Complexité de « recette » :

```
void Stock::ajout_recette(const std::string &nomrecette, const Recette &recette) {
    Recette nouvelle_recette(nomrecette);
    for (auto it = recette.get_ingredients().debut(); it != recette.get_ingredients().fin(); ++it) {
        nouvelle_recette.ajout_ingredient(it.cle(), it.valeur());
    }
    recettes[nomrecette] = nouvelle_recette;
}
```

```
void Recette::ajout_ingredient(const std::string &nomingredient, int quantite) {
    if (ingredients.contient(nomingredient)) {
        ingredients[nomingredient] += quantite;
    } else {
        ingredients[nomingredient] = quantite;
    }
}
```

Soient r le nombre de recettes dans un ArbreMap<std ::string, Recette> de recettes, n le nombre d'ingrédients dans l'ArbreMap<std ::string, int> d'ingrédients d'une recette.

1. Le nom de la recette est ajouté dans l'arbre des recettes. → *O(log r)*
2. A) La boucle for parcourt chaque ingrédient de la recette. → *O(n)*
   B) L'ingrédient est ajouté dans l'arbre des ingrédients de la recette. → *O(log n)*
   → *O(n.log n)*

Ainsi, la complexité totale de recette est de *O(log r + n.log n)*.

Complexité de « ajout » :

```
void Stock::ajout(std::string nomingredient, std::string date, int nombre) {
    inventaire[nomingredient][date] += nombre;
}
```

Soient i le nombre d'ingrédients différents dans l'ArbreMap<std ::string, ArbreMap<std ::string, int>> d'un inventaire, d le nombre de dates de péremption associé à un ingrédient dans son ArbreMap<std ::string, int>.

1. On recherche au premier niveau de l'arbre. → *O(log i)*
2. On insère la date de péremption et la quantité ajoutée dans le sous-arbre. → *O(log d)*

Ainsi, la complexité totale d'ajout est de *O(log i + log d)*.

Complexité de « retrait » :

```
void Stock::retrait(std::string nomingredient, int nombre) {
    auto entrees = inventaire[nomingredient].debut();

    while (entrees) {
        if (nombre <= 0) break;
        if (entrees.valeur() <= nombre) {
            nombre -= entrees.valeur();
            entrees.valeur() = 0;
        } else {
            entrees.valeur() -= nombre;
            nombre = 0;
        }
        ++entrees;
    }
}
```

La commande « retrait » suit le même chemin que « ajout » mais retire des aliments après une recherche dans un arbre puis un sous-arbre.

Ainsi, la complexité totale de retrait est de *O(log i + log d)*.

Complexité de « recommandation » :

```cpp
std::string Stock::recommander_recette() const {
    std::string date_proche;
    std::vector<std::string> recettes_recommandees;

    for (auto it = recettes.debut(); it != recettes.fin(); ++it) {
        const std::string& nomrecette = it.cle();
        const Recette& recette = it.valeur();

        if (!realisable(recette)) {
            continue;
        }

        std::string date = dateExpiration(recette);

        if (date_proche.empty() || date < date_proche) {
            date_proche = date;
            recettes_recommandees.clear();
            recettes_recommandees.push_back(nomrecette);
        } else if (date == date_proche) {
            recettes_recommandees.push_back(nomrecette);
        }
    }

    std::string resultat;
    for (const auto& recette : recettes_recommandees) {
        if (!resultat.empty()) {
            resultat += " ";
        }
        resultat += recette;
    }

    return resultat;
}
```

```
std::string Stock::dateExpiration(const Recette &recette) const {
    std::string date_proche;
    auto it = recette.get_ingredients().debut();

    while (it) {
        if (inventaire.contient(it.cle())) {
            auto sous_it = inventaire[it.cle()].debut();
            while (sous_it) {
                auto date = sous_it.cle();
                if (date_proche.empty() || datePlusProche(date, date_proche)) {
                    date_proche = date;
                }
                ++sous_it;
            }
        }
        ++it;
    }
    return date_proche;
}
```

```
bool Stock::realisable(const Recette &recette) const {
    for (auto it = recette.get_ingredients().debut(); it != recette.get_ingredients().fin(); ++it) {
        const auto& nomingredient = it.cle();
        int qte_necessaire = it.valeur();
        int qte_disponible = 0;

        if (!inventaire.contient(nomingredient)) {
            return false;
        }

        auto sous_it = inventaire[nomingredient].debut();

        while (sous_it) {
            qte_disponible += sous_it.valeur();
            if (qte_disponible >= qte_necessaire) break;
            ++sous_it;
        }

        if (qte_disponible < qte_necessaire) {
            return false;
        }
    }
    return true;
}
```

1. On parcourt toutes les recettes de l'ArbreMap<std ::string, Recette>. → *O(r)*
2. On parcourt dans dateExpiration chaque ingrédient d'une recette et toutes les dates associées à cet ingrédient. → *O(n.d)*
3. On parcourt dans realisable chaque ingrédient d'une recette et chaque instance d'un ingrédient dans l'inventaire. → *O(n.d)*

Ainsi, la complexité de recommandation est de *O(r.n.d)*.

Complexité de « utilisation » :

```
bool Stock::utiliser_recette(const std::string &nomrecette) {
    if (!recettes.contient(nomrecette)) return false;

    const Recette& recette = recettes[nomrecette];

    if (!realisable(recette)) return false;

    return utilisation(recette);
}
```

```
bool Stock::utilisation(const Recette &recette) {
    auto it = recette.get_ingredients().debut();
    while (it) {
        const auto& nomingredient = it.cle();
        int qte_necessaire = it.valeur();
        auto sous_it = inventaire[nomingredient].debut();

        while (sous_it) {
            if (qte_necessaire <= 0) break;

            if (sous_it.valeur() >= qte_necessaire) {
                sous_it.valeur() -= qte_necessaire;
                qte_necessaire = 0;
            } else {
                qte_necessaire -= sous_it.valeur();
                sous_it.valeur() = 0;
            }
            ++sous_it;
        }

        ++it;
    }
    return true;
}
```

1. On vérifie l'existence de la recette sur un arbre AVL. → *O(log r)*
2. On accède à une recette avec comme clé son nom dans un arbre map. → *O(log r)*

3. La fonction realisable a déjà été analysé. → *O(n.d)*
4. Dans utilisation, on parcourt encore une fois chaque ingrédient d'une recette et chacune des dates associés à chaque ingrédient. → *O(n.d)*

Ainsi, la complexité d'utilisation est de *O(log r + n.d) = O(n.d)*.

Complexité pour « affichage » :

```
void Stock::affichage() {
    for (auto it = inventaire.debut(); it != inventaire.fin(); ++it) {
        std::cout << it.cle() << " ";
        int total = 0;
        auto v = it.valeur().debut();
        while (v) {
            total += v.valeur();
            ++v;
        }
        std::cout << total << std::endl;
    }
    std::cout << ";" << std::endl;
}
```

1. On parcourt chaque ingrédient d'un inventaire. → *O(i)*
2. On parcourt chaque quantité associée à chaque date pour un ingrédient. → *O(d)*

Ainsi, la complexité totale d'affichage est de *O(i.d)*.