# Algorithme

## Introduction

Algorithme : ensemble finit d'étape dont le but est de résoudre un problème ou d'accomplir une tâche. Ces étapes sont formées d'un ensemble fini d'opérations.

Un algorithme est constitué d'entrées, de sorties et de dépendance relationnelle qui définit la sortie correspondant à chaque entrée possible.

Une recherche dichotomique a une complexité en O(log(n))

## Complexité

* La complexité caractérise son coût en temps processeur en fonction de la taille de la donnée.
* Complexité de O(1)
* Complexité de O(N)
* Complexité quadratique
* Sont de complexité logarithmique
* La complexité dans le meilleur des cas, dans le pire des cas et en moyenne

Le tri d'un vecteur, par exemple, peut faire appel aux probabilités. Dans le meilleur des cas il est déjà trié, dans le pire des cas il est trié à l'envers. Dans ce genre de cas la complexité moyenne est la même que la complexité dans le pire des cas. La complexité de la recherche séquentielle est dans le meilleur des cas de O(1) si on trouve directement. Dans le pire des cas on parcourt tout l'objet et la complexité est de O(n). Le cas moyen donne ce qui vaut à O(n).

## Notation de Landau

Une fonction f est un grand O de la fonction g. Ce qui signifie que f croît au plus vite que g. De cela découle deux règles simples :

## Notations complémentaires

Si une fonction est la somme de plusieurs termes, on garde que celui qui croît le plus vite. Si une fonction est le produit de plusieurs termes, on peut ignorer le facteur constant.

Dans le cas d'un enchaînement alternatif, il faut calculer le meilleur, le cas moyen et le pire cas. Une recherche séquentielle a dans le pire cas O(n).

f = O(g) f croît au plus vite que g f = o(g) f croît strictement plus lentement que g. f = Ω(g) f croît au moins aussi vite que g . f Θ(g) f et g ont le même ordre de grandeur.

# Récursivité

L'algorithme de Fibonacci ne doit surtout pas être écrite de manière récursive. Quand un appel est tout à la fin de la fonction, l'optimisation du compilateur fait que la fonction a la même exécution qu'une fonction itérative.

La complexité d'une fonction récursive est en général constante si on omet les appels récursifs. Ce qui veut dire que la complexité finale dépend uniquement du nombre d'appel.

Une fonction factorielle a une complexité de O(N), de O(log(n)) si on divise n dans l'appel de la fonction. L'algorithme de Fibonacci est de O(log(n)) avec comme base (1 ± √5) / 2. Le nombre de déplacement pour les tours de Hanoï est de 2^n -1 et sa complexité est exponentielle O(2^n). Le triangle récursif, la complexité est de O(a^2 - b^2). Le PGDC, la complexité est de O(log*x(a)) avec phi comme x. Les fractales, la complexité est de O(4^n + i\*2^n) car le 4^n calcul un carré et le i \* 2^n le triangle. Et pour les conversions de base, la complexité est de O(log*B(n)).

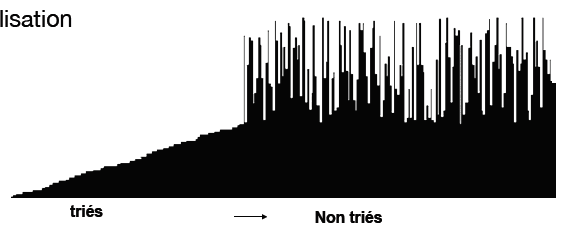
Complexité en général O() x le nombre d'or

|  |  |
| --- | --- |
| Factorielle récursif et itératif | O(N) |
| Fibonacci récursif |  |
| Fibonacci itératif | O(n) |
| PGDC Euclide | O(log(n)) |
| Euclide le pire cas est quand les deux opérandes sont des nombres de fibo |  |
| Tours de Hanoï itératif et récursif |  |
| Permutations | O(n!) |
| Tic Tac Toe | 9! |
| Puissance de 4 profondeurs d'exploration de d tours |  |
| Minimax (Negamax) m mouvement possibles par tour profondeur de d tours |  |
| Triangle récursif |  |
| Fractales |  |
| Conversion de base |  |

# Tris

## Tri à bulles

On prend deux éléments du tableau, on les compare et si elles ne sont pas dans l'ordre, on les permute. Lors du premier passage, le plus élément du tableau est au bout. Lors de chaque passage on travaille avec n-1 élément du tableau. Il est stable. La complexité est de O(n²). Mise en œuvre en place mais requiert une variable tampon.

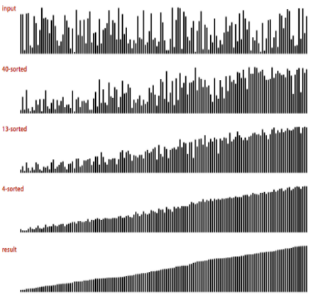


## Tri par sélection

On recherche l'élément le plus petit du tableau est on le permute avec l'élément non trié en première position. Contrairement au tri à bulle on trie depuis le début. La complexité est de O(n²) O(N) permutations. Se fait en place. Ce tri n'est pas stable.

## Tri par insertion

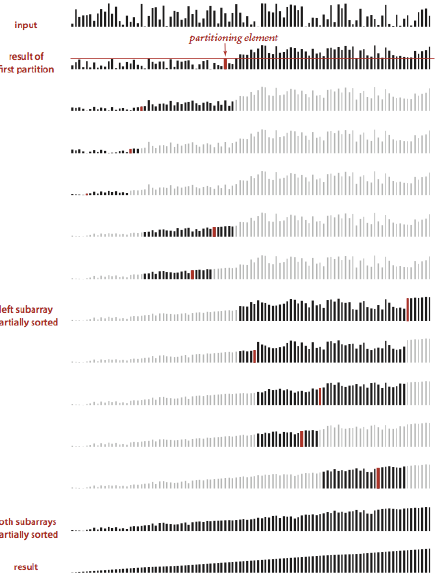
On prend le premier élément non trié et tant qu'il est plus petit que le précédent, on le permute avec celui-ci. Se fait en place, tri stable. La complexité dépend dans le meilleur cas O(N), pire cas et cas moyen O()



## Tri de Shell

On divise le tableau en sous tableaux d'abord très grand, on les trie et on divise en petit tableaux de plus en plus petit. La complexité dépend de la séquence choisie pour h (taille du tableau). La performance est assez proche du tri par fusion. Les petits sous

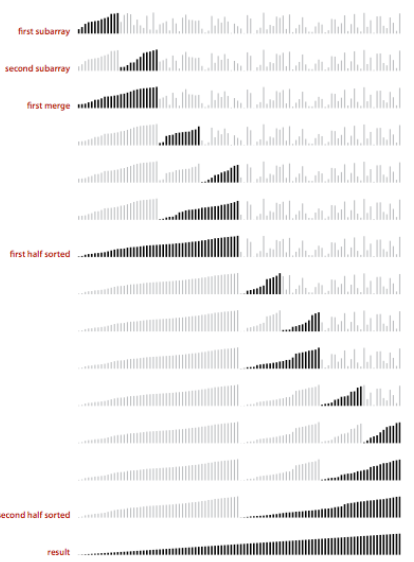
Tableaux sont triés par insertion h(x)=3h(x-1) +1. (1,4,13,40 en général). Complexité au pire de . On trie par insertion les petits tableaux.



## Tri fusion

On divise en sous-tableaux, on trie séparément les deux sous tableaux et on compare à chaque fois les premiers de chacun. Pour les petits tableaux, on fait un tri pas insertion. Complexité de . Ne se fait pas en place mais stable. Il faut utiliser des ≤ au lieu de <. Mis en place dans std::stable\_sort.

N’est pas adapté aux petits tableaux



## Tri rapide

On choisit un élément pivot, d'un côté les éléments plus petits que le pivot et de l'autre ceux plus grands. Pour optimiser le tri rapide, on fait un appel récursif seulement sur la partition la plus petite. Le pire cas est en théorie impossible si on choisit bien le pivot. Meilleur cas , pire cas et cas moyen . Pour les petites partitions on fait un tri par insertion. Ne se fait pas en place, n'est pas stable. Mise en place std::sort

Le C++ sort à besoin de l'opérateur < pour fonctionner, il garantit une complexité de O (n log(n)) mais pas la stabilité contrairement à stable sort.

* Tri par échange : tri à bulle, tri par sélection, tri rapide.
* Tri qui ne dépend pas de la taille des données : tri à bulle, tri par sélection, tri par fusion. Le tri par insertion dépend de l'état de l'entrée.

# Structures linéaires

Les types de données abstrait possèdent des constructeurs, des modificateurs, des sélecteurs et des itérateurs.

|  |  |  |
| --- | --- | --- |
| std::vector | en moyenne | au pire |
| consulter un élément [ ], at | O(1) | O(1) |
| modifier un élément [ ], at | 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(1) | o(1 ou N) |
| suppression en fin pop\_back | O(N) | o(N) |
| ... au milieu erase | O(N) | O(N) |
| ... au début erase(begin()) | O(N) | O(N) |
| push\_back() et erase(v.begin()) dans la même boucle | O(N) |  |
| reserve | O(N) |  |
| vector<> v(M  Loop(N)  v.push\_back + v.erase | O(M\*N) |  |

## Tableau à taille fixe

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

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

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

## Tableau de taille variable et de capacité fixe

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

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

Vector sort O(N\*log(N)). Si erase au début = pop\_back = O(1).

## Listes

### Simplement chaînées

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

|  |  |
| --- | --- |
| Forward\_list | complexité |
| Push*front, pop*front | O(1) |
| sort | O(N\*log(N)) |
| splice\_after en fonction du nombre d'éléments | O(N) |
| erase\_after en fonction du nombre d'éléments | O(N) |

### Doublement chaînées

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

|  |  |
| --- | --- |
| list | Complexité |
| emplace permet d'insérer en place | O(1) |
| push*back, push*front | O(1) |
| splice (constant si un élément ou si liste entière) | O(N) |
| merge | O(N) |
| unique enlève les doublons | O(1) |
| pop*front, push*back | O(1) |
| sort | O(N\*log(N)) |
| insert | O(N) |

### Tri

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

## Deque

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

|  |  |
| --- | --- |
| Deque | Complexité |
| Insertion | O(1) |
| Pire des cas nouveau chunk | O(c) |
| Réallocation de la map (n nombre élément deque) | O(n/c) |
| Donc vraiment dans le pire des cas | O(c+n/c) |
| Suppressions aux extrémités | O(1) |

## Tas

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

push\_heap : L'insertion se fait en fin de tableau. Ensuite il faut faire remonter l'élément jusqu'à que la condition de tas soit respectée. O(log(n)). On part du principe que c’est déjà un tas.

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

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

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

Les conditions de tas sont :

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

## Type de donnée abstrait

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

### Pile

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

### Queue

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

### priority\_queue

Même chose que queue mais avec notion de priorité. Si on rentre une paire d'élément dans la file, on ressort en regardant d'abord le premier mais s’ils sont égaux on regarde le deuxième. la complexité pour push fait appel au push*back du type et à push*heap.

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

## Structures linéaires en C++

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

## Allocation dynamique

|  |  |
| --- | --- |
| resize() |  |
| constructeur sans argument | Construit |
| Réallocation de push\_back | D(args) cp/mv D |
| Si on construit et on affecte | Construire, mv, D |
| swap | Cm, Am, Am, D |

allocation statique, automatique et dynamique

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

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

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

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

Trois sources de fuite de mémoire :

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

## Arbres

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

|  |  |  |  |
| --- | --- | --- | --- |
| Arbres et arbres binaires | Meilleur | Moyen | Pire |
| Recherche (insertion, suppression, min, max, rang, sélection) | O(hauteur) | O(log(n)) | O(n) |
| Parcours (pré, post, symétrique, largeur) |  | O(n) | O(n) |
| Elément suivant |  | O(1) | O(h) |
| Tri (tree sort) |  | O(N\*log(n)) |  |

### Arbres quelconques

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

### Arbres binaires

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

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

## Set

Les sets stockent des éléments dans un ordre spécifique et ne peuvent être modifiés dans la structure. Utilisé pour les ABR. O(n) dans le cas où les éléments sont déjà trié sinon O(n\* log(n)), et constante pour le constructeur vide et constructeur par déplacement

set.insert(rand()%2) O(1)

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

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

## Multimap

Complexité comme Multiset.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Map | Multimap | Set | Multiset | priority\_queue | queue |
| Insertion | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(1) |
| Suppression | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(1) |
| Recherche | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(1) |

# Arbres

### Accélération du parcours exact de l'arbre

L'élagage alpha beta est le fait d'abandonner des branches classées non optimal. Dans le meilleur cas, il permet de doubler la profondeur d'exploration et dans le pire des cas cela n'apporte rien. L'ordre correspond à la branche par laquelle on commence. La table de transposition est de regarder les branches qui ont la même situation.

# Graphes non orientés

Un graphe est un ensemble de sommet connectés deux par deux par des arêtes ou des arcs. L'arbitrage est le fait de trouver un circuit dont la multiplication de tous les poids est plus grands que 1. L'algorithme de Bellman-Ford permet de trouver un circuit dont la somme des poids est négatifs.

Taille : Nombre d'arêtes

Ordre: Nombre de sommets

Graphe simple: pas de boucle, par plus d'une arête par paire de sommets. son inverse : multigraphe

Graphe partiel : on supprime quelques arêtes

Sous graphe : on supprime quelques sommets

Graphe complet : tous les sommets sont connectés

Adjacence : deux sont adjacents s'il y a une arête entre eux

Une API est une interface qui permet d'accéder à la structure du graphe de manière générique. Le contenu des sommets ne fait pas partie de l'API, on fait une table des symboles avec une map par exemple. La liste des arêtes fait partie de l'API du graphe. On peut le faire par une liste, une matrice d'adjacence, par une liste d'adjacence.

## Parcours en profondeur

Méthode récursive qui appelle tous les sommets adjacents. La pile de fait office de fil rouge. On peut lancer un parcours DFS depuis chaque sommet n'ayant pas été atteint au premier passage. Si pré-ordre on le marque en entrant dans la fonction, si post-ordre on le marque à la fin.

## Parcours en largeur

Utilisation d'une queue qui parcourt tous les sommets adjacents avant ceux à une distance de 2. Les sommets passent par une queue FIFO, on les marque en entrant dans la queue. Parcourt les sommets par ordre croissante au sommet de départ mais la file ne contient que des sommets de distance k ou k+1. Le BFS calcul les chaînes les plus courtes.

La complexité : On sort chaque sommet une seule fois quel que soit le parcours. V fois pour tout le graphe. (V étant le nombre de sommets). Considérer la liste de ses voisins dépend de la représentation : Matrice d'adjacence : V liste d'adjacence : degré(V). Au total par matrice , tandis que la liste V+E(E nombre d'arêtes).

Pour éviter les boucles ou les cycles dans le graphe, il faut marquer les sommets déjà atteints par le parcours.

Deux sommets sont connectés s'il existe une chaîne entre eux. Il faut pré-traiter le graphe pour répondre en temps constant. La relation de connectivité est réflexive, transitive et symétrique. Une composante connexe est un ensemble maximum de sommets connectés.

Un Parcours permet d'atteindre tous les sommets connexes à un sommet donné. S’il reste des sommets non marqués, on incrémente l'indice et on recommence. La recherche des composantes connexe à la même complexité que les parcours.

Un graphe Connexe admet un cycle Eulérien si et seulement s'il n’a pas de degré impair. S’il y a un cycle Eulérien, chaque passage par un sommet contribue pour 2 dans le degré de ce sommet. Si l’on cherche une chaîne plutôt qu’un cycle, deux sommets - le départ et l’arrivée - peuvent être de degré impair.

Cycle Hamiltonien idem que le cycle Eulérien mais en passant une et une seule fois par chaque sommet.

# Complexité

|  |  |
| --- | --- |
| std::find | O(n) |
| std:: lower*bound et upper*bound | O(log(n)) |
| advance | O(N) en moyenne |
| std::find | O(n) |
| std::generate() | O(n) |
| std::partial\_sort |  |
| std::sort() |  |
| std::stable\_sort() |  |
| std::stable\_sort() si fusion mémoire |  |
| nth\_element réarrange les éléments | O(n) |
| remove | O(n) |
| sort\_heap | O(log(n)) |
| Min/ max | O(log(n)) |
| Distance si random acess iteator | O(1) |
| Distance | O(n) |

