- **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++;
}
```

En cas d'enchaînement, il faut prendre en compte les différentes parties du code.

---

f = O(g) → f croit au plus aussi vite que g

f = o(g) → f croit strictement plus lentement que g. $\frac{f(n)}{g(n)} = 0$

f = Ω(g) → f croit au moins aussi vite que g (inverse de grand O), ce qui veut dire g = O(f).

f = Θ(g) → f et g ont le même ordre de grandeur.

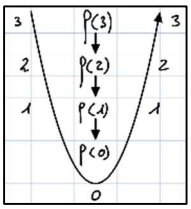
Règles :

1. Si une fonction est la somme de plusieurs termes, si l'un d'eux croit plus vite que les autres, on ne garde que lui et on ignore les autres.
2. Si une fonction est le produit de plusieurs facteurs, on peut ignorer tout facteur constant.

Rappels :

- $O(2^n) > O(n^2) > O\big(n \cdot log(n)\big) > O(n) > O\big(log(n)\big)$
- $O(k \cdot n) = O(n)$
- $O(n^2 + n + 1) = O(n^2)$

Affichage typique si on fait un appel d'une fonction récursive avec 2 affichages



Si on appelle x fois la fonction et que le paramètre diminue de 1 à chaque fois on a $O(x^n)$. S'il diminue de 2, on a $O(x^{n/2})$.

$$f(N-1) + f(N-1) = 2^n$$

On peut faire varier cette formule en appelant X fois la fonction et de diviser par 3 le paramètre à chaque fois, dans ce cas $O\big(x^{log_3(N)}\big)$. Attention aux simplifications possibles entre la puissance et le logarithme

---

Si on fait des appels récursifs et que l'on divise par 2 le paramètre que l'on passe, alors on fait $log(N)$ appels à la fonction.

Fonctions de la STL :

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

Si on a des appels à (f-1) et (f-2), la complexité est équivalent à $O(nbre\ d'or^n)$

lower_bound : Retourne un itérateur du premier élément qui n'est pas plus petit que la valeur passée (plus grand ou égal).

upper_bound : Retourne un itérateur du premier élément qui est plus grand que la valeur

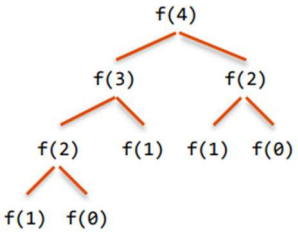
binary_search : Cherche un élément dans une plage donnée, true si trouvé sinon false.

equal_range : Retourne une plage contenant les valeurs à chercher (dans une liste triée) ce sont deux itérateurs, un du début de la plage (first) et un de la fin (second).

Paramètres de récursivité : Paramètres changeant à chaque exécution
Cas trivial : Termine la fonction
Cas général : Effectue un nouvel appel récursif

La complexité de la fonction étant constante, on calcule via le nombre d'appels récursifs. Le mieux reste d'essayer avec un nombre pas trop grand.

Fibonacci est un très bon exemple d'algorithme récursif et qui permet de détailler simplement la manière de les comprend. Pour rappel Fibonacci indique que pour calculer $f(n) = f(n-1) + f(n-2)$. On va donc dessiner un arbre des appels récursifs. De cette manière, on peut établir combien il nous faut d'appel pour chaque niveau.



$$A(0) = 0$$
$$A(1) = 0$$
$$A(2) = 2$$
$$A(3) = 2 + A(2) = 4$$
$$A(4) = 2 + A(3) + A(2) = 8$$
$$A(n) = 2 + A(n\text{-}1) + A(n\text{-}2)$$

Les fonctions de types Fibonacci appelant l'avant dernier et l'avant avant dernier pallier ont une complexité particulière en $O(nbre\ d'or)$. C'est le cas pour le pire cas de l'algorithme d'Euclide (a % b = a - b).

Encore un exemple si on a 4 lettres à placer dans n'importe quel ordre. La complexité sera de n !, donc on a 4 lettres placées en première place, puis 3 à mettre, etc…

L'algorithme **MiniMax** a pour but de minimiser les pertes maximums. C'est-à-dire qu'il assigne à chaque coup possible un certain score et le coup ayant le meilleur score est celui choisi. La complexité de ce type d'algorithme est assez difficile à établir. Mais en général, le joueur peut choisir entre M coups, on explore alors N coups successifs. Ce qui fait un algorithme en $O(M^n)$. **NegaMax** c'est une version où le score d'une position pour un joueur est l'inverse pour l'autre.

L'élagage alpha-beta c'est d'arrêter la recherche de minimum beta s'il est plus petit que le max (alpha) au niveau au-dessus. Et arrêter la recherche de maximum alpha s'il est déjà plus grand le minimum (beta) au niveau au-dessus. Cela permet de couper les branches de l'arbre qui ne donnent pas un résultat satisfaisant. Permet de doubler la profondeur d'exploration et dans le pire cas, il n'apporte aucune amélioration.

Fibonacci → Récursif $O(\phi^n)$ et Itératif $O(n)$. Il faut faire attention à ne pas dépasser la pile d'appels, ce qui peut arriver notamment avec la somme de N premiers entiers qui récursivement explose la pile. Récursion finale, si la dernière opération est un appel récursif alors on peut le faire itérativement.

```
fonction recursion(n)
 faire A
 si B, alors
   faire C
   appeler recursion(..)
 fin si
```

```
fonction iteration(n)
 boucler
   faire A
   si non B, alors
     sortir boucle
   fin si
   faire C
 fin boucle
```

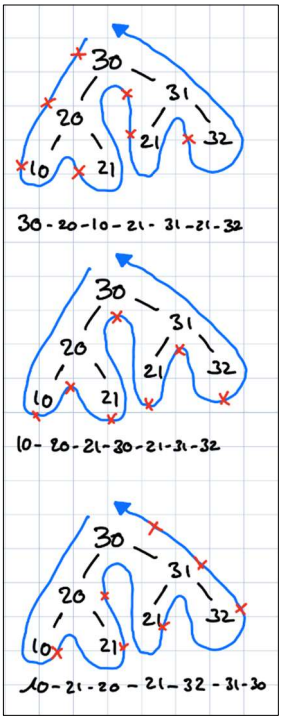
Algorithmes	Complexités
Factorielle récursif	O(n)
Factorielle itératif	O(n)
Fibonacci récursif	$O(1.618^n)$
Fibonacci itératif	O(n)
PGCD (Euclide)	O(log(n))
Tours de Hanoï récursif	$O(2^n)$
Tours de Hanoï itératif	$O(2^n)$
Permutations	O(n!)
Tic Tac Toe	9!
Puissance 4, profondeur d'exploration de d tours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

!! S'il y a un appel à N-1 et N-2, c'est Fibonacci !!

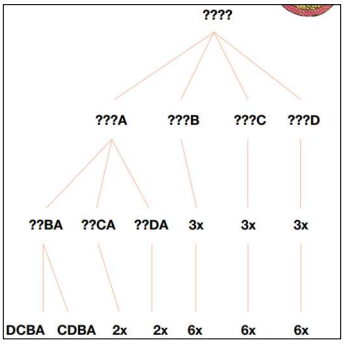
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

Comme pour les complexités, un algorithme qui va s'appeler X fois N-1 fois donne un résultat du type : $(X)^n$.

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



Fibonacci : Appel à (f-1) et (f-2)
Euclide : Appel à f(b, a%b)
Hanoï : Appel 2 fois à f(n-1)
Permutations : (n) appel n-fois (n-1), etc…



Récursion finale : Quand la toute dernière opération de la fonction récursive est l'appel récursif

```
long sumRec(int n) {
    if(n<=0) return 0;
    return n + sumRec(n-1);
}
```

```
long sumTailRec(int n, long r) {
    if(n==0) return r;
    return sumTailRec(n-1,n+r);
}
```

Tri à bulles : $O(n^2)$ comparaisons. Complexité de 0 à $O(n^2)$ échanges dans le pire cas. Stable s'il n'échange pas les éléments égaux. Il faut le voir comme des bulles qui remontent à la surface, il 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. Dans un tri intermédiaire, on retrouve alors toujours les plus grands éléments en fin.

Worst-case performance	$O(n^2)$ comparisons, $O(n^2)$ swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons, $O(n^2)$ swaps

Exemple d'une première passe

$$(\mathbf{5}\,\mathbf{1}\,4\,2\,8) \rightarrow (\mathbf{1}\,\mathbf{5}\,4\,2\,8),$$
$$(1\,\mathbf{5}\,\mathbf{4}\,2\,8) \rightarrow (1\,\mathbf{4}\,\mathbf{5}\,2\,8),$$
$$(1\,4\,\mathbf{5}\,\mathbf{2}\,8) \rightarrow (1\,4\,\mathbf{2}\,\mathbf{5}\,8),$$
$$(1\,4\,2\,\mathbf{5}\,\mathbf{8}) \rightarrow (1\,4\,2\,\mathbf{5}\,\mathbf{8}),$$

Tri par sélection : $O(n^2)$ comparaisons. Complexité en $O(n)$ échanges peu importe l'ordre des entrées. Non-stable. Cherche le minimum et le met à la position courant via un échange. Ramène au début les éléments les plus petits.

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**

Worst-case performance	$O(n^2)$ comparisons, $O(n)$ swaps
Best-case performance	$O(n^2)$ comparisons, $O(1)$ swap
Average performance	$O(n^2)$ comparisons, $O(n)$ swaps

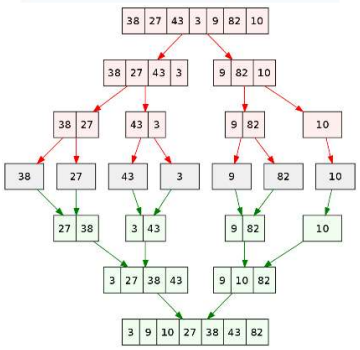
Tri par insertion : Meilleur cas : $O(n)$ pour tout et pire cas/cas moyen $O(n^2)$. A chaque passe, on le prend l'élément suivant et on le compare à tous les éléments de gauche. S'il est plus petit on le décale, sinon on le laisse à sa place. On retrouve tous les éléments triés vers le début, mais pas forcément les plus petits (ce qui serait le cas pour un sélection), voir exemple avec le 1.

Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps

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 par fusion : 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. Si partition pas de la même taille, première toujours plus grande.**

Worst-case performance	$O(n \log n)$
Best-case performance	$\Omega(n \log n)$ typical, $\Omega(n)$ natural variant
Average	$\Theta(n \log n)$



Tri rapide : 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 lorsqu'un élément est à la mauvaise place et les inverse. La manipulation s'arrête lorsqu'ils se croisent. On échange le pivot avec l'itérateur montant.

!! Quand on fait la récursion, c'est avec les éléments restants à gauche et à droite du pivot !!

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)

lo	i	hi	1	2	3	4	5	6	7	8	9	10	11	12
			E	X	E	M	P	L	E	D	E	T	R	I
1	6	12	E	E	E	D	E	I	P	M	X	T	R	L
1	3	5	D	E	E	E	E	I	P	M	X	T	R	L
1	2	2	D	E	E	E	E	I	P	M	X	T	R	L
1		1	D	E	E	E	E	I	P	M	X	T	R	L
4	5	5	D	E	E	E	E	I	P	M	X	T	R	L
4		4	D	E	E	E	E	I	P	M	X	T	R	L
7	7	12	D	E	E	E	E	I	L	M	X	T	R	P
8	9	12	D	E	E	E	E	I	L	M	P	T	R	X
8		8	D	E	E	E	E	I	L	M	P	T	R	X
10	12	12	D	E	E	E	E	I	L	M	P	T	R	X
10	10	11	D	E	E	E	E	I	L	M	P	R	T	X
11		11	D	E	E	E	E	I	L	M	P	R	T	X
			D	E	E	E	E	I	L	M	P	R	T	X

Dans le meilleur cas le pivot c'est la médiane des éléments à partitionner, on divise alors la partition par deux, complexité en $O(n \cdot log(n))$ (pareil en moyenne). Dans le pire cas, c'est une valeur extrême et il faut faire n partitions. La complexité est alors en $O(n^2)$. Pas adapter pour les petits tableaux, on fait alors un tri hybride avec un tri par insertion.

Worst-case performance	$O(n^2)$
Best-case performance	$O(n \log n)$ (simple partition) or $O(n)$ (three-way partition and equal keys)
Average performance	$O(n \log n)$

Si on utilise une comparaison non-stricte et que beaucoup d'éléments se répètent, la complexité est en $O(n^2)$. $O(n)$ échanges si tableau trié. Les échanges se font en place et le tri n'est pas stable.

Sélection rapide : Permet de trouver le k[ième] plus petit élément dans un tableau. On utilise le même algorithme de partition qu'un tri rapide, mais ne continue que du côté pertinent.

Pire cas	$O(n^2)$
Moyenne	$O(n)$
Meilleur cas	$O(n)$

Tri comptage : Utilise les propriétés des données (typiquement avec les cartes). Il va compter chaque type de données et laisse de la place dans un nouveau tableau pour les placer. Il utilise 2 tableaux pour stocker les éléments (source, destination) et un tableau pour les compteurs. Stable, pas en place.

Complexité $O(n+b)$ pour n éléments

pouvant prendre b valeurs distinctes

Tri par base : On trie les tableaux chiffres par chiffres (unités, dizaines, etc…). Il est stable et pas en place.

Complexité $O(d.(n+b))$ pour n éléments pouvant prendre b^d valeurs

La caractéristique que l'on veut utiliser pour regrouper les cartes c'est la dernière que l'on doit trier. Ex : Avoir un jeu de cartes triés avec des groupes d'enseignes et dans ces groupes triés par numéro. On fait d'abord les numéros, puis les enseignes. Ex.2 : Pour des chiffres, on fera d'abord les unités, puis les dizaines.

---

std::qsort : Il n'y a pas de garantie sur la complexité de l'algorithme. En pratique on est sur du linéarithmique ou quadratique dans le pire cas. Pas stable.

```
void qsort (void* base,
            size_t num,
            size_t size,
            int (*comp)(const void*,const void*));
```

- `void *base` : adresse du premier élément du tableau
- `size_t num` : nombre d'éléments à trier
- `size_t size` : taille d'un élément du tableau
- `int (*comp) (void const *a, void const *b)` : adresse de la fonction de comparaison, fournie par l'utilisateur.

std::sort : Complexité linéarithmique, variation du **quicksort**, pas stable.
std::stable_sort : (si on utilise une fonction de comparaison personnalisée, il faut avoir une inégalité stricte : < ou >). C'est une variation du **tri fusion**, donc il faut que std::move soit implémenté pour le type.
std::nth_element : Prend l'élément les itérateurs d'un tableau et au milieu le pilier pour partitionner et fait une partition (même complexité que la sélection rapide).
std::partial_sort : Trie les éléments du tableau donné, mais seulement jusqu'à un nombre d'élément voulu (représenté par un itérateur).

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))

Tips détecté les tris :

- **Tri à bulles :** Les plus grandes valeurs se retrouvent à la fin et le début n'est pas forcément désorganiser.
- **Tri par insertion :** Début organisé, mais la fin ne l'est pas encore (on peut voir des petites valeurs). On peut retrouver les places inversées.
- **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.

Tri rapide avec récursion terminale : Récursion traite la plus petite partition et l'itération traite la plus grande partition.

Les variables **globales** sont construites et allouées avant le main() et détruites et libérées après le main.
Les variables **statiques** sont construites lors du premier passage et détruites et libérées après le main().
Les variables **automatiques** sont construites lors de leur exécution et détruite lors de la sortie du bloc.
Variables **dynamiques** sont construites au passage de l'exécution sur « new » et elles sont 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 mémoire
T* q = new(p) T{};                       // construction objet de type T
```
Peut lever 2 types d'exception, soit std::bad_alloc ou une exception levée à la construction et relayée. On peut les éviter, pour la mauvaise allocation et ce qui permet de retourner un « nullptr ». Impossible pour les exceptions de T().

```
#include <new>

void* p = ::operator new(sizeof(T), std::nothrow);
```

La même manipulation est possible lors de la destruction et libération avec « delete ». « Delete » ne lève pas d'exceptions.
```
delete p;
```
cette ligne devient alors les deux lignes ci-dessous.

```
p->~T();                 // destruction de l'objet de type T
::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.

Sans constructeurs de copie explicite, le langage copie l'adresse, donc lors de la destruction l'objet est effacé 2 fois. Il faut en faire un qui copie les données.

```
RandomString(const RandomString& rs) {
    data = new char[rs.N];
    N = rs.N;
    copy(rs.data,rs.data+N,data); // <algorithm>
}
```

Il faut aussi fournir un opérateur d'affectation. Celui-ci doit copier et libérer les ressources précédentes. Par contre, il faut gérer certains cas limite comme l'auto-affectation, l'affectation des mêmes données (optimisation) ou encore la levée d'une exception.

```
RandomString& operator = (const RandomString& rs) {
    if (this == &rs) return *this;
    if (rs.N != N) {
        char* tmp = new char[rs.N];
        // noexcept après cette ligne
        delete[] data;
        data = tmp;
        N = rs.N;
    }
    copy(rs.data,rs.data+N,data);
    return *this;
}
```

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.

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.

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.

```
RandomString(RandomString&& other) noexcept
        : N(other.N), data(other.data)
{
    other.data = nullptr;
    other.N = 0;
}
```

Rappels :

- **Reserve :** Non-destructif, par contre ça n'agrandit pas le vecteur après coup.
- **Operator new/delete :** N'appel 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 realloue 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 fini 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.**

```
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";
    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
```

```
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
```

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*	

N = Ancienne taille

M = Nouvelle taille

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

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

priority_queue : comme un heap

Autres fonctions :

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

Tableau de taille fixe : std::array<T, N>
Tableau capacité fixe : Insert/Suppr en fin en O(1). Insert/Suppr en i (début) en O(n).
Buffer circulaire : On peut naviguer dans la structure sans s'arrêter grâce au modulo du calcul. Indexé de 0 à taille-1, bien pour insérer en début et fin (nul sinon). File FIFO. Il a deux indice, logique (celui qui est stocké dans sa structure, il peut commencer partout) et physique (l'occupation réelle en mémoire). Il se calcule : (début + i logique + capacité)%capacité.
Tableau de capacité variable : std::vector<T>.

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

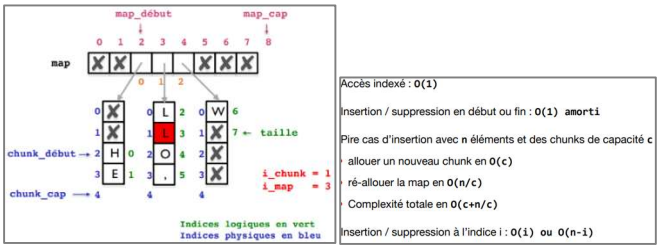
Liste simplement chaînée : std::forward_list<T>. On ne peut aller que vers l'élément suivant. Pas efficace pour insert/suppr autre que fin et début. Permet d'éviter de stocker en mémoire consécutif. On stocke à des endroits différents en mémoire (maillons).

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)

Liste doublement chaînée : std::list<T>. Comme la liste simple, sauf qu'il y a aussi un pointeur vers le 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>. Opération en début et fin en temps amorti constant. Accès indexé en temps constant. Ce sont des tableaux de tableaux.

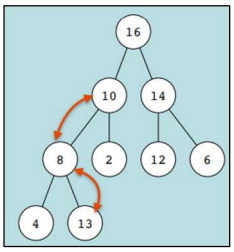


Opération	en moyenne	au pire
Consultation	O(1)	O(1)
Insertion en fin	O(1)	O(c+n/c)
... au milieu	O(min(i, n-i))	O(c+n/c)
... au début	O(1)	O(c+n/c)
Suppression en fin	O(1)	O(1)
... au milieu	O(min(i, n-i))	O(min(i, n-i))
... au début	O(1)	O(1)
Taille	O(1)	O(1)
Suivant(s)	-	-
Distance	-	-

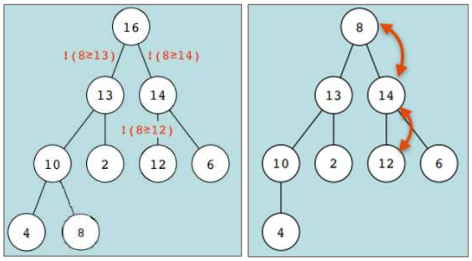
	array	vector	forward_list	list	deque
Mémoire additionnelle	0	3*p + O(n)*t si taille != capacité	(n+1)*p	(2*n+3)*p	O(n/B)*p + O(B)*t + 6*p
operator[]	O(1)	O(1)	N/A	N/A	O(1), mais un peu plus lent
push_front / pop_front	N/A	O(n)	O(1)	O(1)	O(1)
insert / erase au milieu	N/A	O(n)	O(1) si élément connu	O(1)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

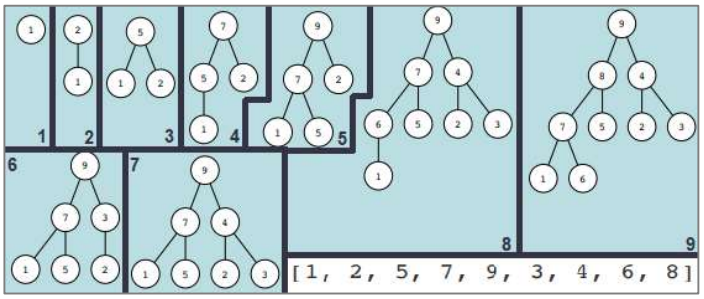
Un tas est un arbre binaire complet dont tous les éléments sont plus petits ou égaux que leurs parents (condition de tas).
push_back : Ajoute la valeur en fin de vecteur, mais pas dans le tas.
pop_back : Supprime le dernier élément du vecteur.
Insertion (push_heap) : On insère toujours un élément à la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



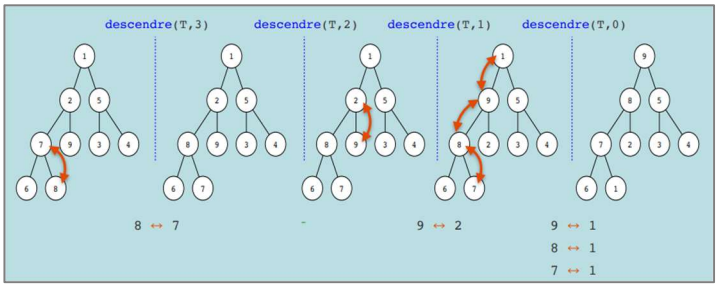
Supprimer le sommet du tas (pop_heap) : Il faut swap le sommet avec le dernier élément et réorganiser le tas. Pour réorganiser, il faut faire descendre le sommet (celui que l'on vient d'échanger) en direction du plus grand des enfants. Attention l'élément supprimer reste dans le tableau qui stocke le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



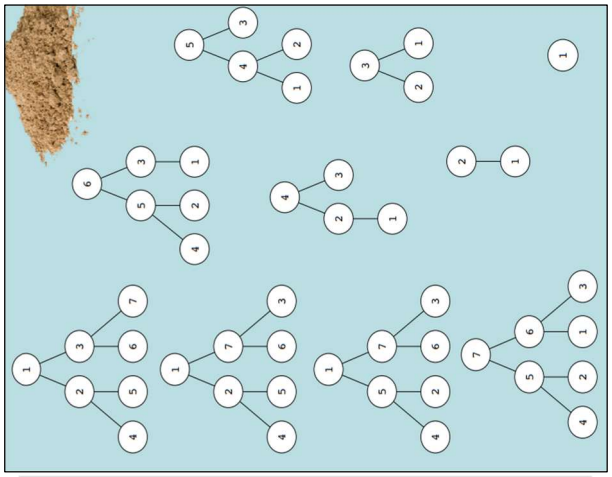
Créer un tas (make_heap) : On insère les éléments en fin et ensuite on les fait remonter à leur place. Ici, on insère les éléments dans l'ordre donc du deuxième au dernier. C'est un algorithme en O(N*log(N)). Remonter éléments.



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 O(log(N)) et tri en O(N*log(N)).
!! POUR LE TRI, IL FAUT QUE LA CDT SOIT RESPECTER SINON MARCHE PAS !!



```
Créer le tas                    Echanges
[1, 2, 3, 4, 5, 6, 7]           3↔7
[1, 2, 7, 4, 5, 6, 3]           2↔5
[1, 5, 7, 4, 2, 6, 3]           1↔7, 1↔6
[7, 5, 6, 4, 2, 1, 3]

Boucle sur k                    7↔3, 3↔6
[6, 5, 3, 4, 2, 1] [7]          6↔1, 1↔5, 1↔4
[5, 4, 3, 1, 2] [6, 7]          5↔2, 2↔4
[4, 2, 3, 1] [5, 6, 7]          4↔1, 1↔3
[3, 2, 1] [4, 5, 6, 7]          3↔1, 1↔2
[2, 1] [3, 4, 5, 6, 7]          2↔1
[1] [2, 3, 4, 5, 6, 7]
```

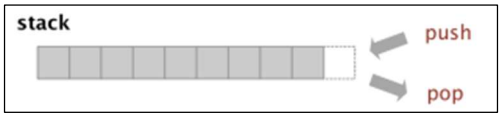
Dans la STL, il existe des fonctions pour faire ces algorithmes (make_heap, sort, pop, push, is). Elles ne vérifient pas que c'est bien un tas, c'est à nous de vérifier (pop et sort → tas entre [first ; last(et push → tas entre [first ; last-1(.

Make_heap	3*std::distance(first, last)
Sort_heap	N*log(N)
Push _heap	log(N)
Pop_heap	log(N)
Is_heap(_until)	N

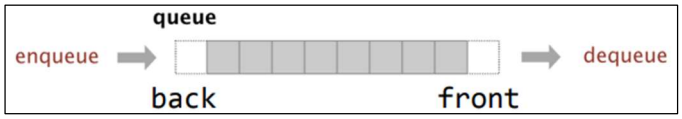
!! Si le tas respecte déjà la CDT, insérer un élément se fait en O(1) !!

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

Pile : 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).

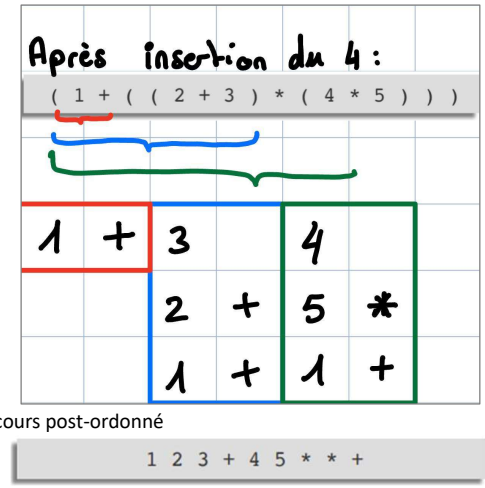


File (queue) : On insère d'un côté et on supprime de l'autre. FIFO. On peut donc accéder aussi accéder au dernier élément contrairement à LIFO.



File de priorité std::priority_queue: 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



NPI = Parcours post-ordonné

```
1 2 3 + 4 5 * * +
```

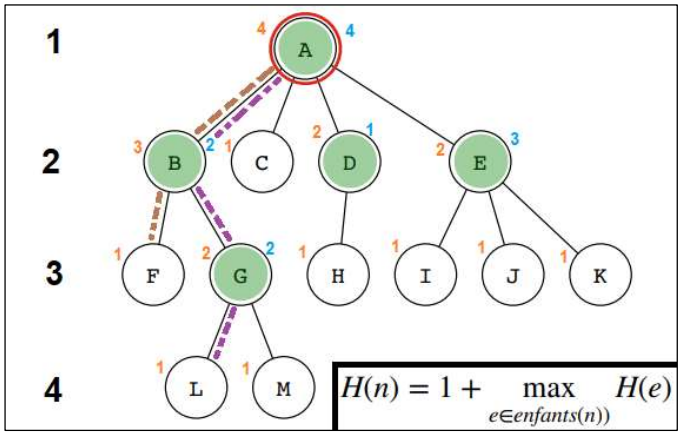
Fonctions forward_list / list:

splice_after(dest, source_list, pos_beg, pos_end) : Place après la position dest. dans la liste source_list les éléments commençant après pos_beg et jusqu'à pos_end (non compris). Si pas de end, prend 1 seul élément. Retire l'élément de la liste de départ. O(1).

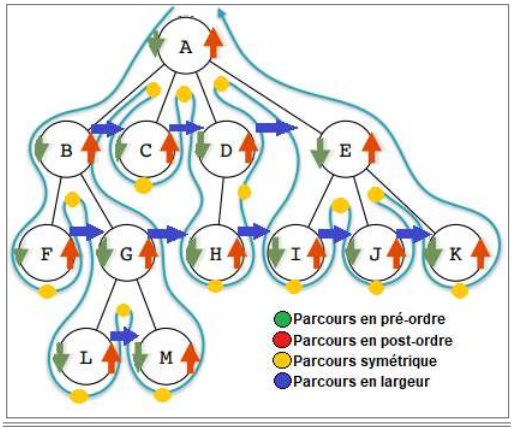
Splice : Pareil que after sauf qu'il fait tout aux positions données.

Merge : O(M+N), M et N étant les tailles des listes.

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.

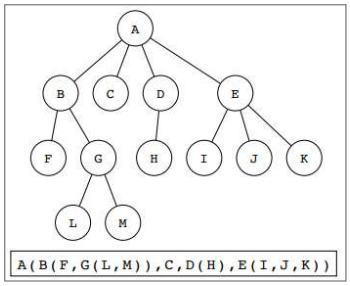


Vide : Sans aucun nœud et a une hauteur de 0.
Plein : Tous les nœuds de niveau inférieur à h-1 sont de degré d. Les nœuds de niveau h-1 ont un degré quelconque. Les nœuds de niveau h sont des feuilles.
Complet : Arbre plein dont le dernier niveau est rempli par la gauche.
Dégénéré : Arbre de degré 1.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



Il ne faut pas oublier qu'un nœud a un pointeur sur ses enfants, son parent et son puiné (enfant droite du même parent). Symétrique que pour **binaire**.
File largeur que pour les nœuds de même niveau et enfant dont on a visité les parents.

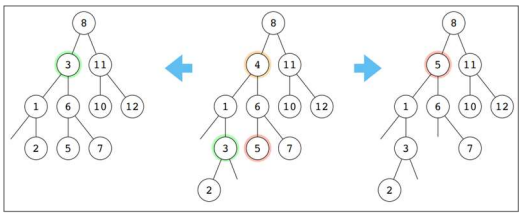
Représentation indentée : Type commande « tree » sur Windows.
Listes imbriquées :



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…

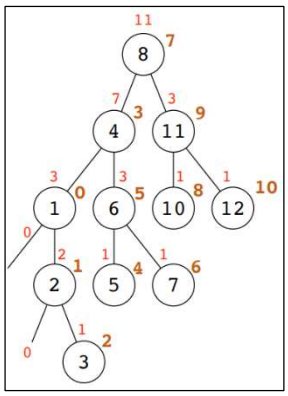
ABR : Arbre binaire où l'enfant gauche est toujours plus petit que son parent et l'enfant droite est toujours plus grand. Pour chercher/insérer/supprimer une clé, on retrouve une complexité en O(log(N)). Ici, on stocke tous les pointeurs comme pour l'arbre normal, mais sans le puiné.
Pour **chercher** et **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).
Parcourir un ABR dans l'ordre, c'est simplement un parcours symétrique.

Pour la suppression, il y a 3 cas. Si on supprime des feuilles il ne se passe rien. Si ce sont des nœuds de degré 1, il suffit de raccrocher le sous-arbre. Si c'est de degré 2 ou plus, il faut soit prendre le min du sous-arbre gauche et inverse.



L'itération se fait de façon similaire à un arbre classique. Iter. en profondeur veut dire que l'on donne tous les nœuds par lesquels on passe.

Taille : nombre de nœuds d'un arbre. C'est la somme des nœuds g + d + 1. (O(N) si taille à calculer et O(1) si taille stockée).
Rang : Nombre d'éléments strictement plus petits que la clé.



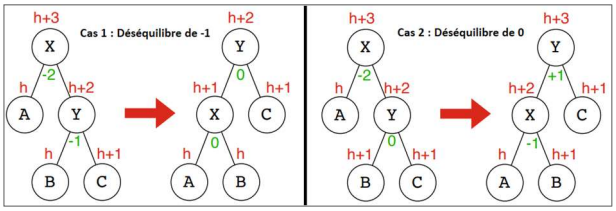
Opération	en moyenne	au pire
Parcours	O(n)	O(n)
Itération sur tout l'arbre	O(n)	O(n)
...suivant	O(1)	O(h)
Recherche, insertion, suppression, min, max	O(p)	O(p)

Parcours en O(N). Si lien parent, itér. sur tout l'arbre en O(N). suivant en O(1) en moy. ou O(h) au pire. Recherche, insert ou suppr max ou min O(p) avec p le niveau du nœud.
Dégénéré en O(N) et Plein en O(log(N)).
h et p sont en O(log(N)) et au pire O(N).

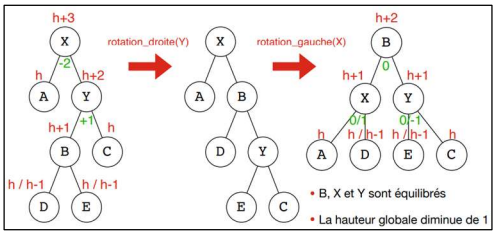
Un ABR est équilibré lorsque la différence de hauteur de ses deux sous-arbre est entre -1 et 1 (gauche – droite).

L'équilibrage AVL permet notamment de garder cette différence comprise dans les bornes à chaque opération. On fait les mesures en O(N) et O(1) si on stocke la hauteur dans les nœuds.

L'équilibrage AVL est composé de 3 cas :
2 cas résous avec une rotation simple. rg(x).



Le troisième cas concerne un déséquilibre de 1. Il faut dans ce cas 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 ensembles : Stockage de données sans répétitions, on peut le parcourir dans l'ordre s'il est mis en œuvre avec un ABR.
Union : On prend nos deux ABR (N et M éléments) et on parcourt du plus petit au plus grand et on avance que dans l'arbre ayant le plus petit élément. On insère ensuite en tête afin d'avoir un arbre dégénéré sans enfant droit. On finit par arboriser. O(N+M).
Intersection : Itération simultanée où on insère les clés présentent dans les deux arbres avec une complexité en O(N+M). On peut réaliser toutes les op.

TDA Tableau associatif : C'est aussi appelé un dictionnaire. Associe des clés uniques à 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. O(log(N)).

std::set : TDA ensemble triées et uniques, on ne peut pas modifier les éléments. On retrouve 3 pointeurs par nœud.
Sur cette structure le **end** est un nœud au-dessus de tous et vide et **begin** est le nœud le plus à gauche. **Retourne** toujours un pointeur sur l'élément insérer.
Insertion par clé en O(log(N)), par indice (itérateur) en O(1), plage de N clés dans un set de taille S en O(N*log(N+S)). Insertion d'un grand nombre N de valeurs, mais les valeurs ont un petit domaine de définition se fait en O(N).
Supprimer par clé et indice comme insertion, supprimer plage de valeur en O(dist(first,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	-	-

std::map : 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.

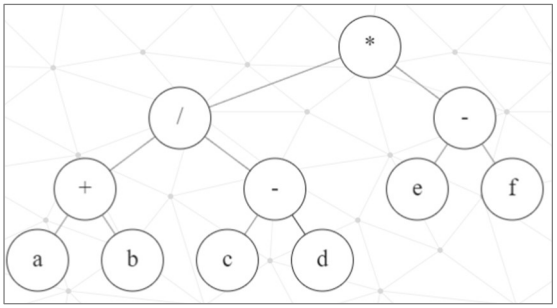
La STL propose déjà les opérations ensemblistes. Pour ces opération (union, intersection), il faut passer les deux plages sur lesquels faire les opérations et la plage stockant le résultat.

Consommation mémoire :

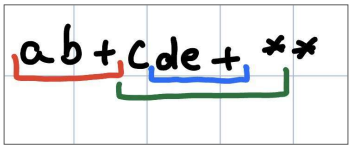
std::array	Aucun coût à part la taille des éléments
std::vector	3 pointeurs (taille à vide d'un vector), éléments
std::forward_list	1 pointeur sur la liste, 1 pointeurs par élément + éléments
std::list	2 pointeurs + 1 size_t (taille à vide), 2 pointeurs par élément + éléments
std::deque	6 pointeurs + 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

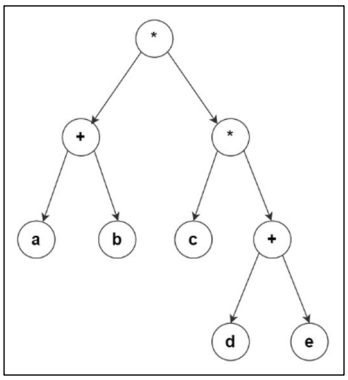
Transformation opérations :
Expression : (((1+2)/(3-4))*(1-2))
Préfixe : */+1 2 -3 4 -1 2
Postfixe : 1 2+ 3 4-/ 1 2-*

Arbre des opérations : (((A+B)/(C-D))*(E-F))



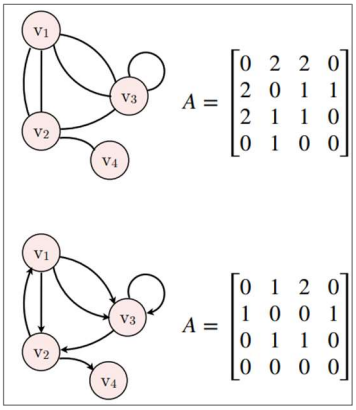
Exemple avec expression postfixe : (a+b)*(c*(d+e))





Graphe non-orienté : Composé de sommets et d'arêtes. Un relie deux sommets dans les deux sens. Les deux sommets sont adjacents.
Une **chaîne** est une alternance de sommets et liens. La **longueur** de cette chaîne est le nombre d'arêtes. Un **cycle** est une chaîne fermée qui commence et se termine au même endroit (élémentaire si aucun sommet se répète et simple si aucune arête se répète).
Graphe orienté : Composé de sommets et d'arcs. Un arc est unidirectionnel, il Relie un sommet A à un sommet B. Si on remplace les arcs par des arêtes, on obtient un graphe sous-jacent.
Un **chemin** est une suite alternée de sommets et d'arcs qui commence et fini au même endroit. **Circuit** pareil mais pour graphe ordonné.
Nomenclature : Une **boucle** est un lien d'un sommet sur lui-même. Des liens sont dits **multiples** s'ils touchent les mêmes sommets. Deux arcs sont **opposés** s'ils ont des sens différents. Un graphe **simple** n'a rien de tout ça.
Le **degré** est le nombre de liens touchant un sommet. On peut faire une distinction de demi-degré entrant/sortant pour les graphes orientés (deg_+/ deg_-).
Un graphe est acyclique s'il n'a pas de cycle simple. On parle de forêt si le graphe est en plusieurs morceaux disjoints, sinon c'est un arbre.
On peut rajouter un poids à chaque lien si qui rend le graphe pondéré.

Matrice d'adjacence : Indique les liens, elle est forcément symétrique pour un non-orienté, mais pas forcément pour un orienté.

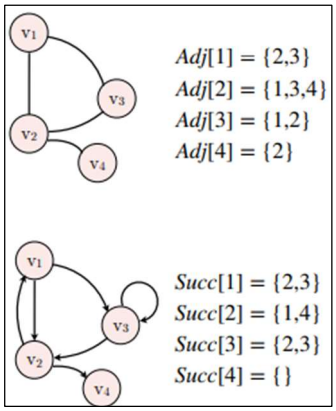


Complexités :

Stockage de la matrice (N sommets et M liens)	$O(N^2)$
Savoir si un arc existe	$O(1)$
Parcourir tous les liens qui partent d'un sommet (parcourir ligne matrice)	$O(N)$
Parcourir tous les liens qui arrivent sur un sommet (parcourir colonne matrice)	$O(N)$
Parcourir tous les liens d'un graphe	$O(N^2)$

NB : Pour un graphe pondéré, on peut stocker le poids directement dans la matrice et les liens inexistants, on met une valeur indiquant l'absence.

Liste d'adjacence : Pour chaque sommet, on stocke tous les sommets auxquels il est lié.

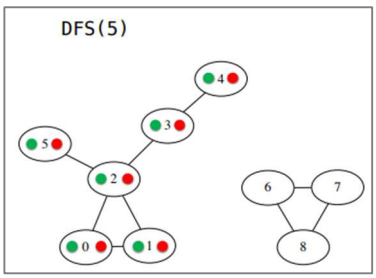


Complexités :

Stocker la liste (N sommets et M liens)	$\boldsymbol{O(N+M)}$
Savoir si un lien existe de A vers B ($deg_+ si\ arcs$)	$\boldsymbol{O(deg(A))}$
Parcourir tous les liens partant de A ($deg_+ si\ arcs$),	$\boldsymbol{O(deg(A))}$
Parcourir tous les liens du graphe	$\boldsymbol{O(N+M)}$
Parcourir tous les liens dont A est la destination	$\boldsymbol{O(N+M)}$
Pareil qu'au-dessus, mais avec listes de prédécesseurs	$\boldsymbol{O(deg\_(A))}$
$O(deg(A)) \equiv O(M/N)$	-

NB : Pour un graphe pondéré, on peut stocker une paire indice/poids. On peut utiliser n'importe quelle structure linéaire, tant qu'un accès par index existe.

!! Les parcours dans des graphes ordonnés ne peuvent pas aller à l'envers !!
Parcours en profondeur (DFS) : méthode récursive qui s'appelle pour tous les sommets adjacents. On parcours tous les sommets. Ici la liste d'adjacence est triée de façon croissante.

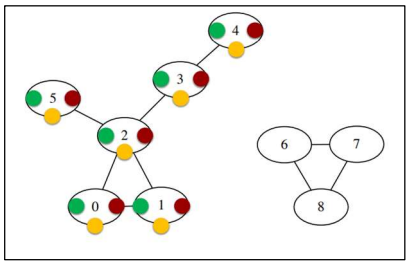



```
Précondition: les sommets sont
 marqués comme non visités

fonction profondeur (sommet v,
     Fn pre, (opt.) Fn post)

pre(v) // en pré-ordre

  marquer v visité
  pour tout w adjacent à v
    si w n'est pas visité
      profondeur(w, pre, post)

post(v) // en post-ordre
```

Pré-ordre : 5-2-0-1-3-4
Post-ordre : 1-0-4-3-2-5
Dans un graphe ordonné, on va prendre en compte le sens des flèches. Si celles-ci sortent, on va suivre. Si celles-ci rentrent, on ne va pas là-bas. Si plus aucune flèche n'est sortant, on va au prochain plus petit nœud restant à traiter

---

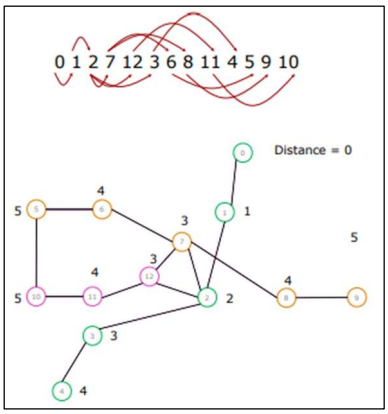
Parcours en largeur (BFS) : Parcours les sommets en fonction de leur distance. FIFO. On prend tous les sommets adjacents au sommet actuel, on les traite, puis on les sort de la file les uns après les autres.
Pour l'exemple ci-dessus. Lorsque l'on traite 2, cela veut dire que l'on ajoute 0,1 et 3 dans la file. Une fois ajoutés, on a fini de traiter 2. On peut refaire cela avec chaque nœud. On ajoute dans la file les nœuds adjacents pas encore traités par un point vert. On traite par distance au point de départ.



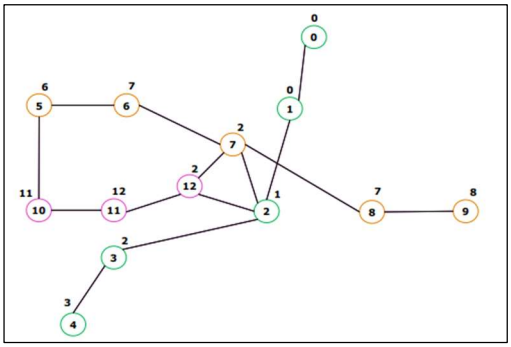

```
fonction largeur (sommet v,
                  Fn action)

  Initialiser une file Q
Q.push(v)
  marquer v

  tant que Q n'est pas vide
    v ← Q.pop()
action(v)
    pour tout w adjacent à v
      si w n'est pas marqué
        Q.push(w)
        marquer w
```

Parcours : 5-2-0-1-3-4
NB : Il faut marquer les sommets déjà visités et tous les sommets ne sont pas toujours visités.

La file ne contient que des sommets à distance K ou K+1.



On peut utiliser l'indice du parent au lieu d'utiliser un boolean pour savoir si on est déjà passé sur un sommet et cela nous permet de savoir d'où on vient. Pour l'image ci-dessous la chaine entre 0 et 5 serait : 0-1-2-7-6-5.



Complexités (N sommets et M liens) :

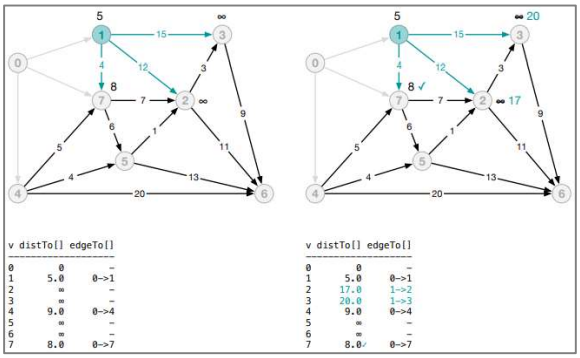
Liste des arêtes incidents à V (matrice)	$\boldsymbol{O(N)}$
Liste des arêtes incidents à V (liste)	$\boldsymbol{O(deg(V))}$
Au total un parcours pour une matrice	$\boldsymbol{O(N^2)}$
Au total un parcours pour une liste	$\boldsymbol{O(N+M)}$

Dijkstra : Avec le parcours en largeur avec parents, on peut calculer le chemin le plus court. Mais Dijkstra permet d'avoir le chemin le plus cours selon un poids (métrique). Il faut forcément des **poids positifs.** Les sommets dans la file de priorité sont classés selon leur poids. Le plus important est le plus léger (court). On les stocke tous sauf ceux traités.

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. A l'initialisation, on donne une distance infinie pour tous les nœuds sauf pour le nœud de départ qui est à 0. On relâche un arc si le chemin courant est le plus court jusqu'au nœud qu'on évalue.

On fait un parcours par distance croissante (largeur), il faut donc traiter tous les nœuds adjacents à un sommet avant de continuer. On peut voir ci-dessous que pour le nœud 7, on garde la distance précédente qui était bien plus courte.

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.



v	distTo[]	edgeTo[]
0	0	–
1	5.0	0->1
2	∞	–
3	∞	–
4	9.0	0->4
5	∞	–
6	∞	–
7	8.0	0->7

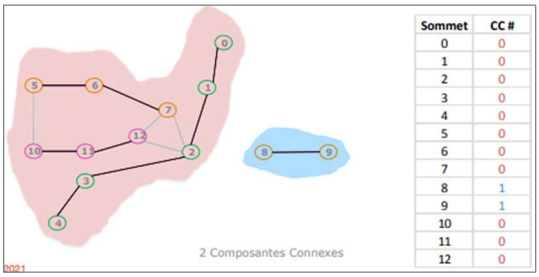
v	distTo[]	edgeTo[]
0	0	–
1	5.0	0->1
2	17.0	1->2
3	20.0	1->3
4	9.0	0->4
5	∞	–
6	∞	–
7	8.0	0->7

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	$O(M \cdot log(N))$

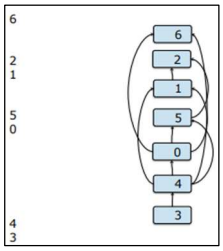
Une composante connexe permet de définir tous les sommets qui sont connectés ensembles. On essaye en quelques sortes d'identifier les groupes distincts d'arbres dans une forêt. (non-orienté)

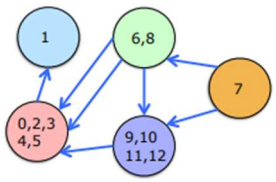
On utilise donc un parcours et on marque tous les sommets atteints avec un indice. On fait cela pour chaque arbre de la forêt et on peut donc savoir rapidement si deux sommets sont connectés. La complexité est identique que celle des parcours.



Sommet	CC #
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	1
9	1
10	0
11	0
12	0

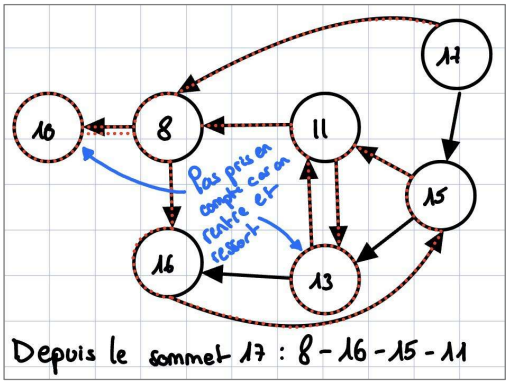
Tri topologique : Les flèches vont dans 1 seul sens. Inverse du post-ordre, on prend ce parcours inversé et on met les sommets les uns après les autres et on a un tri topologique.



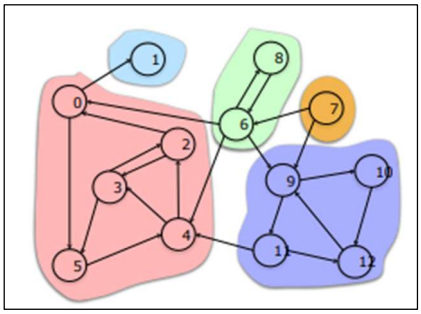


Graphe acyclique si on regroupe tous les sommets d'une CFC. (Tri topologique possible)

Détection de circuits : Parcours en profondeur où l'on garde la trace des sommets présents dans la pile de récursion. Si le parcours nous amène à atteindre un sommet présent dans la pile, alors tous les sommets entre ce sommet que l'on vient d'atteindre et ceux présents dans la pile forment un circuit.

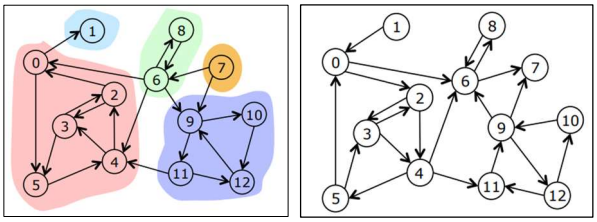


Deux sommets sont fortement connectés s'il existe un chemin (orienté) allant de l'un à l'autre et inversement (orienté). On peut faire des cycles dans les zones, mais pas en sortir.



Pour faire cet algorithme, il faut dans un premier temps inverser le graphe (sens des flèches). Puis faire son post-ordre. Ensuite, on inverse le post-ordre. En prend ensuite chaque nœud un à un et faire un parcours en profondeur à partir de celui-ci sur le graphe (non inversé) et voir tous les nœuds que l'on peut connecter (jusqu'à faire une boucle).

Graphe initial **Graphe inversé**



DFS du graphe inversé :
Pre 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

DFS des sommets dans l'ordre Post inverse jusqu'à ne plus avancer :
DFS(1) -> { 1 }
DFS(0) -> { 0, 5, 4, 2, 3 }
DFS(11) -> { 11, 12, 9, 10}
DFS(6) -> { 6, 8 }
DFS(7) -> { 7 }

NB : Il faut toujours avoir des flèches pour revenir dans une même composante.

Dijkstra pas de poids négatifs ! Bellman-Ford permet les poids négatifs. Si cycles de poids négatifs pas de solution. Par contre plus complexe. Il faut aussi faire attention aux arbres avec des boucles qui ne peuvent pas être traitées.

Composante connexe (CC) : Ensemble maximum de sommets connectés. Chaine entre v et w (que pour non-ordonné)

Tri topologique : Permet de redessiner un graphe orienté acyclique pour que tous ses arcs pointent dans la même direction (pas possible si circuit).

Détecter un circuit, c'est garder dans la pile les sommets présents. Si on croise un sommet déjà présent dans la pile c'est un circuit.

CFC : Chemin de a à b et b à a. (que pour ordonné).

Pour l'algo du plus court chemin, il est possible d'ignorer certains sommets lorsque l'on fait un tri topologique et que ces nœuds ne sont pas entre l'arrivée et la destination. Ex : 0->4 dans 0-1-2-3-4-5-6-7, on peut ignorer 5 à 7.