Rapport du ALGAV

Auteur : ZHOU runlin - 28717281

Introduction au dossier:

- définition des structures : setKey.py, tasPriporite.py, fileBinomial.py, ArbreRecherche.py. Chaque ficher a un fonction main pour lancer une petite exemple du programme, exécute ***python3 file_name*** pour voir la résultat.
- tabHachage.py : le ficher qui implémente MD5 (Exec 4)
- test.py : le ficher pour répondre des questions dans la partie 6 **“Étude expérimentale”**

Structure : Key

Les clés sont des entiers codés sur 128 bits. On peut utiliser n entiers pour les représenter. Nous pouvons traiter la clé comme une chaîne binaire de longueur 128, donc chaque entier représente une chaîne binaire de 128/n bits convertie en décimal.

Dans le structure, il y a deux champs:

- data : le tableau du *type int* qui stocke les entiers
- size: la taille du tableau

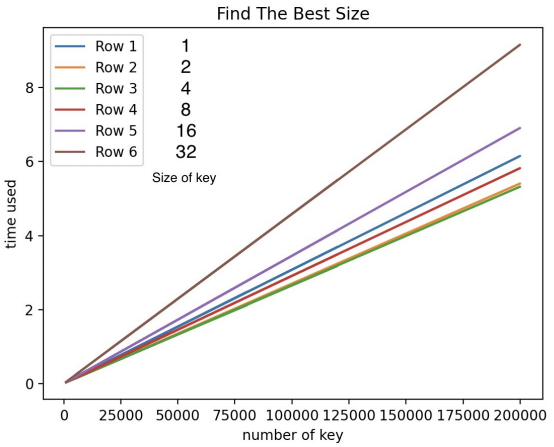
Pour la classe Key, on définie certaines méthodes pour gérer les objets :

- __init__ : Initialiser la taille de la clé avec *size* comme argument et initialiser data à un tableau vide
- storeNum(HexChar) / outputNum() : Transforme la chaîne en base 16 vers un tableau de int ou faire le contraire
- méthodes spéciales : lt, gt, eq, le, ge

En même temps, dans la fonction main du fichier setKey.py, nous effectuons un tests pour explorer la taille plus optimale de la classe *Key.*

L'expérience est mesurée par la somme du temps utilisé par trois étapes : initialisation du clé, remplissage des données et sortie des données, et les données du test proviennent de *cles_alea*.

L'expérience a passé un total de 6 groupes de tests, respectivement, la taille du clé égale à 1, 2, 4, 8, 16, 32. Et chaque groupe a collecté 7 points (nombre d'essai) : 1000, 5000, 10000, 20000, 50000, 80000, 120000, 200000.



Selon cette graphe, nous permet conclure que Key est la plus efficace lorsque size = 4

Structure : Tas priorité min

On réalise le tas priorité min sur la struct tableau.

- data : le tableau du tas
- size: la taille du tableau

Pour la classe TasTab, on définie certaines méthodes pour gérer les objets :

- Ajout(el) / AjoutInteratifs(listeEls) : Ajouter un / une liste de l'élément dans la queue de la tableau, et puis faire le fix-up
- SuperMin() : Supprimer le plus petit elements (data[0]), déplace d'abord le dernier élément du tas vers le sommet, puis faire le *fix-down*. La fonction revoie l'element supprimé.
- Construction(listeEls) : Initialiser le tas avec une liste de l'élément sans tirées. En partant du dernier nœud non feuille, l'opération *heaplify* est effectuée de manière séquentielle, en conservant la propriété de tas minimal.
- Union(otherTas) : Même idée avec Construction. Initialiser le tas avec l'union de deux listes, applique heaplify avec la même algorithme.

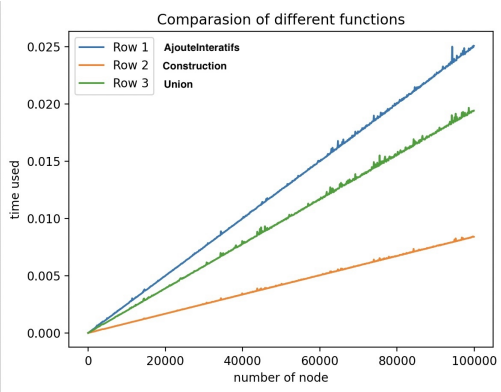
Les fonctions cores :

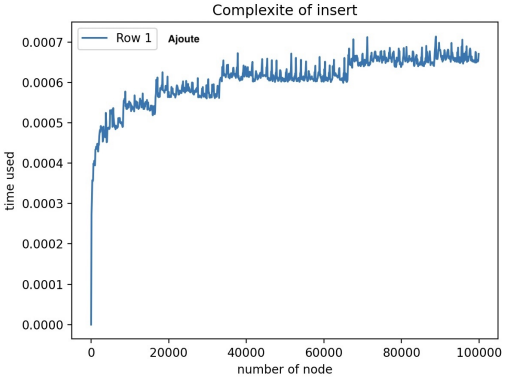
1. **opération fix-up :**

- À partir du dernier élément du tas (élément ajouté au tableau), on commence à ajuster vers le haut jusqu'à ce que la condition du *while* soit satisfaite.
- À chaque itération, on compare avec le nœud parent.
  - si la valeur du parent est inférieure à celle du nœud actuel, la boucle s'arrête
  - sinon : met à jour l'index du nœud actuel avec le parent, et échange leurs positions
- La complexité du fonction est en $O(log(n))$

2. **operation heaplify :**
   - La fonction prend en argument le sommet du tas (index du root) à ajuster
   - - si la valeur du parent est inférieure à celle du nœud actuel (tous les trois élément suivis la règle du Tas propriété), et la fonction retourne 0
     - sinon, échange leurs positions et la fonction retourne 1
   - La complexité du fonction est en $O(1)$
3. **opération fix-down :**
   - On commence à ajuster le tas sur le sommet vers le bas à partir du sommet jusqu'à ce que la condition *while* soit satisfaite.
   - La complexité du fonction est en $O(log(n))$

Representation du résultat:

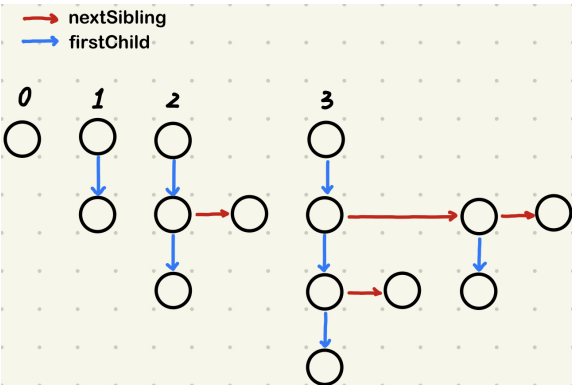




Structure : File Binomiales

Un *tournoi binomial* est un arbre binomial étiqueté croissant. Un arbre binomial est défini récursivement comme suit :

1. l'arbre d'ordre 0 est un seul nœud
2. l'arbre d'ordre *k* possède une racine de degré *k* et ses fils sont racine des arbres d'ordre *k*-1, *k*-2, … , 0.



Pour simplifier la structure, deux champs sont introduit :

- firstChild : le premier fils d'un nœud
- nextSibling : le premier frère d'un nœud

Les primitives du Tournoi :

- Degre(): la profondeur de la chaîne du *firstChild*,
- __str__(): entrer la chaîne de *nextSibling* du chaque *firstChild* pour parcourir tous les nœuds.
- Union2Tid(other): comparer la valeur de deux racines de l'arbre, on choisit la racine A (soit racine A est plus petit) comme la nouvelle racine. Le firstChild de A est la racine B, et le nextSibling de racine B est le firstChild de racine A.

Un *file binomiale* est composé par une suite de *tournois* de tailles strictement décroissantes. Autrement dit que un file binomiale est un forêt de tournois :

- maxSize : nombre de tournoi dans le file binomiale
- rootList : un tableau qui stocker tous les racines de tournoi. Pour initialiser un file binomiale, le rootList est définie comme un tableau remplir par "None", pour représenter un tableau vide.

Les fonctions cores du file binomial :

- Union(F1, F2): réaliser comme l'addition binaire
    1. Interclasser les 2 files en partant des tournois de degré minimum — DegMin().

2. Engendre un tournoi avec 2 tournois de même degré — Union2Tid(). À chaque étape au plus 3 tournois de même degré sont à fusionner
3. Lorsque 3 tournois de même degré *k*, on en retient 2 pour engendrer un tournoi de degré *k* + 1, et l'on garde le troisième comme tournoi de degré *k*.

- Ajout(el): Traiter l'élément comme un file binomial avec un seul élément, puis faire Union()
  Le meilleur cas : si FB_k avec k est un nombre paire, la complexité est en $O(1)$
  Le pire cas: si $FB_k =< TB_m, TB_{m-1}, ..., TB >$ avec $m = \lfloor log_2(k) \rfloor$, la complexité est en $O(logN)$

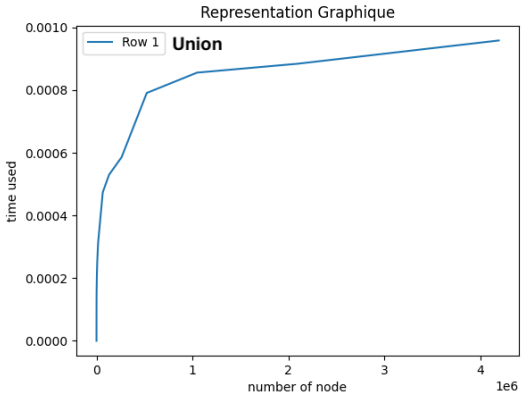
  Coût amorti: La probabilité d'obtenir un nombre pair est de $\frac{1}{2}$, alors que la probabilité d'obtenir $2^k - 1$ est très faible.
  Suppose qu'on faire l'insertion N fois. Où $\frac{N}{2}$ insertions ont un temps d'exécution de 1, $\frac{N}{4}$ insertions ont un temps d'exécution de 2, $\frac{N}{8}$ insertions (2 postes) ont un temps d'exécution de 3, et $\frac{N}{2^k}$ insertions ont un temps d'exécution de k. Donc, en total :

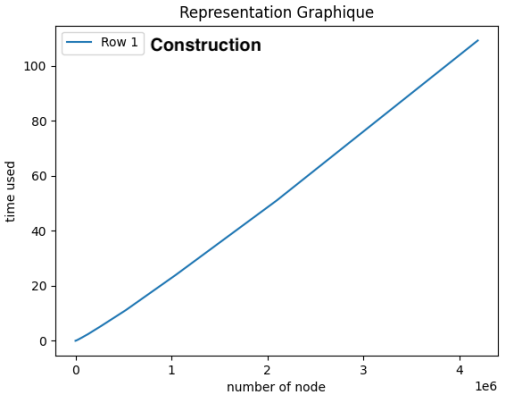
$$Nb_{op} = \sum_{i=0}^{k} i * \frac{N}{2^i} < N * \sum_{i=0}^{\infty} \frac{i}{2^i} = 2N$$

  Donc, le cout amorti de Ajout(el) est $2N/N = O(1)$

- Construction(listeEls): Ajouter les éléments dans listeEls itérativement. La complexité est en $O(N)$, car la fonctions appel N fois de Ajoute(el)

Representation du résultat:





Structure : Arbre de Recherche

Avant construit un arbre, on définie le noeud à l'avance (node):

- data : la valeur de cette noeud
- left : pointée vers le fils gauche du noeud
- right : pointée vers le fils droite du noeud
- height : la hauteur de noeud

Dans cette section, on veut réaliser un AVL (BTree):

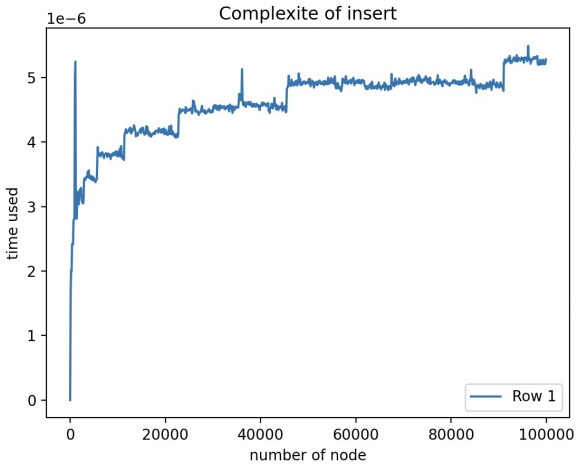
- root : qui correspond le sommet de l'arbre
- Insert(el) : Une fois que l'élément manipulé a trouvé sa position correcte, nous mettons à jour le hauteur de l'arbre et effectuons l'opération equilibre()
- Delete(el) : L'opération de suppression est similaire à l'insertion, sauf que lorsque l'élément supprimé est retrouvé (el == self.data) :
  - s'il n'existe pas le sous-arbre gauche : retourne sous-arbre droite
  - s'il n'existe pas le sous-arbre droite : retourne sous-arbre gauche
  - sinon :
    - si le fils gauche est plus haut que le droite, trouver l'élément le plus gauche (l'élément plus petite a gauche), on met cet élément comme la racine
    - si le fils droite est plus haut que le gauche, trouver l'element le plus droite (l'élément plus grande a droite), on met cet élément comme la racine
- equilibre(root, el) :
  - si facteur d'équilibre (left.height() - right.height()) est plus grand que 1:
    - si le nouveau nœud est inséré/ supprimé dans la sous-arbre droite de sous-arbore gauche et que le facteur d'équilibre de ce nœud est plus grand que 1 (el > left.data), effectuer le RotationGaucheDroite()
    - sinon, effectuer le RotationGauche()
  - si facteur d'équilibre est plus petit que -1:
    - si le nouveau nœud est inséré/ supprimé dans la sous-arbre droite de sous-arbore gauche et que le facteur d'équilibre de ce nœud est plus

grand que 1 (el < right.data), effectuer le RotationDroiteGauche()

- sinon, effectuer le RotationDroite()

- show() : en utilisent traversée hiérarchique, la fonction renvoie une liste de liste, remplie par les éléments de chaque niveau

Representation du résultat:



Fonction de hachage

Dans cette partie, on implémenterait **MD5**(écrire dans *tabHachage.py*). Nous utiliserons cette fonction de hachage pour compter le nombre de mots différents dans l'ouvre de Shakespeare et le nombre de collision.

Pour le réaliser, nous utiliserons la classe : *OrdredSet()* en python. La clé du set est un hachage de Md5 de ce mot, et la valeur est un tableau qui stocke tous les mots qui ont le même hachage valeur.

En conséquence, le nombre de mots différents est la taille de ce set, et le nombre de collision correspond le nombre de clé qui a au plus que un mot dans le tableau.

Representation du résultat:

```
finish process the book : Shakespeare/twelfth_night.txt
finish process the book : Shakespeare/two_gentlemen.txt
finish process the book : Shakespeare/winters_tale.txt
Nb of different mot : 23086
Nb of collision by using MD5 : 0
finish process the book : cles_alea/jeu_1_nb_cles_1000.txt
finish process the book : cles_alea/jeu_1_nb_cles_5000.txt
```

Conclusion :

Tas priorité min :

Ajoute : $O(log(n))$

Union : $O(n * log(n))$

Cons : $O(n)$

File Binomiales :

Ajoute : $O(log(n))$

Union : $O(log(n))$

Cons : $O(n)$

Arbre de Recherche :

Ajoute : $O(log(n))$

Suppr : $O(log(n))$