ASD - Algorithmes et Structures de Données

Leonard Cseres | Juin 2024

Complexité

$\mathcal{O}(1) < \mathcal{O}(\log(n)) < \mathcal{O}(n) < \mathcal{O}(n \cdot \log(n)) < \mathcal{O}(n^c) < \mathcal{O}(2^n) < \mathcal{O}(n!) < \mathcal{O}(n^n)$

---

for $i = 1$ **to** n **step** $i \times \alpha$ **do** $\triangleright\ \mathcal{O}(\log_\alpha(n))$
  $k \leftarrow k + 1$
end for

---

for $i = 1$ **to** n **step** $i \times i$ **do** $\triangleright\ \mathcal{O}(\sqrt{n})$
  $k \leftarrow k + 1$
end for

---

for $i = 0$ **to** n **do** $\triangleright\ \mathcal{O}(n \cdot \frac{(n-1)}{2})$
  for $j = 0$ **to** i **do**
    $k \leftarrow k + 1$
  end for
end for

---

for $i = 0$ **to** n **and** m **do** $\triangleright\ \mathcal{O}(min(n, m))$
  $k \leftarrow k + 1$
end for

---

for $i = 0$ **to** n **or** m **do** $\triangleright\ \mathcal{O}(max(n, m))$
  $k \leftarrow k + 1$
end for

Cas spéciaux

Operation	Complexité
`rand()` $\%\ n == 0$	$P = \frac{1}{n}$
(`rand()` $\%\ n$) $\cdot$ (`rand()` $\%\ n$) $== n$	$P = \sqrt{n}$

Récursivité $C = \text{nb\_appels\_rec}^{\text{profondeur}}$

Algorithmes	Complexités
Factorielle récursif	$\mathcal{O}(n)$
Factorielle itératif	$\mathcal{O}(n)$
Fibonacci récursif	$\mathcal{O}(\phi^n),\ \phi = \frac{\sqrt{5}+1}{2}$
Fibonacci itératif	$\mathcal{O}(n)$
PGCD (Euclide)	$\mathcal{O}(\log(n))$
Tours de Hanoï récursif	$\mathcal{O}(2^n)$
Tours de Hanoï itératif	$\mathcal{O}(2^n)$
Permutations	$\mathcal{O}(n!)$
Tic Tac Toe	9!
Puissance 4 (Profondeur d'exploration de d tours)	$\mathcal{O}(7^d)$
Minimax (negamax) (M mouvements possibles par tour, profondeur de d tours)	$\mathcal{O}(m \cdot d)$

Algorithmes de Tri

Algorithme	Pire cas	Moyen	Meilleur	Stable	En place
Tri à bulles	$\mathcal{O}(n^2)$	$\Theta(n^2)$	$\Omega(n)$	Oui ($\leq$)	Oui
Tri par sélection	$\mathcal{O}(n^2)$	$\Theta(n^2)$	$\Omega(n^2)$	Non	Oui
Tri par insertion	$\mathcal{O}(n^2)$	$\Theta(n^2)$	$\Omega(n)$	Oui	Oui
Tri fusion	$\mathcal{O}(n\log(n))$	$\Theta(n\log(n))$	$\Omega(n\log(n))$	Oui	Non
Tri rapide	$\mathcal{O}(n^2)$	$\Theta(n\log(n))$	$\Omega(n\log(n))$	Non	Oui
Tri comptage	$\mathcal{O}(n+k)$	$\Theta(n+k)$	$\Omega(n+k)$	Oui	Non
Tri par base	$\mathcal{O}(n \cdot k)$	$\Theta(n \cdot k)$	$\Omega(n \cdot k)$	Oui	Non
Selection rapide	$\mathcal{O}(n^2)$	$\Theta(n)$	$\Omega(n)$	Non	Oui

`void qsort(start, count, size, cmp)`
$\Theta(n\log(n))$, $\mathcal{O}(n^2)$, pas stable (`<cstdlib>`)

`void sort(first, last, cmp)`
$\Theta(n\log(n))$, `swap` à implémenter

`void stable_sort(first, last, cmp)`
$\mathcal{O}(n\log(n))$, $\mathcal{O}(n\log^2(n))$ si fait en place et doit utiliser $<>$ pour la stabilité (tri fusion)

`void nth_element(first, nth, last)`
$\mathcal{O}(n)$, où $n =$ `last` $-$ `first`

`void partial_sort(first, middle, last)`
$\mathcal{O}(n\log(m))$, où $n =$ `last` $-$ `first` et $m =$ `middle` $-$ `first`

Structures linéaires

Sequence	
`array, vector`	
`deque`	Double-ended queue
`forward_list`	Liste chaînée simple
`list`	Liste chaînée double
Adaptative	
`stack`	Pile (LIFO)
`queue`	File (FIFO)
`priority_queue`	File de priorité
Associative	*Balanced Binary Tree*
`set`	Ensemble trié
`multiset`	Ensemble trié avec doublons
`map`	Table de hachage
`multimap`	Table de hachage avec doublons
Unordered associative	*Hash Table*
`unordered_set`	Ensemble non trié
`unordered_multiset`	Ensemble non trié avec doublons
`unordered_map`	Table de hachage non triée
`unordered_multimap`	Table de hachage non triée avec doublons

Buffer Circulaire (+) $\mathcal{O}(1)$ pour ajouter, supprimer et accéder, (−) $\mathcal{O}(\min(i, N-i))$ pour insérer ou supprimer à l'index i, (−) capacité fixe

Deque

Opération	Complexité
Allouer un nouveau block	$\mathcal{O}(B)$
Ré-allouer la map	$\mathcal{O}(n/B)$
Insertion	$\mathcal{O}(B + n/B)$
Insertion/suppression à l'indice i	$\mathcal{O}(\min(i, n-i))$

	`array`	`vector`	`forward_list`	`list`	`deque`
Mémoire extra	0	$3p + \mathcal{O}(n) \cdot t$	$p(n+1)$	$p(2n+3)$	$\mathcal{O}(n/B) \cdot p + \mathcal{O}(B) \cdot t + 6p$
operator[]	$\mathcal{O}(1)$	$\mathcal{O}(1)$	N/A	N/A	$\mathcal{O}(1)^2$
push/pop front	N/A	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
insert/erase milieu	N/A	$\mathcal{O}(n)$	$\mathcal{O}(1)^1$	$\mathcal{O}(1)^1$	$\mathcal{O}(n)$
push/pop back	N/A	$\Omega(1)\ \mathcal{O}(n)$	N/A	$\mathcal{O}(1)$	$\mathcal{O}(1)$

$t =$ `sizeof(T)`, $p =$ `sizeof(T*)`, $B =$ `block size`
[1]si l'élément est connu [2]un peu plus lent

Tas $\forall i > 0, T[\text{parent}(i)] \geq T[i]$

Insertion

1. Insérer à la fin
2. Remonter l'élément

Suppression

1. Permuter premier et dernier élément
2. Supprimer le dernier
3. Descendre l'élément permuté

Notation polonaise inversée (RPN) Placer opérateurs après les opérandes et évaluer avec une pile de gauche à droite

---

Arbres

Hauteur $H(n) = 1 + \max(H(n_g), H(n_d))$

Degrés $d = \max(n_g, n_d)$ (nbr. d'enfants pour un noeud donné)

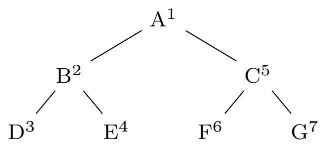
Plein (1) les noeuds de niveau $< h - 1$ sont de degré d, (2) les noeuds de niveau $h-1$ sont de degré quelconque, (3) les feuilles sont de niveau h

Complet Dernier niveau rempli par la gauche

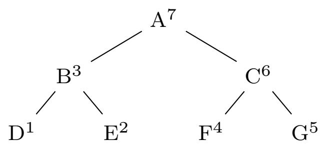
Binaire Arbre de degré ≤ 2

Parcours

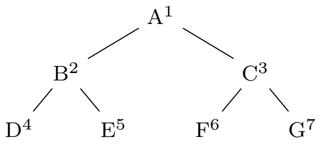
Pré-ordre A, B, D, E, C, F, G



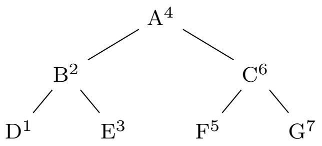
Post-ordre D, E, B, F, G, C, A



Largeur A, B, C, D, E, F, G



Symétrique (Arbres Binaires) D, B, E, A, F, C, G



Arbres binaires de recherche

$n_g \leq n < n_d$, de complexité $\mathcal{O}(\log_2(n))$ si équilibré

Suppression

- Feuille: supprimer
- Degré 1: remplacer par enfant
- Degré 2: choisir un des noeuds descendant comme racine du sous-arbre à raccrocher (1) Minimum du sous-arbre droit, (2) Maximum du sous-arbre gauche

Taille Nombre de noeuds dans l'arbre

Rang d'une clé Nombre de clés plus petites que la clé donnée

Équilibre $-1 \leq \text{hauteur}(n_g) - \text{hauteur}(n_d) \leq 1$

Tables de hachage

Fonction de hachage $h(k) = k \mod M$ avec M premier et éloigné de puissances de 2

Adressage MAD $h(k) = (a \cdot k + b) \mod M$ avec $a, b \in \mathbb{N}$ et non multiples de M

Résolution de collisions

Chainage Liste chaînée à chaque case $M \approx \frac{N}{4}$ avec M listes chainées et N paires clé-valeur ($M < N$)

- Doubler M quand $\frac{N}{M} \geq 8$
- Diviser M par 2 quand $\frac{N}{M} \leq 2$

Insertion Insertion au début de la liste chaînée

Adressage ouvert

$M \approx 2N$ ($M > N$)

Sondage Linéaire Chercher la clé k. Si occupé, placer à la case $h(k) + i \mod M$ avec $i = 1, 2, 3, \ldots$

- Doubler M quand $\frac{N}{M} \geq \frac{1}{2}$
- Diviser M par 2 quand $\frac{N}{M} \leq \frac{1}{8}$

Suppression Ré-insérer les éléments suivants à la case $h(k)$

Comparaisons

	Insérer	Rechercher	Supprimer
Tableau trié	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$
Tableau non trié	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Liste triée	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Liste non triée	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Arbre	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$
Table de hachage	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Graphes

Simple Pas de boucles, pas de multi-arêtes
Degré Nombre de sommets adjacents
Circuit Chemin fermé

Matrice d'adjacence

$M_{ij} = \deg(v_i, v_j)$

Complexité $\mathcal{O}(n^2)$ coût mémoire, $\mathcal{O}(1)$ pour accéder à une arête, $\mathcal{O}(n)$ pour parcourir les voisins, $\mathcal{O}(n^2)$ pour parcourir tous les arêtes

Liste d'adjacence

Liste des voisins

- **Non orienté** $\text{Adj}[u] = \{v \mid (u, v) \in E\}$
- **Orienté** $\text{Succ}[u] = \{v \mid (u, v) \in E\}$
- **Pondéré** $\text{Adj}[u] = \{(v, w) \mid (u, v, w) \in E\}$

Complexité $\mathcal{O}(n + m)$ coût mémoire, $\mathcal{O}(\deg(u))$ pour accéder à une arête, $\mathcal{O}(\deg(u))$ pour parcourir les voisins, $\mathcal{O}(n + m)$ pour parcourir tous les arêtes ($\mathcal{O}(\deg(u)) \approx \mathcal{O}(\frac{m}{n})$)

DFS (pré) En profondeur, récursif

BFS (post) En largeur, file FIFO

Complexité parcours $\mathcal{O}(n^2)$ pour matrice, $\mathcal{O}(n + m)$ pour liste

Tri topologique Inverse du post-ordre d'un DAG (ne fonctionne pas sur un graphe cyclique)

CC (Composantes Connexes) Est-ce que u et v sont connectés?

CFC (Composantes Fortement Connexes) Est-ce que u et v sont connectés dans les deux sens?

1. Calculer post-ordre inverse de l'inverse de G
2. Calculer les CFC par DFS sur G dans cet ordre

Usage Reference

Containers

`<>.insert` Inserts an element before the specified position
`<>.splice` Transfers element(s) from one list to another
`<>.splice_after` Transfers element(s) after the specified position

Algorithms

`lower_bound` (Sorted input) Returns an iterator pointing to first element **not** $<$ value. Return `end()` if no such element
`upper_bound` (Sorted input) Returns an iterator pointing to first element $>$ value. Return `end()` if no such element
`nth_element` Rearranges the elements in the range [first, last) such that the element at the `nth` position is the element that would be in that position in a sorted sequence

`partition` Rearranges the elements in the range [first, last) such that `pred` is **True** for elements before the partition and **False** for elements after. Returns an iterator to the first element of the second group
`partition_point` Returns an iterator to the first element in the range [first, last) for which `pred` is **False**

`partial_sort` Sort until `middle` in the range [first, last)
`adjacent_find` Returns an iterator to the first element in the range [first, last) that is equal to the next element

`push_heap` (Is already heap) Inserts `last−1` element into the correct position of the heap
`pop_heap` (Is already heap) Swaps the `first` and `last−1` element and rearranges the heap

Iterators

<table>
<tr><td colspan="4">All</td><td>incrementation
copy-constructible/assignable
destructible</td></tr>
<tr><td rowspan="5">RA</td><td rowspan="4">Bi</td><td rowspan="3">Fw</td><td>In</td><td>==, !=
dereferenced as an rvalue</td></tr>
<tr><td>Out</td><td>dereferenced as an lvalue</td></tr>
<tr><td></td><td>default-constructible
multi-pass[1]</td></tr>
<tr><td colspan="2"></td><td>decrementation</td></tr>
<tr><td colspan="3"></td><td>+, -, +=, -=, <, >, ≤, ≥, []</td></tr>
</table>

[1] neither dereferencing nor incrementing affects dereferenceability

`advance`, `distance`, `next`, `prev` COMPLEXITY $\mathcal{O}(1)$ for `RA`, otherwise $\mathcal{O}(n)$

Permuter les n premiers caractères

```
function Permuter(S, n)                    ▷ e = n · n!
    if n = 1 then
        traiter(S)
    else
        for i = 1 to n do
            Permuter(S, n − 1)
            swap(S[i], S[n])
        end for
    end if
end function
```

. . . .

```
function Permuter(S, n)                    ▷ e = 3.43 · n!
    if n = 1 then
        traiter(S)
    else
        for i = 1 to n do
            swap(S[i], S[n])
            Permuter(S, n − 1)
            swap(S[i], S[n])
        end for
    end if
end function
```

. . . .

```
function Permuter(S, n)                    ▷ e = 2(n! − 1)
    if n = 1 then
        traiter(S)
    else
        for i = 1 to n − 1 do
            swap(S[i], S[n])
            Permuter(S, n − 1)
            swap(S[i], S[n])
        end for
        Permuter(S, n − 1)
    end if
end function
```

. . . .

```
function Permuter(S, n)                    ▷ e = n! − 1
    if n = 1 then
        traiter(S)
    else
        Permuter(S, n − 1)
        for i = 1 to n − 1 do
            if n is even then
                swap(S[i], S[n])
            else
                swap(S[1], S[n])
            end if
            Permuter(S, n − 1)
        end for
    end if
end function
```

Tris

```
function BubbleSort(A, n)
    for i = 1 to n − 1 do
        for j = 1 to n − i do
            if A[j + 1] < A[j] then
                swap(A[j], A[j + 1])
            end if
        end for
    end for
end function
```

. . . .

```
function SelectionSort(A, n)
    for i = 1 to n − 1 do
        imin ← i
        for j = i + 1 to n do
            if A[j] < A[imin] then
                imin ← j
            end if
        end for
        swap(A[i], A[imin])
    end for
end function
```

. . . .

```
function InsertionSort(A, n)
    for i = 2 to n do
        tmp ← A[i]
        j ← i
        while j − 1 ≥ 1 and A[j − 1] > tmp
do
            A[j] ← A[j − 1]
            j ← j − 1
        end while
        A[j] ← tmp
    end for
end function
```

. . . .

```
function Fusionner(A, p, q, r)
    L ← copy of subarray A[p . . . q]
    R ← copy of subarray A[q + 1 . . . r]
    insert a sentinel value ∞ at the end of L
and R
    i ← 1, j ← 1
    for k = p to r do
        if L[i] ≤ R[j] then
            A[k] ← L[i]
            i ← i + 1
        else
            A[k] ← R[j]
            j ← j + 1
        end if
    end for
end function
```

```
function TriFusion(A, lo, hi)
    if hi ≤ lo then
        return
    end if
    mid ← lo + (hi − lo)/2

    TriFusion(A, lo, mid)
    TriFusion(A, mid + 1, hi)
    Fusionner(A, lo, mid, hi)
end function
```

. . . .

```
function Partition(A, lo, hi)
    i ← lo − 1, j ← hi
    while True do
        repeat
            i ← i + 1
        until A[i] ≥ A[hi]
        repeat
            j ← j − 1
        until j ≤ lo or A[hi] ≥ A[j]
        if i ≥ j then
            break
        end if
        swap(A[i], A[j])
    end while
    swap(A[i], A[hi])
    return i
end function
```

. . . .

```
function TriRapide(A, lo, hi)
    if lo < hi then
        p ← choose pivot element
        swap(A[hi], A[p])
        i ← Partition(A, lo, hi)

        QuickSort(A, lo, i − 1)
        QuickSort(A, i + 1, hi)
    end if
end function
```

. . . .

```
function QuickSort(A, lo, hi)
    while lo < hi do
        p ← choose pivot element
        swap(A[hi], A[p])
        i ← Partition(A, lo, hi)

        if i − lo < hi − i then
            QuickSort(A, lo, i − 1)
            lo ← i + 1
        else
            QuickSort(A, i + 1, hi)
            hi ← i − 1
        end if
    end while
end function
```

```
function SelectionRapide(A, n, k)
    lo ← 1
    hi ← n

    while hi > lo do
        i ← Partition(A, lo, hi)
        if i < k then
            lo ← i + 1
        else if i > k then
            hi ← i − 1
        else                                   ▷ i = k
            return A[k]
        end if
    end while

    return A[k]
end function
```

. . . .

```
function TriComptage(A, n, b, key)
    C ← array of b counters initialized to zero
    for each e in A do
        C[key(e)] ← C[key(e)] + 1
    end for

    idx ← 1
    for i = 1 to b do
        tmp ← C[i]
        C[i] ← idx
        idx ← idx + tmp
    end for

    B ← array of the same size as A
    for each e in A do
        B[C[key(e)]] ← move e
        C[key(e)] ← C[key(e)] + 1
    end for
    return B
end function
```

. . . .

```
function TriParBase(T, d)
    for i = d to 1 do
        Sort T with a stable sort according to
        the i-th digit
    end for
end function
```

Tas (Heap)

```
function Parent(i) return (i − 1)/2
end function
```

```
function E1(i) return 2 · i + 1                 ▷ Enfant gauche
end function
```

```
function E2(i) return 2 · i + 2                 ▷ Enfant droit
end function
```

```
function Remonter(T, i)
    while i > 0 and T[Parent(i)] < T[i] do
        swap(T[i], T[Parent(i)])
        i ← Parent(i)
    end while
end function
```

```
function PGE(T, i, k = taille(T))               ▷ Plus grand enfant
    if E2(i) < k and T[E2(i)] > T[E1(i)] then
        return E2(i)
    end if
end function
```

```
function Descendre(T, i, k = taille(T))
    while E1(i) < k and T[PGE(T, i, k)] > T[i] do
        swap(T[PGE(T, i, k)], T[i])
        i ← PGE(T, i, k)
    end while
end function
```

```
function CreerTas(T)                            ▷ O(n log(n))
    for i = 1 to taille(T) − 1 do
        Remonter(T, i)
    end for
end function
```

```
function CreerTas(T)                            ▷ O(n)
    p ← Parent(taille(T) − 1)
    for i = p to 0 do
        Descendre(T, i)
    end for
end function
```

```
function TriParTas(T)                           ▷ O(n log(n))
    N ← taille(T)
    CreerTas(T)
    for k = N − 1 to 1 do
        swap(T[k], T[0])
        Descendre(T, 0, k)
    end for
end function
```

Arbres Binaires

```
function Hauteur(r)
    if r = ∅ then return 0
    else
        return 1+max(Hauteur(r.gauche), Hauteur(r.droit))
    end if
end function
```

```
function Equilibre(r)
    if r = ∅ then return 0
    else
        return Hauteur(r.gauche) − Hauteur(r.droit)
    end if
end function
```

```
function Lineariser(r, ref out, ref n)          ▷ n compteur noeuds
    if r ≠ ∅ then
        Lineariser(r.droit, out, n)
        r.droit ← out, out ← r, n ← n + 1
        Lineariser(r.gauche, out, n)
        r.gauche ← ∅
    end if
end function
```

```
function Arboriser(ref out, n)
    if n ≠ 0 then
        rg ← Arboriser(out, (n − 1)/2)
        r ← out, out.gauche ← rg, out ← out.droit
        r.droit ← Arboriser(out, n/2)
        return r
    end if
end function
```

```
function Equilibrer(r)                          ▷ O(n)
    out ← ∅, n ← 0
    Lineariser(r, out, n)
    return Arboriser(out, n)
end function
```

```
function RotGauche(ref r)
    tmp ← r.droit
    r.droit ← tmp.gauche
    tmp.gauche ← r
    r ← tmp
end function
```

```
function RotDroite(ref r)
    tmp ← r.gauche
    r.gauche ← tmp.droit
    tmp.droit ← r
    r ← tmp
end function
```

```
function RetablirEquilibre(ref r)
    if r = ∅ then return
    end if
    if Equilibre(r) < −1 then
        if Equilibre(r.droit) > 0 then
            RotDroite(r.droit)
        end if
        RotGauche(r)
    else if Equilibre(r) > 1 then
        if Equilibre(r.gauche) < 0 then
            RotGauche(r.gauche)
        end if
        RotDroite(r)
    else
        CalculerHauteur(r)      ▷ Mettre à jour attrib. hauteur
    end if
end function
```

```
function Inserer(ref r, k)
    if r = ∅ then
        r ← new Noeud(k)
    else if k = r.cle then
        return                                  ▷ Déjà présent
    else
        if k < r.cle then
            Inserer(r.gauche, k)
        else
            Inserer(r.droit, k)
        end if
        RetablirEquilibre(r)
    end if
end function
```

```
function Supprimer(ref r, k)
    if r = ∅ then
        return
    end if
    if k = r.cle then
        Suppression de r avec 3 cas
    else
        if k < r.cle then
            Supprimer(r.gauche, k)
        else
            Supprimer(r.droit, k)
        end if
    end if
    RetablirEquilibre(r)
end function
```

Graphes

```
Require: Les sommets sont marqués comme non visités
  function PROFONDEUR(sommet v, Fn pre, (opt.) Fn post)
    pre(v)                                  ▷ en pré-ordre
    v ← visité
    for tout w adjacent à v do
      if w n'est pas visité then
        PROFONDEUR(w, pre, post)
      end if
    end for
    post(v)                                 ▷ en post-ordre
  end function
```

```
  function PROFONDEURG(Graphe G, Fn action)
    Marquer tous les sommets comme non visités
    for tout sommet v de G do
      if v n'est pas visité then
        PROFONDEUR(v, action)
      end if
    end for
  end function
```

```
  function LARGEUR(sommet v, Fn action)
    Initialiser une file Q vide
    Q.push(v)
    v ← visité
    while Q n'est pas vide do
      v ← Q.pop()
      action(v)
      for tout w adjacent à v do
        if w n'est pas visité then
          Q.push(w)
          w ← visité
        end if
      end for
    end while
  end function
```

```
  function PARENTSENLARGEUR(sommet v)
    Initialiser un tableau Parents à −1
    Initialiser une file Q vide
    Q.push(v)
    Parents[v] ← v                          ▷ sommet d'origine
    while Q n'est pas vide do
      v ← Q.pop()
      for tout w adjacent à v do
        if Parents[w] = −1 then             ▷ w non marqué
          Q.push(w)
          Parents[w] ← v
        end if
      end for
    end while
    return Parents
  end function
```

```
function CHAINE(Parents P, sommet w)
  Initialiser chaine vide
  if P[w] = −1 then
    return chaine vide
  else
    while P[w] ≠ w do
      ajouter w à la chaine
      w ← P[w]
    end while
  end if
  return chaine
end function
```

```
function PARENTSENLARGEUR(Sommets S)
  Initialiser tableau Parents à −1
  Initialiser file Q vide
  for tout sommet v de S do
    Q.push(v)
    Parents[v] ← v
  end for
  while Q n'est pas vide do
    v ← Q.pop()
    for tout w adjacent à v do
      if Parents[w] = −1 then
        Q.push(w)
        Parents[w] ← v
      end if
    end for
  end while
  return Parents
end function
```

```
function DIJKSTRA(graphe G(V, E), sommet v0)
  Q ← PriorityQueue()
  distTo[|V|], edgeTo[|V|]
  distTo[v0] ← 0, Q.push(v0, 0)

  for v ∈ G do
    if v ≠ v0 then
      distTo[v] ← ∞, edgeTo[v] ← NULL
    end if
  end for
  while Q n'est pas vide do
    v ← Q.top(), Q.pop()
    for e : v → w ∈ adj(v) do
      d ← distTo[v] + weight(e)
      if d < distTo[w] then
        distTo[w] ← d, edgeTo[w] ← e
        Q.add_or_modify(w, −distTo[w])
      end if
    end for
  end while
  return distTo[], edgeTo[]
end function
```

```
function COMPOSANTESCONNEXES(Graphe G)
  id ← 0
  Initialiser tableau CC à −1
  for tout sommet v de G do
    if CC[v] = −1 then
      PROFONDEUR(v, fn(v) { CC[v] ← id; })
      id ← id + 1
    end if
  end for
  return CC[]
end function
```

```
function DETECTIONCYCLE(sommet v)
  static foundCycle ← False
  Initialiser tableau visited à True
  Initialiser tableau parents à True

  for tout sommet w ∈ adj(v) do
    if foundCycle then
      return
    else if not visited[w] then
      DETECTIONCYCLE(w)
    else if parents[w] then
      foundCycle ← True
    end if
  end for
  parents[v] ← False
end function
```

Reference

$\mathcal{O}(1)$
$\mathcal{O}(\log(n))$
$\mathcal{O}(n)$
$\mathcal{O}(n\log(n))$

1 Array

1.01 `ref at(size_t n); const_ref at(size_t n) const;` complexity Constant.

1.02 `ref back(); const_ref back()const;` complexity Constant.

1.03 `iter begin()noexcept; const_iter begin() const noexcept;` complexity Constant.

1.04 `const_iter cbegin()const noexcept;` complexity Constant.

1.05 `const_iter cend()const noexcept;` complexity Constant.

1.06 `const_reverse_iter crbegin()const noexcept;` complexity Constant.

1.07 `const_reverse_iter crend()const noexcept;` complexity Constant.

1.08 `value_t* data()noexcept; const value_t* data ()const noexcept;` complexity Constant.

1.09 `constexpr bool empty()noexcept;` complexity Constant.

1.10 `iter end()noexcept; const_iter end()const noexcept;` complexity Constant.

1.11 `void fill(const value_t& val);` complexity Linear: Performs as many assignment operations as the size of the array object.

1.12 `ref front(); const_ref front()const;` complexity Constant.

1.13 `constexpr size_t max_size()noexcept;` complexity Constant.

1.14 `ref operator[](size_t n); const_ref operator [](size_t n)const;` complexity Constant.

1.15 `reverse_iter rbegin()noexcept; const_reverse_iter rbegin()const noexcept;` complexity Constant.

1.16 `reverse_iter rend()noexcept; const_reverse_iter rend()const noexcept;` complexity Constant.

1.17 `constexpr size_t size()noexcept;` complexity Constant.

1.18 `void swap(array& x)noexcept(noexcept(swap( declval<value_t&>(),declval<value_t&>())));` complexity Linear: Performs as many individual swap operations as the size of the arrays. validity The validity of all iterators, references and pointers is not changed: They remain associated with the same positions in the same container they were associated before the call, but the elements they still refer to have the swapped values.

2 Vector

2.01 `void assign(init_list<value_t> il);` complexity Linear on initial and final sizes (destructions, constructions).Additionally, in the range version (1), if InputIterator is not at least of a forward iterator category (i.e., it is just an input iterator) the new capacity cannot be determined beforehand and the operation incurs in additional logarithmic complexity in the new size (reallocations while growing). validity All iterators, pointers and references related to this container are invalidated.

2.02 `ref at(size_t n); const_ref at(size_t n) const;` complexity Constant.

2.03 `ref back(); const_ref back()const;` complexity Constant.

2.04 `iter begin()noexcept; const_iter begin() const noexcept;` complexity Constant.

2.05 `size_t capacity()const noexcept;` complexity Constant.

2.06 `const_iter cbegin()const noexcept;` complexity Constant.

2.07 `const_iter cend()const noexcept;` complexity Constant.

2.08 `void clear()noexcept;` complexity Linear in size (destructions).This may be optimized to constant complexity for trivially-destructible types (such as scalar or PODs), where elements need not be destroyed. validity All iterators, pointers and references related to this container are invalidated.

2.09 `const_reverse_iter crbegin()const noexcept;` complexity Constant.

2.10 `const_reverse_iter crend()const noexcept;` complexity Constant.

2.11 `value_t* data()noexcept; const value_t* data ()const noexcept;` complexity Constant.

2.12 `template <class... Args>iter emplace( const_iter pos, Args&&... args);` complexity Linear on the number of elements after position (moving).If a reallocation happens, the reallocation is itself up to linear in the entire size. validity If a reallocation happens, all iterators, pointers and references related to this container are invalidated.Otherwise, only those pointing to position and beyond are invalidated, with all iterators, pointers and references to elements before position guaranteed to keep referring to the same elements they were referring to before the call.

2.13 `template <class... Args> void emplace_back( Args&&... args);` complexity Constant (amortized time, reallocation may happen).If a reallocation happens, the reallocation is itself up to linear in the entire size. validity If a reallocation happens, all iterators, pointers and references related to this container are invalidated.Otherwise, only the end iterator is invalidated, and all other iterators, pointers and references to elements are guaranteed to keep referring to the same elements they were referring to before the call.

2.14 `bool empty()const noexcept;` complexity Constant.

2.15 `iter end()noexcept; const_iter end()const noexcept;` complexity Constant.

2.16 `iter erase(const_iter pos); iter erase( const_iter first, const_iter last);` complexity Linear on the number of elements erased (destructions) plus the number of elements after the last element deleted (moving). validity Iterators, pointers and references pointing to position (or first) and beyond are invalidated, with all iterators, pointers and references to elements before position (or first) are guaranteed to keep referring to the same elements they were referring to before the call.

2.17 `ref front(); const_ref front()const;` complexity Constant.

2.18 `allocator_t get_allocator()const noexcept;` complexity Constant.

2.19 `iter insert(const_iter pos, init_list< value_t> il);` complexity Linear on the number of elements inserted (copy/move construction) plus the number of elements after position (moving).Additionally, if InputIterator in the range insert (3) is not at least of a forward iterator category (i.e., just an input iterator) the new capacity cannot be determined beforehand and the insertion incurs in additional logarithmic complexity in size (reallocations).If a reallocation happens, the reallocation is itself up to linear in the entire size at the moment of the reallocation. validity If a reallocation happens, all iterators, pointers and references related to the container are invalidated.Otherwise, only those pointing to position and beyond are invalidated, with all iterators, pointers and references to elements before position guaranteed to keep referring to the same elements they were referring to before the call.

2.20 `size_t max_size()const noexcept;` complexity Constant.

2.21 `ref operator[](size_t n); const_ref operator [](size_t n)const;` complexity Constant.

2.22 `vector& operator=(init_list<value_t> il);` complexity Linear in size. validity All iterators, references and pointers related to this container before the call are invalidated.In the move assignment, iterators, pointers and references referring to elements in x are also invalidated.

2.23 `void pop_back();` complexity Constant. validity The end iterator and any iterator, pointer and reference referring to the removed element are invalidated.Iterators, pointers and references referring to other elements that have not been removed are guaranteed to keep referring to the same elements they were referring to before the call.

2.24 `void push_back(const value_t& val); void push_back(value_t&& val);` complexity Constant (amortized time, reallocation may happen).If a reallocation happens, the reallocation is itself up to linear in the entire size. validity If a reallocation happens, all iterators, pointers and references related to the container are invalidated.Otherwise, only the end iterator is invalidated, and all iterators, pointers and references to elements are guaranteed to keep referring to the same elements they were referring to before the call.

2.25 `reverse_iter rbegin()noexcept; const_reverse_iter rbegin()const noexcept;` complexity Constant.

2.26 `reverse_iter rend()noexcept; const_reverse_iter rend()const noexcept;` complexity Constant.

2.27 `void reserve(size_t n);` complexity If a reallocation happens, linear in vector size at most. validity If a reallocation happens, all iterators, pointers and references related to the container are invalidated.Otherwise, they all keep referring to the same elements they were referring to before the call.

2.28 `void resize(size_t n); void resize(size_t n, const value_t& val);` complexity Linear on the number of elements inserted/erased (constructions/destructions).If a reallocation happens, the reallocation is itself up to linear in the entire vector size. validity In case the container shrinks, all iterators, pointers and references to elements that have not been removed remain valid after the resize and refer to the same elements they were referring to before the call.If the container expands, the end iterator is invalidated and, if it has to reallocate storage, all iterators, pointers and references related to this container are also invalidated.

2.29 `void shrink_to_fit();` complexity At most, linear in container size. validity If a reallocation happens, all iterators, pointers and references related to the container are invalidated.Otherwise, no changes.

2.30 `size_t size()const noexcept;` complexity Constant.

2.31 `void swap(vector& x);` complexity Constant. validity All iterators, pointers and references referring to elements in both containers remain valid, and are now referring to the same elements they referred to before the call, but in the other container, where they now iterate.Note that the end iterators do not refer to elements and may be invalidated.

3 List

3.01 `void assign(init_list<value_t> il);` complexity Linear in initial and final sizes (destructions, constructions). validity All iterators, references and pointers related to this container are invalidated, except the end iterators.

3.02 `ref back(); const_ref back()const;` complexity Constant.

3.03 `iter begin()noexcept; const_iter begin() const noexcept;` complexity Constant.

3.04 `const_iter cbegin()const noexcept;` complexity Constant.

3.05 `const_iter cend()const noexcept;` complexity Constant.

3.06 `void clear()noexcept;` complexity Linear in list::size (destructions). validity All iterators, references and pointers related to this container are invalidated, except the end iterators.

3.07 `const_reverse_iter crbegin()const noexcept;` complexity Constant.

3.08 `const_reverse_iter crend()const noexcept;` complexity Constant.

3.09 `template <class... Args> iter emplace( const_iter pos, Args&&... args);` complexity Constant.

3.10 `template <class... Args> void emplace_back( Args&&... args);` complexity Constant.

3.11 `template <class... Args> void emplace_front(Args&&... args);` COMPLEXITY Constant. VALIDITY No changes.Member begin returns a different iterator value.

3.12 `bool empty()const noexcept;` COMPLEXITY Constant.

3.13 `iter end()noexcept; const_iter end()const noexcept;` COMPLEXITY Constant.

3.14 `iter erase(const_iter pos); iter erase(const_iter first, const_iter last);` COMPLEXITY Linear in the number of elements erased (destructions). VALIDITY Iterators, pointers and references referring to elements removed by the function are invalidated.All other iterators, pointers and references keep their validity.

3.15 `ref front(); const_ref front()const;` COMPLEXITY Constant.

3.16 `allocator_t get_allocator()const noexcept;` COMPLEXITY Constant.

3.17 `iter insert(const_iter pos, init_list<value_t> il);` COMPLEXITY Linear in the number of elements inserted (copy/move construction).

3.18 `size_t max_size()const noexcept;` COMPLEXITY Up to linear.Constant.

3.19 `template <class Compare> void merge(list& x, Compare comp); template <class Compare> void merge(list&& x, Compare comp);` COMPLEXITY At most, linear in the sum of both container sizes minus one (comparisons). VALIDITY No changes on the iterators, pointers and references related to the container before the call.The iterators, pointers and references that referred to transferred elements keep referring to those same elements, but iterators now iterate into the container the elements have been transferred to.

3.20 `list& operator=(init_list<value_t> il);` COMPLEXITY Linear in size. VALIDITY All iterators, references and pointers related to this container are invalidated, except the end iterators.In the move assignment, iterators, pointers and references referring to elements in x are also invalidated.

3.21 `void pop_back();` COMPLEXITY Constant. VALIDITY Iterators, pointers and references referring to element removed by the function are invalidated.All other iterators, pointers and reference keep their validity.

3.22 `void pop_front();` COMPLEXITY Constant. VALIDITY Iterators, pointers and references referring to the element removed by the function are invalidated.All other iterators, pointers and reference keep their validity.

3.23 `void push_back(const value_t& val); void push_back(value_t&& val);` COMPLEXITY Constant.

3.24 `void push_front(const value_t& val); void push_front(value_t&& val);` COMPLEXITY Constant.

3.25 `reverse_iter rbegin()noexcept; const_reverse_iter rbegin()const noexcept;` COMPLEXITY Constant.

3.26 `void remove(const value_t& val);` COMPLEXITY Linear in container size (comparisons). VALIDITY Iterators, pointers and references referring to elements removed by the function are invalidated.All other iterators, pointers and reference keep their validity.

3.27 `template <class Pred> void remove_if(Pred pred);` COMPLEXITY Linear in list size (applications of pred). VALIDITY Iterators, pointers and references referring to elements removed by the function are invalidated.All other iterators, pointers and reference keep their validity.

3.28 `reverse_iter rend()nothrow; const_reverse_iter rend()const nothrow;` COMPLEXITY Constant.

3.29 `void resize(size_t n); void resize(size_t n, const value_t& val);` COMPLEXITY If the container grows, linear in the number number of elements inserted (constructor).If the container shrinks, linear in the number of elements erased (destructions), plus up to linear in the size (iterator advance). VALIDITY Iterators, pointers and references referring to elements removed by the function are invalidated.All other iterators, pointers and references keep their validity.

3.30 `void reverse()noexcept;` COMPLEXITY Linear in list size.

3.31 `size_t size()const noexcept;` COMPLEXITY Up to linear.Constant.

3.32 `template <class Compare> void sort(Compare comp);` COMPLEXITY Approximately NlogN where N is the container size.

3.33 `void splice(const_iter pos, list& x, const_iter first, const_iter last); void splice(const_iter pos, list&& x, const_iter first, const_iter last);` COMPLEXITY Constant for (1) and (2).Up to linear in the number of elements transferred for (3). VALIDITY No changes on the iterators, pointers and references related to the container before the call.The iterators, pointers and references that referred to transferred elements keep referring to those same elements, but iterators now iterate into the container the elements have been transferred to.

3.34 `void swap(list& x);` COMPLEXITY Constant. VALIDITY All iterators, pointers and references referring to elements in both containers remain valid, but now are referring to elements in the other container, and iterate in it.Note that the end iterators do not refer to elements and may be invalidated.

3.35 `template <class BinaryPred> void unique(BinaryPred binary_pred);` COMPLEXITY Linear in container size minus one. VALIDITY Iterators, pointers and references referring to elements removed by the function are invalidated.All other iterators, pointers and references keep their validity.

4 Forward List

4.01 `void assign(init_list<value_t> il);` COMPLEXITY Linear in initial and final container sizes (destructions, constructions). VALIDITY All iterators, references and pointers related to this container are invalidated, except the end iterators.

4.02 `iter before_begin()noexcept; const_iter before_begin()const noexcept;` COMPLEXITY Constant.

4.03 `iter begin()noexcept; const_iter begin() const noexcept;` COMPLEXITY Constant.

4.04 `const_iter cbefore_begin()const noexcept;` COMPLEXITY Constant.

4.05 `const_iter cbegin()const noexcept;` COMPLEXITY Constant.

4.06 `const_iter cend()const noexcept;` COMPLEXITY Constant.

4.07 `void clear()noexcept;` COMPLEXITY Linear in size (destructions). VALIDITY All iterators, references and pointers related to this container are invalidated, except the end iterators.

4.08 `template <class... Args> iter emplace_after(const_iter pos, Args&&... args);` COMPLEXITY Constant.

4.09 `template <class... Args> void emplace_front(Args&&... args);` COMPLEXITY Constant. VALIDITY No changes.Member begin returns a different iterator value.

4.10 `bool empty()const noexcept;` COMPLEXITY Constant.

4.11 `iter end()noexcept; const_iter end()const noexcept;` COMPLEXITY Constant.

4.12 `iter erase_after(const_iter pos); iter erase_after(const_iter pos, const_iter last);` COMPLEXITY Linear in the number of elements erased (destructions). VALIDITY Iterators, pointers and references referring to elements removed by the function are invalidated.All other iterators, pointers and references keep their validity.

4.13 `ref front(); const_ref front()const;` COMPLEXITY Constant.

4.14 `allocator_t get_allocator()const noexcept;` COMPLEXITY Constant.

4.15 `iter insert_after(const_iter pos, init_list<value_t> il);` COMPLEXITY Linear on the number of elements inserted (copy/move construction).

4.16 `size_t max_size()const noexcept;` COMPLEXITY Constant.

4.17 `template <class Compare> void merge(forward_list& fwdlst, Compare comp); template <class Compare> void merge(forward_list&& fwdlst, Compare comp);` COMPLEXITY At most, linear in the sum of both container sizes minus one (comparisons). VALIDITY No changes on the iterators, pointers and references related to the container before the call.The iterators, pointers and references that referred to transferred elements keep referring to those same elements, but iterators now iterate into the container the elements have been transferred to.

4.18 `forward_list& operator=(init_list<value_t> il);` COMPLEXITY Linear in the number of elements. VALIDITY All iterators, references and pointers related to this container are invalidated, except the end iterators.In the move assignment, iterators, pointers and references referring to elements in x are also invalidated.

4.19 `void pop_front();` COMPLEXITY Constant. VALIDITY Iterators, pointers and references referring to element removed by the function are invalidated.All other iterators, pointers and reference keep their validity.

4.20 `void push_front(const value_t& val); void push_front(value_t&& val);` COMPLEXITY Constant.

4.21 `void remove(const value_t& val);` COMPLEXITY Linear in container size (comparisons). VALIDITY Iterators, pointers and references referring to elements removed by the function are invalidated.All other iterators, pointers and reference keep their validity.

4.22 `template <class Pred> void remove_if(Pred pred);` COMPLEXITY Linear in container size (applications of pred). VALIDITY Iterators, pointers and references referring to elements removed by the function are invalidated.All other iterators, pointers and reference keep their validity.

4.23 `void resize(size_t n); void resize(size_t n, const value_t& val);` COMPLEXITY Linear in the number number of elements inserted/erased (constructor/destructor), plus up to linear in the size (iterator advance). VALIDITY Iterators, pointers and references referring to elements removed by the function are invalidated.All other iterators, pointers and references keep their validity.

4.24 `void reverse()noexcept;` COMPLEXITY Linear in container size.

4.25 `template <class Compare> void sort(Compare comp);` COMPLEXITY Approximately NlogN where N is the container size.

4.26 `void splice_after(const_iter pos, forward_list& fwdlst, const_iter first, const_iter last); void splice_after(const_iter pos, forward_list&& fwdlst, const_iter first, const_iter last);` COMPLEXITY Up to linear in the number of elements transferred. VALIDITY No changes on the iterators, pointers and references related to the container before the call.The iterators, pointers and references that referred to transferred elements keep referring to those same elements, but iterators now iterate into the container the elements have been transferred to.

4.27 `void swap(forward_list& fwdlst);` COMPLEXITY Constant. VALIDITY All iterators, pointers and references referring to elements in both containers remain valid, but now are referring to elements in the other container, and iterate in it.Note that the end iterators (including before_begin) do not refer to elements and may be invalidated.

4.28 `template <class BinaryPred> void unique(BinaryPred binary_pred);` COMPLEXITY Linear in container size minus one. VALIDITY Iterators, pointers and references referring to elements removed by the function are invalidated.All other iterators, pointers and references keep their validity.

5 Queue

5.01 `ref& back(); const_ref& back()const;` COMPLEXITY Constant (calling back on the underlying container).

5.02 `template <class... Args> void emplace(Args&&... args);` COMPLEXITY One call to emplace_back on the underlying container.

5.03 `bool empty()const;` COMPLEXITY Constant (calling empty on the underlying container).

5.04 `ref& front(); const_ref& front()const;` COMPLEXITY Constant (calling front on the underlying container).

5.05 `void pop();` COMPLEXITY Constant (calling pop_front on the underlying container).

5.06 `void push(const value_t& val); void push(value_t&& val);` COMPLEXITY One call to push_back on the underlying container.

5.07 `size_t size()const;` COMPLEXITY Constant (calling size on the underlying container).

5.08 `void swap(queue& x)noexcept(/*see below*/);` COMPLEXITY Constant.

6 Stack

6.01 `template <class... Args> void emplace(Args&&... args);` COMPLEXITY One call to emplace_back on the underlying container.

6.02 `bool empty()const;` COMPLEXITY Constant (calling empty on the underlying container).

6.03 `void pop();` COMPLEXITY Constant (calling pop_back on the underlying container).

6.04 `void push(const value_t& val); void push(value_t&& val);` COMPLEXITY One call to push_back on the underlying container.

6.05 `size_t size()const;` COMPLEXITY Constant (calling size on the underlying container).

6.06 `void swap(stack& x)noexcept(/*see below*/);` COMPLEXITY Constant.

6.07 `ref top(); const_ref top()const;` COMPLEXITY Constant (calling back on the underlying container).

7 Deque

7.01 `void assign(init_list<value_t> il);` COMPLEXITY Linear in initial and final sizes (destructions, constructions). VALIDITY All iterators, pointers and references related to this container are invalidated.

7.02 `ref at(size_t n); const_ref at(size_t n) const;` COMPLEXITY Constant.

7.03 `ref back(); const_ref back()const;` COMPLEXITY Constant.

7.04 `iter begin()noexcept; const_iter begin() const noexcept;` COMPLEXITY Constant.

7.05 `const_iter cbegin()const noexcept;` COMPLEXITY Constant.

7.06 `const_iter cend()const noexcept;` COMPLEXITY Constant.

7.07 `void clear()noexcept;` COMPLEXITY Linear in size (destructions). VALIDITY All iterators, pointers and references related to this container are invalidated.

7.08 `const_reverse_iter crbegin()const noexcept;` COMPLEXITY Constant.

7.09 `const_reverse_iter crend()const noexcept;` COMPLEXITY Constant.

7.10 `template <class... Args> iter emplace(const_iter pos, Args&&... args);` COMPLEXITY Depending on the particular library implemention, up to linear in the number of elements between position and one of the ends of the deque. VALIDITY If the insertion happens at the beginning or the end of the sequence, all iterators related to this container are invalidated, but pointers and references remain valid, referring to the same elements they were referring to before the call.If the insertion happens anywhere else in the deque, all iterators, pointers and references related to this container are invalidated.

7.11 `template <class... Args> void emplace_back(Args&&... args);` COMPLEXITY Constant. VALIDITY All iterators related to this container are invalidated, but pointers and references remain valid, referring to the same elements they were referring to before the call.

7.12 `template <class... Args> void emplace_front(Args&&... args);` COMPLEXITY Constant. VALIDITY All iterators related to this container are invalidated, but pointers and references remain valid, referring to the same elements they were referring to before the call.

7.13 `bool empty()const noexcept;` COMPLEXITY Constant.

7.14 `iter end()noexcept; const_iter end()const noexcept;` COMPLEXITY Constant.

7.15 `iter erase(const_iter pos); iter erase(const_iter first, const_iter last);` COMPLEXITY Linear on the number of elements erased (destructions). Plus, depending on the particular library implemention, up to an additional linear time on the number of elements between position and one of the ends of the deque. VALIDITY If the erasure operation includes the last element in the sequence, the end iterator and the iterators, pointers and references referring to the erased elements are invalidated.If the erasure includes the first element but not the last, only those referring to the erased elements are invalidated.If it happens anywhere else in the deque, all iterators, pointers and references related to the container are invalidated.

7.16 `ref front(); const_ref front()const;` COMPLEXITY Constant.

7.17 `allocator_t get_allocator()const noexcept;` COMPLEXITY Constant.

7.18 `iter insert(const_iter pos, init_list<value_t> il);` COMPLEXITY Linear on the number of elements inserted (copy/move construction). Plus, depending on the particular library implemention, up to an additional linear in the number of elements between position and one of the ends of the deque. VALIDITY If the insertion happens at the beginning or the end of the sequence, all iterators related to this container are invalidated, but pointers and references remain valid, referring to the same elements they were referring to before the call.If the insertion happens anywhere else in the deque, all iterators, pointers and references related to this container are invalidated.

7.19 `size_t max_size()const noexcept;` COMPLEXITY Constant.

7.20 `ref operator[](size_t n); const_ref operator[](size_t n)const;` COMPLEXITY Constant.

7.21 `deque& operator=(init_list<value_t> il);` COMPLEXITY Linear in size. VALIDITY All iterators, references and pointers related to this container before the call are invalidated.In the move assignment, iterators, pointers and references referring to elements in x are also invalidated.

7.22 `void pop_back();` COMPLEXITY Constant. VALIDITY The end iterator and any iterator, pointer and reference referring to the removed element are invalidated.Iterators, pointers and references referring to other elements that have not been removed are guaranteed to keep referring to the same elements they were referring to before the call.

7.23 `void pop_front();` COMPLEXITY Constant. VALIDITY The iterators, pointers and references referring to the removed element are invalidated.Iterators, pointers and references referring to other elements that have not been removed are guaranteed to keep referring to the same elements they were referring to before the call.

7.24 `void push_back(const value_t& val); void push_back(value_t&& val);` COMPLEXITY Constant. VALIDITY All iterators related to this container are invalidated. Pointers and references to elements in the container remain valid, referring to the same elements they were referring to before the call.

7.25 `void push_front(const value_t& val); void push_front(value_t&& val);` COMPLEXITY Constant. VALIDITY All iterators related to this container are invalidated. Pointers and references to elements in the container remain valid, referring to the same elements they were referring to before the call.

7.26 `reverse_iter rbegin()noexcept; const_reverse_iter rbegin()const noexcept;` COMPLEXITY Constant.

7.27 `reverse_iter rend()noexcept; const_reverse_iter rend()const noexcept;` COMPLEXITY Constant.

7.28 `void resize(size_t n); void resize(size_t n, const value_t& val);` COMPLEXITY Linear on the number of elements inserted/erased (constructions/destructions). VALIDITY In case the container shrinks, all iterators, pointers and references to elements that have not been removed remain valid after the resize and refer to the same elements they were referring to before the call.If the container expands, all iterators are invalidated, but existing pointers and references remain valid, referring to the same elements they were referring to before.

7.29 `void shrink_to_fit();` COMPLEXITY At most, linear in the container size.

7.30 `size_t size()const noexcept;` COMPLEXITY Constant.

7.31 `void swap(deque& x);` COMPLEXITY Constant. VALIDITY All iterators, pointers and references referring to elements in both containers remain valid, and are now referring to the same elements they referred to before the call, but in the other container, where they now iterate.Note that the end iterators do not refer to elements and may be invalidated.

8 Map

8.01 `mapped_t& at(const key_t& k); const mapped_t& at(const key_t& k)const;` COMPLEXITY Logarithmic in size.

8.02 `iter begin()noexcept; const_iter begin() const noexcept;` COMPLEXITY Constant.

8.03 `const_iter cbegin()const noexcept;` COMPLEXITY Constant.

8.04 `const_iter cend()const noexcept;` COMPLEXITY Constant.

8.05 `void clear()noexcept;` COMPLEXITY Linear in size (destructions). VALIDITY All iterators, pointers and references related to this container are invalidated.

8.06 `size_t count(const key_t& k)const;` COMPLEXITY Logarithmic in size.

8.07 `const_reverse_iter crbegin()const noexcept;` COMPLEXITY Constant.

8.08 `const_reverse_iter crend()const noexcept;` COMPLEXITY Constant.

8.09 `template <class... Args> pair<iter,bool> emplace(Args&&... args);` COMPLEXITY Logarithmic in the container size.

8.10 `template <class... Args> iter emplace_hint(const_iter pos, Args&&... args);` COMPLEXITY Generally, logarithmic in the container size.Amortized constant if the insertion point for the element is position.

8.11 `bool empty()const noexcept;` COMPLEXITY Constant.

8.12 `iter end()noexcept; const_iter end()const noexcept;` COMPLEXITY Constant.

8.13 `pair<const_iter,const_iter> equal_range(const key_t& k)const; pair<iter,iter> equal_range(const key_t& k);` COMPLEXITY Logarithmic in size.

8.14 `iter erase(const_iter first, const_iter last);` COMPLEXITY For the first version (erase(position)), amortized constant.For the second version (erase(val)), logarithmic in container size.For the last version (erase(first,last)), linear in the distance between first and last. VALIDITY Iterators, pointers and references referring to elements removed by the function are invalidated.All other iterators, pointers and references keep their validity.

8.15 `iter find(const key_t& k); const_iter find(const key_t& k)const;` COMPLEXITY Logarithmic in size.

8.16 `allocator_t get_allocator()const noexcept;` COMPLEXITY Constant.

8.17 `void insert(init_list<value_t> il);` COMPLEXITY If a single element is inserted, logarithmic in size in general, but amortized constant if a hint is given and the position given is the optimal.If N elements are inserted, Nlog(size+N) in general, but linear in size+N if the elements are already sorted according to the same ordering criterion used by the container.If N elements are inserted, Nlog(size+N).Implementations may optimize if the range is already sorted.

8.18 `key_compare key_comp()const;` COMPLEXITY Constant.

8.19 `iter lower_bound(const key_t& k); const_iter lower_bound(const key_t& k)const;` COMPLEXITY Logarithmic in size.

8.20 `size_t max_size()const noexcept;` COMPLEXITY Constant.

8.21 `mapped_t& operator[](const key_t& k); mapped_t& operator[](key_t&& k);` COMPLEXITY Logarithmic in size.

8.22 `map& operator=(init_list<value_t> il);` COMPLEXITY For the copy assignment (1): Linear in sizes (destructions, copies).For the move assignment (2): Linear in current container size (destructions).* For the initializer list assignment (3): Up to logarithmic in sizes (destructions, move-assignments) – linear if il is already sorted.* Additional complexity for assignments if allocators do not propagate. VALIDITY All iterators, references and pointers related to this container are invalidated.In the move assignment, iterators, pointers and references referring to elements in x are also invalidated.

8.23 `reverse_iter rbegin()noexcept; const_reverse_iter rbegin()const noexcept;` COMPLEXITY Constant.

8.24 `reverse_iter rend()noexcept; const_reverse_iter rend()const noexcept;` COMPLEXITY Constant.

8.25 `size_t size()const noexcept;` COMPLEXITY Constant.

8.26 `void swap(map& x);` COMPLEXITY Constant. VALIDITY All iterators, pointers and references referring to elements in both containers remain valid, but now are referring to elements in the other container, and iterate in it.Note that the end iterators do not refer to elements and may be invalidated.

8.27 `iter upper_bound(const key_t& k); const_iter upper_bound(const key_t& k)const;` COMPLEXITY Logarithmic in size.

8.28 `value_compare value_comp()const;` COMPLEXITY Constant.

9 Set

9.01 `iter begin()noexcept; const_iter begin() const noexcept;` COMPLEXITY Constant.

9.02 `const_iter cbegin()const noexcept;` COMPLEXITY Constant.

9.03 `const_iter cend()const noexcept;` COMPLEXITY Constant.

9.04 `void clear()noexcept;` COMPLEXITY Linear in size (destructions). VALIDITY All iterators, pointers and references related to this container are invalidated.

9.05 `size_t count(const value_t& val)const;` COMPLEXITY Logarithmic in size.

9.06 `const_reverse_iter crbegin()const noexcept;` COMPLEXITY Constant.

9.07 `const_reverse_iter crend()const noexcept;` COMPLEXITY Constant.

9.08 `template <class... Args> pair<iter,bool> emplace(Args&&... args);` COMPLEXITY Logarithmic in the container size.

9.09 `template <class... Args> iter emplace_hint( const_iter pos, Args&&... args);` COMPLEXITY Generally, logarithmic in the container size.Amortized constant if the insertion point for the element is position.

9.10 `bool empty()const noexcept;` COMPLEXITY Constant.

9.11 `iter end()noexcept; const_iter end()const noexcept;` COMPLEXITY Constant.

9.12 `pair<const_iter,const_iter> equal_range (const value_t& val)const; pair<iter,iter> equal_range(const value_t& val);` COMPLEXITY Logarithmic in size.

9.13 `iter erase(const_iter first, const_iter last );` COMPLEXITY For the first version (erase(position)), amortized constant.For the second version (erase(val)), logarithmic in container size.For the last version (erase(first,last)), linear in the distance between first and last. VALIDITY Iterators, pointers and references referring to elements removed by the function are invalidated.All other iterators, pointers and references keep their validity.

9.14 `const_iter find(const value_t& val)const; iter find(const value_t& val);` COMPLEXITY Logarithmic in size.

9.15 `allocator_t get_allocator()const noexcept;` COMPLEXITY Constant.

9.16 `void insert(init_list<value_t> il);` COMPLEXITY If a single element is inserted, logarithmic in size in general, but amortized constant if a hint is given and the position given is the optimal.If N elements are inserted, Nlog(size+N) in general, but linear in size+N if the elements are already sorted according to the same ordering criterion used by the container.If N elements are inserted, Nlog(size+N).Implementations may optimize if the range is already sorted.

9.17 `key_compare key_comp()const;` COMPLEXITY Constant.

9.18 `iter lower_bound(const value_t& val); const_iter lower_bound(const value_t& val)const;` COMPLEXITY Logarithmic in size.

9.19 `size_t max_size()const noexcept;` COMPLEXITY Constant.

9.20 `set& operator=(init_list<value_t> il);` COMPLEXITY For the copy assignment (1): Linear in sizes (destructions, copies).For the move assignment (2): Linear in current container size (destructions).* For the initializer list assignment (3): Up to logarithmic in sizes (destructions, move-assignments) – linear if il is already sorted.* Additional complexity for assignments if allocators do not propagate. VALIDITY All iterators, references and pointers related to this container are invalidated.In the move assignment, iterators, pointers and references referring to elements in x are also invalidated.

9.21 `reverse_iter rbegin()noexcept; const_reverse_iter rbegin()const noexcept;` COMPLEXITY Constant.

9.22 `reverse_iter rend()noexcept; const_reverse_iter rend()const noexcept;` COMPLEXITY Constant.

9.23 `size_t size()const noexcept;` COMPLEXITY Constant.

9.24 `void swap(set& x);` COMPLEXITY Constant. VALIDITY All iterators, pointers and references referring to elements in both containers remain valid, but now are referring to elements in the other container, and iterate in it.Note that the end iterators do not refer to elements and may be invalidated.

9.25 `iter upper_bound(const value_t& val); const_iter upper_bound(const value_t& val)const;` COMPLEXITY Logarithmic in size.

9.26 `value_compare value_comp()const;` COMPLEXITY Constant.

10 Unordered Map

10.01 `mapped_t& at(const key_t& k); const mapped_t& at(const key_t& k)const;` COMPLEXITY Average case: constant.Worst case: linear in container size.

10.02 `local_iter begin(size_t n); const_local_iter begin(size_t n)const;` COMPLEXITY Constant.

10.03 `size_t bucket(const key_t& k)const;` COMPLEXITY Constant.

10.04 `size_t bucket_count()const noexcept;` COMPLEXITY Constant.

10.05 `size_t bucket_size(size_t n)const;` COMPLEXITY Linear in the bucket size.

10.06 `const_local_iter cbegin(size_t n)const;` COMPLEXITY Constant.

10.07 `const_local_iter cend(size_t n)const;` COMPLEXITY Constant.

10.08 `void clear()noexcept;` COMPLEXITY Linear on size (destructors). VALIDITY All iterators, pointers and references are invalidated.

10.09 `size_t count(const key_t& k)const;` COMPLEXITY Average case: linear in the number of elements counted.Worst case: linear in container size.

10.10 `template <class... Args>pair<iter, bool> emplace(Args&&... args);` COMPLEXITY Average case: constant.Worst case: linear in container size.May trigger a rehash (not included). VALIDITY On most cases, all iterators in the container remain valid after the insertion. The only exception being when the growth of the container forces a rehash. In this case, all iterators in the container are invalidated.A rehash is forced if the new container size after the insertion operation would increase above its capacity threshold (calculated as the container's bucket_count multiplied by its max_load_factor).References to elements in the unordered_map container remain valid in all cases, even after a rehash.

10.11 `template <class... Args>iter emplace_hint( const_iter pos, Args&&... args);` COMPLEXITY Average case: constant.Worst case: linear in container size.May trigger a rehash (not included). VALIDITY On most cases, all iterators in the container remain valid after the insertion. The only exception being when the growth of the container forces a rehash. In this case, all iterators in the container are invalidated.A rehash is forced if the new container size after the insertion operation would increase above its capacity threshold (calculated as the container's bucket_count multiplied by its max_load_factor).References remain valid in all cases, even after a rehash.

10.12 `bool empty()const noexcept;` COMPLEXITY Constant.

10.13 `local_iter end(size_t n); const_local_iter end(size_t n)const;` COMPLEXITY Constant.

10.14 `pair<iter,iter> equal_range(const key_t& k ); pair<const_iter,const_iter> equal_range(const key_t& k)const;` COMPLEXITY Average case: constant.Worst case: linear in container size.

10.15 `iter erase(const_iter first, const_iter last);` COMPLEXITY Average case: Linear in the number of elements removed (which is constant for versions (1) and (2)).Worst case: Linear in the container size. VALIDITY Only the iterators and references to the elements removed are invalidated.The rest are unaffected.Only the iterators and references to the elements removed are invalidated.The rest are unaffected.The relative order of iteration of the elements not removed by the operation is preserved.

10.16 `iter find(const key_t& k); const_iter find (const key_t& k)const;` COMPLEXITY Average case: constant.Worst case: linear in container size.

10.17 `allocator_t get_allocator()const noexcept;` COMPLEXITY Constant.

10.18 `hasher hash_function()const;` COMPLEXITY Constant.

10.19 `void insert(init_list<value_t> il);` COMPLEXITY Single element insertions:Average case: constant.Worst case: linear in container size.Multiple elements insertion:Average case: linear in the number of elements inserted.Worst case: N*(size+1): number of elements inserted times the container size plus one.May trigger a rehash (not included in the complexity above). VALIDITY On most cases, all iterators in the container remain valid after the insertion.

The only exception being when the growth of the container forces a rehash. In this case, all iterators in the container are invalidated.A rehash is forced if the new container size after the insertion operation would increase above its capacity threshold (calculated as the container's bucket_count multiplied by its max_load_factor).References to elements in the unordered_map container remain valid in all cases, even after a rehash.

10.20 key_equal key_eq()const; COMPLEXITY Constant.

10.21 float load_factor()const noexcept; COMPLEXITY Constant.

10.22 size_t max_bucket_count()const noexcept; COMPLEXITY Constant.

10.23 void max_load_factor(float z); COMPLEXITY Constant.May trigger a rehash (not included). VALIDITY No changes, unless a change in this value forces a rehash. In this case, all iterators in the container are invalidated.A rehash is forced if the new container max_load_factor is set below the current load_factor.References to elements in the unordered_map container remain valid in all cases, even after a rehash.

10.24 size_t max_size()const noexcept; COMPLEXITY Constant.

10.25 mapped_t& operator[](const key_t& k); mapped_t& operator[](key_t&& k); COMPLEXITY Average case: constant.Worst case: linear in container size.May trigger a rehash if an element is inserted (not included in the complexity above). VALIDITY On most cases, all iterators in the container remain valid after the insertion. The only exception being when this function inserts a new element and this forces a rehash. In this case, all iterators in the container are invalidated.A rehash is forced if the new container size after the insertion operation would increase above its capacity threshold (calculated as the container's bucket_count multiplied by its max_load_factor).References to elements in the unordered_map container remain valid in all cases, even after a rehash.

10.26 unordered_map& operator=(intitializer_list<value_t> il); COMPLEXITY For the copy assignment (1): Linear in sizes (destructions, copies).For the move assignment (2): Linear in current container size (destructions).* For the initializer list assignment (3): On average, linear in sizes (destructions, move-assignments) – worst case: quadratic.* Additional complexity for assignments if allocators do not propagate. VALIDITY All iterators, references and pointers to elements that were in the container before the call are invalidated.

10.27 void rehash(size_t n); COMPLEXITY In case of rehash,Average case: linear in container size.Worst case: quadratic in container size. VALIDITY If a rehash happens, all iterators are invalidated, but references and pointers to individual elements remain valid.If no actual rehash happens, no changes.

10.28 void reserve(size_t n); COMPLEXITY In case of rehash,Average case: linear in container size.Worst case: quadratic in container size. VALIDITY If a rehash happens, all iterators are invalidated, but references and pointers to individual elements remain valid.If no actual rehash happens, no changes.

10.29 size_t size()const noexcept; COMPLEXITY Constant.

10.30 void swap(unordered_map& ump); COMPLEXITY Constant. VALIDITY All iterators, pointers and references remain valid, but now are referring to elements in the other container, and iterate in it.

11 Unordered Set

11.01 local_iter begin(size_t n); const_local_iter begin(size_t n)const; COMPLEXITY Constant.

11.02 size_t bucket(const key_t& k)const; COMPLEXITY Constant.

11.03 size_t bucket_count()const noexcept; COMPLEXITY Constant.

11.04 size_t bucket_size(size_t n)const; COMPLEXITY Linear in the bucket size.

11.05 const_local_iter cbegin(size_t n)const; COMPLEXITY Constant.

11.06 const_local_iter cend(size_t n)const; COMPLEXITY Constant.

11.07 void clear()noexcept; COMPLEXITY Linear on size (destructors). VALIDITY All iterators, pointers and references are invalidated.

11.08 size_t count(const key_t& k)const; COMPLEXITY Average case: constant.Worst case: linear in container size.

11.09 template <class... Args>pair <iter,bool> emplace(Args&&... args); COMPLEXITY Average case: constant.Worst case: linear in container size.May trigger a rehash (not included). VALIDITY On most cases, all iterators in the container remain valid after the insertion. The only exception being when the growth of the container forces a rehash. In this case, all iterators in the container are invalidated.A rehash is forced if the new container size after the insertion operation would increase above its capacity threshold (calculated as the container's bucket_count multiplied by its max_load_factor).References to elements in the unordered_set container remain valid in all cases, even after a rehash.

11.10 template <class... Args>iter emplace_hint(const_iter pos, Args&&... args); COMPLEXITY Average case: constant.Worst case: linear in container size.May trigger a rehash (not included). VALIDITY On most cases, all iterators in the container remain valid after the insertion. The only exception being when the growth of the container forces a rehash. In this case, all iterators in the container are invalidated.A rehash is forced if the new container size after the insertion operation would increase above its capacity threshold (calculated as the container's bucket_count multiplied by its max_load_factor).References remain valid in all cases, even after a rehash.

11.11 bool empty()const noexcept; COMPLEXITY Constant.

11.12 local_iter end(size_t n); const_local_iter end(size_t n)const; COMPLEXITY Constant.

11.13 pair<iter,iter> equal_range(const key_t& k); pair<const_iter,const_iter> equal_range(const key_t& k)const; COMPLEXITY Average case: constant.Worst case: linear in container size.

11.14 iter erase(const_iter first, const_iter last); COMPLEXITY Average case: Linear in the number of elements removed (which is constant for versions (1) and (2)).Worst case: Linear in the container size. VALIDITY Only the iterators and references to the elements removed are invalidated.The rest are unaffected.Only the iterators and references to the elements removed are invalidated.The rest are unaffected.The relative order of iteration of the elements not removed by the operation is preserved.

11.15 iter find(const key_t& k); const_iter find (const key_t& k)const; COMPLEXITY Average case: constant.Worst case: linear in container size.

11.16 allocator_t get_allocator()const noexcept; COMPLEXITY Constant.

11.17 hasher hash_function()const; COMPLEXITY Constant.

11.18 void insert(init_list<value_t> il); COMPLEXITY Single element insertions:Average case: constant.Worst case: linear in container size.Multiple elements insertion:Average case: linear in the number of elements inserted.Worst case: N*(size+1): number of elements inserted times the container size plus one.May trigger a rehash (not included). VALIDITY On most cases, all iterators in the container remain valid after the insertion. The only exception being when the growth of the container forces a rehash. In this case, all iterators in the container are invalidated.A rehash is forced if the new container size after the insertion operation would increase above its capacity threshold (calculated as the container's bucket_count multiplied by its max_load_factor).References to elements in the unordered_set container remain valid in all cases, even after a rehash.

11.19 key_equal key_eq()const; COMPLEXITY Constant.

11.20 float load_factor()const noexcept; COMPLEXITY Constant.

11.21 size_t max_bucket_count()const noexcept; COMPLEXITY Constant.

11.22 void max_load_factor(float z); COMPLEXITY Constant.May trigger a rehash (not included). VALIDITY No changes, unless a change in this value forces a rehash. In this case, all iterators in the container are invalidated.A rehash is forced if the new container max_load_factor is set below the current load_factor.References to elements in the unordered_set container remain valid in all cases, even after a rehash.

11.23 size_t max_size()const noexcept; COMPLEXITY Constant.

11.24 unordered_set& operator=(intitializer_list<value_t> il); COMPLEXITY For the copy assignment (1): Linear in sizes (destructions, copies).For the move assignment (2): Linear in current container size (destructions).* For the initializer list assignment (3): On average, linear in sizes (destructions, move-assignments) – worst case: quadratic.* Additional complexity for assignments if allocators do not propagate. VALIDITY All iterators, references and pointers to elements that were in the container before the call are invalidated.

11.25 void rehash(size_t n); COMPLEXITY In case of rehash,Average case: linear in container size.Worst case: quadratic in container size. VALIDITY If a rehash happens, all iterators are invalidated, but references and pointers to individual elements remain valid.If no actual rehash happens, no changes.

11.26 void reserve(size_t n); COMPLEXITY In case of rehash,Average case: linear in container size.Worst case: quadratic in container size. VALIDITY If a rehash happens, all iterators are invalidated, but references and pointers to individual elements remain valid.If no actual rehash happens, no changes.

11.27 size_t size()const noexcept; COMPLEXITY Constant.

11.28 void swap(unordered_set& ust); COMPLEXITY Constant. VALIDITY All iterators, pointers and references remain valid, but now are referring to elements in the other container, and iterate in it.

12 Algorithm

12.01 template <class InIter, class UnaryPred> bool all_of(InIter first, InIter last, UnaryPred pred); COMPLEXITY Up to linear in the distance between first and last: Calls pred for each element until a mismatch is found.

12.02 template <class InIter, class UnaryPred> bool any_of(InIter first, InIter last, UnaryPred pred); COMPLEXITY Up to linear in the distance between first and last: Calls pred for each element until a match is found.

12.03 template <class InIter, class UnaryPred> bool none_of(InIter first, InIter last, UnaryPred pred); COMPLEXITY Up to linear in the distance between first and last: Calls pred for each element until a match is found.

12.04 template <class InIter, class Function> Function for_each(InIter first, InIter last, Function fn); COMPLEXITY Linear in the distance between first and last: Applies fn to each element.

12.05 template <class InIter, class T> InIter find(InIter first, InIter last, const T& val); COMPLEXITY Up to linear in the distance between first and last: Compares elements until a match is found.

12.06 template <class InIter, class UnaryPred > InIter find_if(InIter first, InIter last, UnaryPred pred); COMPLEXITY Up to linear in the distance between first and last: Calls pred for each element until a match is found.

12.07 template <class InIter, class UnaryPred> InIter find_if_not(InIter first, InIter last, UnaryPred pred); COMPLEXITY Up to linear in the distance between first and last: Calls pred for each element until a mismatch is found.

12.08 template <class FwIter1, class FwIter2, class BinaryPred> FwIter1 find_end(FwIter1 first1 , FwIter1 last1, FwIter2 first2, FwIter2 last2, BinaryPred pred); COMPLEXITY Up to linear in count2*(1+count1-count2), where countX is the distance between firstX and lastX: Compares elements until the last matching subsequence is found.

12.09 template <class InIter, class FwIter, class BinaryPred> InIter find_first_of(InIter first1 , InIter last1, FwIter first2, FwIter last2, BinaryPred pred); COMPLEXITY Up to linear in count1*count2 (where countX is the distance between firstX and lastX): Compares elements until a match is found.

12.10 template <class FwIter, class BinaryPred> FwIter adjacent_find(FwIter first, FwIter last, BinaryPred pred); COMPLEXITY Up to linear in the distance between first and last: Compares elements until a match is found.

12.11 template <class InIter, class T> typename iter_traits<InIter>::difference_t count(InIter first, InIter last, const T& val); COMPLEXITY Linear in the distance between first and last: Compares once each element.

12.12 template <class InIter, class UnaryPred > typename iter_traits<InIter>::difference_t count_if(InIter first, InIter last, UnaryPred pred); COMPLEXITY Linear in the distance between first and last: Calls pred once for each element.

12.13 template <class InIter1, class InIter2, class BinaryPred> pair<InIter1, InIter2> mismatch (InIter1 first1, InIter1 last1, InIter2 first2, BinaryPred pred); COMPLEXITY Up to linear in the distance between first1 and last1: Compares elements until a mismatch is found.

12.14 template <class InIter1, class InIter2, class BinaryPred> bool equal(InIter1 first1, InIter1 last1, InIter2 first2, BinaryPred pred); COMPLEXITY Up to linear in the distance between first1 and last1: Compares elements until a mismatch is found.

12.15 template <class FwIter1, class FwIter2, class BinaryPred> bool is_permutation(FwIter1 first1, FwIter1 last1, FwIter2 first2, BinaryPred pred); COMPLEXITY If both sequence are equal (with the elements in the same order), linear in the distance between first1 and last1.Otherwise, up to quadratic: Performs at most N2 element comparisons until the result is determined (where N is the distance between first1 and last1).

12.16 template <class FwIter1, class FwIter2, class BinaryPred> FwIter1 search(FwIter1 first1 , FwIter1 last1, FwIter2 first2, FwIter2 last2, BinaryPred pred); COMPLEXITY Up to linear in count1*count2 (where countX is the distance between firstX and lastX): Compares elements until a matching subsequence is found.

12.17 template <class FwIter, class Size, class T, class BinaryPred> FwIter search_n(FwIter first, FwIter last, Size count, const T& val, BinaryPred pred); COMPLEXITY Up to linear in the distance between first and last: Compares elements until a matching subsequence is found.

12.18 template <class InIter, class OutIter> OutIter copy(InIter first, InIter last, OutIter result); COMPLEXITY Linear in the distance between first and last: Performs an assignment operation for each element in the range.

12.19 template <class InIter, class Size, class OutIter> OutIter copy_n(InIter first, Size n, OutIter result); COMPLEXITY Linear in the distance between first and last: Performs an assignment operation for each element in the range.

12.20 template <class InIter, class OutIter, class UnaryPred> OutIter copy_if(InIter first, InIter last, OutIter result, UnaryPred pred); COMPLEXITY Linear in the distance between first and last: Applies pred to each element in the range and performs at most that many assignments.

12.21 template <class BiIter1, class BiIter2> BiIter2 copy_backward(BiIter1 first, BiIter1 last , BiIter2 result); COMPLEXITY Linear in the distance between first and last: Performs an assignment operation for each element in the range.

12.22 template <class InIter, class OutIter> OutIter move(InIter first, InIter last, OutIter result); COMPLEXITY Linear in the distance between first and last: Performs a move-assignment for each element in the range.

12.23 template <class BiIter1, class BiIter2> BiIter2 move_backward(BiIter1 first, BiIter1 last, BiIter2 result); COMPLEXITY Linear in the distance between first and last: Performs a move-assignment for each element in the range.

12.24 template <class T, size_t N> void swap(T(& a)[N], T(&b)[N])noexcept(noexcept(swap(*a,*b))); COMPLEXITY Non-array: Constant: Performs exactly one construction and two assignments (although notice that each of these operations works on its own complexity).Array: Linear in N: performs a swap operation per element.

12.25 template <class FwIter1, class FwIter2> FwIter2 swap_ranges(FwIter1 first1, FwIter1 last1 , FwIter2 first2); COMPLEXITY Linear in the distance between first and last: Performs a swap operation for each element in the range.

12.26 template <class FwIter1, class FwIter2> void iter_swap(FwIter1 a, FwIter2 b); COMPLEXITY Constant: Calls swap once.

12.27 template <class InIter1, class InIter2, class OutIter, class BinaryOperation> OutIter transform(InIter1 first1, InIter1 last1, InIter2 first2, OutIter result, BinaryOperation binary_op); COMPLEXITY Linear in the distance between first1 and last1: Performs one assignment and one application of op (or binary_op) per element.

12.28 template <class FwIter, class T> void replace(FwIter first, FwIter last, const T& old_value, const T& new_value); COMPLEXITY Linear in the distance between first and last: Compares each element and assigns to those matching.

12.29 template <class FwIter, class UnaryPred, class T> void replace_if(FwIter first, FwIter last, UnaryPred pred, const T& new_value); COMPLEXITY Linear in the distance between first and last: Applies pred to each element and assigns to those matching.

12.30 template <class InIter, class OutIter, class T> OutIter replace_copy(InIter first, InIter last, OutIter result, const T& old_value , const T& new_value); COMPLEXITY Linear in the distance between first and last: Performs a comparison and an assignment for each element.

12.31 template <class InIter, class OutIter, class UnaryPred, class T> OutIter replace_copy_if (InIter first, InIter last, OutIter result, UnaryPred pred, const T& new_value); COMPLEXITY Linear in the distance between first and last: Applies pred and performs an assignment for each element.

12.32 template <class FwIter, class T> void fill (FwIter first, FwIter last, const T& val); COMPLEXITY Linear in the distance between first and last: Assigns a value to each element.

12.33 template <class OutIter, class Size, class T> OutIter fill_n(OutIter first, Size n, const T& val); COMPLEXITY Linear in n: Assigns a value to each element.

12.34 template <class FwIter, class Generator > void generate(FwIter first, FwIter last, Generator gen); COMPLEXITY Linear in the distance between first and last: Calls gen and performs an assignment for each element.

12.35 template <class OutIter, class Size, class Generator> OutIter generate_n(OutIter first, Size n, Generator gen); COMPLEXITY Linear in n: Calls gen and performs an assignment for each element.

12.36 template <class FwIter, class T> FwIter remove(FwIter first, FwIter last, const T& val); COMPLEXITY Linear in the distance between first and last: Compares each element, and possibly performs assignments on some of them.

12.37 template <class FwIter, class UnaryPred > FwIter remove_if(FwIter first, FwIter last, UnaryPred pred); COMPLEXITY Linear in the distance between first and last: Applies pred to each element, and possibly performs assignments on some of them.

12.38 template <class InIter, class OutIter, class T> OutIter remove_copy(InIter first, InIter last, OutIter result, const T& val); COMPLEXITY Linear in the distance between first and last: Compares each element, and performs an assignment operation for those not removed.

12.39 template <class InIter, class OutIter, class UnaryPred> OutIter remove_copy_if(InIter first, InIter last, OutIter result, UnaryPred pred); COMPLEXITY Linear in the distance between first and last: Applies pred to each element, and performs an assignment operation for those not removed.

12.40 template <class FwIter, class BinaryPred > FwIter unique(FwIter first, FwIter last, BinaryPred pred); COMPLEXITY For non-empty ranges, linear in one less than the distance between first and last: Compares each pair of consecutive elements, and possibly performs assignments on some of them.

12.41 template <class InIter, class OutIter, class BinaryPred> OutIter unique_copy(InIter first, InIter last, OutIter result, BinaryPred pred); COMPLEXITY Up to linear in the distance between first and last: Compares each pair of elements, and performs an assignment operation for those elements not matching.

12.42 template <class BiIter> void reverse(BiIter first, BiIter last); COMPLEXITY Linear in half the distance between first and last: Swaps elements.

12.43 template <class BiIter, class OutIter> OutIter reverse_copy(BiIter first, BiIter last, OutIter result); COMPLEXITY Linear in the distance between first and last: Performs an assignment for each element.

12.44 template <class FwIter> FwIter rotate(FwIter first, FwIter middle, FwIter last); COMPLEXITY Up to linear in the distance between first and last: Swaps (or moves) elements until all elements have been relocated.

12.45 template <class FwIter, class OutIter> OutIter rotate_copy(FwIter first, FwIter middle, FwIter last, OutIter result); COMPLEXITY Linear in the distance between first and last: Performs an assignment for each element.

12.46 template <class RAIter, class RandomNumberGenerator> void random_shuffle(RAIter first, RAIter last, RandomNumberGenerator&& gen); COMPLEXITY Linear in the distance between first and last minus one: Obtains random values and swaps elements.

12.47 template <class RAIter, class URNG> void shuffle(RAIter first, RAIter last, URNG&& g); COMPLEXITY Linear in the distance between first and last minus one: Obtains random values and swaps elements.

12.48 template <class InIter, class UnaryPred> bool is_partitioned(InIter first, InIter last, UnaryPred pred); COMPLEXITY Up to linear in the distance between first and last: Calls pred for each element until a mismatch is found.

12.49 template <class FwIter, class UnaryPred > FwIter partition(FwIter first, FwIter last, UnaryPred pred); COMPLEXITY Linear in the distance between first and last: Applies pred to each element, and possibly swaps some of them (if the iterator type is a bidirectional, at most half that many swaps, otherwise at most that many).

12.50 template <class BiIter, class UnaryPred> BiIter stable_partition(BiIter first, BiIter last , UnaryPred pred); COMPLEXITY If enough extra memory is available, linear in the distance between first and last: Applies pred exactly once to each element, and performs up to that many element moves.Otherwise, up to linearithmic: Performs up to N*log(N) element swaps (where N is the distance above). It also applies pred exactly once to each element.

12.51 `template <class InIter, class OutIter1 , class OutIter2, class UnaryPred pred> pair< OutIter1,OutIter2> partition_copy(InIter first , InIter last, OutIter1 result_true, OutIter2 result_false, UnaryPred pred);` COMPLEXITY Linear in the distance between first and last: Calls pred and performs an assignment once for each element.

12.52 `template <class FwIter, class UnaryPred> FwIter partition_point(FwIter first, FwIter last , UnaryPred pred);` COMPLEXITY On average, logarithmic in the distance between first and last: Performs approximately log2(N)+2 element comparisons (where N is this distance).On non-random-access iterators, the iterator advances produce themselves an additional linear complexity in N on average.

12.53 `template <class RAIter, class Compare> void sort(RAIter first, RAIter last, Compare comp);` COMPLEXITY On average, linearithmic in the distance between first and last: Performs approximately N*log2(N) (where N is this distance) comparisons of elements, and up to that many element swaps (or moves).

12.54 `template <class RAIter> void stable_sort (RAIter first, RAIter last); template <class RAIter, class Compare> void stable_sort(RAIter first, RAIter last, Compare comp);` COMPLEXITY If enough extra memory is available, linearithmic in the distance between first and last: Performs up to N*log2(N) element comparisons (where N is this distance), and up to that many element moves.Otherwise, polyloglinear in that distance: Performs up to N*log22(N) element comparisons, and up to that many element swaps.

12.55 `template <class RAIter, class Compare> void partial_sort(RAIter first, RAIter middle, RAIter last, Compare comp);` COMPLEXITY On average, less than linearithmic in the distance between first and last: Performs approximately N*log(M) comparisons of elements (where N is this distance, and M is the distance between first and middle). It also performs up to that many element swaps (or moves).

12.56 `template <class InIter, class RAIter, class Compare> RAIter partial_sort_copy(InIter first,InIter last, RAIter result_first, RAIter result_last, Compare comp);` COMPLEXITY On average, less than linearithmic in the distance between first and last: Performs approximately N*log(min(N,M)) comparisons of elements (where N is this distance, and M is the distance between result_first and result_last). It also performs up to that many element swaps (or moves) and min(N,M) assignments between ranges.

12.57 `template <class FwIter, class Compare> bool is_sorted(FwIter first, FwIter last, Compare comp);` COMPLEXITY Up to linear in one less than the distance between first and last: Compares pairs of elements until a mismatch is found.

12.58 `template <class FwIter, class Compare> FwIter is_sorted_until(FwIter first, FwIter last , Compare comp);` COMPLEXITY Up to linear in the distance between first and last: Calls comp for each element until a mismatch is found.

12.59 `template <class RAIter, class Compare> void nth_element(RAIter first, RAIter nth, RAIter last, Compare comp);` COMPLEXITY On average, linear in the distance between first and last: Compares elements, and possibly swaps (or moves) them, until the elements are properly rearranged.

12.60 `template <class FwIter, class T, class Compare> FwIter lower_bound(FwIter first, FwIter last, const T& val, Compare comp);` COMPLEXITY On average, logarithmic in the distance between first and last: Performs approximately log2(N)+1 element comparisons (where N is this distance).On non-random-access iterators, the iterator advances produce themselves an additional linear complexity in N on average.

12.61 `template <class FwIter, class T, class Compare> FwIter upper_bound(FwIter first, FwIter last, const T& val, Compare comp);` COMPLEXITY On average, logarithmic in the distance between first and last: Performs approximately log2(N)+1 element comparisons (where N is this distance).On non-random-access iterators, the iterator advances produce themselves an additional linear complexity in N on average.

12.62 `template <class FwIter, class T, class Compare> pair<FwIter,FwIter> equal_range(FwIter first, FwIter last, const T& val, Compare comp);` COMPLEXITY On average, up to twice logarithmic in the distance between first and last: Performs approximately 2*log2(N)+1 element comparisons (where N is this distance).On non-random-access iterators, the iterator advances produce themselves an additional up to twice linear complexity in N on average.

12.63 `template <class FwIter, class T, class Compare> bool binary_search(FwIter first, FwIter last, const T& val, Compare comp);` COMPLEXITY On average, logarithmic in the distance between first and last: Performs approximately log2(N)+2 element comparisons (where N is this distance).On non-random-access iterators, the iterator advances produce themselves an additional linear complexity in N on average.

12.64 `template <class InIter1, class InIter2, class OutIter, class Compare> OutIter merge( InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, OutIter result, Compare comp);` COMPLEXITY Up to linear in (1+count1-count2), where countX is the distance between firstX and lastX: Compares and assigns all elements.

12.65 `template <class BiIter, class Compare> void inplace_merge(BiIter first, BiIter middle, BiIter last, Compare comp);` COMPLEXITY If enough extra memory is available, linear in the distance between first and last: Performs N-1 comparisons and up to twice that many element moves.Otherwise, up to linearithmic: Performs up to N*log(N) element comparisons (where N is the distance above), and up to that many element swaps.

12.66 `template <class InIter1, class InIter2> bool includes(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2); template <class InIter1, class InIter2, class Compare> bool includes(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, Compare comp);` COMPLEXITY Up to linear in twice the distances in both ranges: Performs up to 2*(count1+count2)-1 comparisons (where countX is the distance between firstX and lastX).

12.67 `template <class InIter1, class InIter2, class OutIter, class Compare> OutIter set_union (InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, OutIter result, Compare comp);` COMPLEXITY Up to linear in 2*(count1+count2)-1 (where countX is the distance between firstX and lastX): Compares and assigns elements.

12.68 `template <class InIter1, class InIter2 , class OutIter, class Compare> OutIter set_intersection(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, OutIter result , Compare comp);` COMPLEXITY Up to linear in 2*(count1+count2)-1 (where countX is the distance between firstX and lastX): Compares and assigns elements.

12.69 `template <class InIter1, class InIter2 , class OutIter, class Compare> OutIter set_difference(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, OutIter result , Compare comp);` COMPLEXITY Up to linear in 2*(count1+count2)-1 (where countX is the distance between firstX and lastX): Compares and assigns elements.

12.70 `template <class InIter1, class InIter2 , class OutIter, class Compare> OutIter set_symmetric_difference(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, OutIter result, Compare comp);` COMPLEXITY Up to linear in 2*(count1+count2)-1 (where countX is the distance between firstX and lastX): Compares and assigns elements.

12.71 `template <class RAIter, class Compare> void push_heap(RAIter first, RAIter last, Compare comp);` COMPLEXITY Up to logarithmic in the distance between first and last: Compares elements and potentially swaps (or moves) them until rearranged as a longer heap.

12.72 `template <class RAIter, class Compare> void pop_heap(RAIter first, RAIter last, Compare comp );` COMPLEXITY Up to twice logarithmic in the distance between first and last: Compares elements and potentially swaps (or moves) them until rearranged as a shorter heap.

12.73 `template <class RAIter, class Compare> void make_heap(RAIter first, RAIter last, Compare comp);` COMPLEXITY Up to linear in three times the distance between first and last: Compares elements and potentially swaps (or moves) them until rearranged as a heap.

12.74 `template <class RAIter, class Compare> void sort_heap(RAIter first, RAIter last, Compare comp);` COMPLEXITY Up to linearithmic in the distance between first and last: Performs at most N*log(N) (where N is this distance) comparisons of elements, and up to that many element swaps (or moves).

12.75 `template <class RAIter, class Compare> bool is_heap(RAIter first, RAIter last, Compare comp );` COMPLEXITY Up to linear in one less than the distance between first and last: Compares pairs of elements until a mismatch is found.

12.76 `template <class RAIter, class Compare> RAIter is_heap_until(RAIter first, RAIter last Compare comp);` COMPLEXITY Up to linear in the distance between first and last: Compares elements until a mismatch is found.

12.77 `template <class T> constexpr T min( init_list<T> il); template <class T, class Compare> constexpr T min(init_list<T> il, Compare comp);` COMPLEXITY Linear in one less than the number of elements compared (constant for (1) and (2)).

12.78 `template <class T> constexpr T max( init_list<T> il); template <class T, class Compare> constexpr T max(init_list<T> il, Compare comp);` COMPLEXITY Linear in one less than the number of elements compared (constant for (1) and (2)).

12.79 `template <class T> constexpr pair<T,T> minmax(init_list<T> il); template <class T, class Compare> constexpr pair<T,T> minmax(init_list<T> il, Compare comp);` COMPLEXITY Up to linear in one and half times the number of elements compared (constant for (1) and (2)).

12.80 `template <class FwIter, class Compare> FwIter min_element(FwIter first, FwIter last, Compare comp);` COMPLEXITY Linear in one less than the number of elements compared.

12.81 `template <class FwIter, class Compare> FwIter max_element(FwIter first, FwIter last, Compare comp);` COMPLEXITY Linear in one less than the number of elements compared.

12.82 `template <class FwIter, class Compare> pair<FwIter,FwIter> minmax_element(FwIter first , FwIter last, Compare comp);` COMPLEXITY Up to linear in 1.5 times one less than the number of elements compared.

12.83 `template <class InIter1, class InIter2, class Compare> bool lexicographical_compare( InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2, Compare comp);` COMPLEXITY Up to linear in 2*min(count1,count2) (where countX is the distance between firstX and lastX): Compares elements symmetrically until a mismatch is found.

12.84 `template <class BiIter, class Compare> bool next_permutation(BiIter first, BiIter last, Compare comp);` COMPLEXITY Up to linear in half the distance between first and last (in terms of actual swaps).

12.85 `template <class BiIter, class Compare> bool prev_permutation(BiIter first, BiIter last, Compare comp);` COMPLEXITY Up to linear in half the distance between first and last (in terms of actual swaps).