Allocation dynamique

On retourne des pointeurs pour que l'objet existe à la fin de la fonction, dans le sens l'adresse d'un emplacement mémoire que l'on aura alloué dans la fonction.

Quand on détruit un objet pointé, on fait en quelque sorte de la place à cet emplacement mémoire, on peut donc faire pointer cet emplacement sur un objet nouvellement créé (réassigner le pointeur).

L'opérateur new pour allouer de la mémoire + créer un objet ne prend pas de paramètre, on écrit l'objet à la suite, tandis que si l'on sépare les choses, l'opérateur deux points new prend en paramètre la taille en bytes du type d'élément, ensuite on appelle new avec en paramètre le pointeur et l'objet à la suite.

Pour détruire, on utilise l'opérateur flèche sur le pointeur, et on appelle le destructeur de l'objet.

Quand on doit doubler la capacité, on commence par construire au nouvel endroit mémoire le nouvel élément, on va ensuite "move" (si constructeur par déplacement il y a) les éléments restants "de droite à gauche", donc "de la fin au début".

Si l'on veut insérer au milieu et que l'on a pas la place, on construit/move l'élément au nouvel emplacement mémoire et ensuite, l'on construit autour, vers la gauche et ensuite vers la droite.

Si l'on veut insérer au milieu et que l'on a la place, on "move" le dernier élément plus loin et ensuite on utiliser l'opérateur d'affectation par déplacement.

On détruit depuis la fin du vecteur.

Complexités containers (tableau complexités)

Attention swap utilise certainement move.

	Meilleur des cas	Moyenne	Pire des cas
Bubble sort	O(n)	$O(n^2)$	
Selection sort	$O(n^2)$		
Insertion sort	O(n)	$O(n^2)$	
Merge sort std::sort, std::stable_sort	O(nlog(n))		
Quick sort	O(nlog(n))		$O(n^2)$
Counting sort (comptage)	O(n+b)		
Radix sort (base)	O(d·(n+b))		
Euclide (pgcd), std::upper_bound	O(log(n))		
Quick select, std::nth_element, std::min_element	O(n)		

Opération	std::vector	en moyenne	au pire
Consulter un élément	at, operator[]	$O(1)$	$O(1)$
Modifier un élément	at, operator[]	$O(1)$	$O(1)$
Insertion en fin	push_back	$O(1)$	$O(n)$
… au milieu	insert	$O(n)$	$O(n)$
… au début	insert(begin())	$O(n)$	$O(n)$
Suppression en fin	pop_back	$O(1)$	$O(1)$ ou $O(n)$ *
… au milieu	erase	$O(n)$	$O(n)$
… au début	erase(begin())	$O(n)$	$O(n)$

Tas (parent toujours plus grand qu'enfant)

Insertion dans un tas : on ajoute la valeur en queue de tas et on la fait remonter pour que parent(val) > val soit respecté. push_heap met le dernier élément du vecteur dans le tas (à sa place, suppose que c'est un tas, donc ne corrige rien du tout une fois qu'il a placé l'élément au bon endroit).

Suppression du sommet d'un tas : on échange sommet et queue, on supprime queue et on corrige le tas. pop_heap "supprime" le sommet mais si l'on demande l'affichage, il apparaît tout de même à la fin, car il faudrait faire un pop_back pour le supprimer définitivement. pop_heap en O(log(N)).

Création d'un tas : on commence à parent(taille(T)-1), c'est-à-dire au parent du dernier élément. Ici, le 1, et on le descend à la position où il doit être (si besoin). Pareil avec le prochain parent (7), et le suivant (5), etc. make_heap en O(N).

Tri par tas : on crée le tas (si demandé, si juste sort_heap on fait direct le tri). On échange sommet <-> queue, on descend le nouveau sommet s'il est plus petit que ses enfants directs, on met de côté la queue == max, on s'arrête dès que l'on a un tas d'un élément. Pas oublier d'écrire chaque étape de création de tas également.

Coût en mémoire des structures de donnée

Array : ne coûte rien, uniquement le coût de chaque élément

Vector : 24B (1 ptr + 2 size_t (taille + capacité), ou 3 pointeurs), ensuite juste le coût de chaque élément.
Effacer au début en temps linéaire O(N). next en temps constant car random access iterator.

Forward list : 8B (1 ptr sur la tête) de base, et pour chaque élément encore 8 (ptr sur le prochain élément) + son coût

List : 24B (1 size_t (taille) + 1ptr sur la tête + 1 ptr sur la queue), pour chaque élément 16B (ptr sur le prochain élément et sur l'élément précédent) + son coût
Effacer au début en temps constant O(1). next en temps linéaire car il doit parcourir la List.

Attention au doublement de capacité sur un vecteur.

Itérateurs et splice

Attention aux changements de capacité pouvant invalider un itérateur par déplacement de zone mémoire.

Penser à utiliser next avec splice pour éviter une boucle infinie. Pareil avec forward_list pour vérifier si l'on est à la fin, car sinon il va vérifier après la fin s'il y a un élément.

Si l'on insère un élément dans une List, l'itérateur colle à l'élément où il se trouvait, tandis que dans un vector, l'itérateur reste en place.

Algorithme de Dijkstra

Notation polonaise inverse : tant qu'on rencontre des parenthèses ouvrantes ou des opérateurs, on écrit les chiffres (ordre gauche -> droite), dès qu'on rencontre une parenthèse fermante, on écrit le dernier opérateur rencontré et on continue ainsi, on déroule les opérateurs (ordre droite -> gauche).

Contenu des piles valeurs et opérateurs : on les écrit élément par élément dans la bonne colonne, dès qu'on rencontre une parenthèse fermante, on effectue l'opération en utilisant les 2 dernières valeurs et le dernier opérateur. On supprime l'opérateur et on remplace les 2 valeurs par le résultat de l'opération.

Arbres

Hauteur : en partant du bas, niveau selon l'enfant le plus bas du sous-arbre (max = hauteur de l'arbre).

Nœuds internes : parents.

Chemin : suite des nœuds reliant racine au nœud.

Niveau : longueur de son chemin, on compte en partant du haut.

Taille : nombre de nœuds d'un arbre.

Rang d'une clé k : nombre d'éléments strictement plus petits que la clé.

Arbre vide : aucun nœud, hauteur 0.

Arbre plein : hauteur h, degré d, nœuds de niveau (!) inférieur à h-1 sont de degré d, nœuds de niveau h-1 ont degré quelconque.

Arbre complet : arbre plein dont le dernier niveau est rempli par la gauche (un tas est un arbre binaire complet).

Arbre dégénéré : degré 1, topologiquement équivalent à une liste chaînée, pire cas pour des algorithmes dont la complexité dépend de la hauteur, parfois utile comme structure intermédiaire (équilibrage par linéarisation/arborisation).

Arbre binaire : degré <= 2, pour nœuds de degré 1 -> distinguer si l'enfant est à gauche ou à droite, éléments plus petits que racine à gauche, plus grands à droite. Pas oublier les branches vides lorsqu'un parent n'a qu'un seul enfant ! (pour distinguer si gauche ou droite).

Parcours en profondeur : pré-ordonné.

Suppression de la clé k (sans rotation ensuite)

Feuilles : rien à faire. **Degré 1 :** comme pour le minimum. **Degré 2 :** choisir un des nœuds descendant comme racine du sous-arbre à raccrocher (minimum du sous-arbre droit ou maximum du sous-arbre gauche).

Rotation

Si le nœud que l'on doit faire remonter penche du côté inverse du nœud au-dessus, il faudra faire une double-rotation (si un nœud penche à gauche et un en-dessous penche à droite, même +1 ou -1, il faut la faire).

Coût mémoire set et map

Fixe : 1 ptr sur le début, 1 ptr sur la fin, 1 size_t pour la taille
Par nœud : 2 ptrs vers les enfants (ainé (gauche) et puiné (droit)), 1 ptr vers le parent, (coût clé +) coût valeur

ordered_map : même chose avec une valeur de plus, comme un dictionnaire, une clé et une valeur dans chaque noeud d'arbre

Insertion en O(log(N)). Tri automatique en O(N*log(N)).

```
fonction linéariser (r, ref L, ref n)
  si r != ∅,
    linéariser(r.droit, L, n)
    r.droit ← L,  L ← r,  n ← n + 1
    linéariser(r.gauche, L, n)
    r.gauche ← ∅
```

```
fonction équilibrer (r)
  L ← ∅,  n ← 0
  linéariser(r, L, n)
  retourner arboriser(L, n)
```

```
fonction arboriser (ref L, n)
  si n != 0,
    rg ← arboriser(L, (n-1)/2)   (division entière)
    r ← L,  r.gauche ← rg,  L ← L.droit
    r.droit ← arboriser(L, n/2)  (division entière)
    retourner r
```

```
fonction rotation_gauche (ref r)
  t ← r.droit
  r.droit ← t.gauche
  t.gauche ← r
  r ← t
```

```
fonction rotation_droite (ref r)
  t ← r.gauche
  r.gauche ← t.droit
  t.droit ← r
  r ← t
```

```
fonction calculer_hauteur (r)
  si r != ∅,
    r.hauteur ← 1 +
 max(hauteur(r.gauche), hauteur(r.droit))
```

```
fonction arbre_depuis_postfixe (flux)
  e ← lire(flux inversé)
  r ← nouveau noeud d'étiquette e
  si e est un opérateur
    r.droit ← arbre_depuis_postfixe (flux)
    r.gauche ← arbre_depuis_postfixe (flux)
  retourner r
```

Ou bien : depuis notation postfixe, regarder l'ordre des opérations, noter les 2 premiers opérandes et leur parent, et dérouler de cette façon.