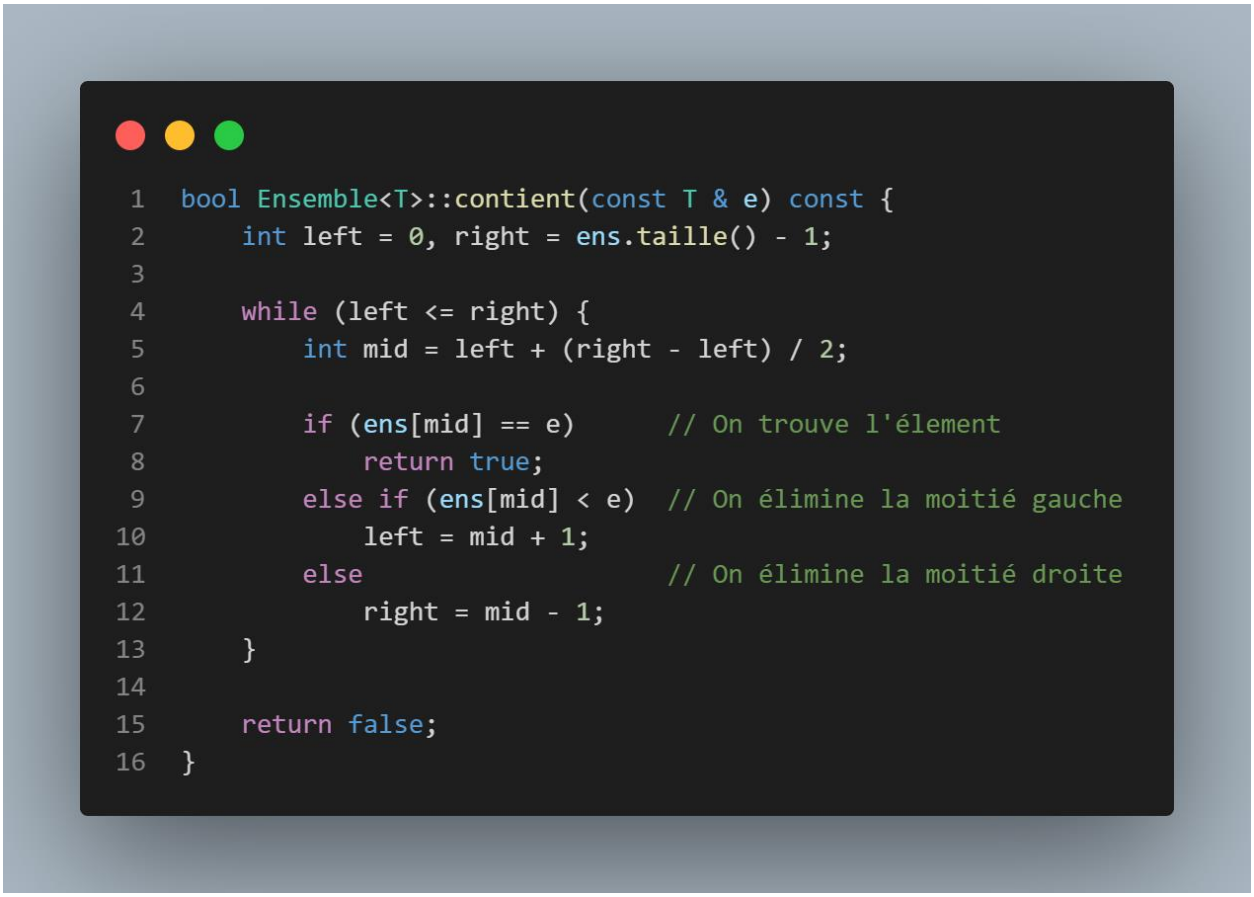
Glazyrin Leonid GLAL77080105

Analyse des algorithmes:

`bool contient(const T & e) const;` **O(log n)**

La complexité temporelle du pire cas de cette méthode est déterminée par le nombre maximum d'itérations de la boucle `while`. Dans le pire des cas, la boucle while sera exécutée jusqu'à ce que `left` soit égal à `right`. Par conséquent, le temps d'exécution total de la méthode est de O(log n).

Cette complexité temporelle est efficace pour les ensembles de grande taille, car le nombre d'itérations nécessaires pour trouver un élément est logarithmique par rapport à la taille de l'ensemble.



```cpp
bool Ensemble<T>::contient(const T & e) const {
    int left = 0, right = ens.taille() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (ens[mid] == e)      // On trouve l'élement
            return true;
        else if (ens[mid] < e)  // On élimine la moitié gauche
            left = mid + 1;
        else                    // On élimine la moitié droite
            right = mid - 1;
    }

    return false;
}
```

`bool operator == (const Ensemble & autre) const;` **O(min(n, m))**

La méthode ne fait qu'utiliser la l'opérateur == de la classe Tableau, qui elle vérifie en O(1) si leur nombres d'éléments sont égaux et puis pour chaque indice, vérifie si le contenu est égale. Ainsi, la complexité temporelle totale de cette méthode est de O(min(n, m)), où m et n sont les tailles des deux ensembles.

Il me semble que le min() ne fait aucun sens, puisque les deux Ensemble seront toujours de même longueur pour procéder à la verification indice par indice. Sinon la complexité sera de O(1), car en vérifiant les tailles non égales, on retournera tout de suite false.

```cpp
bool Ensemble<T>::operator == (const Ensemble<T> & autre) const {
    return ens == autre.ens;
}
```

```cpp
bool Tableau<T>::operator == (const Tableau<T> & autre) const {
    if(this==&autre)
        return true;
    if (nbElements != autre.nbElements)
        return false;

    for (int i = 0; i < nbElements; i++)
        if (elements[i] != autre.elements[i])
            return false;

    return true;
}
```

`Ensemble fusion(const Ensemble & autre) const;` **O(m + n)**

Dans le pire des cas, tous les éléments des deux ensembles sont différents, et chaque élément doit être comparé à chaque élément de l'autre ensemble pour fusionner les deux ensembles.

La boucle principale parcourt les deux ensembles simultanément en comparant les éléments de chaque ensemble à chaque itération. Le temps d'exécution de la boucle principale est donc proportionnel à la taille totale des deux ensembles. Par conséquent, la complexité temporelle de cette méthode est de O(m + n).

```cpp
Ensemble<T> Ensemble<T>::fusion(const Ensemble<T> & autre) const {
    Ensemble<T> result;
    int i = 0, j = 0;

    while (i < ens.taille() && j < autre.ens.taille()) {
        if (ens[i] < autre.ens[j]) {
            result.ens.ajouter(ens[i++]);
        } else if (autre.ens[j] < ens[i]) {
            result.ens.ajouter(autre.ens[j++]);
        } else {
            result.ens.ajouter(ens[i++]);
            j++;
        }
    }

    // Rajouter les element restant des deux ensembles
    while (i < ens.taille())
        result.ens.ajouter(ens[i++]);
    while (j < autre.ens.taille())
        result.ens.ajouter(autre.ens[j++]);

    return result;
}
```

`Ensemble inter(const Ensemble & autre) const;` **O(m + n)**

Dans le pire des cas, les deux ensembles sont différents et ont des éléments distincts. Par conséquent, le nombre maximum de comparaisons nécessaires pour trouver l'intersection de deux ensembles de tailles m et n est de m + n.

La boucle principale parcourt les deux ensembles simultanément en comparant les éléments de chaque ensemble à chaque itération. Si les éléments sont égaux, l'élément est ajouté au résultat. Sinon, l'indice d'un des tableaux est incrémenté pour continuer la comparaison. Par conséquent, la complexité temporelle de cette méthode est de O(m + n), où m et n sont les tailles des deux ensembles sur lesquels on cherche l'intersection.

```cpp
Ensemble<T> Ensemble<T>::inter(const Ensemble<T> & autre) const {
    Ensemble<T> result;

    int i = 0, j = 0;
    while (i < ens.taille() && j < autre.ens.taille()) {
        if (ens[i] < autre.ens[j]) {
            i++;
        } else if (autre.ens[j] < ens[i]) {
            j++;
        } else {
            result.ens.ajouter(ens[i]);
            i++;
            j++;
        }
    }

    return result;
}
```

Bonus:

À chaque fois que le nombre d'éléments atteint la capacité, on double la taille du tableau et on re-affecte chaque élément dans le tableau, car on re-copie tout les élément de l'ancien tableau au nouveau. Sachant que la taille initiale est de 4, par exemple si on insere 4 éléments nous 4 affectations. Pour 5 élément, par contre, nous aurons 4 + 4 + 1 = 9 affectations. Pour 8, nous aurons 12 affectations. Pour 9, nous aurons 12(nombre d'affectations précédent) + 8(longueur du tableau précédent) + 1(l'élément 9) = 21.

Pour ce qui est de la formule ça doit genre être une série géométrique.

Aussi 256 * 256 * 256 * 2 – 4 = 33554428.