RAPPORT SDA PROJET

NOM ET PRENOM : LE Richard, PHYU THANT THAR Rémi, HENOUNE Asma, TAMAZIRT Sarah

LICENCE 2 – SEMESTRE 1

MATIERE : Structures de données

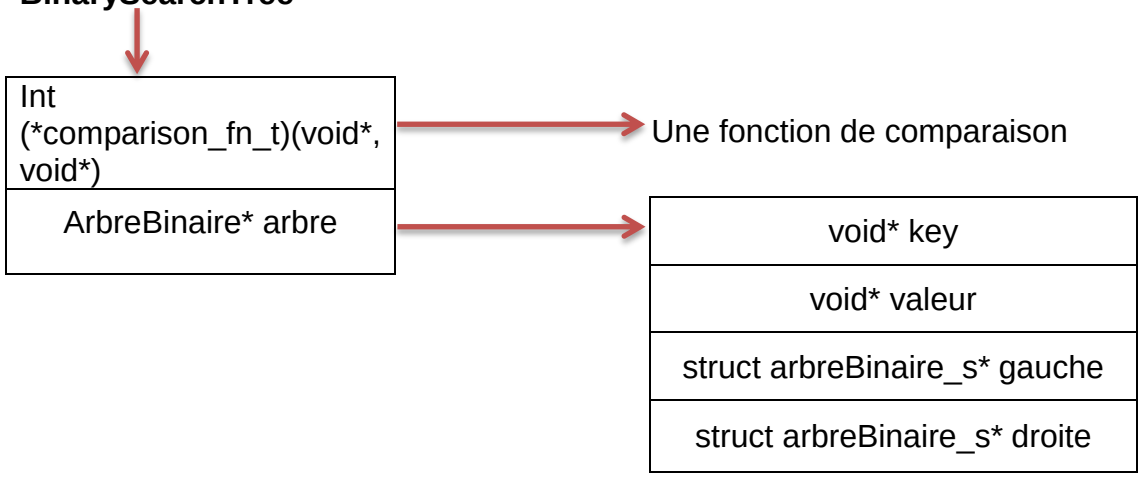
UNIVERSITE PARIS 13, INSTITUT GALILÉE

ANNÉE : 2017/2018

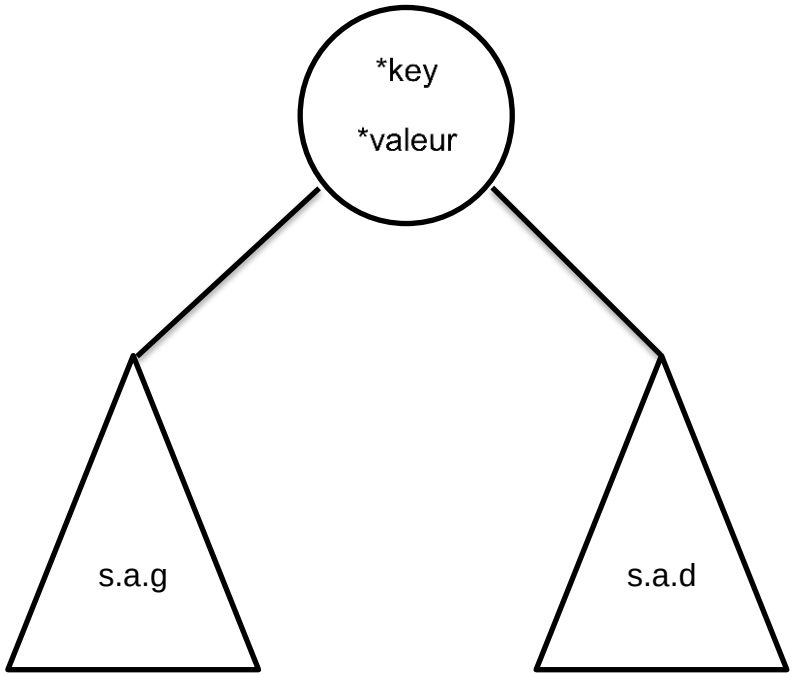
1) Choix d'implémentation :

Schéma des structures de données :

BinarySearchTree*



ArbreBinaire*



Pour les fonctions de construction de l'arbre binaire de recherche, on a dû en déclarer une autre pour faire les appelles récursif notamment. Par exemple, pour la fonction insertInBST, on a déclaré et implémenté la fonction insertArbreBinaire pour faciliter l'appelle récursif. Il suffira donc d'appeler la fonction InsertArbreBinaire dans insertInBST et le tour est joué.

On explique à la suite brièvement les 3 fonctions de recherche. Dans les 3 cas, il faudra au moins construire 1 arbre binaire de recherche et renvoyer une linkedList.

LinkedList* findCities1BST(LinkedList* ll, double latitudeMax, …){ … }

On construit un seul arbre binaire de recherche contenant toute les villes en fonction de leurs latitudes.

On extrait de l'arbre les villes comprises entre 2 latitudes (passé en paramètre) avec la fonction getInRange().

On filtre les villes extraites avec filterLinkedList(…).

LinkedList* findCities2BST(LinkedList* ll, double latitudeMax, …){ … }

On construit 2 arbres distincts, un en fonction de la latitude et l'autre de la longitude.

On extrait des 2 arbres de listes distincts avec getInRange.

On effectue l'intersection des 2 listes avec intersect(...).

LinkedList* findCitiesZBST(LinkedList* ll, double latitudeMax, …){ … }

On construit 1 arbre en fonction des zscore, grâce à la fonction zEncode().

On extrait de l'arbre les villes comprises entre 2 zscore avec getInRange.

On filtre pour récupérer seulement les bonnes villes filterLinkedList(…).

2) Pseudo code getInRange

LinkedList* ll = NewLinkedList() ;

void getInRangeArbreBinaire(…) /* appelle d'une sous fonction comme décrit juste avant */

if(arbre == NULL)

return ;

if(arbre->cle ∈ intervalle){

getInRangeArbreBinaire(arbre->gauche, …) ;

Traitement

getInRangeArbreBinaire(arbre->droite) ;

}

if(arbre->cle est en dessous de l'intervalle)

getInRangeArbreBinaire(arbre->droite, …) ;

sinon

getInRangeArbreBinaire(arbre->gauche,…) ;

Fin

Complexité : Soit N le nombre de nœuds. Dans le meilleur des cas, N == 0 donc la complexité est constante O(1). Dans le pire des cas, on parcours tout les noeuds de l'arbre, O(N).

5)Pseudo code intersect

while(ListeA != NULL){

while(ListeB != NULL){ Traitement; ListeB->next }

ListeA->next } ret LinkedList* l;

Fin

Complexité: Soit M et N la taille des 2 listes et M <= N. Dans le meilleur des cas, la première liste est NULL, M == 0 donc complexité constant O(1). Dans le pire des

cas, on parcours pour chaque élément de la première liste tous les éléments de la 2ème liste, donc complexité O(M*N) qui est une complexité quadratique. Cette complexité en temps est très mauvaise pour des nombres très grands (D'ailleurs on l'a vu lors des tests de temps de recherche).

6) Comparaison des trois recherches

Pour la comparaison des 3 recherches, on a pris K = 10 et testé avec 10000, 100000, 1000000 de villes. Quatre points sont tirés au hasard pour délimiter une zone.

Ici, il s'agit des résultats avec **un arbre NON AVL**.

(R1 = recherche1 ; R2 = recherche2 ; RZ = rechercheZ)

CONDITIONS	ARBRE NON AVL	**Nb villes: 10 000**	temps: secondes
K = 10	recherche1	rechercher2	rechercheZ
1	0,156 s	0,203 s	0,172 s
2	0,047 s	0,156 s	0,109 s
3	0,469 s	0,610 s	0,078 s
4	0,188 s	0,078 s	0,141 s
5	0,078 s	0,109 s	0,063 s
6	0,062 s	0,750 s	0,578 s
7	0,062 s	0,078 s	0,063 s
8	0,094 s	0,750 s	0,094 s
9	0,109 s	0,078 s	0,563 s
10	0,063 s	0,063 s	0,094 s
moyenne	0,133 s	0,288 s	0,196 s

	| |	**Nb villes: 100 000**	| |
K = 10	R1	R2	RZ
1	0,266 s	51,958 s	0,328 s
2	0,281 s	0,422 s	0,250 s
3	0,293 s	395,241 s	0,297 s
4	0,297 s	0,431 s	0,297 s
5	0,281 s	104,756 s	0,328 s
6	0,281 s	0,484 s	0,313 s
7	0,344 s	2,883 s	0,281 s
8	0,281 s	3,263 s	0,294 s
9	0,313 s	0,531 s	0,266 s
10	0,289 s	134,747 s	0,313 s
moyenne	0,293 s	69,472 s	0,297 s

	| |	**Nb villes: 1 000 000**	| |
K = 10	R1	R2	RZ
1	5,366 s	161,362 s	3,286 s
2	4,833 s	285,171 s	3,837 s
3	4,897 s	15,856 s	3,658 s
4	4,625 s	>1h	3,765 s
5	4,732 s	TROP LONG	3,458 s
6	4,680 s		3,377 s
7	4,699 s		3,431 s
8	4,462 s		3,483 s
9	4,410 s		3,344 s
10	4,929 s		3,277 s
moyenne	4,763 s		3,492 s

On n'a pas effectué les recherches pour 1000 villes, car les résultats ne seraient pas

intéressants. De plus, pour 1 000 000 de villes, la recherche2 (ici R2) est beaucoup trop long (l'ordinateur tourne pendant des heures, donc ce serais un peu exhaustif de le faire 10 fois) .

Soit n le nombre de villes. Pour n = 10000, la différence des moyennes de temps de recherches est négligeable. Pour n = 100 000, on peut noter que la recherche2 prend 230 fois plus de temps en moyenne que les deux autre recherches (69s contre ~0.3s). Pour n = 1 000 000, la différence est d'autant plus grande que ça devient inutile de faire la recherche. (>1h contre 4.763s et 3.492s) .

Pour la recherche1 et la rechercheZ(zscore) la différence n'est pas notable, mis à part quand n = 1 000 000. La recherche prend +25% plus de temps que rechercheZ.

D'après les résultats, on peut affirmer que la recherche2 est complètement obsolète quand n est très grand. Cela peut tout simplement s'expliquer par la complexité de la fonction intersect(), qui a une complexité quadratique. De plus la rechercheZ à l'air plus efficace (-25% en temps) ce qui n'est pas négligeable.

Ici, les résultats avec **un arbre AVL.**

Nb Villes: 10 000			
K = 10	R1	R2	RZ
1	0,109 s	0,109 s	0,109 s
2	0,047 s	0,938 s	0,078 s
3	0,078 s	0,141 s	0,063 s
4	0,078 s	0,094 s	0,109 s
5	0,063 s	0,078 s	0,078 s
6	0,078 s	0,078 s	0,062 s
7	0,047 s	0,109 s	0,047 s
8	0,078 s	0,125 s	0,078 s
9	0,078 s	0,078 s	0,078 s
10	0,047 s	0,063 s	0,063 s
moyenne	0,070 s	0,181 s	0,077 s

Nb Villes: 1 000 000			
K = 10	R1	RZ	R2
1	3,749 s	3,884 s	-
2	3,478 s	3,560 s	-
3	3,667 s	3,683 s	-
4	3,520 s	3,618 s	-
5	3,415 s	3,580 s	-
6	3,706 s	3,543 s	-
7	3,527 s	3,661 s	-
8	3,692 s	3,668 s	-
9	3,596 s	3,582 s	-
10	3,424 s	3,572 s	-
moyenne	3,577 s	3,635 s	-
hauteur	23	23	-
diff en %	**1,59%**		

Comme prévu, les résultats pour n = 10 000 sont très similaires quand l'arbre est NON AVL. Par contre, quand n = 1 000 000, R2 est toujours trop long, mais la différence en % des moyennes de R1 et RZ est très petite (1.59%). On peut en déduire qu'en fait la différence de temps n'est dû qu'à la différence de hauteur des arbres, qui peut ralentir la recherche. Donc RZ n'est pas plus rapide que R1, il permet juste une construction d'arbre plus équilibré, qui est plus efficace pour effectuer des recherches.

Pour résumé, la recherche2 est inefficace quand n est très grand et qu'il peut y avoir des différences d'efficacités seulement à cause de la façon dont on construit l'arbre.