- **Une opération** telle qu'une incrémentation a une complexité en $O(1)$.

```
++k;
```

- **Une boucle** que l'on va enchaîner N-fois aura une complexité en $O(N)$.

```
for(int i = 1; i <= n; ++i) {
  ++k;
}
```

- **Une imbrication de M-boucles** allant jusqu'à N aura une complexité en $O(N^M)$.

```
for(int i = 0; i <= n; ++i) {
  for(int j = 0; j <= n; ++j) {
    ++k;
  }
}
```

- **Une imbrication de 2 boucles qui vont jusqu'à M et N** aura une complexité en $O(M \cdot N)$.

```
for(int i = 0; i < n; ++i) {
  for(int j = 0; j < m; ++j) {
    k++;
  }
}
```

- **Une imbrication de 2 boucles dont la deuxième va jusqu'au paramètre de la première** aura une complexité en $O(N^2)$, car on fait $\frac{(N-1)\cdot N}{2}$ opérations.

```
for(int i = 0; i < n; ++i) {
  for(int j = 0; j < i; ++j) {
    k++;
  }
}
```

- **Si l'opération de la boucle est une multiplication ou une division par M**, alors la complexité sera en $O(log_M(N))$. Mais la base importe peu, on peut donc l'enlever à chaque fois.

```
for(int i = 1; i <= n; i *= 2) {
  k++;
}
```

```
for(int i = n; i > 0; i /= 3) {
  k++;
}
```

•Somme de termes - ignorer les termes croissant moins vite que les autres

$$1 + n^2 + n^5 + n^8 = \mathcal{O}(n^8)$$

•Produits de facteurs - ignorer les facteurs constants

$$3.1415 \cdot n^2 = \mathcal{O}(n^2)$$

•Ordre de grandeur

$$\mathcal{O}(2^n) > \mathcal{O}(n^2) > \mathcal{O}(n \cdot log(n)) > \mathcal{O}(n) > \mathcal{O}(log(n))$$

ATTENTION : Si les boucles ne sont pas imbriquées, on prend la complexité la plus élevée.

NE PAS OUBLIER : Des codes à la suite les complexités s'additionnent, mais les codes qui s'imbriquent, elles se multiplient.

Fonctions de la STL		
Fonctions	**complexité**	**détails**
std::nth_element	$\mathcal{O}(n)$	
std::find_XX	$\mathcal{O}(n)$	
std::generate	$\mathcal{O}(n)$	n étant la taille du vecteur à remplir
std::distance	$\mathcal{O}(n)$	
std::max/min_element	$\mathcal{O}(n)$	
std::unique	$\mathcal{O}(N)$	
std::search	$\mathcal{O}(N)$	
std::search_n	$\mathcal{O}(N)$	
std::swap	$\mathcal{O}(1)$	
std::merge	$M + N$	M et N étant la taille des listes
std::sort	$\mathcal{O}(n \log n)$	
std::stable_sort	$\mathcal{O}(n \log n)$	si assez de mémoire
std::partial_sort	$\mathcal{O}(n \log m)$	où $n = last - first$ et $m = middle - first$
std::upper/lower_bound	$\mathcal{O}(\log_2 n + 1)$	
std::equal_range	$\mathcal{O}(\log_2 N)$	
std::binary_search	$\mathcal{O}(\log_2 N)$	

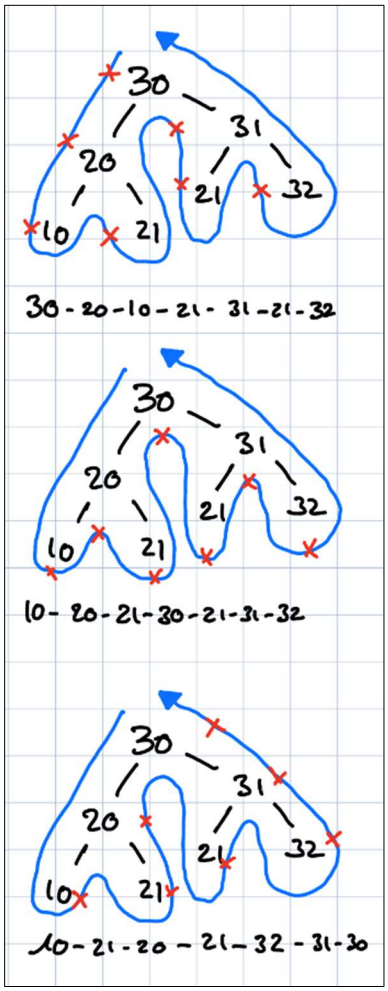
Si on appelle x fois la fonction et que le paramètre diminue de 1 à chaque fois on $\mathcal{O}(x^n)$: f(n-1) + f(n-1) + f(n-1) et que n = 4 la réponse sera de $(x)^n$ donc $(3)^4$

Un second exemple pourrait être d'appeler x fois la fonction et de diviser par 3 à chaque fois, dans ce cas $\mathcal{O}(x^{log_3(N)})$

Si on a des appels à (f-1) et (f-2), la complexité est équivalent à $\mathcal{O}(\varphi^n)$ ou φ = le nombre d'or.

Algorithmes	Complexités
Factorielle récursif	O(n)
Factorielle itératif	O(n)
Fibonacci récursif	$O(1.618^n)$
Fibonacci itératif	O(n)
PGCD (Euclide)	O(log(n))
Tours de Hanoï récursif	$O(2^n)$
Tours de Hanoï itératif	$O(2^n)$
Permutations	O(n!)
Tic Tac Toe	9!
Puissance 4, profondeur d'exploration de d tours	$O(7^d)$
Minimax (negamax), m mouvements possibles par tour, profondeur de d tours	$O(m^d)$

Arbre des appels récursifs : Affichage avant les 2 appels (1), affichage entre les deux appels (2) et affichage après les 2 appels (3).



Tri à bulles (Bubble sort) $\mathcal{O}(n^2)$ **:** parcourt le tableau du début vers la fin et amène à la fin l'élément le plus grand. Il fait cela en boucle jusqu'à avoir parcouru la taille du tableau.

Tri par sélection (Selection sort) $\mathcal{O}(n^2)$ **:** Rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0 puis rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1. Continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

64 25 12 22 **11**

11 **25** **12** 22 64

11 12 **25** **22** 64

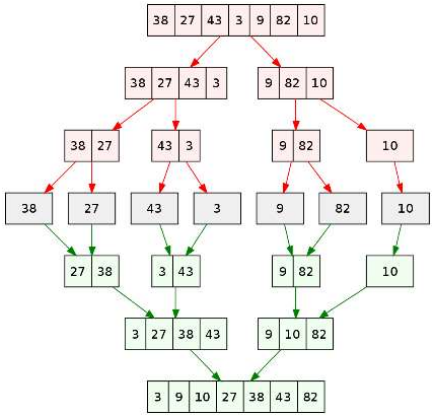
11 12 22 **25** 64

11 12 22 25 **64**

Tri par insertion (Insertion sort) $\mathcal{O}(n^2)$ **:** Pour trouver la place où insérer un élément parmi les précédents, il faut le comparer à ces derniers, et les décaler afin de libérer une place où effectuer l'insertion. Le décalage occupe la place laissée libre par l'élément considéré. En pratique, ces deux actions s'effectuent en une passe, qui consiste à faire « remonter » l'élément au fur et à mesure jusqu'à rencontrer un élément plus petit.

3	7	4	9	5	2	6	1
3*	7	4	9	5	2	6	1
3	7*	4	9	5	2	6	1
3	4*	7	9	5	2	6	1
3	4	7	9*	5	2	6	1
3	4	5*	7	9	2	6	1
2*	3	4	5	7	9	6	1
2	3	4	5	6*	7	9	1
1*	2	3	4	5	6	7	9

Tri fusion (Merge sort) $\mathcal{O}(\mathbf{n}\log\mathbf{n})$ **:** Algorithme récursif dans lequel on essaye de trier un tableau en le séparant en tableaux de 1 élément trié. Une fois les tableaux séparés, on les fusionne au fur et à mesure. Les petits tableaux sont dans l'ordre, il suffit de comparer à chaque fois le premier élément de chaque tableau et de le mettre dans le tableau de destination. La complexité est linéarithmique. Il n'est pas en place, mais il est stable. Pas adapter aux petits tableaux.



Tri rapide (Quick sort) $\mathcal{O}(n\log n)$ **:** On choisit un pivot et on le met en fin de tableau, puis on met à gauche tous les éléments plus petits et à droite les plus grand. Il est stable et en place. Pour se faire, on a deux itérateurs qui s'arrêtent lorsque un élément est à la mauvaise place et les inverse. La manipulation s'arrête lorsqu'ils se croisent. **Si les valeurs sont égales on inverse !!!**

E X E M P L E D E T R I	i=0, j=12
E X E M P L E D E T R I	++i = 1
E X E M P L E D E T R I	++i = 2
E X E M P L E D E T R I	--j = 11
E X E M P L E D E T R I	--j = 10
E X E M P L E D E T R I	--j = 9
E E E M P L E D X T R I	echange(2,9)
E E E M P L E D X T R I	++i = 3
E E E M P L E D X T R I	++i = 4
E E E M P L E D X T R I	--j = 8
E E E D P L E M X T R I	echange(4,8)
E E E D P L E M X T R I	++i = 5
E E E D P L E M X T R I	--j = 7
E E E D E L P M X T R I	echange(5,7)
E E E D E L P M X T R I	++i = 6
E E E D E L P M X T R I	--j = 6
E E E D E L P M X T R I	--j = 5
E E E D E I P M X T R L	echange(6,12)

Tips détecté les tris :

- **Tri à bulles :** Les plus grandes valeurs se retrouvent à la fin.
- **Tri par insertion :** Début organisé, mais la fin ne l'est pas encore (on peut voir des petites valeurs).
- **Tri par sélection :** Début organisé et il n'y pas de plus petites valeurs autre part qu'au début.
- **Tri par fusion :** On retrouve des morceaux triés dans certains partie des éléments. Séparer en deux à chaque fois.
- **Tri de Shell :** Le tri n'a aucun sens.
- **Tri rapide :** Il faut essayer de trouver quel était le pivot et selon quoi le tableau est plus ou moins mélanger.

---

std::qsort : En moyenne $\mathcal{O}(\mathbf{n}\log\mathbf{n})$ dans le pire des cas $\mathcal{O}(n^2)$. Pas stable.
std::sort : $\mathcal{O}(\mathrm{n}\log\mathrm{n})$, souvent une variation du **tri rapide**, pas stable.
std::stable_sort : garantit une complexité temporelle en $\mathcal{O}(\mathbf{n}\log\mathbf{n})$, si il y a assez de mémoire ou une complexité temporelle en $\mathcal{O}(\mathbf{n}\log^2\mathbf{n})$ si le tri fusion doit être réalisé en place. Il est stable, moins rapide que **sdt::sort**, les complexités sont garanties dans le pire des cas. Le **std::move(T)** doit être implémenté pour le type T car c'est une variante du **tri fusion**.
std::nth_element : $\mathcal{O}(n^2)$ Prend l'élément les itérateurs d'un tableau et au milieu le pilier pour partitionner et fait une partition.
std::partial_sort : $\mathcal{O}(n\log m)$ avec ***n = last - first*** et ***m = middle – first***

Tableau résumé de la complexité des comparaisons

Tri	Ascendant	Partiellement ordonné	Non-ordonné	Descendant
À bulles	O(n)	O(n^2)	O(n^2)	O(n^2)
Insertion	O(n)	O(n^2)	O(n^2)	O(n^2)
Sélection	O(n^2)	O(n^2)	O(n^2)	O(n^2)
std::sort	O(n * log(n))	O(n * log(n))	O(n * log(n))	O(n * log(n))
std::stable_sort	O(n * log(n))	O(n * log(n))	O(n * log(n))	O(n * log(n))

Tableau résumé des affectations

Tri	Ascendant	Partiellement ordonné	Non-ordonné	Descendant
À bulles	O(1)	O(n^2)	O(n^2)	O(n^2)
Insertion	O(n)	O(n^2)	O(n^2)	O(n^2)
Sélection	O(n)	O(n)	O(n)	O(n)
std::sort	O(n)	O(n * log(n))	O(n * log(n))	O(n)
std::stable_sort	O(n * log(n))	O(n * log(n))	O(n * log(n))	O(n * log(n))

Variables globales : construites et allouées avant le main() et détruites et libérées après le main().
Variables statiques : construites lors du premier passage et détruites et libérées après le main().
Variables automatiques : construites lors de leur exécution et détruite lors de la sortie du bloc.
Variables dynamiques : construites au passage de l'exécution sur « new » et détruites et libérées au passage sur « delete ».

`T* ident = new T;`	`T* ident = new T[N];`
`delete ident;`	`delete[] ident;`

On détruit dans le sens inverse de la construction. Le premier élément créé est le dernier élément détruit.

Le mot clé « new » alloue et construit, mais il est possible de séparer les deux.

```
T* p = new T{};
```

cette ligne devient alors les deux lignes ci-dessous

```
void* p = ::operator new(sizeof(T));     // allocation mémoir
T* q = new(p) T{};                       // construction objet de type
```

La même manipulation est possible lors de la destruction et libération avec « delete ».

```
delete p;
```

cette ligne devient alors les deux lignes ci-dessous.

```
p->~T();                    // destruction de l'objet de type
::operator delete(p);       // libération de la mémoire
```

Fuites mémoires : Perdre l'adresse de l'objet à détruire, perdre le pointeur en sortant du scope ou une exception est levée. Pour les éviter, il faut encapsuler les allocations dynamiques et détruire dans les destructeurs.

Il faut implémenter **std::swap** si on souhaite faire des tris (tri à bulles en a besoin, par contre insertion et fusion non). C'est un simple échange des adresses mémoire. Il faut aussi implémenter tout le reste des opérateurs. Comme l'opérateur d'affichage et de comparaison pour la réalisation d'un tri à bulles.

Il faut implémenter **std::move** si on souhaite réaliser un tri par insertion générique. Le but étant de déplacer les ressources de a à b, mais en laissant a dans un état valide. **std::move** cast les paramètres en rvalue référence (T&&). Il va donc voler les ressources de la source et les rendre valide.

std::push_back appelle le constructeur std::move et std::emplace_back non.
Si on std::push_back et qu'il n'y a pas assez de place, la taille double et on recopie tous les éléments et on n'oublie pas de détruire derrière.

```
v.push_back(C(rand())); // construction du paramètre
                        // construction par copie
                        // destruction du paramètre
v.emplace_back(rand()); // construction en place
```

Le mieux est d'utiliser std::swap pour tous les types simples de notre classe.

```
C c; // variable globale

void f() {
    cout << "f() : début\n";
    cout << "f() : fin\n »;
}

int main() {
    cout << "main() : début\n";
    cout << "f() : avant\n";
    f();
    cout << "f() : après\n";
    cout << "main() : fin\n";
}
```

```
--> Constructeur
main() : début
 f() : avant
  f() : début
  f() : fin
 f() : après
main() : fin
--> Destructeur
```

```
void f() {
    cout << "  f() : début\n";
    C c;
    cout << "  f() : fin\n";
}

int main() {
    cout << "main() : début\n";
    cout << " f() : avant\n";
    f();
    cout << " f() : après\n";
    cout << "main() : fin\n";
}
```

```
main() : début
 f() : avant
  f() : début
--> Constructeur
  f() : fin
--> Destructeur
 f() : après
main() : fin
```

```
void f() {
    cout << "  f() : début\n";
    static C c;
    cout << "  f() : fin\n";
}

int main() {
    cout << "main() : début\n";
    cout << " f() : avant\n";
    f();
    f();
    cout << " f() : après\n";
    cout << "main() : fin\n";
}
```

```
main() : début
 f() : avant
  f() : début
--> Constructeur
  f() : fin
  f() : début
  f() : fin
 f() : après
main() : fin
--> Destructeur
```

Rappels :

- **Reserve :** Non-destructif, par contre ça n'agrandit pas le vecteur après coup.
- **Operator new/delete :** N'appelle pas les constructeurs/destructeurs
- **new T(params.) :** Appel au constructeur avec params.
- **Static :** Alloc. Comme les autres, par contre destruction à la fin du programme.
- **vector<T> v(N)** : Construit N objets de type T
- **push_back** : Il réalloue de la mémoire s'il n'y a pas assez de place en doublant la capacité. Il appelle le constructeur de copie pour mettre les objets dans le nouveau vecteur. Il finit par détruire les anciens éléments. Si l'objet existe déjà et qu'on l'ajoute, il passe par le constructeur de copie.
- **push_back**(T(…)) : Il créer un objet avec le constructeur normal, puis appelle le constructeur de déplacement et détruit le temp.
- **emplace_back**(arguments de la classe) : Construit directement dans le vecteur. Pas de move, appelle constructeur basique. Juste un appel au constructeur normal (en place).
- **resize(n, val) :**
  - n < taille : On détruit les éléments
  - n > taille : On crée un objet val. avec le constructeur et on fait des copies avec le constructeur de copie. Si pas de valeur précisée, alors on crée un objet avec le constructeur par défaut et on copie.
  - n > capacité : On realloue comme push_back
- **swap** : Fait une construction avec le constructeur de déplacement et fait deux affectations avec l'opérateur d'affectation par déplacement. Puis détruit l'objet temporaire.
- **v[i] = T(…)** : Construit l'objet normalement et fait une affectation par déplacement, puis détruit.
- **V[i] = T** : Appel l'affectation par copie.
- Si on initialise un objet avec un **std::move**, ça appelle le constructeur de déplacement. Et si on fait une affectation avec, ça appelle l'affectation par déplacement. Un move permet d'éviter une destruction.

Si on construit un objet à l'aide d'un objet temporaire, le constructeur ne construit qu'un seul objet.
A la fin d'un bloc on détruit tous les objets existants. Dans le cas d'un vecteur, on détruit tous les objets dans ce vecteur un à un.

Constructions, destructions et assignations
N = ancienne taille avant modification
M = nouvelle taille

Fonction	Sans redimension	Avec redimension
`Vector<T> v(M)`	M * « Cd »	
Constructeur sans param (`T t1;` et `new T`)	« Cd »	
Destruction (`delete t;` et `stack`)	« D »	
Fin de bloc	*Détruire tout ce qui est dans la stack*	
`T t2(<param>)`	« Cp »	
`Fn(T(<params>));` (obj temp.)	« Cp » + Action de la fonction + « D ». *Privilégie le Move si existant car temporaire*	
`Push_back(t1)`	« Cc »	(N + 1)* « Cc » + N * « D »
`Push_back(T(<params>)`	« CpCmD »	« CpCm » + N * « Cc » + (N + 1) * « D »
`Emplace_back(<params>)`	« Cp	« Cp » + N * « Cc » + N * « D »
`Resize(M)`	*Si* M < N : (N – M) * « D ». *Sinon si* M > N : (M – N) * « Cd » + N * « D ». *Sinon si* (M == N) : *rien*	
`Resize(M, t)`	*Si* M < N : (N – M) * « D ». *Sinon si* M > N : M * « Cc» + N * « D ». *Sinon si* (M == N) : *rien*	
`Swap(t1, t2)`	« CmAmAmD »	
`T1 = T(<params>)`	« CpAmD » (obj temp.), *sinon* « Ac »	
`Reserve(M)`	*Si* M > N : N * « Cc » + N * « D », *Sinon si* (M <= N) : rien	
`Shrink_to_fit() ;`	*rien*	

Déclaration statique	Alloc. comme les autres, par contre destruction à la fin du programme. Avec params ou défaut.	Cp ou Cd
Reserve(M)	Non-destructif. Appel du constructeur de déplacement (Cm) et destructeur (D), N fois. Par exemple s'il y avait 3 objets dans le vecteur, on aurait CmCmCmDDD. **!! Alloue de la taille au vecteur pas de doulage !!**	M * (Cm D)
new T(params./void)	Appel au constructeur avec params	Cp ou Cd
Operator new/delete	N'appelle pas les constructeurs/destructeurs	-
vector<T> v(N)	Construit M objets de type T. Donc utilise le constructeur par défaut.	M * Cd
push_back	Il réalloue de la mémoire s'il n'y a pas assez de place en doublant la capacité. Il appelle le constructeur de copie pour mettre les objets dans le nouveau vecteur. Il finit par détruire les anciens éléments. Si l'objet existe déjà et qu'on l'ajoute, il passe par le constructeur de copie.	
push_back(T(…))	Il créer un objet avec le constructeur normal, puis appelle le constructeur de déplacement et détruit le temp.	CpCmD
emplace_back(T(…))	Construit directement dans le vecteur. Pas de move, appelle constructeur basique. Juste un appel au constructeur normal (en place).	Cp/Cd
resize(M, val)	M < taille : On détruit les éléments M > taille : On crée un objet val. avec le constructeur et on fait des copies avec le constructeur de copie. Si pas de valeur précisée, alors on crée un objet avec le constructeur par défaut et on copie. M > capacité : On réalloue comme push_back	
swap	Fait une construction avec le constructeur de déplacement et fait deux affectations avec l'opérateur d'affectation par déplacement. Puis détruit l'objet temporaire.	CmAmAmD
v[i] = T(…)	Construit l'objet normalement et fait une affectation par déplacement, puis détruit.	CpAmD
V[i] = T	Affecter un objet qui existe déjà. Appel l'affectation par copie.	Ac
std::move	Si on initialise un objet avec un std::move, ça appelle le constructeur de déplacement. Et si on fait une affectation avec, ça appelle l'affectation par déplacement. Un move permet d'éviter une destruction	Cm ou Am
Erase(begin())	Déplace N-1 objet vers la gauche avec affectation par déplacement et détruit 1 fois	(N-1)*Am D
T* toto	Ne fait rien, car ça déclare un pointeur et non un objet.	
Destruction	En sortie des accolades ou en fin de programme. Sauf allocation dynamique et static qui doit être free.	D

- **Si on construit un objet à l'aide d'un objet temporaire, le constructeur ne construit qu'un seul objet.**
- **A la fin d'un bloc on détruit tous les objets existants. Dans le cas d'un vecteur, on détruit tous les objets dans ce vecteur un à un.**

Complexités

	array	vector	forward_list	list	deque	set	map
Mémoire structure vide	Élément * nb d'éléments	3 pointeurs : début, taille, capacité = 24B	1 pointeur : 1er maillon = 8B	2 pointeurs : 1er et dernier maillon + 1 size_t (taille) = 24B	6 pointeurs = 48B	2 pointeurs (begin() + end()) + 1 size_t (taille) = 24B	24B
Mémoire par élément	Élément	élément	1 pointeur + élément	2 pointeurs + élément		3 pointeurs + élément	3 pointeurs + clé + valeur
operator[]	O(1)	O(1)	-	-	O(1)	-	O(log(n)) (par clé)
push_front	O(n)*	O(n)	O(1)	O(1)	O(1)	O(log(n))*	O(log(n))*
pop_front	O(n)*	O(n)	O(1)	O(1)	O(1)	O(log(n))*	O(log(n))*
insert indice i	O(n-i)*	O(n-i)	O(1)*	O(1)*	O(min(i, n-i))	O(log(n))	O(log(n))
erase indice i	O(n-i)*	O(n-i)	O(1)*	O(1)*	O(min(i, n-i))	O(log(n))	O(log(n))
push_back	O(1)*	O(1) / O(n)	O(n)*	O(1)	O(1)	O(log(n))*	O(log(n))*
pop_back	O(1)*	O(1) / O(n)	O(n)*	O(1)	O(1)	O(log(n))*	O(log(n))*
size	O(1)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)
merge				O(1)			
next	O(1)	O(1)	O(1) - O(n)	O(1) - O(n)		O(n)	
distance	O(1)	O(1)	O(n)	O(n)			

(*) = fonction n'existe pas nativement sur la structure

Ex de notation : O(1) - O(n) / O(n^2) : meilleure - moyenne / au pire

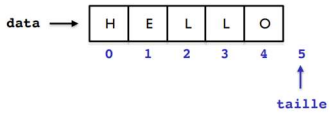
priority_queue : comme un heap

	heap
make_heap	O(n)
push_heap	O(1) - O(log(n))
pop_heap	O(log(n))
sort_heap	O(nlog(n))

Autres fonctions :

- generate() : O(n)
- is_sorted : O(n)
- resize() : O(n)
- unique() : O(n)
- splice() et splice_after (sur listes) : O(1)

Tableaux de taille fixe (std::array<T, N>) : Pas d'insertion / suppression d'éléments, Accès aux éléments en O(1) en calculant leur adresse.




```
classe TableauTailleFixe
  Entier constant taille
  Element[] data[taille]
```

Buffer circulaire : Tableau de taille variable indexé de 0 à taille-1 efficace pour insérer / supprimer en fin pas efficace pour insérer / supprimer ailleurs qu'en fin. File FIFO, Le buffer circulaire remplace l'index 0 par un index début.

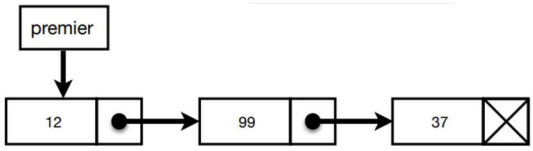
Avantages : $\mathcal{O}(1)$ pour insérer/supprimer en début ou en fin , accéder en toute position

Inconvénients : Insérer ou supprimer à l'indice i en $\mathcal{O}(min(i, taille - i))$, capacité fixe.

Tableaux de capacité variable (std::vector<T>) :

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)

Listes simplement chainées (std::forward_list<T>) : Tableaux inefficaces pour insérer / supprimer en positions autres que fin (et début). Ne pas stocker les éléments dans des emplacements mémoire consécutifs, on perd la relation suivant/précédent implicite. Chaque élément est stocké dans une structure appelée maillon qui inclut un pointeur vers le maillon suivant (et précédent) L'ensemble de ces maillons constitue une liste chainée.




```
structure Maillon
   Element valeur
   Maillon* suivant
```

Opération	en moyenne	au pire
Consultation	-	-
Insertion en fin	O(n)	O(n)
... au milieu	O(1)	O(1)
... au début	O(1)	O(1)
Suppression en fin	O(n)	O(n)
... au milieu	O(1)	O(1)
... au début	O(1)	O(1)
Taille	O(n)	O(n)
Suivant(s)	O(1)	O(n)
Distance	O(n)	O(n)

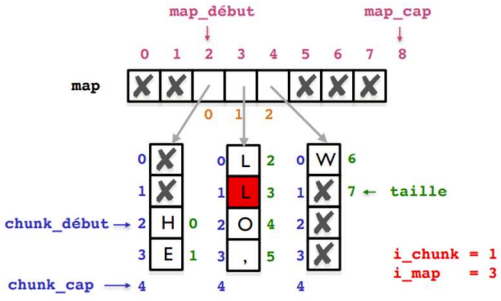
Listes doublements chainées (std::list<T>) : Comme la liste simple, sauf qu'il y a aussi un pointeur vers le maillon précédent.


```
structure Maillon
   Element valeur
   Maillon* suivant
   Maillon* précédent
```

```
structure MaillonVide
  Maillon* suivant
  Maillon* précédent
```

Opération	en moyenne	au pire
Consultation	-	-
Insertion en fin	O(1)	O(1)
... au milieu	O(1)	O(1)
... au début	O(1)	O(1)
Suppression en fin	O(1)	O(1)
... au milieu	O(1)	O(1)
... au début	O(1)	O(1)
Taille	O(1)	O(1)
Suivant(s)	O(1)	O(n)
Distance	O(n)	O(n)

Double ended queue (std::deque<T>) : Opérations en début et en fin en temps amorti constant, accès indexé en temps constant



Indices logiques en vert
Indices physiques en bleu

Opération	en moyenne	au pire
Consultation	O(1)	O(1)
Insertion en fin	O(1)	O(c+n/c)
... au milieu	O(min(i, n-i))	O(c+n/c)
... au début	O(1)	O(c+n/c)
Suppression en fin	O(1)	O(1)
... au milieu	O(min(i, n-i))	O(min(i, n-i))
... au début	O(1)	O(1)
Taille	O(1)	O(1)
Suivant(s)	-	-
Distance	-	-

	array	vector	forward_list	list	deque
Mémoire additionnelle	0	3*p + O(n)*t si taille != capacité	(n+1)*p	(2*n+3)*p	O(n/B)*p + O(B)*t + 6*p
operator[]	O(1)	O(1)	N/A	N/A	O(1), mais un peu plus lent
push_front / pop_front	N/A	O(n)	O(1)	O(1)	O(1)
insert / erase au milieu	N/A	O(n)	O(1) si élément connu	O(1) si élément connu	O(n)
push_back / pop_back	N/A	O(1) amorti O(n) au pire	N/A	O(1)	O(1)

t = sizeof(T); p = sizeof(T*); B = __deque_block_size<T,size_t>::value;

std::list<T>

Expression	Description
push_back(t) pop_back()	insère ou supprime un élément en fin
push_front(t) pop_front()	insère ou supprime un élément au début
insert(pos,...)	insère des éléments devant l'itérateur pos
erase(pos) erase(first,last)	supprime des éléments et retourne un itérateur suivant le dernier supprimé
emplace(pos,...) emplace_front() emplace_back()	comme push_back, push_front et insert mais l'objet est construit en place
clear()	supprime tous les éléments
splice(...)	opération d'épissure
sort()	tri stable en place en Θ(n.log(n))
merge(...)	opération de fusion du tri fusion
remove(...) remove_if(...) unique()	suppression d'éléments selon divers critères
reverse()	inversion de l'ordre des éléments de la liste

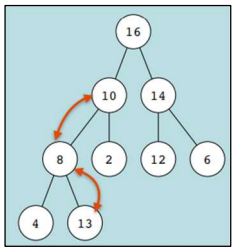
std::forward_list<T>

Expression	Description
insert_after(it,t) emplace_after(it,...)	insertion après un élément
erase_after(it)	suppression de l'élément suivant
splice_after(it,...)	épissure derrière un élément
before_begin()	itérateur avant le premier élément

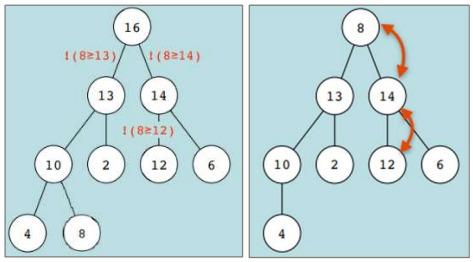
std::deque<T>

Méthode	Description
push_front(t) emplace_front(...)	insère un élément au début
pop_front()	supprime le premier élément

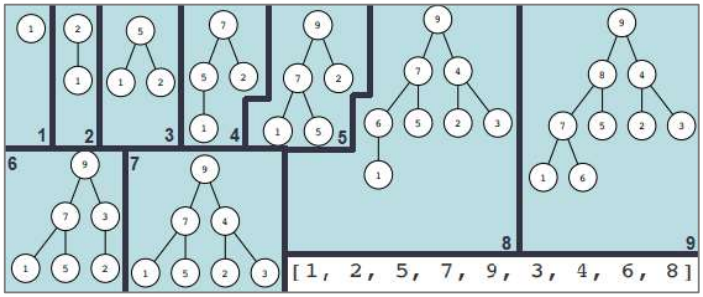
Un tas (heap) : est un arbre binaire complet dont tous les éléments sont plus petits ou égaux que leurs parents, c'est la condition de tas (CDT).
push_back : Ajoute la valeur en fin de vecteur, mais pas dans le tas.
pop_back : Supprime le dernier élément du vecteur.
Insertion (push_heap) : On insère toujours un élément à la fin du tas et on le remonte après. L'insertion est en $\mathcal{O}(1)$ et la remontée est en $\mathcal{O}(\log n)$.



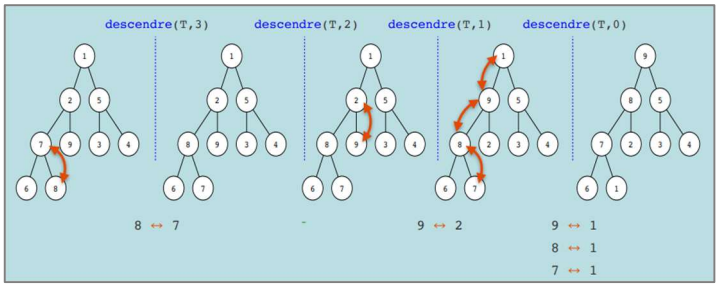
Supprimer le sommet du tas (pop_heap) : Il faut swap le sommet avec le dernier élément et réorganiser le tas. Pour réorganiser, il faut faire descendre le sommet (celui que l'on vient d'échanger) en direction du plus grand des enfants. **Attention :** l'élément supprimer reste dans le tableau qui stocke le tas.



Créer un tas (make_heap) : On insère les éléments en fin et ensuite on les fait remonter à leur place. Ici, on insère les éléments dans l'ordre donc du deuxième au dernier. C'est un algorithme en $\mathcal{O}(\mathrm{n}\log n)$. Remonter éléments.



On fait les nœuds du dernier au premier, mais on prend en compte que les nœuds qui ont des enfants et si la CDT n'est pas respectée pour le nœud, alors on le fait descendre le nœud jusqu'à sa place en swappant avec le plus grand enfant. Complexité en O(N). Descendre éléments.



Tri par tas (sort_heap) : Il faut déjà avoir un tas, puis on swap le premier et dernier élément et on ré-organise la condition de tas en descendant le premier élément. Sélection maximum en $\mathcal{O}(\log n)$ et tri en $\mathcal{O}(\mathrm{n}\log n)$.

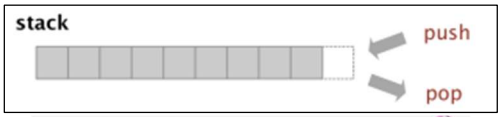
!! POUR LE TRI, IL FAUT QUE LA CDT SOIT RESPECTER SINON MARCHE PAS !!
Dans la STL, il existe des fonctions pour faire ces algorithmes (make_heap, sort, pop, push, is). Elles ne vérifient pas que c'est bien un tas, c'est à nous de vérifier (pop et sort → tas entre [first ; last(et push → tas entre [first ; last-1(.

Make_heap	3*std::distance(first, last)
Sort_heap	$\mathcal{O}(\mathrm{n}\log n)$
Push _heap	$\mathcal{O}(\log n)$
Pop_heap	$\mathcal{O}(\log n)$
Is_heap(_until)	$\mathcal{O}(\mathrm{n})$

!! Si le tas respecte déjà la CDT, insérer un élément se fait en O(1) !!

!! Ces structures ont la même complexité qu'un tas !!

Pile (std::stack<T, Container>) : structure où l'on insère et supprime du même côté (au sommet). LIFO. Elle permet notamment d'évaluer des expressions. NPI c'est lorsque l'on met l'opérateur à la fin de l'expression (permet d'éliminer les parenthèses).



pileVal	pileOp	Expression restante à évaluer
		(1 + ((2 + 3) * (4 * 5)))
1		+ ((2 + 3) * (4 * 5)))
1	+	((2 + 3) * (4 * 5)))
1 2	+	+ 3) * (4 * 5)))
1 2	+ +	3) * (4 * 5)))
1 2 3	+ +	) * (4 * 5)))
1 5	+	* (4 * 5)))
1 5	+ *	(4 * 5)))
1 5 4	+ *	* 5)))
1 5 4	+ * *	5)))
1 5 4 5	+ * *	)))
1 5 20	+ *	))
1 100	+	)
101		

Notations polonaise inverse : Pour évaluer, l'ordre opérateur / opérande n'importe pas.

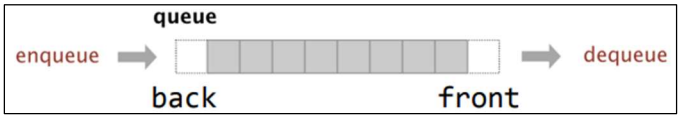
(1 + ((2 + 3) * (4 * 5)))

(1 ((2 3 +) (4 5 *) *) +)

En plaçant opérateur après ses opérandes, plus besoin de parenthèses

1 2 3 + 4 5 * * +

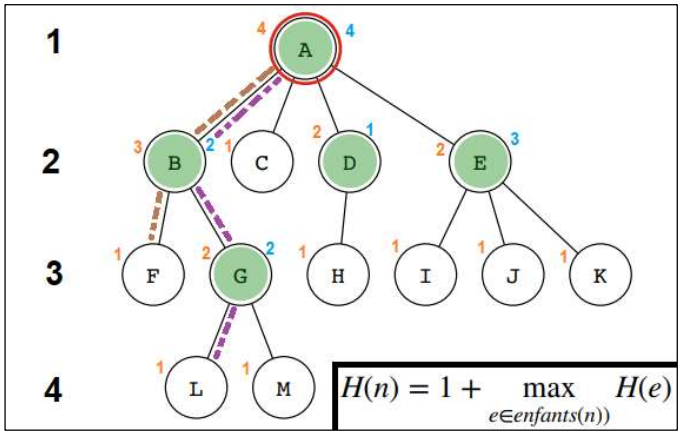
File (std::queue <T, Container>) : On insère d'un côté et on supprime de l'autre, FIFO. On peut donc accéder aussi accéder au dernier élément contrairement à LIFO.



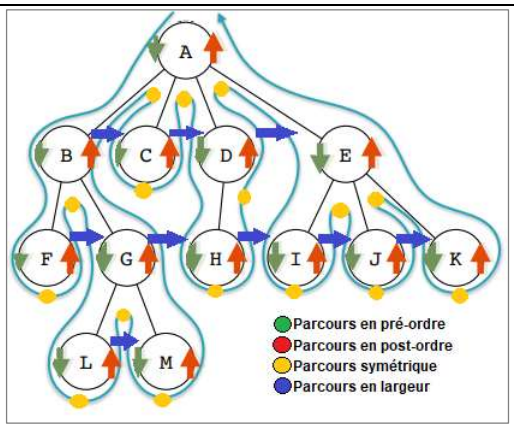
File de priorité (std::priority_queue<T, Container, Compare>) : File organisée selon un critère de priorité, mise en œuvre avec un tas. On peut toujours accéder à l'élément le plus prioritaire. Le sommet de la file est l'élément le plus prioritaire. Le sommet est donc l'élément ayant le plus grand indice. Si il y a une égalité, on compare les valeurs.

Top (front)	Accède au sommet (+ grand/prior.)
Back	Accède au dernier élément (file)
Push	Insérer un élément
Pop	Supprime le sommet

Racine : N'a pas de parent.
Nœuds internes : Nœuds qui ont un ou plusieurs enfants.
Feuilles : N'ont pas d'enfants.
Degré nœud : Nombre d'enfants. / Degré arbre : Degré max de ses nœuds.
Chemin d'un nœud : Suite des nœuds reliant la racine au nœud.
Niveau d'un nœud : Longueur de son chemin (Ex : L → 4).
Hauteur d'un arbre : Niveau maximum parmi ses nœuds.



Vide : Sans aucun nœud et a une hauteur de 0.
Plein : Tous les nœuds de niveau inférieur à h-1 sont de degré d. Les nœuds de niveau h-1 ont un degré quelconque. Les nœuds de niveau h sont des feuilles.
Complet : Arbre plein dont le dernier niveau est rempli par la gauche.
Dégénéré : Arbre de degré 1. Type liste chaînée.
Binaire : Arbre de degré <= 2. On distingue enfant gauche et droite.



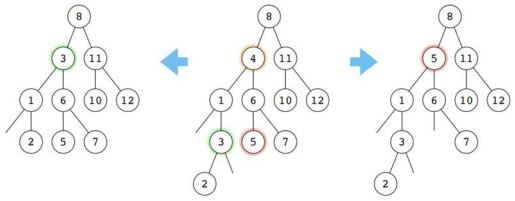
Attention : Il ne faut pas oublier qu'un nœud a un pointeur sur ses enfants, son parent et son puiné (enfant droite du même parent). Parcours symétrique que pour binaire.
Arbre à partir de deux parcours : On utilise le parcours post ou pré pour connaître la racine de l'arbre et les racines des sous-arbres. Et on utilise le parcours symétrique pour savoir quel sommet est à gauche ou à droite du quel autre sommet. On va donc regarder le début du pré ordre pour trouver la racine. Ensuit le symétrique pour trouver les sous arbres gauche et droite. Après on regarde le pré ordre de gauche à droite pour savoir quel est le sommet qui vient en premier dans le sous-arbre gauche, etc…
Expressions arithmétiques : Arbre binaire avec 2 types de nœuds, feuille = valeurs et nœuds interne = opérateurs binaires tels que +, -, *, /. Pas de nœud de degré 1.

- Infixe : (((1+2)/(3-4))*(1-2))
- Préfixe : */+12-34-12
- Postfixe : 12+34-/12-*

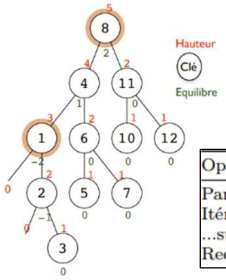
Arbre binaire de recherche (ABR) : Arbre binaire où l'enfant gauche est toujours plus petit que son parent et l'enfant droite est toujours plus grand que son parent. Pour chercher/insérer/supprimer une clé, on retrouve une complexité en $\mathcal{O}(\log n)$.
ABR chercher/insérer : il faut comparer la clé avec la racine et descendre en fonction de si la clé est plus petite (à gauche) ou plus grande (à droite).
ABR supprimer : Si on supprime des :

- **Feuilles :** il ne se passe rien,
- **Nœuds de degré 1 :** il suffit de raccrocher le sous-arbre.
- **Nœuds de degré 2 ou plus :** utiliser la technique de Hibbard.

Technique de Hibbard : on prend le minimum du sous-arbre droit ou le maximum du sous-arbre gauche.



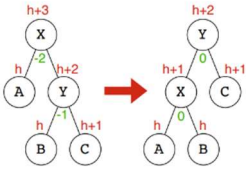
Equilibre : Un arbre est équilibré si $-1 \leq \text{équilibre(n)} \leq 1$



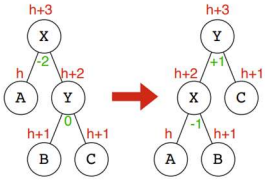
Opération	en moyenne	au pire
Parcours	O(n)	O(n)
Itération sur tout l'arbre	O(n)	O(n)
...suivant	O(1)	O(h)
Recherche, insertion, suppression, min, max	O(p)	O(p)

Arbre AVL (AVL tree) : Un arbre AVL est un arbre binaire de recherche qui garantit qu'aucun nœud n'a un déséquilibre autre que -1, 0 ou 1 en se rééquilibrant à chaque opération d'insertion / suppression.
AVL insérer : même que pour ABR avec un plus un rééquilibrage après insertion.
AVL supprimer : même que pour ABR avec aussi un rééquilibrage après suppression.
Équilibrage AVL : l'équilibrage AVL est composé de 3 cas.

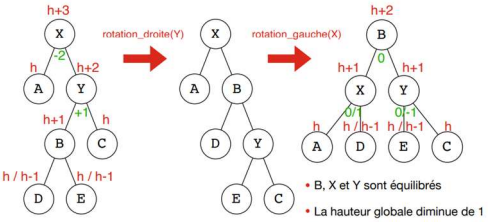
- Cas 1 : déséquilibre de -1, cas résous avec une rotation simple. rg(x).



- Cas 2 : déséquilibre de 0, cas résous avec une rotation simple. rg(x).



- Cas 3 : déséquilibre de +1, il faut réaliser une double rotation. On effectue une rotation dans le sens inverse du nœud du bas et ensuite on effectue la rotation dans l'autre sens du nœud du dessus.



Niveau(hauteur) → Meilleur : $2^h - 1$; Pire → φ^h

TDA : type de données abstrait
TDA Ensemble : TDA qui stocke les données sans répétition. Un ensemble est dit ordonné si on peut le parcourir dans l'ordre. C'est le cas quand il est mis en œuvre avec un ABR.
TDA Tableau associatif : Aussi appelé dictionnaire, Associe des clés uniques et des valeurs. Ce sont des std::map où on donne un type à la clé et un type à la valeur à stocker. C'est faisable avec des ABR en stockant clé et valeur. $\mathcal{O}(\log n)$.

Ensemble trié et unique (std::set<T, Compare>) : 3 pointeurs par nœud.
End : est un nœud au-dessus et il est vide.
Begin : est le nœud le plus à gauche
Insertion : par clé en $\mathcal{O}(\log n)$, par indice (itérateur) en $\mathcal{O}(1)$, plage de N clés dans un set de taille S en $\mathcal{O}(n \cdot \log(n+s))$. Insertion d'un grand nombre n de valeurs, mais les valeurs ont un petit domaine de définition se fait en $\mathcal{O}(n)$.
Suppression : par clé et indice comme insertion, supprimer plage de valeur en $\mathcal{O}(\text{dist(fist, last)})$.
Complexités :

Opération	en moyenne	au pire
Consultation	-	-
Insertion en fin	O(log(n))	O(log(n))
... au milieu	O(log(n))	O(log(n))
... au début	O(log(n))	O(log(n))
Suppression en fin	O(log(n))	O(log(n))
... au milieu	O(log(n))	O(log(n))
... au début	O(log(n))	O(log(n))
Taille	O(1)	O(1)
Suivant(s)	O(n)	O(n)
Distance	-	-

Tableau associatif (std::map<Key, Value, Compare>) : Comme un set, sauf qu'il contient des éléments de type « pair ». Il faut aussi une fonction de comparaison qui tri les clés. On accède à l'indice avec **.first** et à la valeur avec **.second**. Il fournit aussi l'opérateur[]. Il faut faire attention avec celui-ci, car une lecture d'une **valeur qui n'est pas présente l'ajoute dans la map** (utiliser find).

```
string chaine = "abracadabra";
std::map<char,size_t> m;

size_t i = 0;
for(char c : chaine) m[c] = i++;

for(auto p : m)
    cout << "m[" << p.first << "]=" << p.second << "\n";
```

```
m[a]=10
m[b]=8
m[c]=4
m[d]=6
m[r]=9
```

Complexités :

Opération	en moyenne	au pire
Consultation	O(log(n))	O(log(n))
Insertion en fin	O(log(n))	O(log(n))
... au milieu	O(log(n))	O(log(n))
... au début	O(log(n))	O(log(n))
Suppression en fin	O(log(n))	O(log(n))
... au milieu	O(log(n))	O(log(n))
... au début	O(log(n))	O(log(n))
Taille	O(1)	O(1)
Suivant(s)	-	-
Distance	-	-

Multiset/multimap : Comme set et map, mais permet de stocker plusieurs fois la même clé. Pas d'opérateur [] pour les multimap et equal_range retourne toutes les clés équivalentes.

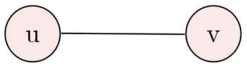
STL : propose déjà les opérations ensemblistes. Pour ces opération (union, intersection), il faut passer les deux plages sur lesquels faire les opérations et la plage stockant le résultat.

Consommation mémoire :

std::array	Aucun coût à part la taille des éléments
std::vector	3 pointeurs (taille à vide d'un vector), éléments
std::forward_list	1 pointeur sur la liste, 1 pointeurs par élément + éléments
std::list	2 pointeurs + 1 size_t (taille à vide), 2 pointeurs par élément + éléments
std::deque	6 pointeurs + n/c pointeurs + c éléments
std::set	2 pointeurs + 1 size_t (à vide), 1 maillon du set stocke 3 pointeurs et l'élément
std::map	2 pointeurs + 1 size_t (à vide), (3 pointeurs + clé) par élément + élément

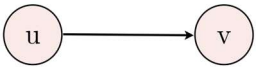
---

Graphes non orientés : Un graphe non orienté $G = (V, E)$ est formé, d'un ensemble V dont les éléments sont appelés **sommets**. D'un ensemble E dont les éléments sont appelés **arêtes**. D'une fonction d'incidence qui associe à chaque arête e une paire $\{u(e), v(e)\}$ de sommets appelés extrémités de l'arête e.



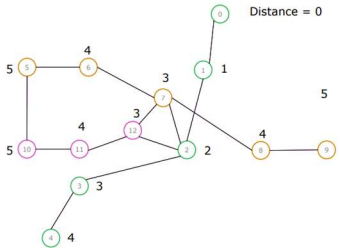
Une arête e relie ses deux extrémités u et v et est **incidente** avec elles.
Deux sommets u et v sont dits **adjacents** s'il existe une arête e qui les relie. Ils sont incidents à l'arête e.

Graphes orientés : Un graphe orienté $G = (V, E)$ est formé, d'un ensemble V dont les éléments sont appelés **sommets**. D'un ensemble E dont les éléments sont appelés **arcs**. D'une fonction d'incidence qui associe à chaque arête e une paire $\{u(e), v(e)\}$ de sommets appelés extrémités de l'arête e.

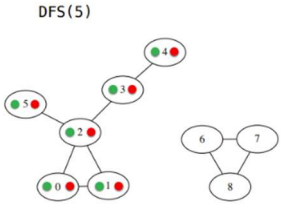


On appelle u **l'extrémité initiale** et v **l'extrémité finale** de l'arc e.
On remplaçant tous les arcs de G par des arêtes de même extrémités, on obtient son graphe sous-jacent (non orienté).

Parcours en largeur (breadth first search ou BFS) : utilisation d'une file FIFO qui parcourt d'abord tous les sommets adjacents avant ceux à distance 2, puis 3, ... BFS parcourt les sommets par ordre de distance croissante au sommet de départ.



Parcours en profondeurs (depth first search ou DFS) : méthode récursive qui s'appelle pour tous les sommets adjacents, Le parcours depuis un sommet n'atteint pas nécessairement tous les sommets du graphe.



Pré-ordre : 5-2-0-1-3-4
Post-ordre : 1-0-4-3-2-5

Complexités (N sommets et M liens) :

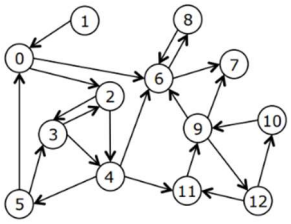
Liste des arêtes incidents à V (matrice)	$\boldsymbol{O(N)}$
Liste des arêtes incidents à V (liste)	$\boldsymbol{O(deg(V))}$
Au total un parcours pour une matrice	$\boldsymbol{O(N^2)}$
Au total un parcours pour une liste	$\boldsymbol{O(N+M)}$

Dijkstra : On stocke dans chaque sommet 2 informations. La distance (distTo(V)) jusqu'au sommet de départ et le dernier arc (edgeTo(V)) du chemin le plus court. On traite les sommets dans l'ordre de priorité de leur poids. Le poids le plus faible est celui à privilégier. Les poids sur le chemin le plus court final, sont les poids initiaux et pas cumulés.
Complexités (N sommets et M arcs) :

Accès arcs sortant	$O(deg_+(V))$
Modifier la priorité d'un sommet (réalisé au pire à chaque relâchement de sommet donc M fois)	$O(log(N))$
Complexité globale de Dijkstra	$O(M \cdot log(N))$

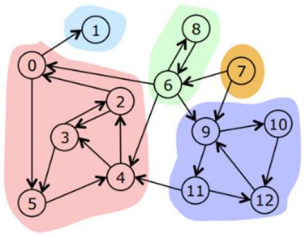
Tri topologique (topological sort) : redessiner un graphe orienté acyclique (Directed Acyclic Graph ou DAG) pour que tous ses arcs pointent dans la même direction, Ce n'est évidemment pas possible pour un graphe comprenant un circuit. **Tri topologique = Inverse du post-ordre.**
Détection de circuit : Si le parcours nous amène à atteindre un sommet présent dans cette pile, tous les éléments de la pile entre ce sommet (vertex) et le sommet (top) de la pile forment un circuit.

Algorithme de Kosaraju-Sharir : Calculer le post ordre inverse du parcours en profondeur pour le graphe inverse de G



Pré-ordre : 0 2 3 4 5 11 9 6 7 8 12 10 1
Post-ordre : 5 7 8 6 10 12 9 11 4 3 2 0 1
Post inverse : 1 0 2 3 4 11 9 12 10 6 8 7 5

Finalement Calculer les composantes connexes par parcours en profondeur DFS sur G dans cet ordre :



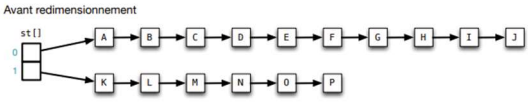
DFS(1)->{ 1 }
DFS(0)->{ 0,5,4,2,3 }
DFS(11)->{ 11,12,9,10 }
DFS(6)->{ 6,8 }
DFS(7)->{ 7 }

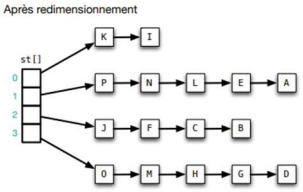
Étape :

1. Trouver le DFS en post ordre et l'inverser.
2. Inverser les flèches sur le graphe.
3. Refaire des DFS avec le post ordre inverse pour trouver les composant connexes.

Tables de Hachage :

Résolution de collisions par chainage : Utilise un tableau de M listes simplement chaînées pour stocker N paires clé/valeur, avec $M <$ N. Transforme la clé k en un entier $0 \leq h(k) < M$





- Pour garder une longueur moyenne de liste N/M relativement constante, on va
- doubler M quand $\frac{N}{M} \geq 8$
- diviser M par deux quand $\frac{N}{M} \leq 2$
- Attention, il faut re-hacher toutes les clés quand change M

Résolution de collisions par sondage linéaire : insérer une clé k si l'emplacement à $h(k)$ l'indice est occupé, essayer $(h(k)+1)\ mod\ M$, $(h(k)+2)\ mod\ M$, … jusqu'à trouver un emplacement vide.

Chercher la clé k si l'emplacement à l'indice $h(k)$ est occupé mais ne correspond pas à k,essayer $(h(k)+1)\ mod\ M$, $(h(k)+2)\ mod\ M$, … jusqu'à trouver soit k un emplacement vide.

---

std::upper_bound : Returns an iterator pointing to the first element in the range [first,last) which compares greater than val.

std::lower_bound : Returns an iterator pointing to the first element in the range [first,last) which does not compare less than val.