IFT339
Structures de données

Thème 5 : La bibliothèque normalisée en C++

Aïda Ouangraoua
Département d'informatique

UNIVERSITÉ DE
SHERBROOKE

Bibliothèque normalisée en C++

- Bibliothèque de classes génériques de conteneurs:
  vector, deque, list, set, map, stack, queue, etc.

- Conteneurs : composés de données de même type

- Uniformité dans la définition des conteneurs
  - Plusieurs opérations de bases en commun :
    Exemple: **size** : Ø → size_t : retournant le nombre d'éléments d'un conteneur
    Note : size_t est un alias pour un type d'entiers non signés

- Des différences dans l'implémentation et la complexité des opérations

- Conteneurs pouvant s'utiliser comme des types primitifs
  (affectation, paramètre et retour de fonctions)

4 Principaux groupes de conteneurs

- **Conteneurs séquentiels** : ordre des éléments défini par l'usager, en ajoutant les éléments au début, à la fin ou à une position donnée du conteneur.
  - Array : tableau statique aux éléments contigus en mémoire
  - Vector : tableau dynamique aux éléments contigus en mémoire
  - Deque : double file aux éléments semi-contigus en mémoire
  - List : liste aux éléments non contigus en mémoire
  - Forward_list : liste unidirectionnelle aux éléments non contigus en mémoire

- **Conteneurs associatifs** : ordre des éléments défini par le conteneur qui décide où ajouter les éléments en fonctions de leur valeur.
  - Set : ensemble d'éléments (valeurs)
  - Mutliset : ensemble avec doublons possibles
  - Map : fonction, ensemble de paires (clé, valeur)
  - Multimap : fonction avec doublons de clés possibles

4 Principaux groupes de conteneurs

- **Conteneurs associatifs sans ordre** : pas d'ordre défini sur les éléments
  - Unordered set
  - Unordered mutliset
  - Unordered map
  - Unordered multimap

- **Conteneurs adaptés**: utilisent comme représentation un autre conteneur
  - Stack : pile (adapté de list, vector ou deque)
  - Queue : file (adapté de list ou deque)
  - Priority_queue : file de priorité (adapté de vector)

Des opérations communes

- **Constructeur sans paramètre** : Ø → Ø : permet de créer un conteneur « this » vide (aucun espace allouée dynamiquement)
- **Constructeur avec paramètre** : size_t → Ø : permet de créer un conteneur « this » contenant un nombre n d'éléments
- **Constructeur par copie** : TypeConteneur → Ø
- **Affectateur** : TypeConteneur → Ø
- **Clear** : Ø → Ø : ramène le conteneur « this » à l'état de création avec le constructeur sans paramètre
- **Destructeur** : Ø → Ø : appelle l'opération « clear » pour nettoyer « this »

Des opérations communes

- **Size** : Ø → size_t : retourne le nombre d'éléments contenus dans le conteneur « this » (**dimension** du conteneur, pas dans forward_list)
- **Empty** : Ø → Bool : indique si le conteneur « this » est vide ou pas.
- **Swap** : TypeConteneur → Ø : échange « this » avec le conteneur en paramètre explicite
- **Max_size** : Ø → size_t : retourne le nombre d'éléments maximum
- **iterator** : iterateur sur les éléments

Conteneurs séquentiels

- **Array** : dimension (nombre d'éléments) statique fixée à la déclaration
  - Impossible d'ajouter ou retirer des éléments
  - Accès au ième élément en O(1) ; Recherche par valeur en O(n)
  - Représentation : tableau alloué statiquement (**éléments contigus en mémoire**)

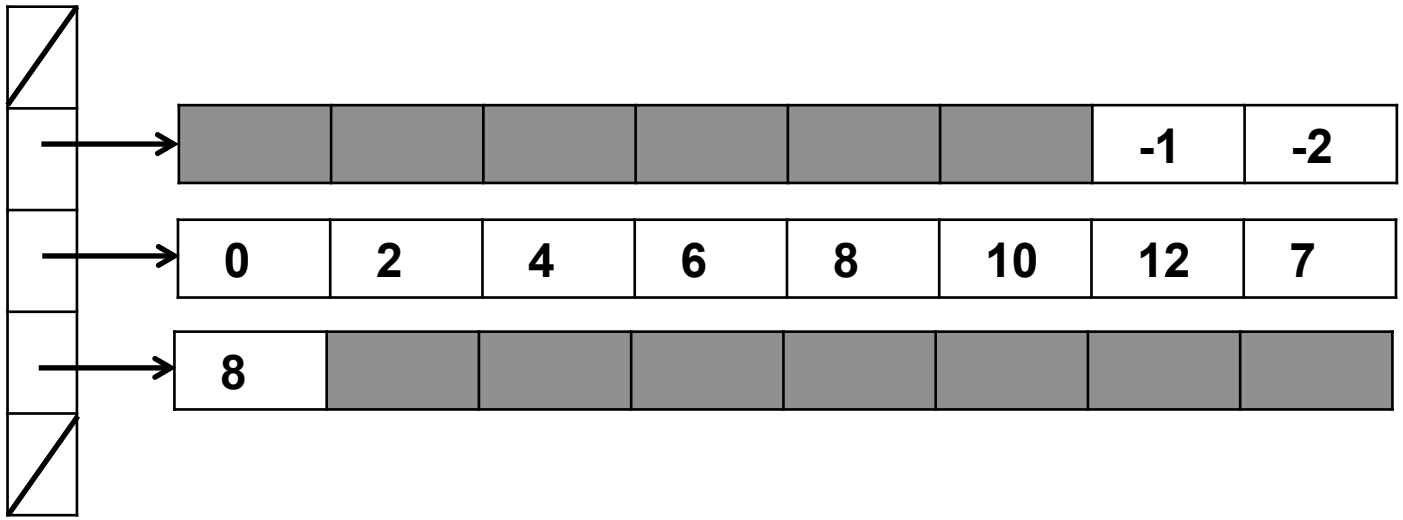
```
#include <array>
array<int,6> A;
//iteration a l'aide d'un entier et l'operator[]
for(int i = 0; i < A.size() ; i++)
    A[i] = 2*i;
```

- **Vector** : dimension dynamique (occupe plus d'espace mémoire que requis pour ses éléments)
  - Ajout/retrait d'élément à la fin du conteneur en O(1)
  - Ajout/retrait d'élément partout en O(n) (déplacement des éléments en mémoire)
  - Accès au ième élément en O(1) ; Recherche par valeur en O(n)
  - Représentation : tableau alloué dynamiquement, avec espace réservé à la fin pour les futurs ajouts d'éléments (**éléments contigus en mémoire**)

```
#include <vector>
vector<int> V(5);
//iteration
for(int i = 0; i < V.size() ; i++)
    V[i] = 2*i;
V.push_back(7);
```

- **Deque** : dimension dynamique (a également une réserve pour les futurs ajouts d'éléments)
  - Ajout/retrait d'éléments au début et à la fin du conteneur en O(1)
  - Ajout/retrait d'élément partout en O(n) (déplacement des éléments en mémoire)
  - Accès au ième élément en O(1) ; Recherche par valeur en O(n)
  - Représentation : plusieurs tableaux alloués statiquement, avec des espaces réservés au début et à la fin pour les futurs ajouts d'éléments (**éléments semi-contigus en mémoire**)

```
#include <deque>
deque<int> D(7);
//iteration a l'aide d'un entier
et l'operator[]
for(int i = 0; i < D.size() ; i++)
    V[i] = 2*i;
D.push_front(-1);
D.push_front(-2);
D.push_back(7);
D.push_back(8);
```

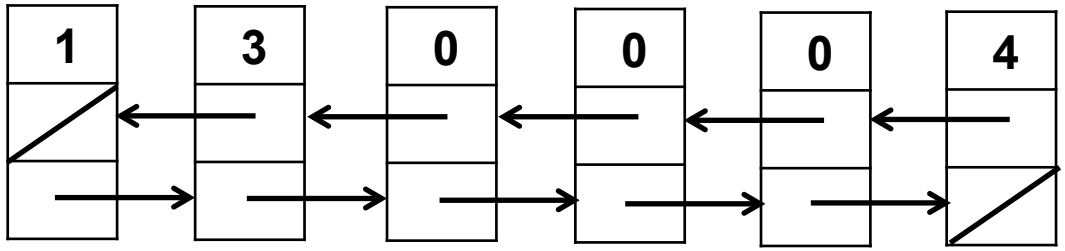


Opérations communes à Array (A), Vector (V), Deque (D)

- **Operator[] :** size_t → type : retourne une référence non vérifiée à un élément identifié par sa position
- **At :** size_t → type : retourne une référence vérifiée à un élément identifié par sa position
- **Shrink_to_fit** : ∅ → ∅ : retire la réserve (pas dans A)

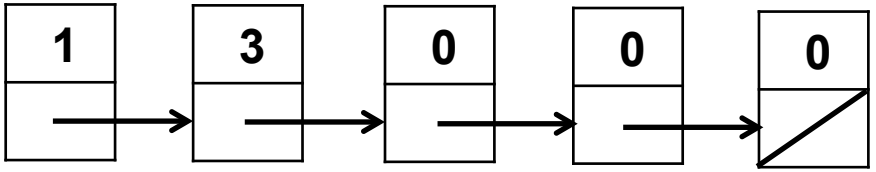
- **List** : dimension dynamique (ajout et retrait d'éléments possibles)
  - Ajout/retrait d'éléments partout en O(1)
  - Accès au ième élément en O(n) ; Recherche par valeur en O(n)
  - Représentation : cellules doublement chaînées (**éléments non contigus en mémoire**)

```
#include <list>
list<int> L(3);
L.push_front(1);
L.push_back(4);
list<int>::iterator i=L.begin();
i++;
L.insert(i,3);
//iteration a l'aide d'un iterator
for(i=L.begin(); i != L.end() ; i++)
  cout << *i << endl;
```



- **Forward_list** : dimension dynamique (ajout et retrait d'éléments possibles)
  - Ajout/retrait au début, ou après une position en O(1)
  - Accès au ième élément en O(n)
  - Recherche par valeur en O(n)
  - Représentation : cellules simplement chaînées (**éléments non contigus en mémoire**)

```
#include <forward_list>
list<int> F(3);
F.push_front(1);
forward_list<int>::iterator i=F.begin();
i++;
F.insert_after(i,3);
for(i=F.begin(); i != F.end() ; i++)
  cout << *i << endl;
```



Opérations communes à Array (A), Vector (V), Deque (D), List (L), Forward_list (F)

- **front :** Ø → type : retourne une référence au premier élément
- **back :** Ø → type : retourne une référence au dernier élément (pas dans F)
- **push_front :** type → Ø : ajoute un élément au début (pas dans A et V)
- **push_back :** type → Ø : ajoute un élément à la fin (pas dans A et F)
- **insert** : iterator x type → Ø : ajoute un élément à une position (pas dans A et F)
- **erase :** iterator → Ø : retire un élément à une position donnée (pas dans A)
- **pop_front :** Ø → Ø : retire le premier élément (pas dans A et V)
- **pop_back :** Ø → Ø : retire le dernier élément (pas dans A et F)
- **resize** : size_t → Ø : limite le nombre d'éléments à la taille donnée (pas dans A)

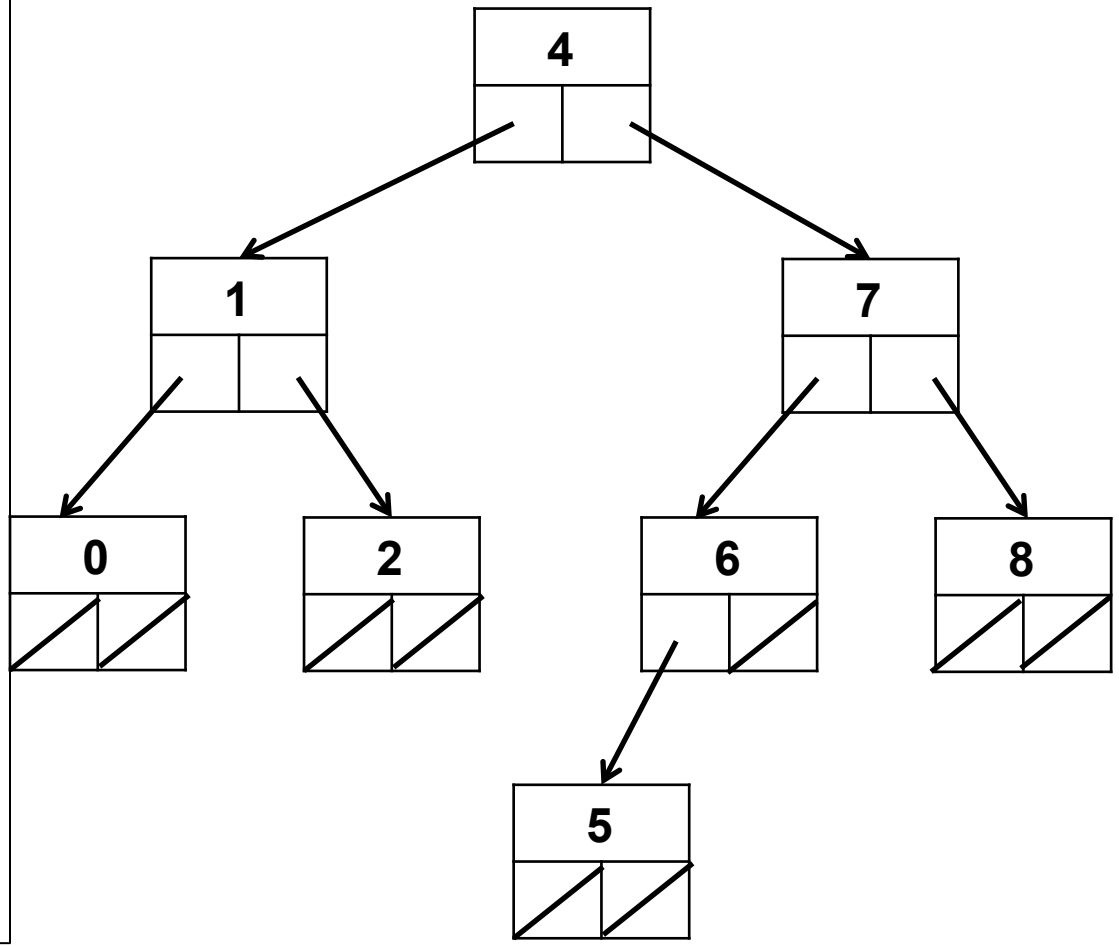
Opérations spécifiques aux listes : List (L), Forward_list (F)

- **remove :** type → Ø: retire toutes les occurrences d'un élément
- **insert_after :** iterator x type → Ø : ajoute un élément après une position donnée (seulement dans F)
- **erase_after :** iterator → Ø : retire un élément après une position (seulement dans F)
- **merge :** Liste (L ou F) → Ø : fusionne deux listes
- **reverse** : Ø → Ø : inverse l'ordre des éléments
- **sort** : Ø → Ø : trie les éléments

Conteneurs associatifs

- **Set** : dimension dynamique (on peut ajouter et retirer des éléments)
  - Ordre des éléments déterminé par l'ordre des valeurs (pas d'ajout d'élément en spécifiant la position)
  - Ajout/retrait d'éléments en O(1) (après recherche de la position)
  - Recherche par valeur en O(log(n))
  - Représentation : structure de recherche

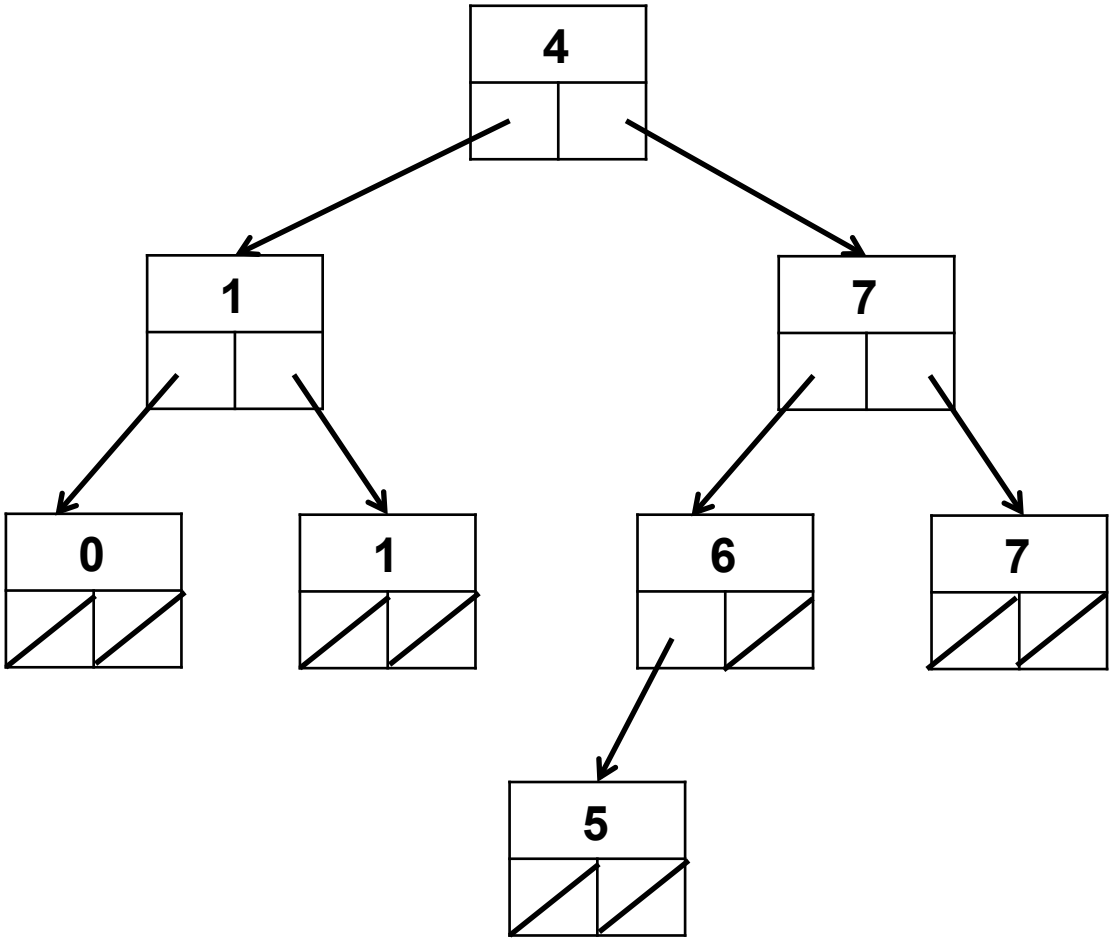
```
#include <set>
set<int> S;
S.insert(7); S.insert(1);
S.insert(4); S.insert(3);
S.insert(2); S.insert(8);
S.insert(0); S.insert(6);
S.insert(5);
set<int>::iterator i;
i = S.find(3);
S.erase(i);
//iteration a l'aide d'un
iterator
for(i=S.begin(); i !=
S.end() ; i++)
  cout << *i << endl;
```



Profondeur de l'arbre AVL: log(n)

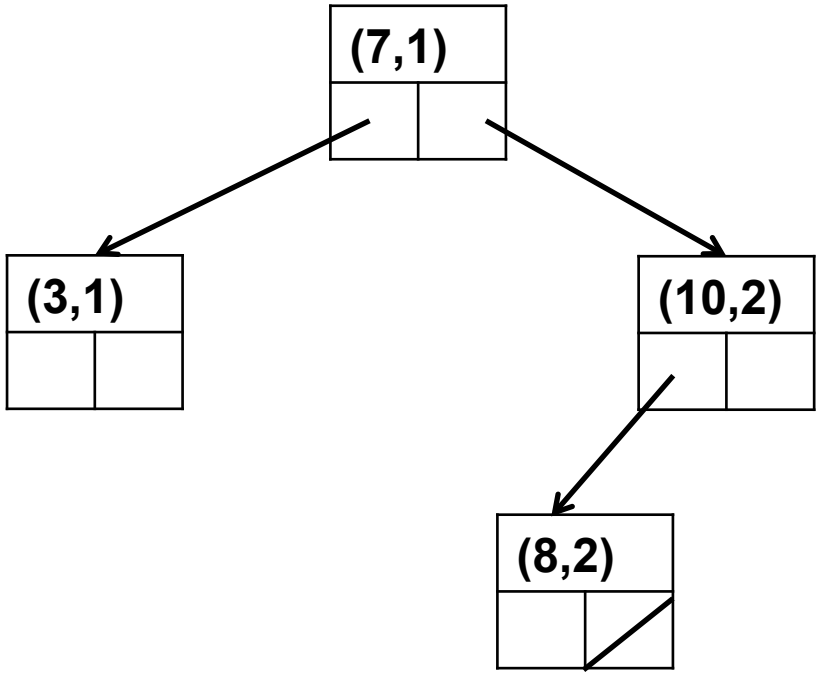
- **Multiset** : similaire à **set** avec possibilité d'avoir des valeurs multiples
  - Ordre des valeurs identiques déterminé par l'ordre d'insertion

```
#include <set>
multiset<int> MS;
MS.insert(7); MS.insert(1);
MS.insert(7); MS.insert(1);
MS.insert(4); MS.insert(3);
MS.insert(0); MS.insert(6);
MS.insert(5);
multiset<int>::iterator i = MS.find(3);
MS.erase(i);
//iteration a l'aide d'un iterator
for(i=MS.begin(); i != MS.end() ; i++)
   cout << *i << endl;
```



- **Map** : similaire à **set** avec des éléments <clé,valeur>
  - Ordre des éléments déterminé par l'ordre des clés (pas d'ajout d'élément en spécifiant la position)

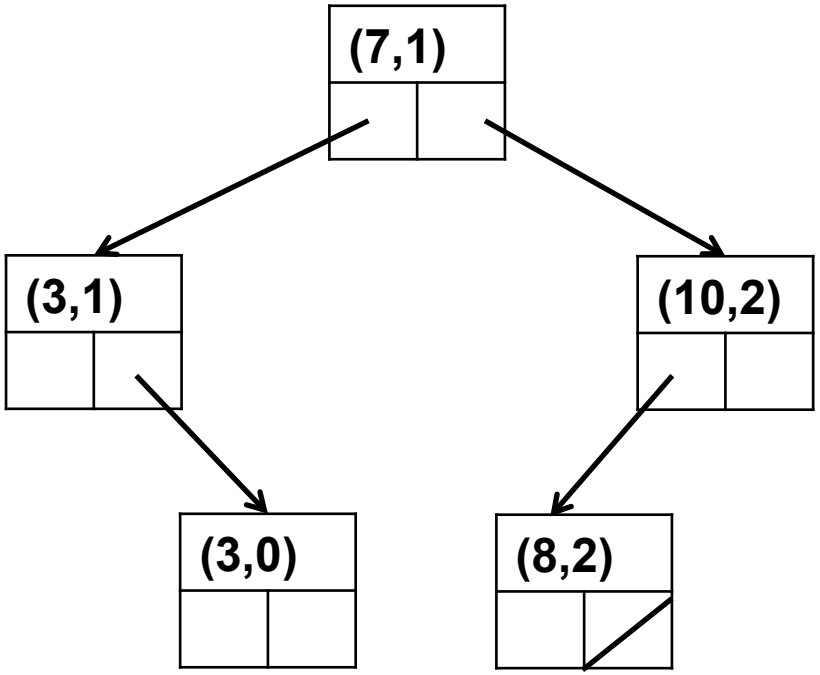
```
#include <map>
map<int,int> M;
M.insert(pair<int, int>(7, 1));
M.insert(pair<int, int>(3, 1));
M.insert(pair<int, int>(10, 2));
M.insert(pair<int, int>(8, 2));
//iteration a l'aide d'un iterator
map<int,int>::iterator i;
for(i=M.begin(); i != M.end() ; i++){
    cout << (*i).first << " " << (*i).second << endl;
  }
```



Conteneurs associatifs

- **Multimap** : similaire à **multiset** avec des éléments <clé,valeur> (possibilité d'avoir des clés multiples ; ordre des clés identiques déterminé par l'ordre d'insertion)

```
#include <map>
map<int,int> M;
M.insert(pair<int, int>(7, 1));
M.insert(pair<int, int>(3, 1));
M.insert(pair<int, int>(3, 0));
M.insert(pair<int, int>(10, 2));
M.insert(pair<int, int>(8, 2));
//iteration a l'aide d'un iterator
map<int,int>::iterator i;
for(i=M.begin(); i != M.end() ; i++){
    cout << (*i).first << " " << (*i).second << endl;
  }
```



Conteneurs associatifs sans ordre

Unordered_set, Unordered_multiset, Unordered_map, Unordered_multimap

- Ordre des éléments déterminé par adressage dispersé
- Aucun ordre garanti à l'itération, sauf pour les doublons de valeurs ou clés dans unordered_multiset et unordered_multimap qui se suivent
- Ajout/retrait d'éléments et recherche par valeur ou clé en O(1) amorti
- Représentation : table de hachage
- Même opérations que les conteneurs associatifs sauf les opérateurs relatives à l'ordre des éléments (exemple : élément précédent, élément suivant)

Conteneurs adaptés

- Stack (s): pile adaptée de list, vector ou deque (par défaut)
- Queue (q): file adaptée de list ou deque (par défaut)
- Priority_queue (p): file de priorité (monceau) adaptée de vector (par défaut) ou deque
- **Opérations spécificiques:**
  - **push :** type → Ø: ajouter un élément (au début pour « stack », à la fin pour « queue », ou suivant l'ordre des valeurs pour « priority_queue »)
  - **pop** : Ø → Ø : retire un élément au début
  - **top** : Ø → type : retourne l'élément se trouvant au début (pas dans « queue »)
  - **front** (resp. **back**) : Ø → type : retourne l'élément se trouvant au début (resp. à la fin) (seulement pour « queue »)