## Tableau à taille fixe

Un tableau de taille fixe est constitué d'un pointeur sur le premier élément et de sa taille. Les éléments sont accédés par l'indice physique. L'indice pour accéder à chaque élément est valide de 0 à N compris. On peut accéder à son emplacement mémoire via la formule : adresse <-- tableau + i \*taille(type).

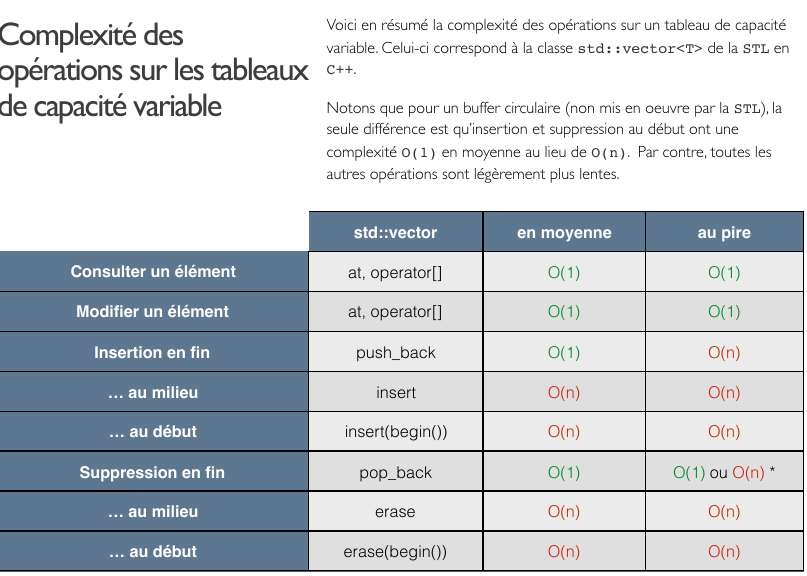
La notion de capacité fixe permet de pouvoir insérer et supprimer en fin en changeant la variable taille. Si ces actions ne sont pas possibles, il faut alerter l'utilisateur. Même chose s'il on veut insérer et supprimer en position quelconque.

Le buffer circulaire permet de se libérer des contraintes de commencer à l'indice 0. Ce dernier possède un indice physique pour l'emplacement mémoire et un indice logique pour les indices utilisateur.

## Tableau de taille variable et de capacité fixe

Cela permet d'allouer dynamiquement de la mémoire si la capacité n'est plus suffisante. S'il n'y a plus de capacité suffisante, il faut allouer un espace mémoire plus grand, déplacer les éléments contenus de l'ancien espace data puis détruire l'ancienne espace de data.

Complexité de l'insertion en fin : si assez de mémoire O (1) dans le pire des cas, il est de O(N). Le cas moyen se situe dans l'intervalle 2 et 3 on dit alors que la complexité et constante en temps amorti. Une bonne méthode pour gérer une capacité plus grande que la taille est d'attendre que taille sois 4 fois plus petite que capacité.



vector sort O(N\*log(N)). Si erase au début = pop\_back = O(1). Generate est linéaire.

# Listes

## Simplement chaînées

Les éléments ne sont pas forcément consécutifs. Mis en œuvre avec une structure générique. Pour le parcours il est préférable de faire de manière itérative. L'encapsulation permet de ne pas avoir de problème lors de la manipulation des éléments. Le mieux serait d'avoir des itérateurs ainsi que des opérateurs pour accéder et passer au suivant (approche de la STL).

## Doublement chaînées

L'utilisation d'un attribut vide pour stocker l'adresse du premier et l'adresse du dernier. Pour une liste vide, les deux pointeurs pointent sur lui-même. La méthode dernier retourne l'adresse de apad.

La méthode emplace permet d'insérer en place avec une complexité constante

La méthode splice insert à l'endroit voulu en une complexité constante à linéaire

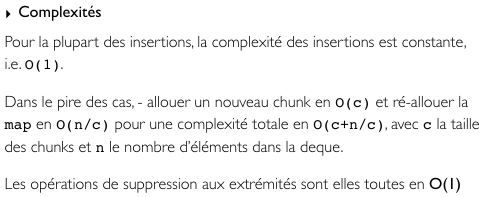
la méthode insert est de complexité linéaire

## Tri

On ne peut pas appliquer les algorithmes de tri vu précédemment car il n'y pas d'accès indexé. Les tris par sélection, par insertion et à bulles peuvent être adapté pour les listes. Le tri se fait en réordonnant les éléments. Il ne faut pas changer la valeur de l'élément. Splice after permet de déplacer des maillons pour le tri par insertion. La fonction splice nous permet de mettre en œuvre le tri fusion. Cette dernière a une complexité temporelle et spatiale constante.

# Deque

Les données sont stockées dans plusieurs petits tableaux de capacité fixe. Ils sont appelés chunk. Les adresse de ces chunk sont stocké dans un buffer circulaire, appelé map, qui est a capacité variable. Pour accéder aux éléments, il faut pratiquer un double déréférencement. La map et le chunk possède chacun un méthode pour le calcul de l'indic physique.



# Tas

Le tas est un tableau de taille variable que l'on représente comme un arbre binaire. Chaque élément a un parent sauf la racine, aucun élément n'a plus de deux enfants. La complexité pour trouver l'élément le plus grand est constante car c'est le premier. L'insertion et la suppression se font avec une complexité logarithmique.

push\_heap : L'insertion se fait en fin de tableau. Ensuite il faut faire remonter l'élément jusqu'à que la condition de tas soit respectée.

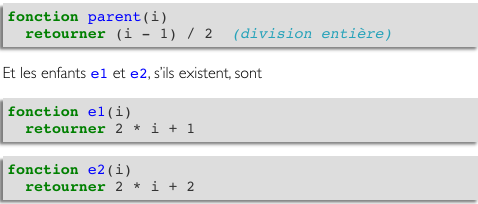
pop\_heap : La suppression s'effectue en général sur l'élément le plus grand. Pour ce faire on échange la racine et le plus petit élément et ensuite on supprime le dernier. Pour rétablir la condition de tas, on fait redescendre l'élément par le plus grand de ses enfants jusqu'a respecté la condition de tas.

make\_Heap sert à faire le tas, il fait descendre les parents, s'ils sont plus petits, vers le plus grand enfant. La complexité est de O(N\*log(N)) car chaque insertion est logarithmique.

Sort\_heap: il faut créer le tas avec make heap. Ensuite on prend à chaque fois la racine et on la met en dernière position et on fait remonter le plus grand des enfants. Ce tri est instable et s'effectue en place.

Les conditions de tas sont :

* Tout élément est plus grand (ou égal) que ses enfants
* Réciproquement, tout élément est plus petit (ou égal) que son parent



# Type de donnée abstrait

Ces types restreignent la manière d'insérer, d'accéder et de supprimer les éléments.

## Pile

Principe LIFO. Opérations de base push, pop et top. Les évaluations d'expressions arithmétiques se fait lorsqu'on rencontre des ')'. Pour stocker les opérandes et les opérateurs, on utilise deux piles et on évalue dès qu'une ')' est là. La notation polonaise inverse permet d'éviter d’avoir deux piles.

## Queue

Principe FIFO, opérations de bases push, pop, front et back

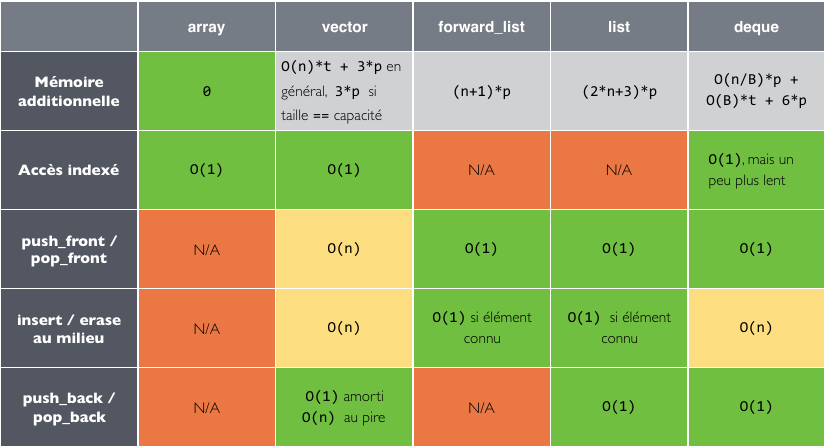
## priority\_queue

Même chose que queue mais avec notion de priorité. Si on rentre une paire d'élément dans la file, on ressort en regardant d'abord le premier mais s’ils sont égaux on regarde le deuxième.

emplace sort les éléments dans l'ordre inverse de pop.

# Structures linéaires en C++

* Sequence container : Array, Vector, Deque, List, Forward\_list.
* Container adaptors: Stack, Queue, priority\_queue.
* Les fonctions de tas sont aux nombres de 6. push*heap, pop*heap, make*heap, sort*heap, is*heap et is*heap*until.*
* Allocation dynamique



T\* ident = new T; // ident est un pointeur sur la variable de type T  
int\* i = new int(42); // ou {42}  
delete ident;  
  
T\* ident = new T[N]; // ident est un pointeur sur le tableau de type T  
// on doit initialiser chaque élément   
delete[] ident;  
// Il est autoriser un tableau de taille nulle

Il faut appeler delete dans le flot d'exécution pour éviter une fuite de mémoire. Il est de bonne pratique de mettre le pointeur supprimé à nullptr. L'opérateur new appelle le constructeur spécifié, malloc et calloc allouent juste de la mémoire sans appeler le constructeur. Même différence entre delete et free.

T\* Ptr = (T\*) ::operator new(N\*sizeof(T)); // ne construit rien mais alloue la mémoire  
::operator delete(Ptr);// ne détruit pas mais libère la mémoire

Les destructeurs ne lèvent pas d'exceptions. Les constructeurs lèvent soient des std::bad\_alloc ou des exceptions du constructeurs.

Trois sources de fuite de mémoire :

* Écraser la valeur du pointeur
* Perdre le pointeur
* Le programme ne s'exécute pas de manière attendue et fait perdre le pointeur Si une exception est levée elle doit appeler les destructeurs. Le constructeur de copie par défaut va copier tous les attributs d'une classe. Cela pose problème lors de leurs destructions car ils ont les mêmes pointeurs. Pour pouvoir trier une classe, il faut mettre en œuvre les opérateurs > et <. Un emplace*back appelle juste le constructeur avec élément, le push*back appelle le constructeur vide puis l'opérateur de copie, et le destructeur. Si on affecte vecteur avec un type complexe, il va construire un élément, affecter cet élément puis le détruire. La complexité de std::swap est de O(N). Si on fait une méthode qui une fonction qui swap simplement les éléments, on passe à O(1). Une rvalue reference est une rvalue dont l'existence est temporaire, une expression en cours d'évaluation, une valeur en retour de fonction. Le move est noexecpt contrairement à l'opérateur d'affectation.

# Arbres

Le but est d'insérer et de supprimer en O(log(N)) et qui permet de chercher en O(log(N)).

## Arbres quelconques

* Degré de nœud : nombre de fils
* Racine : seul nœud sans père
* Étiquette : information associée à un nœud
* Feuille : nœud sans fils
* Chemin : suite de nœuds reliant la racine à un nœud
* Niveau : nombre de nœuds sur le chemin du nœud
* Nœud interne : nœud avec fils
* Hauteur longueur du chemin le plus long
* Degré de l’arbre : degrés max de ses nœuds
* Sous-arbre : sous ensemble de nœuds ayant une structure d'arbre
* Arbre plein : degré(N) = degré(arbre), en tout nœud interne N
* Arbre complet : arbre plein, rempli depuis la gauche.
* Arbre dégénéré : degré de chaque nœud interne vaut 1 Le parcours en profondeur est un algorithme récursif qui permet de parcourir de haut en bas et de gauche à droite. Le parcours en largeur est un algorithme nécessitant une queue où on fait chaque niveau de gauche à droite.

## Arbres binaires

C'est un arbre de degré 2. Un parcours préordonné est un parcours en profondeur. Un parcours symétrique consiste à prendre l'enfant gauche, la racine et l'enfant droit. Le parcours post ordonné prend l'enfant gauche, droit et la racine. Puis parcours en largeur. Les expressions arithmétiques sont équivalentes à des arbres dont les feuilles sont des valeurs et les nœuds internes des opérations. On peut rechercher, insérer, supprimer, un élément dans un ABR. Le rang d'une clé est toujours de 1 pour une feuille.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Complexité min | Complexité moyenne | Complexité max |
| Recherche | O(h) | O(log(N)) | O(N) |
| Insertion | O(h) | O(log(N)) | O(N) |
| Suppression | O(h) | O(log(N)) | O(N) |
| Min et max | O(h) | O(log(N)) | O(N) |
| Rang | O(h) | O(log(N)) | O(N) |
| Sélection | O(h) | O(log(N)) | O(N) |
| Parcours (pré, post, sym) | - | O(N) | O(N) |
| Élément suivant | - | O(1) | O(h) |
| Tri |  | O(n\*log(n)) |  |

L'équilibrage permet d'éviter de grande complexité pour avoir une hauteur en O(log(N)). On considère qu'un arbre est équilibré si la différence entre la hauteur de l'enfant gauche et droit est plus grand ou égal à 1.

# Set

Les sets stockent des éléments dans un ordre spécifique et ne peuvent être modifiés dans la structure. Utilisé pour les ABR.

# Multiset

Si on insère des éléments non triés c'est une complexité linéarithmique sinon linéaire.

# Map

Structure formée par une combinaison de clé et de valeur. Les clés sont utilisées pour triés la map.

# Multimap

Complexité comme Multiset.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Map | Multimap | Set | Multiset | priority\_queue |
| Insertion | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) |
| Suppression | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) |  |
| Recherche | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) |  |

