# Compte rendu de la séquence 4 - SkipList

Durant cette séquence, nous avons implémenté une SkipList, une structure de données probabiliste pour la gestion efficace de collections ordonnées. Le but était d'implémenter les opérations classiques de dictionnaire (insertion, recherche, suppression), ainsi qu'un système d'itérateur permettant de parcourir la liste dans les deux sens.S

## Implémentation



## Opérateurs constructeurs et destruction

### `skiplist_create`

Cette fonction crée une SkipList avec un nombre spécifié de niveaux.

```
SkipList* skiplist_create(int nblevels) {
    SkipList *list = unwrapMalloc(sizeof(SkipList) + sizeof(Node) +
sizeof(DoubleLink) * nblevels);
    /*assignation des diférents membres de la skiplist*/

    for (int i = 0; i < nblevels; i++) {
        /*assignation des lien doublement chaînés vers la sentinelle*/
    }

    return list;
}
```

**Choix d'implémentation**

La sentinelle est initialisée au début de la liste avec des liens pointant vers elle-même pour tous les niveaux. Cela simplifie la gestion des cas limites. Une allocation contiguë est utilisée pour minimiser la fragmentation mémoire et améliorer l'efficacité des accès.

**Gestion de la mémoire**

Une seule allocation est réalisée pour la structure principale et la sentinelle. Cela permet de simplifier le processus de libération dans la fonction de destruction.

### `skiplist_delete`

Cette fonction libère la mémoire utilisée par la SkipList et ses nœuds.

```
void skiplist_delete(SkipList** d) {
    Node *current = node_next((*d)->sentinel);
    while (current != (*d)->sentinel) {
        /*Suppression de tous les noeuds un par un*/
    }

    free(*d);
    *d = NULL;
}
```

**Choix d'implémentation**

La liste est parcourue en commençant par le premier nœud après la sentinelle, et chaque nœud est supprimé individuellement avant de libérer la structure principale.

**Gestion de la mémoire**

La fonction utilise un seul free pour libérer la mémoire allouée pour la SkipList, conformément à la conception d'une allocation contiguë.

## Opérateurs

### skiplist_size

Cette fonction retourne la taille actuelle de la SkipList.

```
unsigned int skiplist_size(const SkipList *d) {
    return d->size;
}
```

**Choix d'implémentation**

Le champ size dans la structure SkipList est mis à jour à chaque insertion ou suppression, ce qui permet un accès en temps constant pour connaître le nombre d'éléments.

**Complexité**

La conmpléxiter est en O(1), comme demander dans l'exercice.

### skiplist_at

Cette fonction retourne la valeur de l'élément à une position donnée.

```
int skiplist_at(const SkipList* d, unsigned int i) {
    return skiplist_node_at(d, i)->val;
}
```

**Choix d'implémentation**

Utilisation de la fonction auxiliaire skiplist_node_at pour parcourir les nœuds jusqu'à la position spécifiée. Garantit que les indices sont parcourus séquentiellement à partir du premier niveau, ce qui est adapté pour des accès peu fréquents.

**Complexité**

O(n), avec n la position de l'élément.

### skiplist_search

Cette fonction cherche une valeur dans la SkipList et retourne un booléen indiquant si la valeur est trouvée. Elle comptabilise aussi le nombre de nœuds visités.

```
bool skiplist_search(const SkipList* d, int value, unsigned int
*nb_operations) {
    Node *cur_pos = d->sentinel;
    int level_pos = d->sentinel->level - 1;

    *nb_operations = 0; // Initialisation du compteur

    while (level_pos >= 0) {
        Node *next = node_nth_next_node(cur_pos, level_pos);

        if (next == d->sentinel || next->val > value) {
            level_pos--; // Descente d'un niveau
        } else if (next->val < value) {
            cur_pos = next; // Avancée dans le même niveau
            (*nb_operations)++;
        } else {
            (*nb_operations)++;
            return true; // Valeur trouvée
        }
    }

    return false; // Valeur absente
}
```

**Choix d'implémentation**

L'algorithme suit une descente probabiliste à partir du niveau le plus haut jusqu'au niveau 1, en sautant les nœuds inutiles, ce qui optimise la recherche. Le compteur nb_operations est mis à jour pour suivre les performances et le nombre de noeuds visités.

**Complexité**

- Moyenne : O(logN)

- Pire cas : O(N), si tous les niveaux convergent vers un seul chemin.

### skiplist_insert

Cette fonction insère une nouvelle valeur dans la SkipList.

```
SkipList* skiplist_insert(SkipList* d, int value) {
    Node *to_insert_after[d->sentinel->level];
    Node *cur_pos = d->sentinel;
    int level_pos = d->sentinel->level - 1;

    /*Recherche du noeud de la même manière que search*/

    Node *new_node = create_node(value, rng_get_value(&d->rng) + 1);
    for (unsigned int i = 0; i < new_node->level; ++i) {
        /*Ajout des liens au noeuds nouvellement creer*/
    }

    d->size++;
    return d;
}
```

**Choix d'implémentation**

Les niveaux sont calculés aléatoirement à l'aide de rng_get_value. La position où insérer est calculée en parcourant les niveaux supérieurs en premier, pour minimiser les sauts inutiles.

#### Gestion des erreurs

Une duplication de clé est détectée et l'insertion est ignorée.

#### Complexité

- Moyenne : O(logN)
- Pire cas : O(N), si tous les niveaux convergent.

### skiplist_remove

Cette fonction supprime une valeur dans la SkipList.

```
SkipList* skiplist_remove(SkipList* d, int value) {
    Node *cur_pos = d->sentinel;
    int cur_level = d->sentinel->level - 1;

    /*Recherche du noeud, retourne la liste si le noeud n'est pas trouver*/

    for (unsigned int i = 0; i < cur_pos->level; ++i) {
        /*Supression des liens*/
    }
```

```
    d->size--;
    delete_node(&cur_pos);
    return d;
}
```

**Choix d'implémentation**

L'algorithme descend dans les niveaux pour localiser et supprimer la valeur. Les pointeurs des nœuds adjacents sont mis à jour pour préserver l'intégrité de la structure.

### Complexité

- Moyenne : O(logN)
- Pire cas : O(N)

## Map et itérateur

### Map

La fonction skiplist_map permet d'appliquer une fonction à chaque élément de la SkipList. Cela permet de manipuler les valeurs sans avoir à retirer et réinsérer les éléments un par un.

`skiplist_map`

```
void skiplist_map(const SkipList* d, ScanOperator f, void *user_data) {
    Node *cur_pos = node_next(d->sentinel);
    while (cur_pos != d->sentinel) {
        //On parcours tous les noeuds un par un
        f(cur_pos->val, user_data);
        cur_pos = node_next(cur_pos);
    }
}
```

**Choix d'implémentation**

Cette fonction parcourt tous les éléments de la liste, en appliquant la fonction f à chaque valeur. Le paramètre user_data permet de passer des informations supplémentaires à la fonction. L'utilisation de node_next permet de parcourir la liste de manière séquentielle. La liste étant doublement chaînée, il est possible de la traverser de manière efficace.

**Complexité**

O(n), car chaque élément de la liste est visité une fois.

### Itérateur

L'itérateur permet de parcourir la SkipList de manière séquentielle, soit dans le sens avant, soit dans le sens arrière. L'implémentation fournit des fonctions pour créer, déplacer et accéder aux éléments de la liste via un itérateur.

### `skiplist_iterator_create`

Cette fonction crée un itérateur pour la SkipList. L'itérateur peut être configuré pour parcourir la liste dans l'une des deux directions : avant (FORWARD_ITERATOR) ou arrière (BACKWARD_ITERATOR).

```
SkipListIterator* skiplist_iterator_create(SkipList* d, IteratorDirection
w) {
    SkipListIterator *result = unwrapMalloc(sizeof(SkipListIterator));
    result->direction = w;
    result->list = d;
    result->pos = node_next(d->sentinel); // Démarre à la première position
    return result;
}
```

#### Choix d'implémentation

L'itérateur est initialisé avec la direction choisie. En fonction de la direction, il est positionné au début de la liste `(node_next(d->sentinel))`, ce qui permet de commencer le parcours dans le sens spécifié. L'utilisation de unwrapMalloc garantit une allocation sécurisée de l'itérateur.

### `skiplist_iterator_delete`

Cette fonction supprime l'itérateur de la mémoire.

```
void skiplist_iterator_delete(SkipListIterator** it) {
    free(*it);
    *it = NULL;
}
```

**Choix d'implémentation :**

Une simple libération de mémoire est effectuée ici. La mise à NULL du pointeur évite des erreurs de segmentation si l'itérateur est utilisé après suppression.

### `skiplist_iterator_begin`

Cette fonction place l'itérateur au début de la liste (soit le premier élément, soit le dernier en fonction de la direction).

```
SkipListIterator* skiplist_iterator_begin(SkipListIterator* it) {
    it->pos = it->direction == FORWARD_ITERATOR
```

```
            ? it->list->sentinel->dl_tab[0].next
            : it->list->sentinel->dl_tab[0].previous;

    return it;
}
```

#### Choix d'implémentation

La direction choisie permet de déterminer si l'itérateur commence à l'élément suivant ou précédent par rapport à la sentinelle. Cette fonction ajuste simplement la position de l'itérateur en fonction de la direction.

### skiplist_iterator_end

Cette fonction vérifie si l'itérateur a atteint la fin de la liste (la sentinelle).

```
bool skiplist_iterator_end(SkipListIterator* it) {
    return (it->pos == it->list->sentinel);
}
```

#### Choix d'implémentation

L'itérateur est considéré comme arrivé à la fin lorsqu'il atteint la sentinelle, ce qui permet de contrôler la condition d'arrêt du parcours.

### skiplist_iterator_next

Cette fonction fait avancer l'itérateur d'une position dans la direction spécifiée.

```
SkipListIterator* skiplist_iterator_next(SkipListIterator* it) {
    it->pos = it->direction == FORWARD_ITERATOR
        ? it->pos->dl_tab[0].next
        : it->pos->dl_tab[0].previous;

    return it;
}
```

#### Choix d'implémentation

L'itérateur se déplace soit vers l'élément suivant, soit vers l'élément précédent en fonction de la direction. Cette fonction est essentielle pour avancer dans la liste de manière séquentielle.

### skiplist_iterator_value

Cette fonction permet d'accéder à la valeur de l'élément sur lequel l'itérateur est actuellement positionné.

```
int skiplist_iterator_value(SkipListIterator* it) {
    return it->pos->val;
}
```

#### Choix d'implémentation

L'itérateur retourne simplement la valeur de l'élément pointé par pos.

#### Recherche d'une valeur avec l'itérateur

Un utilisateur peut décider de chercher une valeur avec l'itérateur de la manière suivante :

```
SkipListIterator * e = skiplist_iterator_create(list,FORWARD_ITERATOR);
    for (e = skiplist_iterator_begin(e);
        !(skiplist_iterator_end(e) || skiplist_iterator_value(e) == value);
        e = skiplist_iterator_next(e));
    printf("value %d found :%s",value ,skiplist_iterator_value(e) == value
? "true" : "false);
```

Ceci est beaucoup moins efficace que la recherche avec la fonction `skiplist_search`. La compléxiter dans ce cas est de O(n), avec n la longeurs de la liste, alors qu'avec `skiplist_search` nous avons une compléxiter de O(log(n)) dans le cas moyen (il est rare d'avoir une compléxiter en O(n)). Une différence de temps significatif est donc présente entre les deux façons de procéder.

## Fonction auxiliaire

### unwrapMaloc

Pour éviter les comportements non définis qui peuvent subvenir quand une allocation mémoire échoue avec la fonction `malloc`

Une fonction `unwrapMalloc` est définie.

```
void  *  unwrapMalloc(size_t  size){
void  * result =  malloc(size);

if (!result){
  fprintf(stderr,"could not initialize %ld bytes into the heap\n",size);
  perror("stopping program ");
  exit(1);
 }
 return result;
}
```

Elle fait s'arrêter le programme si le malloc échoue, en nous donnant un message d'erreur.