Containers

List

C’est une liste doublement chaînée, donc chaque élément a un pointeur vers l’élément suivant et un autre vers l’élément précédent.

Usage

On l’utilise principalement quand on doit stocker de nombreux éléments dont on ne connaît pas le nombre, car on peut insérer/supprimer efficacement les éléments, de par les pointeurs, évitant alors de devoir décaler toute la liste.

Cependant, il est à éviter si on veut accéder à des valeurs aléatoires simplement et rapidement, parce qu'on ne peut pas indiquer l’index. De plus, si on connaît à l’avance la taille de nos données, on préfèrera utiliser **vector**

Avantages :

- Optimisé pour l’insertion/suppression aléatoire
- Conserve l’ordre d’insertion
- Taille dynamique

Inconvénients

- Accès par index impossible
- Il utilise plus de mémoire

Vector

Fonctionnement

C’est un tableau dynamique. Il contient le nombre d'éléments, les éléments, ainsi que le type des éléments stockés (classe utilisant les templates <T>)

Usage

C’est souvent le conteneur utilisé par défaut, car très simple d’utilisation, cependant il n’est pas le plus optimisé, et pour des besoins bien précis, on préfèrera utiliser d’autres conteneurs pour de meilleures performances (par exemple lorsque l’on ne sait pas la taille, ou que l’on doit faire beaucoup de suppressions/insertions).

Avantages

- Accès par index
- Stocke les éléments de façon contigüe
- Accès en complexité presque constante
- Taille dynamique
- Parfait pour les insertions/suppressions en fin

Inconvénients

- Pour les insertions/suppressions au milieu, c'est moins performant parce qu'il doit décaler tout ce qui suit
- Quand on redimensionne un vector, les références vers les éléments ne sont plus forcément valides

DEQUE

Fonctionnement

Deque (Double Ended QUEue) est une file doublement chainée.

Usage

Il convient lorsqu'on n'est pas sûr du nombre d'éléments et permet un accès aléatoire rapide à n'importe quel élément. Contrairement à un vecteur, deque permet de prendre en charge l'insertion et la suppression efficace à l'avant.

Avantages

- L'insertion et la suppression des éléments en première et en dernière position se fait avec un coût constant
- Optimisée pour les parcours dans les deux sens
- Ne stockent pas les éléments de façon contigüe, il est donc plus efficace en termes de réallocation mémoire importante

Inconvénients

- Après modification, certaines références vers les éléments ne sont plus forcément valides
- L'insertion et la suppression aux autres positions se fait avec une complexité O(n)
- Il utilise plus de mémoire

EXEMPLE

Le deque est vraiment utile dans tout ce qui se rapproche à une gestion de file d'attente (colis, file d'attente dans les pharmacies ect)

Par exemple avec l'attente dans un lieu, chaque personne entrante et qui commence à faire la queue sera gérée avec push_back. Et lorsqu'une personne est accueillie et sort donc de la queue, on utilise pop_front pour gérer les personnes qui partent de la queue.

MAP

Fonctionnement

C'est le fonctionnement d'un dictionnaire : Une unique clé (n'importe quel type) est associée à chaque élément.

Usage

C'est un conteneur assez pratique lorsqu'on utilise autre chose que des int pour l'indexage. (Exemple : Vocabulaire/Traduction, avec l'indexage dans une langue, et la traduction en valeurs)

Avantages

- Insertion suppression aléatoire avec complexité constante (par l'utilisation d'une table de hachage)
- Clés triées automatiquement
- Modification du tableau très optimisé qu'importe la taille

Inconvénients

- Il utilise plus de mémoire par sa structuration
- Sans doublon de clé

EXEMPLE

Nous l'avons utilisé pour notre projet en synthèse d'image. Nous avions des types bien précis d'ennemis écris dans un fichier texte, et c'était plus pratique d'écrire 3 lettres que d'écrire encore et encore les mêmes caractéristiques.

```
void initializeEnemyMap(std::map<std::string, std::tuple<int, int, int, int, int, int>>& enemyMap)
{
    // int w, int h, int d, int points, int left, int up
    /* ********** V E R T I C A L ********** */
    enemyMap["V_L_L"] = std::make_tuple(7, H, 10, 90, 1, -1);
    enemyMap["V_L_R"] = std::make_tuple(7, H, 10, 90, 0, -1);
    enemyMap["V_M_L"] = std::make_tuple(10, H, 10, 70, 1, -1);
    enemyMap["V_M_R"] = std::make_tuple(10, H, 10, 70, 0, -1);
    enemyMap["V_B_L"] = std::make_tuple(L, H, 10, 40, 1, -1);
    enemyMap["V_B_R"] = std::make_tuple(L, H, 10, 40, 0, -1);

    /* ********** H O R I Z O N T A L ********** */
    enemyMap["H_L_U"] = std::make_tuple(L, 4, 10, 90, -1, 1);
    enemyMap["H_L_B"] = std::make_tuple(L, 4, 10, 90, -1, 0);
    enemyMap["H_M_U"] = std::make_tuple(L, 8, 10, 70, -1, 1);
    enemyMap["H_M_B"] = std::make_tuple(L, 8, 10, 70, -1, 0);
    enemyMap["H_B_U"] = std::make_tuple(L, H, 10, 40, -1, 1);
    enemyMap["H_B_B"] = std::make_tuple(L, H, 10, 40, -1, 0);

    /* ********** S Q U A R E ********** */
    enemyMap["L_L_B"] = std::make_tuple(7, 7, 10, 100, 1, 0);
    enemyMap["L_R_B"] = std::make_tuple(7, 7, 10, 100, 0, 0);
    enemyMap["L_L_U"] = std::make_tuple(7, 7, 10, 100, 1, 1);
    enemyMap["L_R_U"] = std::make_tuple(7, 7, 10, 100, 0, 1);

    enemyMap["M_L_B"] = std::make_tuple(10, 10, 10, 80, 1, 0);
    enemyMap["M_R_B"] = std::make_tuple(10, 10, 10, 80, 0, 0);
    enemyMap["M_L_U"] = std::make_tuple(10, 10, 10, 80, 1, 1);
    enemyMap["M_R_U"] = std::make_tuple(10, 10, 10, 80, 0, 1);

    enemyMap["B_L_B"] = std::make_tuple(15, 15, 10, 50, 1, 0);
    enemyMap["B_R_B"] = std::make_tuple(15, 15, 10, 50, 0, 0);
    enemyMap["B_L_U"] = std::make_tuple(15, 15, 10, 50, 1, 1);
    enemyMap["B_R_U"] = std::make_tuple(15, 15, 10, 50, 0, 1);
}
```