La notation de Landau « O »

Définition : La fonction f est un grand O de la fonction g, («f croît au plus aussi vite que g») et l'on note f = O(g) ou f(n)=O(g(n)) s'il existe une constante réelle positive c et un entier positif n0 tels que f(n) <= c*g(n), pour tout n >= n0. On dit que g(n) est une borne supérieure asymptotique pour f(n).

log(n)<n<n*log(n)<n^2<n^3<2^n

f = o(g) : $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

f = Ω(g) si et seulement si g = O(f)

Formellement **f = Θ(g)** si et seulement si

f = O(g) et f = Ω(g)

Si f(n) = 1 + 3·n2 + 4·n3, alors

f(n) = O(f(n)), O(n3), O(25·n3), O(n6), O(2n)

f(n) = o(n6), o(2n), mais pas o(f(n)) ou o(n3)

f(n) = Ω(f(n)), Ω(n3), Ω(n2), Ω(log2(n))

f(n) = Θ(f(n)), Θ(n3), Θ(1000·n3)

	Moyen	Pire	Meilleure
Rech. Dicho	O(log(n))	O(log(n))	O(1)
Rech. Linéaire	O(n)	O(n)	O(1)
std::find	O(n)		
std::lower_bound	O(log(n))		
std::upper_bound	O(log(n))		
std::sort	O(n*log(n))		
std::stable_sort	O(n*log(n))	O(n*log(n)^2)	
std::partial_sort	O(n*log(m)), n = nb élém & m = nb à trier		
std::nth_element	O(n)		
std::generate	O(n)		

Complexité sur un vecteur	
Algo	Moyen
v.reserve(N)	O(N)
v.insert()	O(N^2)
v.push_back()	O(N)
v.erase()	O(N^2)
Si boucle v.push_back + v.erase	O(N)
vector<> v(M) Loop(N) v.push_back + v.erase	O(M*N)
V Loop(n) v.push_back() + erase(begin))	O(N)

Complexités diverses	
Factorielle récursif + itératif	O(n)
Répétition d'une fonction F(N-A)	O((x*^{n}) x = nb de répétitions
Répétition d'une fonction F(N-A,B)	O((x^n * B) x = nb de répétitions
Attention lors des répétitions : F(N/x) + F(N/x)	O(x$^{Log_x(n)}$) x = nb de répét. = au log Si nb rep = log => O(n)
Attention lors des répétitions : F(N/X , B) + F(N/X , B)	O(x$^{Log_x(n)}$ * B) x = nb de répét. = au log Si nb rep = log => O(N * B)
Fibonacci récursif F(N-2) + F(N-1)	O((1+sqrt(5))/2)^n) = O(1.618^n)
Fibonacci itératif	O(n)
Loop(n) - loop(i)	O(n^2)
PGDC (euclide) - [récursif] Inversion a avec b et b avec r	O(log(n))
Hanoï (récursif/itératif) Nb déplacement = 2^{nb_disque} - 1	O(2^n)
Permutation	O(n!)
Morpion	9!
Puissance 4	O(7^d)
Minimax - Negamax (m mouvements possibles par tour et une profondeur de d tours)	O(m^d)
Triangle récursif	O(a^2 - b^2)
Fractale	O(4^n + i*2^n)
n + log(n) => n > log(n)	O(n)
A + B	Max(a,b)

Conteneur de séquences	
array	tableau statique contigu
vector	tableau dynamique contigu
deque	file d'attente à deux bouts
forward_list	liste simplement chaînée
list	liste doublement chaînée
Conteneur associatif	
set	collection de clés **uniques**, triées par les clés
map	collection de paires clé-valeur, triées par les clés, les clés sont uniques
multiset	collection de clés, triées par les clés
multimap	collection de paires clé-valeur, triées par les clés
Adaptateurs de conteneurs	
stack	adapte un conteneur pour fournir une pile (structure de données LIFO)
queue	adapte un conteneur pour fournir une file d'attente (structure de données FIFO)
priority_queue	adapte un conteneur pour fournir une file d'attente prioritaire

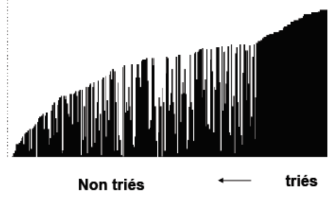
Tri à bulle, on compare 2 éléments successifs, si le 2ème est plus petit que le 1er, on switch. Ceci jusqu'à n.

- Dans l'ordre : on ne modifie pas
- Dans le désordre : on permute
- À Faire jusqu'à N

Tri d'échange

Complexité : $O(N^2)$
Stable : il garde le même ordre qu'au départ : A1B1B2A2 -> A1A2B1B2.
Mémoire :

- mise en œuvre en place sans tableau annexe.
- permutation requiert variable tampon O(1)

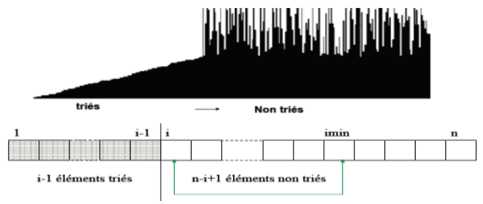



```
void BubbleSort(int *A,int n){
    for(int i = 0; i < n-1; ++i){
        for(int j = 0; j < n-1; ++j){
            if(A[j] > A[j+1])
                swap(A[j],A[j+1]);
        }
    }
} A = tableau / n = taille (nb éléments)
```

Tri par sélection, Recherche la position du minimum parmi les éléments non triés. Permuter minimum avec élément non trié en position i.
Elément jusqu'à i sont triés.
Répéter : i+1 à n.

Tri d'échange

Complexité : $O(N^2)$ comparaison et $O(n)$ permutations
Stable : Non
Mémoire : en place
Signe du tri : Entièrement trié début, fin non touché. Aucun élément plus petit que i peut être derrière.



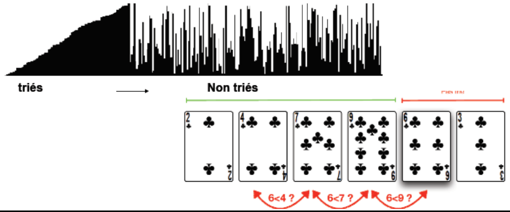

```
void SelectionSort(int *A,int n){
    for(int i = 0; i < n-1; ++i){
        int imin = i;
        for(int j = i+1; j < n; ++j){
            if(A[j] < A[imin])
                imin = j;
        }
        swap(A[i], A[imin]);
    }
} A = tab | n = taille (nb éléments)
```

Tri par insertion, On prend premier élément non trié. Tant qu'il est plus petit que élément précédent, on permute avec celui-ci jusqu'à ce qu'il trouve sa place dans la partie triée du tableau.

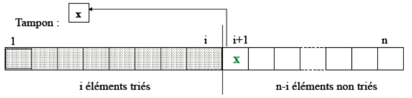
Complexité :

- Meilleur cas : tableau trié O(N)
- Pire cas : tableau trié à l'envers $O(N^2)$
- Cas moyen : ordre aléatoire $O(N^2)$

Stable : Oui
Mémoire : en place
Mieux que tri rapide pour tab [5-10]




```
void InsertionSort(int *A,int n){
    for(int i = 1; i < n; i++){
        int tmp = A[i]; int j = i;
        while(j-1 >= 0 && A[j-1] > tmp){
            A[j] = A[j-1];
```

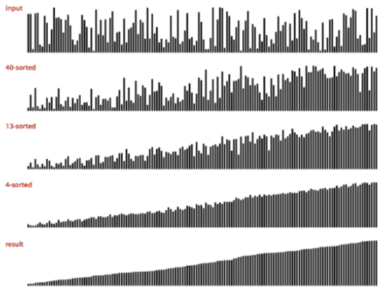


Tri de Shell, comparaison ne fait bouger un élément que d'une position à la fois.
On divise le tableau en h sous-tableau par saut de h éléments. On trie chaque sous-tableau, h = 1,4,13,40,121,364,1090,… 3h(x-1)+1

```
E H D E L I E L P L E M R X E T S
E ——— L ——— P ——— R ——— S
  H ——— I ——— L ——— X
    D ——— E ——— E ——— E
      E ——— L ——— M ——— T
```

Complexité : au pire $O(N^{3/2})$
Mémoire : en place
Stable : oui

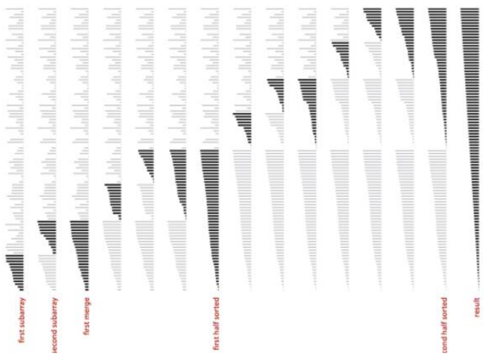
Pourquoi ce tri : quand h est grand, les h sont petits et ce tri est efficace. Amélioration du tri par insertion.




```
void ShellSort(int *A, int n){
    int h = 1;
    while(h < n/3){ h = 3*h+1; }
    while(h >= 1){
        for(int i = h+1; i < n; i++){
            int tmp = A[i]; int j = i;
            while(j-h >= 0 && A[j-h] > tmp){
                A[j] = A[j-h]; j -= h;
            } A[j] = tmp;
        } h = h/3;
    }
}
```

Tri par fusion, Découpage du tableau en 2.
Chaque tableau est trié.
Puis on compare les éléments (1 de chaque) et place dans un 3eme tableau (fusion).
Recommencer n fois.

Complexité : $O(N*\log_2(N))$
Mémoire : pas en place (nécessite un tableau temporaire de la même taille que l'original
Stable : oui
En C++ : std::stable_sort()
Appel récursif : si n>1, divise en 2 tableaux de n/2 éléments. Trier les 2 tableaux, fusionner les tableaux triés
Note : pas adapter pour les petits tableaux




```
fonction MergeSort(A,lo,hi)

  si hi <= lo, alors
    retourner
  fin si

  mid ← lo + (hi-lo)/2

  MergeSort(A,lo,mid)
  MergeSort(A,mid+1,hi)

  Fusionner(A,lo,mid,hi)
```

Entrée
- Tableau A
- indices lo et hi entre lesquels il faut trier

Sortie
- Tableau A trié des indices lo à hi

```
fonction Fusionner(A,p,q,r)

  L ← copie du tableau A de p à q
  R ← copie du tableau A de q+1 à r
  L(q-p+2) ← ∞      (sentinelles)
  R(r-q+1) ← ∞

  i ← 1, j ← 1
  pour k de p à r boucler
    si L(i) ≤ R(j), alors
      A(k) ← L(i)
      incrémenter i
    sinon
      A(k) ← R(j)
      incrémenter j
    fin si
  fin pour
```

Entrée
- Tableau A
- les éléments p à q du tableau A sont triés
- les éléments q+1 à r du tableau A sont triés

Sortie
- Tableau A modifié
- les éléments p à r du tableau A sont triés.

Tri rapide, choix d'un élément de pivot, répartition du tableau en deux. D'un côté + petit que pivot et l'autre les plus grands. Puis récursion dans chacune des deux partitions. Quand les 2 compteurs se croisent, on remplace pivot avec la place i.

1) Echange du pivot avec dernier

2) It moins gauche, it grand droite → quand arrêt on swap

3) Quand it - et it + croisent -> swap position i avec pivot

Complexité :

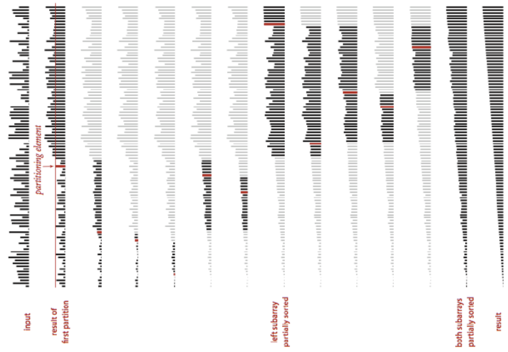
- Meilleur cas, le tableau se sépare toujours en 2 = taille égale. **$O(n*\log_2(n))$**
- Pire cas : découpage très inégale (1 et N-1). **$O(N^2)$**
- Moyen : **O(N)**

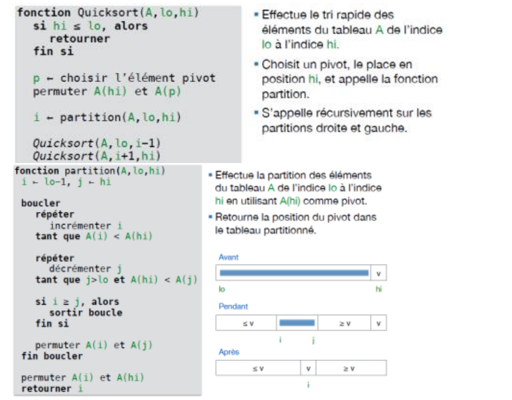
Mémoire : en place

Stable : non

Trouver pivot : valeur médiane = O(N) -> prendre médiane d'échantillon (3 éléments au hasard)

Trouver $K^{ième}$ plus petit élément du tableau de N élément : Min (K = 1), Max(K=N), médiane (K=N/2)





```
fonction Quicksort(A,lo,hi)
  si hi ≤ lo, alors
    retourner
  fin si

  p ← choisir l'élément pivot
  permuter A(hi) et A(p)

  i ← partition(A,lo,hi)

  Quicksort(A,lo,i-1)
  Quicksort(A,i+1,hi)
```

- Effectue le tri rapide des éléments du tableau A de l'indice lo à l'indice hi.
- Choisit un pivot, le place en position hi, et appelle la fonction partition.
- S'appelle récursivement sur les partitions droite et gauche.

```
fonction partition(A,lo,hi)
 i ← lo-1, j ← hi

 boucler
   répéter
     incrémenter i
   tant que A(i) < A(hi)

   répéter
     décrémenter j
   tant que j>lo et A(hi) < A(j)

   si i ≥ j, alors
     sortir boucle
   fin si

   permuter A(i) et A(j)
 fin boucler

 permuter A(i) et A(hi)
 retourner i
```

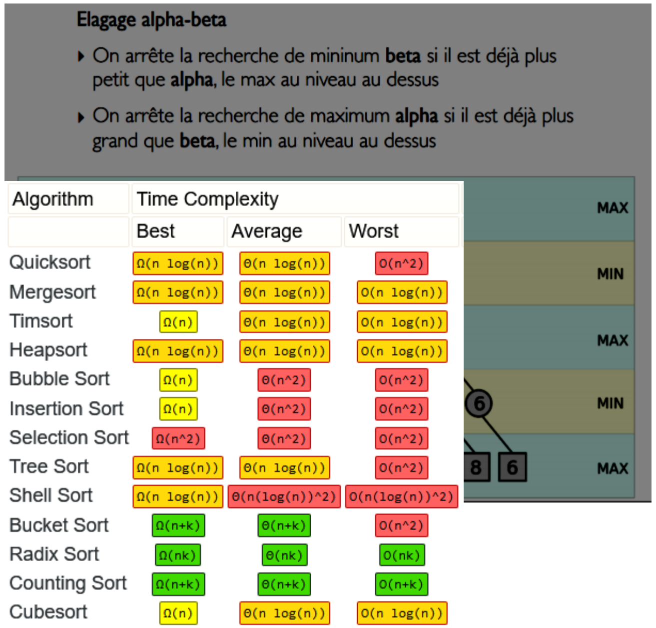
- Effectue la partition des éléments du tableau A de l'indice lo à l'indice hi en utilisant A(hi) comme pivot.
- Retourne la position du pivot dans le tableau partitionné.

Tris efficaces (complexité linéaire moyenne) : Tri fusion, rapide, par tas

Tris	Stable	Mémoire	Meilleur	Pire	Moyen
Bulle	Oui	place	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bulle stop	Oui	place	O(n)	$O(n^2)$	$O(n^2)$
Sélection	Non	place	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion	Oui	place	O(n)	$O(n^2)$	$O(n^2)$
Fusion	Oui	Non place	O(n*log(n))	O(n*log(n))	O(n*log(n))
Shell	Non	place	O(n*log(n))	$O(n^{3/2})$	Non calculable
Quick pivot centre	Non	place	O(n*log(n))	O(n*log(n))	O(n*log(n))
Quick début	Non	place	O(n*log(n))	$O(n^2)$	O(n*log(n))
Radix M casier	Oui	Non place	O(m*n)	O(m*n)	O(m*n)
Counting N → K val	Oui	Non place	O(k+n)	O(k+n)	O(k+n)

Meilleur cas = tableau déjà trié | Pire cas = tableau trié à l'inverse

Proposition : un algorithme de tri basé sur des comparaisons a au minimum une complexité $O(N*\log_2(N))$ comparaisons pour trier N éléments



Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))

list<T> l;	Création d'une liste vide	O(1)
list<T> l(begin, end);	Make a list and copy the values from begin to end.	O(n)
l.size();	Return current number of elements.	O(1)
l.empty();	Return true if list is empty.	O(1)
l.begin();	Return bidirectional iterator to start.	O(1)
l.end();	Return bidirectional iterator to end.	O(1)
l.front();	Return the first element.	O(1)
l.back();	Return the last element.	O(1)
l.push_front(value);	Add value to front.	O(1)
l.push_back(value);	Add value to end.	O(1)
l.insert(iterator, value);	Insert value after position indexed by iterator.	O(1)
l.pop_front();	Remove value from front.	O(1)
l.pop_back();	Remove value from end.	O(1)
l.erase(iterator);	Erase value indexed by iterator.	O(1)
l.erase(begin, end);	Erase the elements from begin to end.	O(1)
l.remove(value);	Remove all occurrences of value.	O(n)
l.remove_if(test);	Remove all element that satisfy test.	O(n)
l.reverse();	Reverse the list.	O(n)
l.sort();	Sort the list.	O(n log n)
l.sort(comparison);	Sort with comparison function.	O(n logn)
l.merge(l2);	Merge sorted lists.	O(n)
l.splice	Tranfert elements from list to list l = liste de dest. Pos = celle donnée	

queue< container<T> > q;	Création d'une file vide	O(1)
q.front();	Return the front element.	O(1)
q.back();	Return the rear element.	O(1)
q.size();	Return current number of elements.	O(1)
q.empty();	Return true if queue is empty.	O(1)
q.push(value);	Add value to end. Same as push_back() for underlying container.	O(1)
q.pop();	Remove value from front.	O(1)

stack<container<T>> s;	Création d'une stack vide	O(1)
s.top();	Return the top element.	O(1)
s.size();	Return current number of elements.	O(1)
s.empty();	Return true if stack is empty.	O(1)
s.push(value);	Push value on top. Pareil que push_back pour le conteneur sous-jacent	O(1)
s.pop();	Pop value from top.	O(1)

deque<T> d;	Création d'une deque vide	O(1)
deque<T> d(n);	Make a deque with N elements.	O(n)
deque<T> d(n, value);	Make a deque with N elements, initialized to value.	O(n)
deque<T> d(begin, end);	Make a deque and copy the values from begin to end.	O(n)
d[i];	Return (or set) the I'th element.	O(1)
d.at(i);	Return (or set) the I'th element, with bounds checking.	O(1)
d.size();	Return current number of elements.	O(1)
d.empty();	Return true if deque is empty.	O(1)
d.begin();	Return random access iterator to start.	O(1)
d.end();	Return random access iterator to end.	O(1)
d.front();	Return the first element.	O(1)
d.back();	Return the last element.	O(1)
d.push_front(value);	Add value to front.	O(1) (amortized)
d.push_back(value);	Add value to end.	O(1) (amortized)
d.insert(iterator, value);	Insert value at the position indexed by iterator.	O(n)
d.pop_front();	Remove value from front.	O(1)
d.pop_back();	Remove value from end.	O(1)
d.erase(iterator);	Erase value indexed by iterator.	O(n)
d.erase(begin, end);	Erase the elements from begin to end.	O(n)
d.generate(begin,end)		O(n)

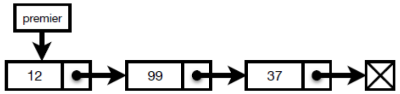
priority_queue<T, container<T> , comparison<T> > q;	Make an empty priority queue using the given container to hold values,and comparison to compare values. container defaults to vector<T> and comparison defaults to less<T>.	O(1)
q.top();	Return the "biggest" element.	O(1)
q.size();	Return current number of elements.	O(1)
q.empty();	Return true if priority queue is empty.	O(1)
q.push(value);	Add value to priority queue.	O(log n)
q.pop();	Remove biggest value.	O(log n)

vector<T> v;	Création d'un vector vide	O(1)
vector<T> v(n);	Make a vector with N elements.	O(n)
vector<T> v(n, value);	Make a vector with N elements, initialized to value.	O(n)
vector<T> v(begin, end);	Make a vector and copy the elements from begin to end.	O(n)
v[i];	Return (or set) the I'th element.	O(1)
v.at(i);	Return (or set) the I'th element, with bounds checking.	O(1)
v.size();	Return current number of elements.	O(1)
v.empty();	Return true if vector is empty.	O(1)
v.begin();	Return random access iterator to start.	O(1)
v.end();	Return random access iterator to end.	O(1)
v.front();	Return the first element.	O(1)
v.back();	Return the last element.	O(1)
v.capacity();	Return maximum number of elements.	O(1) amortized
v.push_front(value)	Ajoute un élément à l'avant	O(n)
v.pop_front	Ajout un élément à la fin	O(n)
v.push_back(value);	Add value to end. (Au pire O(n) -> réalloc)	O(1)
v.insert(iterator, value);	Insert value at the position indexed by iterator.	O(n)
v.pop_back();	Remove value from end. (Au pire O(n))	O(1)
v.erase(iterator);	Erase value indexed by iterator.	O(n)
v.erase(begin, end);	Erase the elements from begin to end.	O(n)
v.generate(b,e)	Assigns a generate value in the container	O(n)

Forward_list		
fl.splice_after()	Transfère les éléments de la fwlist (param) à la fwlist de dest après l'élément pointée par position	
Fw.push_front() Fw.pop_front()	Insérer au début Supprime l'élément au début	O(1)
Fw.insert(milieu) Fw.erase(milieu)	Insert un élément au milieu Supprimer l'élément au milieu (si élément connu)	O(1)
Mémoire additionelle		(n+1)*p

Listes

Simplement chainée : O(1), mémoire prend max 2x sa taille

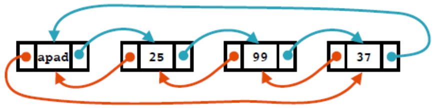


Insertion en fin / devant

```
Maillon<int>* premier      = new Maillon<int> {12, nullptr};
premier->suivant           = new Maillon<int> {99, nullptr};
```

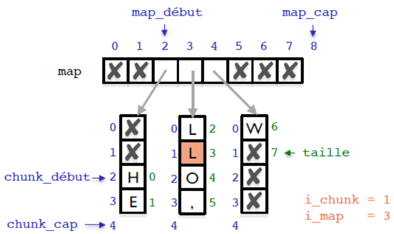
```
Maillon<int>* premier = new Maillon<int> {37, nullptr};
premier               = new Maillon<int> {99, premier};
```

Doublement chainée



Tableaux (Deque / Tas)

Double ended queue (deque)



Indices logiques en vert / physique en bleu

Supprimer extremité/insertion : **O(1)**

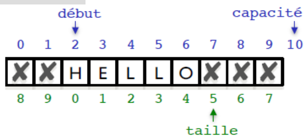
Pire cas (insertion) : pour n élém dans des chunks de taille c :

- Allouer new chunk : **O(c)**
- Réallouer la map : **O(n/c)**
- Complexité total : **O(c+n/c)**

Tableau taille fixe, Buffer circulaire :

Buffer circulaire est uniquement différent pour insertion/suppression au début : O(1) en moyenne au lieu de **O(N)**

```
méthode i_physique(i_logique)
  ip ← (début + i_logique) % capacité
  si ip >= 0, retourner ip
  sinon,      retourner ip + capacité
```



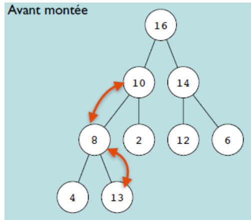
TAS, représenter comme ABR binaire

Condition de tas :

- Tout élément est plus grand/égale que ses enfants
- Tout élément est plus petit/égale que son parent

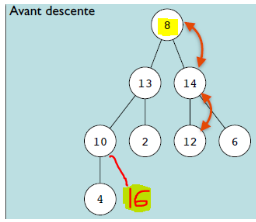
Insertion, place le new élément. À la fin puis le remonte pour qu'il corresponde aux conditions.

Remonter : **O(log(n))** selon la taille de l'arbre



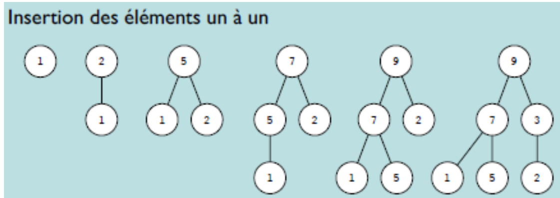
Suppression élément plus grand, on inverse la tête avec le dernier puis on remonte le plus grand des sous arbre jusqu'à respecter condition. Le dernier est supprimé.

Descente : **O(log(n))**



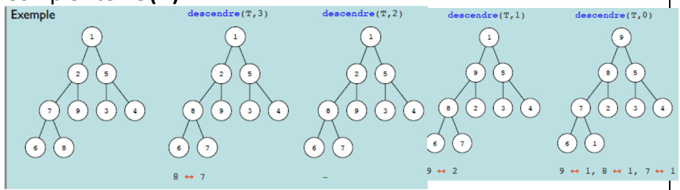
Création d'un tas (cas 1), similaire à une insertion. On ajoute à la fin et on rétablie la condition l'arbre. Ceci pour chaque élément

Complexité : **O(n * log(n))**



Création d'un tas (cas 2), insertion de tous les éléments et on descend uniquement ceux qui peuvent l'être. On fait ça pour chaque sous arbre.

Complexité : **O(N)**



Le tri par tas, est instable comme le tri rapide. Il a une meilleure complexité dans le pire des cas mais effectue plus d'échanges en moyenne => donc moins rapide en pratique.

Types de données abstraits

Pile (Stack)

Structure concrète	Sommet
tableau de taille variable	fin
tableau de capacité variable	fin
liste simplement chainée	début
liste doublement chainée	début ou fin
deque	début ou fin

File (Queue)

Structure concrète	Enfiler	Défiler
Buffer circulaire	début ou fin	fin ou début
Liste simple avec pointeur sur le dernier élément	fin	début
Liste doublement chainée	début ou fin	fin ou début
deque	début ou fin	fin ou début

File de priorité (priority Queue)

Structure linéaire associée à un critère de priorité permettant d'ordonner les éléments.

TAS, suite

Fonction	Description
std::push_heap	Ajoute un élément dans le tas.
std::pop_heap	Descend l'élément en tête à la fin et effectue une condition de tas. Attention l'élément n'est pas supprimé !!!!!!!!!
std::make_heap	Crée un tas à partir d'un tableau quelconque.
std::sort_heap	Trie les éléments à partir d'un tas. 1. Inverse premier/dernier 2. Effectue condition tas
std::is_heap	Vérifie si les éléments respectent la condition de tas.
std::is_heap_until	Trouve le premier élément ne respectant pas la condition de tas.

Notation polonaise inversée

pileValeur	pileOp	1 + ((2 + 3) * 5)
1		+ ((2 + 3) * 5)
1	+	((2 + 3) * 5)
1 2	+	+ 3) * 5)
1 2	+ +	3) * 5)
1 2 3	+ +	) * 5)
1 5	+	* 5)
1 5	+ *	5)
1 5 5	+ *	)
1 25	+	
26		

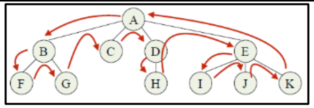
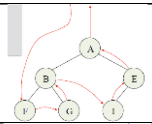
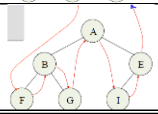
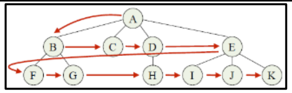
1 + ((2 + 3) * 5) = 1 2 3 + 5 * +

set< type > s;	Constructor par défaut / déplacement	O(1)
set< type > s(begin, end);	Copie depuis une séqu. Ou liste Si element à insérer est trié ou même. Sinon (et/ou différent)	O(n) O(n log n)
s.find(key)	Return an iterator pointing to an occurrence of key in s, or s.end() if key is not in s.	O(log n)
s.lower_bound(key)	Return an iterator pointing to the first occurrence of an item in s not less than key, or s.end() if no such item is found..(compar O(log n))	O(n)
s.upper_bound(key)	Return an iterator pointing to the first occurrence of an item greater than key in s, or s.end() if no such item is found.(compar O(log n))	O(n)
s.equal_range(key)	Returns pair<lower_bound(key), upper_bound(key)>.	O(log n)
s.count(key)	Returns the number of items equal to key in s.	O(log n)
s.size();	Return current number of elements.	O(1)
s.empty();	Return true if set is empty.	O(1)
s.begin()	Return an iterator pointing to the first element.	O(1)
s.end()	Return an iterator pointing one past the last element.	O(1)
s.insert(iterator, key)	Inserts key into s. iterator is taken as a "hint" but key will go in the correct position no matter what. Returns an iterator pointing to where keywent.	O(log n)
s.insert(key)	Insertion d'une donnée Si position est donnée	O(log n) O(1)
Operator++		Moy O(1) Pire O(log n)

Multiset<type> s		
multiset< type > s(begin, end);		O(n log(n))

map< key_type, value_type, key_compare > m;	Make an empty map. key_compare should be a binary predicate for ordering the keys. It's optional and will default to a function that uses operator<.	O(1)
map< key_type, value_type, key_compare > m(begin, end);	Make a map and copy the values from begin to end.	O(n log n)
m[key]	Return the value stored for key. This adds a default value if key not in map.	O(log n)
m.find(key)	Return an iterator pointing to a key-value pair, or m.end() if key is not in map.	O(log n)
m.lower_bound(key)	Return an iterator pointing to the first pair containing key, or m.end() if key is not in map.	O(log n)
m.upper_bound(key)	Return an iterator pointing one past the last pair containing key, or m.end() if key is not in map.	O(log n)
m.equal_range(key)	Return a pair containing the lower and upper bounds for key. This may be more efficient than calling those functions separately.	O(log n)
m.size();	Return current number of elements.	O(1)
m.empty();	Return true if map is empty.	O(1)
m.begin()	Return an iterator pointing to the first pair.	O(1)
m.end()	Return an iterator pointing one past the last pair.	O(1)
m[key] = value;	Store value under key in map.	O(log n)
m.insert(pair)	Inserts the <key, value> pair into the map. Equivalent to the above operation.	O(log n)

Vocabulaire
Set : objet unique et trié (ordre croissant sans répétition)
Multiset : objet unique et trié (ordre croissant avec répét.)
Map : set<clé, valeur>. On peut chercher le n^{éme} ou clé^{éme} valeur avec []. Tableau associatif utilisant des clés triées uniques associées à des valeurs
Multimap : Tableau associatif utilisant des clés triées non-uniques associées à des valeurs
Stack : push/pop = toujours au début
Queue : enqueue (au début)/dequeue(à la fin) => ordre décroissant des éléments
Ancêtre d'un nœud : atteignable en remontant l'arbre de fils en père
Arboriser : Le milieu devient la racine, puis le milieu de chaque demi-arbre et ainsi de suite
Arbre complet : arbre plein dont le dernier niveau est rempli depuis la gauche
Arbre dégénéré : est un arbre tel que le degré de chaque nœud interne vaut 1 ce qui le rend équivalent à une liste chaînée.
Arbre plein : degré de tout nœud interne = degré de l'arbre à l'exception (éventuelle) de l'avant dernier niveau
Chemin : suite de nœuds reliant la racine à un nœud
Degré : nombre de fils
Degré de l'arbre : on cherche le père ayant le plus d'enfants
Descendant : nœud atteignable en descendant de père en fils
Equilibre : Un arbre (binaire) de racine R est équilibré si pour chacun de ses sous-arbres : |hauteur (R.gauche) – hauteur (R.droit)| ≤ 1
Etiquette : (label) information associée au nœud
Feuille : nœud sans fils
Frères : nœuds ayant le même père
Hauteur : nbre de nœuds du chemin le plus long
Linéarisation : arbre du plus petit au plus grand
Niveau : nombre de nœuds sur le chemin du nœud (ce dernier compris)
Nœud interne : nœud (père) ayant des fils
Racine : le seul nœud sans père
Sous-arbre : sous ensemble de nœuds ayant un père et des fils en dessous de la racine
Taille d'un sous-arbre : compteur ajouté à chaque nœud qui addition le nombre d'éléments de chaque sous nœud + 1

Parcours	
Pré-ordonné (profondeur) Racine-gauche-droite	
Post-ordonné Gauche-droite-racine	
Symétrique Gauche-racine-droite	
Largeur (horizontal de gauche à droite)	

Parcours Arbre binaire	**Complexité**
Pré-ordonné/Post-ordonné/ Symétrique	On passe par chaque lien entre nœud à aller/retour. Au plus 3N liens d'ARB Moy/pire : **O(N)**
Recherche/insertion/sup. Min/max/rang/selection	O(h) = hauteur arbre Moy : **O(log(N)** Pire : **O(N)** ARB dégé
Elément suivant	Moy : **O(1)** Pire : **O(h),** h = hauteur
Tri par arbre (Tree sort)	**O(N*log(N))**
Linéariser/Arboriser	**O(N)**

Linéariser/Arboriser	**Complexité**
Réorganiser en Abr dégénéré (sans enfant gauche). Compte le nombre de ses éléments, effectue un parcours décroissant	**O(n)**
Arboriser l'arbre dégénéré. Réorganiser en arbre équilibré. Racine = médian Effectue un parcours croissant	**O(n)**

Chapitre 6 - Graphes non orientés

Chapitre 10 - Allocation dynamique

Emplace_back() = insertion direct avec le constructeur de déplacement

Push_back() création de valeurs temporaire puis déplacement (x déplacement et destruction)

méthode	Supporter	Complexité	constructeur	fonctionnement
At	Array,vect,deque	1	/	accède[]
push_front	Vecteur	N	D(args) cp/mv D	Insert(0,n)
push_front	(forward_)list deque	1	D(args) cp/mv D	push_front
Insert middle	Vecteur deque	N	D(args) cp/mv elems D	insert(n/2,n)
Insert middle	(forward_)list, si elem connu	1	D(arg) change pointeurs	
push_back	Vect, list, deque	1	D(args) cp/mv D	Si max, déplace tous les éléms puis add
emplace_back	Vect,list,deque	1	D(args)	push_back en place
Resize	vect	M ou m =nb elem add	D(args) jusqua taile	Créer des éléments tant que size pas atteint
splice	List, forward	1 – n (si range) modifie les pointeurs	/	Enleve l'élément/un groupe d'une liste et le met dans l'autre