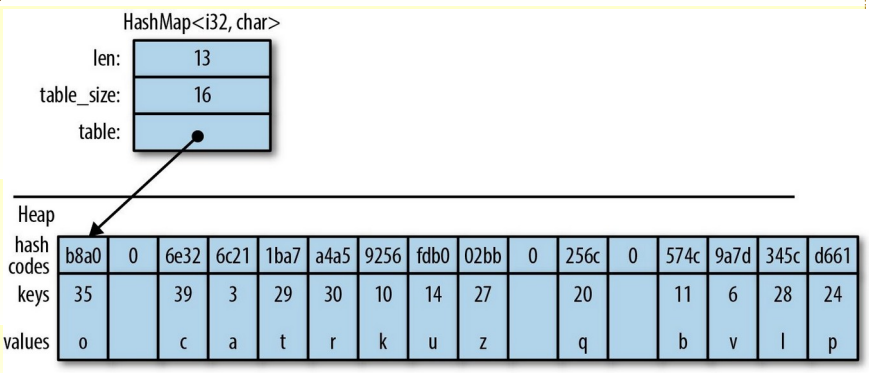
```
[8, 5] => plus forte = 12
[5] => plus forte = 8
[] => plus forte = 5
```

HashMap<K, V> et BtreeMap<K, V>

Une **mappe** est une collection de **paires clé-valeur** qui s'appellent des entrées. La **clé** est différente pour chaque entrée (**unique**) et les entrées sont stockées de telle manière qu'à partir de la **clé** vous pouvez facilement trouver la **valeur** dans la mappe. Une **mappe** s'apparente donc à un **dictionnaire** ou à une table de recherche (**lookup**).

***Rust** propose deux genres de **mappes** : une mappe de hachage **HashMap<K, V>** et une mappe d'arbre binaire **BtreeMap<K, V>**. La majorité de leurs méthodes sont identiques. Ce qui les distingue est la façon dont sont organisées les entrées pour permettre une recherche rapide.*

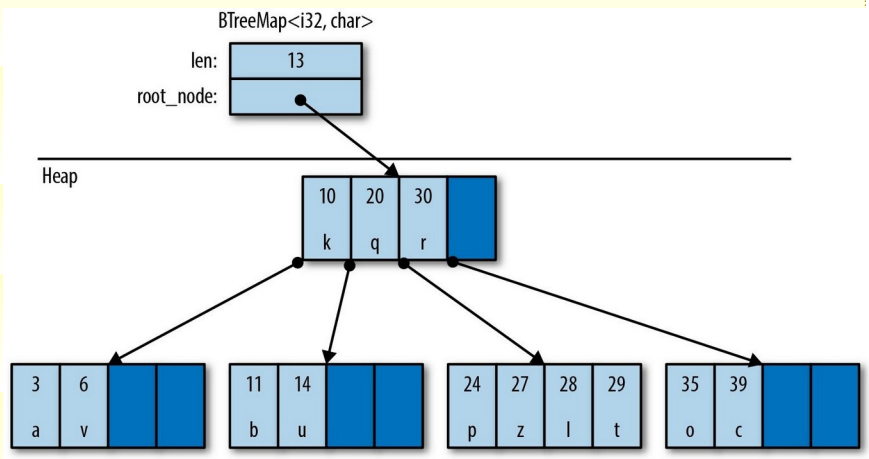
*Pour une **HashMap**, les **clés** et **valeurs** sont stockées dans une table de hachage. Les clés doivent donc être du type **K** qui implémente **Hash** et **Eq**, c'est-à-dire les deux traits pour les opérations de hachage et les tests d'égalité.*



*La figure ci-contre montre l'implantation mémoire d'une mappe **HashMap**. Toutes les **clés**, toutes les **valeurs** et tous les codes de hachage en cache sont placés dans une seule table implanté dans le tas.*

Lorsque l'insertion d'une nouvelle entrée provoque un débordement, la totalité de la table est déplacée dans un nouveau bloc avec tout son contenu.

*La mappe **BTreeMap** stocke ses entrées dans l'ordre des **clés** dans une structure arborescente. Le type **K** de la clé doit donc implémenter **Ord**.*



La figure ci-contre montre l'implantation mémoire avec les zones en bleu foncé pour les espaces libres.

*Dans une mappe **BTreeMap**, les entrées correspondent à des nœuds et la plupart de ces nœuds ne contiennent qu'une paire **clé-valeur**.*

*En revanche, les nœuds qui ne sont pas des feuilles terminales, et notamment le nœud racine de la figure, acceptent également des pointeurs vers les nœuds enfants. Dans l'exemple, le pointeur **(20, 'q')** et **(30, 'r')** pointe sur un nœud enfant qui contient toutes les clés entre **20** et **30**.*

*L'insertion d'une entrée suppose souvent de décaler les entrées existantes vers la droite pour maintenir l'ordre de tri, ce qui amène parfois à réserver de l'espace pour les nouveaux nœuds. Précisons que la figure est simplifiée. Les vrais nœuds de **BTreeMap** offrent **11** entrées et non seulement **4**.*

La librairie standard Rust utilise de préférence les arbres-B et non des arbres binaires équilibrés car ils sont plus rapides sur les machines modernes. Les arbres binaires demandent moins de comparaison pour une recherche donnée, mais les recherches dans un arbre-B restent plus locales, c'est-à-dire que les accès mémoire sont proches et non disséminés dans toute la zone du tas. Il en résulte un moins grand nombre d'erreurs de cache processeur, ce qui profite grandement aux performances.

- ***HashMap::new()** et **BtreeMap :: new() :** crée une nouvelle mappe vide.*
- ***iter.collect() :** permet de créer puis renseigner une nouvelle mappe **HashMap** ou **BTreeMap** grâce à des paires clé-valeur. **Iter** doit être du type **Iterator<Item=(K, V)>**.*
- ***HashMap::width_capacity(n) :** crée une nouvelle mappe de hachage vide offrant assez d'espace pour au moins **n** entrées. Comme un vecteur, une table de hachage stocke les données dans une seule réservation dans le tas. Sa capacité est donc prédéfinie et elle dispose des méthodes **capacity()**, **reserve(autre)** et **shrink_to_fit()** (alors que les mappes binaires **BTreeMap** n'en disposent pas).*
- ***map.len()** : renvoie le nombre d'entrées.*
- ***map.is_empty()** : renvoie **true** si la mappe est vide.*
- ***map.contains_key(&clé)** : renvoie **true** si la mappe contient une entrée pour la clé fournie.*
- ***map.get(&clé)** : recherche l'entrée spécifiée dans la mappe. Si elle est trouvée, renvoie **Some(r)** avec **r** une référence à la valeur ou bien renvoie **None**.*
- ***map.get_mut(&clé)** : est similaire à la précédente en renvoyant une référence mutable. En général, vous pouvez accéder de façon mutable aux valeurs stockées dans une mappe, mais pas aux **clés**. Vous avez le contrôle sur les valeurs mais*

*les **clés** appartiennent à la mappe et il est indispensable qu'elles ne changent pas puisqu'elles déterminent l'ordre de stockage. Si vous modifiez une **clé** en place, vous provoquez un bogue.*

- ***map.insert(clé, valeur)** : insère une entrée **(clé, valeur)** dans la mappe. Si elle existe déjà avec la même **clé**, la nouvelle valeur remplace l'ancienne. La méthode renvoie l'ancienne valeur éventuelle et le type renvoyé est **Option<V>**.*
- ***map.extend(itérable)** : réalise une itération parmi les **(K, V)** éléments de **itérable** pour insérer chacune des paires **clé-valeur** dans la mappe.*
- ***map.append(&mut map2)** : vide **map2** en déplaçant toutes ses entrées dans **map**.*
- ***map.remove(&clé)** : localise puis supprime une entrée pour la **clé** fournie dans la mappe. Renvoie la valeur supprimée si c'est le cas avec un type renvoyé **Option<V>**.*
- ***map.clear()** : supprime toutes les entrées d'une mappe.*

*Vous pouvez utiliser la syntaxe basée sur les crochets droits pour rechercher dans une mappe : **map[&clé]**. Autrement dit, les mappes implémentent le trait standard **Index** mais vous provoquez une panique si la clé demandée ne correspond pas à une entrée, ce qui équivaut à un accès en dehors des bornes. N'utilisez donc cette syntaxe que si vous êtes certain que l'entrée recherchée existe.*

*Pour toutes les méthodes **contains_key(), get(), get_mut()** et **remove()**, le paramètre de clé n'est pas nécessairement du type exact **&K**. En effet, ces méthodes sont génériques et acceptent tous les types pouvant être empruntés à partir de **K**. Vous pouvez donc faire un appel **poisson_map.contains_key(« congrès »)** sur une mappe **HashMap<String, Poisson>**, alors que **« congrès »** n'est pas exactement du type **String**, puisque **String** implémente **Borrow<&str>**.*

*Les mappes à arbre binaire **BTreeMap<K, V>** maintiennent les entrées dans l'ordre de tri des **clés**, ce qui leur permet de supporter une opération de plus :*

- ***btree_map.split_at(&clé)** : découpe la mappe en deux. Les entrées dont la **clé** est inférieure à celle demandée sont maintenues dans la mappe de départ et la méthode renvoie une nouvelle mappe **BtreeMap<K, V>** contenant les autres entrées.*

Répertoire téléphonique

```
fn main() {
  let mut telephones = std::collections::BTreeMap::new();
  telephones.insert("Marcel", "06-89-77-56-21");
  telephones.insert("Bruno", "06-96-12-44-02");
  telephones.insert("Aurore", "06-47-63-25-78");
  for (prenom, telephone) in telephones {
    println!("Prénom={}, Téléphone={}", prenom, telephone);
  }
}
```

Résultat

```
Prénom=Aurore, Téléphone=06-47-63-25-78
Prénom=Bruno, Téléphone=06-96-12-44-02
Prénom=Marcel, Téléphone=06-89-77-56-21
```

Entrées d'une mappe

Les deux styles de mappes, **HashMap** et **BtreeMap** sont liés au type **Entry**. Le but de ce dernier est d'éviter les recherches redondantes. Prenons comme hypothèse l'extrait suivant qui permet de connaître ou de créer un nouveau numéro de téléphone :

Répertoire téléphonique

```
fn main() {
  let mut telephones = std::collections::BTreeMap::new();
  telephones.insert("Marcel", "06-89-77-56-21");
  telephones.insert("Bruno", "06-96-12-44-02");
  telephones.insert("Aurore", "06-47-63-25-78");
  for (prenom, telephone) in &telephones {
    println!("Prénom={}, Téléphone={}", prenom, telephone);
  }
  // Cette personne a-t-elle déjà un numéro de téléphone ?
  if !telephones.contains_key("Michel") {
    // Non : Création d'une nouvelle entrée
    telephones.insert("Michel", "06-89-96-74-14");
  }
  let telephone = telephones["Michel"];
  println!("Numéro de Michel : {}", telephone);
}
```

Résultat

```
Prénom=Aurore, Téléphone=06-47-63-25-78
Prénom=Bruno, Téléphone=06-96-12-44-02
Prénom=Marcel, Téléphone=06-89-77-56-21
Numéro de Michel : 06-89-96-74-14
```

*Le code fonctionne parfaitement, mais nous accédons deux ou trois fois à **téléphones**, pour répéter exactement le même type de recherche.*

*Le but des entrées est de maintenir en quelque sorte une porte ouverte. Une fois que la recherche est faite, nous produisons une valeur **Entry** qui est utilisée pour les recherches suivantes. L'expression sur une ligne qui suit équivaut à la totalité du code ci-dessus, en ne faisant qu'une seule fois la recherche.*

Répertoire téléphonique

```
fn main() {
  let mut telephones = std::collections::BTreeMap::new();
  telephones.insert("Marcel", "06-89-77-56-21");
  telephones.insert("Bruno", "06-96-12-44-02");
  telephones.insert("Aurore", "06-47-63-25-78");
  for (prenom, telephone) in &telephones {
    println!("Prénom={}, Téléphone={}", prenom, telephone);
  }
  let telephone = telephones.entry("Michel").or_insert("06-89-96-74-14");
  println!("Numéro de Michel : {}", telephone);
  let telephone = telephones.entry("Aurore").or_insert("06-89-96-74-14");
  println!("Numéro de Aurore : {}", telephone);
}
```

Résultat

```
Prénom=Aurore, Téléphone=06-47-63-25-78
Prénom=Bruno, Téléphone=06-96-12-44-02
Prénom=Marcel, Téléphone=06-89-77-56-21
Numéro de Michel : 06-89-96-74-14
Numéro de Aurore : 06-47-63-25-78
```

*La valeur de type **Entry** qui est renvoyée par **telephones.entry(« Michel »)** incarne une référence mutable vers la position dans la mappe qui soit occupée par une valeur **clé-valeur**, soit libre, ce qui signifie que cette personne n'est pas encore connu. Dans ce dernier cas, nous appelons à la méthode **or_insert()** pour insérer un nouveau numéro de téléphone.*

*La plupart des opérations concernant les entrées prennent le même aspect : bref et élégant. Toutes les valeurs de type **Entry** sont créées par la même méthode :*

- ***map.entry(clé)** : renvoie une entrée pour la **clé** fournie. Si la **clé** n'existe pas dans la mappe, renvoie une entrée vide, vacante. La méthode reçoit son paramètre **self** sous forme de référence mutable puis renvoie une valeur **Entry** dont la durée de vie est synchronisée :*

  ```
  pub fn entry<'a>(&'a mut self, key: K) -> Entry<'a, K, V>
  ```

  *Le type **Entry** possède un paramètre de durée de vie **'a** parce que c'est une sorte de référence mutable empruntée à la mappe. Tant que cet élément **Entry** existe, il dispose d'un accès exclusif à cette mappe.*
- ***map.entry(clé).or_insert(valeur)** : garantit que la mappe va contenir une entrée avec la **clé** fournie, en insérant une nouvelle entrée si nécessaire. Elle renvoie une référence mutable à la valeur trouvée ou venant d'être insérée.*
- ***map.entry(clé).or_insert_with(défaut_fn)** : ressemble à la **précédente** sauf qu'en cas de création d'une entrée, elle appelle **défaut_fn()** pour produire la valeur à insérer. Cet appel n'est pas réalisé si une entrée existe déjà pour la **clé**.*

*Le type de **Entry** est une énumération, il est défini comme ci-dessous pour **HashMap**, et il est défini comme ci-dessous pour **HashMap** ainsi que pour **BtreeMap** :*

Énumération Entry

```
pub enum Entry<'a, K: 'a, V: 'a> {
  Occupied(OccupiedEntry<'a, K, V>),
  Vacant(VacantEntry<'a, K, V>)
}
```

*Les deux types **OccupiedEntry** et **VacantEntry** sont dotés de méthodes permettant d'insérer, d'enlever et d'accéder aux entrées sans répéter la recherche. Ces méthodes sont utiles pour éliminer une recherche redondante ou deux, mais en général **or_insert()** et **or_insert_width()** répondent aux besoins courants.*

Itérations dans une mappe

Tout comme les vecteurs, les mappes disposent des deux méthodes **iter()** et **iter_mut()** qui savent renvoyer un itérateur par référence et itérer parmi des mappes **&map** ou **&mut map**. Plusieurs approches sont possibles pour parcourir le contenu d'une mappe.

- *Une itération par **valeur** – **for (K, V) in map** – produit des paires **(K, V)** et consomme la mappe.*
- *Une itération par **référence partagée** – **for (K, V) in &map** – produit des paires **(&K, &V).***
- *Une itération par **référence mutable** – **for (K, V) in &mut map** – produit des paires **(&K, &mut V).** Rappelons qu'il est impossible d'obtenir un accès mutable aux **clés** d'une mappe parce que l'ordre des entrées se base sur ces **clés**.*
- ***map.keys()** : renvoie un itérateur pour parcourir les **clés**, par **référence**.*
- ***map.values()** : renvoie un itérateur pour parcourir les **valeurs**, par **référence**.*

- ***map.values_mut()*** *: renvoie un itérateur pour parcourir les* ***valeurs****, par* ***référence mutable****.*

Tous les itérateurs de HashMap consultent les entrées de la mappe dans un ordre non figé. Les itérateurs de BTreeMap consultent les entrées dans l'ordre des clés.

Répertoire téléphonique

```
fn main() {
  let mut telephones = std::collections::BTreeMap::new();
  telephones.insert("Marcel", "06-89-77-56-21");
  telephones.insert("Bruno", "06-96-12-44-02");
  telephones.insert("Aurore", "06-47-63-25-78");
  print!("Prénoms: ");
  for prenom in telephones.keys() { print!("{}  ", prenom);  }
  print!("\nNuméros: ");
  for numero in telephones.values() { print!("{}  ", numero);  }
  println!("\nRépertoire :");
  for (prenom, telephone) in telephones {
    println!("Prénom = {}, Téléphone = {}", prenom, telephone);
  }
}
```

Résultat

```
Prénoms: Aurore  Bruno  Marcel
Numéros: 06-47-63-25-78  06-96-12-44-02  06-89-77-56-21
Répertoire :
Prénom = Aurore, Téléphone = 06-47-63-25-78
Prénom = Bruno, Téléphone = 06-96-12-44-02
Prénom = Marcel, Téléphone = 06-89-77-56-21
```

HashSet<T> et BTreeSet<T>

Un ensemble (**set**) est une collection de valeurs organisées afin que les tests de présence dans la collection soient rapides. Le principe fondamental d'un ensemble est que chaque élément est unique, et en conséquence un ensemble ne contient jamais deux exemplaires de la même valeur.

Les mappes et les ensembles ne possèdent pas les mêmes méthodes, mais en coulisses, un ensemble équivaut à une mappe ne contenant que des ***clés*** *et non des paires* ***clé-valeur****. D'ailleurs, les deux types d'ensembles de Rust* ***HashSet<T>*** *et* ***BTreeSet<T>*** *sont des définitions enveloppes créées autour de* ***HashMap<T, ()>*** *et* ***BtreeMap<T, ()>****.*

- ***HashSet::new()*** *et* ***BtreeSet::new()*** *: servent à créer un nouvel ensemble.*
- ***iter.collect()*** *: crée un ensemble à partir d'un itérateur. Si* ***iter*** *produit des doublons, ils sont automatiquement supprimés.*
- ***HashSet::with_capacity(n)*** *: crée un ensemble* ***HashSet*** *vide avec assez de place pour au moins* ***n*** *valeurs.*
- ***set.len()*** *: renvoie le nombre de valeurs dans l'ensemble* ***set****.*
- ***set.is_empty()*** *: renvoie* ***true*** *si l'ensemble est vide.*
- ***set.contains(&valeur)*** *: renvoie* ***true*** *si l'ensemble contient la* ***valeur****.*
- ***set.insert(valeur)*** *: insère la valeur dans l'ensemble et renvoie* ***true*** *s'il a réussi ou* ***false*** *si elle existait déjà.*
- ***set.remove(&valeur)*** *: enlève une valeur dans l'ensemble et renvoie* ***true*** *s'il a réussi ou* ***false*** *si elle n'était pas présente.*
- ***set.iter()*** *: renvoie un itérateur sur les membres de l'ensemble par référence.*

Comme dans le cas des mappes, les méthodes pour chercher une valeur par référence sont génériques pour tous les types qui peuvent être empruntés depuis T. Deux approches sont possibles pour parcourir un ensemble.

- *L'itération par* ***valeur*** *–* ***for V in set*** *– permet de produire les membres de l'ensemble en consommant celui-ci.*
- *L'itération par* ***référence partagée*** *–* ***for V in &set*** *– sert à produire des références partagées sur les membres de l'ensemble.*

Vous ne pouvez pas parcourir un ensemble avec des références mutables. Vous ne pouvez pas obtenir une référence mutable sur une valeur de l'ensemble.

Les itérateurs ***HashSet*** *comme ceux de* ***HashMap*** *produisent leurs valeurs dans un ordre quelconque. Les itérateurs* ***BTreeSet*** *produisent les valeurs dans l'ordre, comme dans le cas d'un vecteur trié.*

Ensemble de nombres aléatoires

```
use std::collections::HashSet;
use rand::prelude::*;

fn main() {
  let mut aleatoire = thread_rng();
  let mut nombres : HashSet<i32> = (1..10).collect();
  println!("{:?}", nombres);
  for _ in 0..10 {
    let enleve = aleatoire.gen_range(1..10);
    let ajoute = aleatoire.gen_range(1..10);
```

```
    nombres.remove(&enleve);
    nombres.insert(ajoute);
    println!("{:?} -({}) +({})", nombres, enleve, ajoute);
  }
}
```

Résultat

```
{3, 8, 9, 1, 5, 4, 6, 2, 7}
{3, 8, 9, 1, 5, 4, 2, 7} -(6) +(5)
{3, 8, 9, 1, 5, 4, 2} -(7) +(5)
{3, 9, 1, 5, 4, 2, 7} -(8) +(7)
{9, 1, 5, 4, 2, 7} -(3) +(9)
{8, 9, 1, 5, 4, 7} -(2) +(8)
{8, 9, 1, 5, 4, 7} -(3) +(9)
{8, 9, 1, 5, 7} -(4) +(9)
{8, 9, 1, 5, 7} -(5) +(5)
{8, 9, 1, 5, 2, 7} -(3) +(2)
{8, 9, 1, 4, 2, 7} -(5) +(4)
```

Valeurs égales mais différentes

Les ensembles disposent de quelques méthodes qui ne serviront que si vous avez besoin de distinguer entre des valeurs égales. Ce genre de différence est assez fréquent. Par exemple, deux valeurs de type **String** stockent leurs caractères dans deux emplacements mémoire différents. Dans la majorité des cas, cela n'a pas d'importance.

*Mais si vous avez besoin de faire cette distinction, il est possible d'accéder aux valeurs qui sont stockées dans un ensemble au moyen d'une des méthodes suivantes. Chacune renvoie un **Option** qui vaut **None** si l'ensemble ne contenait pas la valeur recherchée.*

- ***set.get(&valeur)** : renvoie une référence partagée aux membres de l'ensemble égaux à la valeur et renvoie un **Option<&T>**.*
- ***set.take(&valeur)** : ressemble à **set.remove(&valeur)**, mais renvoie la valeur enlevée si elle existe sous la forme d'un **Option<T>**.*
- ***set.replace(valeur)** : ressemble à **set.insert(&valeur)**, sauf que si l'ensemble contient déjà une valeur égale à celle fournie, elle est remplacée et l'ancienne est renvoyée sous la forme d'un **Option<T>**.*

Ensemble de nombres aléatoires

```
use std::collections::HashSet;
use rand::prelude::*;

fn main() {
  let mut aleatoire = thread_rng();
  let mut nombres : HashSet<i32> = (1..10).collect();
  println!("{:?}", nombres);
  for indice in 1..11 {
    if let Some(enleve) = nombres.take(&&aleatoire.gen_range(1..10)) {
      println!("{:?} n°{} ({})", nombres, indice, enleve);
    }
  }
}
```

Résultat

```
{1, 7, 8, 9, 5, 2, 3, 4, 6}
{1, 8, 9, 5, 2, 3, 4, 6} n°1 (7)
{1, 8, 9, 5, 3, 4, 6} n°2 (2)
{1, 8, 9, 5, 3, 4} n°4 (6)
{1, 8, 9, 5, 4} n°5 (3)
{1, 9, 5, 4} n°7 (8)
```

Opérations globales sur les ensembles

Les méthodes que nous venons de décrire s'intéressent à une valeur dans un ensemble, mais vous pouvez également créer des opérations qui combinent des ensembles complets. Effectivement, lorsque nous abordons la théorie des ensembles, nous avons souvent besoin des notions d'intersection, d'union, de différences, etc..

*Mais si vous avez besoin de faire cette distinction, il est possible d'accéder aux valeurs qui sont stockées dans un ensemble au moyen d'une des méthodes suivantes. Chacune renvoie un **Option** qui vaut **None** si l'ensemble ne contenait pas la valeur recherchée.*

- ***set1.intersection(&set2)** : renvoie un itérateur sur toutes les valeurs qui existent dans les deux ensembles. Nous pouvons par exemple afficher les noms de tous les étudiants qui sont inscrits simultanément aux cours de neurochirurgie et d'astronautique. Nous disposons même de l'opérateur « **&** » qui effectue l'intersection entre deux ensembles.*

  ***&set1 & &set2** renvoie l'ensemble correspondant à l'intersection des deux ensembles, ce qui correspond à l'opérateur binaire **ET** appliqué à deux références.*

- ***set1.union(&set2)*** *: renvoie un itérateur sur les valeurs qui sont soit dans* ***set1****, soit dans* ***set2****, soit dans les deux.*

  &set1 | &set2 *renvoie un nouvel ensemble contenant ces valeurs, ce qui correspond à* ***set1 OU set2.***
- ***set1.difference(&set2)*** *: renvoie un itérateur sur les valeurs trouvées dans* ***set1*** *mais pas dans* ***set2****.*

  &set1 - &set2 *renvoie un nouvel ensemble contenant ces valeurs****.***
- ***set1.symmetric_difference(&set2)*** *: renvoie un itérateur sur les valeurs qui sont soit dans* ***set1****, soit dans* ***set2****, mais pas les deux.*

  &set1 ^ &set2 *renvoie un nouvel ensemble contenant ces valeurs. L'opération correspond au* ***OU*** *exclusif.*
- ***set1.is_disjoint(set2)*** *: renvoie* ***true*** *si* ***set1*** *et* ***set2*** *n'ont aucunes valeur en commun (intersection vide).*
- ***set1.is_subset(set2)*** *: renvoie* ***true*** *si* ***set1*** *est un sous-ensemble de* ***set2****, c'est-à-dire que toutes ses valeurs sont dans* ***set2****.*
- ***set1.is_superset(set2)*** *: est l'inverse : elle renvoie* ***true*** *si* ***set1*** *contient entièrement* ***set2****.*
- ***==*** *et* ***!=*** *: permettent de tester l'égalité entre deux ensembles. Deux ensembles sont considérés comme égaux s'ils contiennent exactement les mêmes valeurs.*

Théorie des ensembles

```
use std::collections::BTreeSet;

fn main() {
  let premiers : BTreeSet<i32> = (1..10).collect();
  let deuxiemes : BTreeSet<i32> = (5..15).collect();
  println!("{:?} - {:?}", &premiers, &deuxiemes);
  println!("Intersection : {:?}", &premiers & &deuxiemes);
  println!("Union : {:?}", &premiers | &deuxiemes);
  println!("Différence : {:?}", &premiers - &deuxiemes);
  println!("Ou exclusif : {:?}", &premiers ^ &deuxiemes);
}
```

Résultat

```
{1, 2, 3, 4, 5, 6, 7, 8, 9} - {5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
Intersection : {5, 6, 7, 8, 9}
Union : {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
Différence : {1, 2, 3, 4}
Ou exclusif : {1, 2, 3, 4, 10, 11, 12, 13, 14}
```

Création de consonnes minuscules

```
use std::collections::BTreeSet;

fn main() {
  let minuscules : BTreeSet<char> = ('a'..='z').collect();
  let voyelles : BTreeSet<char> = "aeiouy".chars().collect();
  let consonnes = &minuscules - &voyelles;
  println!("{:?}", &consonnes);
  let bienvenue = "Bienvenue!";
  let mut nombre_voyelles = 0;
  let mut nombre_consonnes = 0;
  let mut nombre_caracteres = 0;
  for caractere in bienvenue.to_ascii_lowercase().chars() {
    if voyelles.contains(&caractere) { nombre_voyelles+=1; }
    else if consonnes.contains(&caractere) { nombre_consonnes+=1; }
    else { nombre_caracteres+=1; }
  }
  println!("{} possède {} voyelles {} consonnes et {} symbole",
           bienvenue, nombre_voyelles, nombre_consonnes, nombre_caracteres);
}
```

Résultat

```
{'b', 'c', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'q', 'r', 's', 't', 'v', 'w', 'x', 'z'}
Bienvenue! possède 5 voyelles 4 consonnes et 1 symbole
```