Pràctica 2

Estructures de dades no lineals. Arbres.

Niub – Nom
16799506 - Rubén Ballester Bautista
16856733 – Oriol Rabasseda Alcaide
Grup de Pràctiques F **– Parella** 1
Professora:
Maria Salomó

Exercici 1

Explicació exercici: aquest exercici consisteix en el desenvolupament del TAD Binary Search Tree que serveix per tenir ordenats un seguit de nombres (o qualsevol altre tipus d'objecte que pugui ser representat amb nombres) en un arbre binari, d'aquesta manera, la seva cerca és O(log n) (i no O(n) com en una llista).

Per a desenvolupar aquest TAD també és necessari crear el node amb el que seran representats tots els nombres, aquest node s'anomena Position.

Finalment, la dada més rellevant de la implementació és que ha sigut implementada amb templates per tal de fer més genèric l'ús d'aquest TAD.

TAD Position:

Representació:

- Position<E>* leftP Punter al fill esquerre.
- Position<E>* rightP; Punter al fill dret.
- Position<E>* parentP; Punter al pare de l'element..
- E elementP Element que representa la posició de l'arbre.

Utilitzem aquesta representació per tenir a mà sempre tots els elements relacionats amb el node seleccionat. La inclusió del pare és opcional encara que per poder realitzar amb més facilitat les operacions de rebalanceig (en un futur) és preferible tenir-ho ja inclòs.

El cost d'aquesta representació és d'O(n).

TAD Binary Search Tree:

Especificació e implementació (Mètodes).

- const int size() const – Cost O(n) Recursivament ha de visitar tots els nodes.
- const int height() const – O(n) Recorre tots els nodes possibles.
- bool empty() const – O(1) Comprova únicament si existeix arrel.
- const Position<E>* root() const – O(1) Retorna un punter
- void insert (const E& element) – Cas promig O(log n) Ja que el arbre promig de cerca binaria té altura de O(log n).
- bool search (const E& element) const – Cas promig O(log n) Ja que l'arbre promig de cerca binaria té altura de O(log n).
- void printPreorder() O(n).
- void printPostorder() O(n).
- void printInorder() O(n).

Els recorreguts fan una visita a tots els nodes.
No s'ha creat cap mètode extra, tret del fet que molts mètodes s'han sobrecarregat per tal de fer possible que en les crides recursives la primera és faci directament amb el root sense necessitat que

l'usuari l'introdueixi específicament (per exemples els recorreguts o el size).

Exercici 2

Explicació de l'exercici: basant-nos en el TAD creat a l'exercici 2, cal implementar un buscador de paraules (omplir l'arbre de l'exercici anterior amb un conjunt de paraules que s'ordenen alfabèticament). Els textos són donats i l'algorisme del main consisteix en comparar les paraules d'un diccionari amb les trobades en el text (que es troben a l'arbre).

La classe que implementa les accions que es pot fer amb l'arbre de cerca binària no és template perquè no tindria cap sentit ja que, per definició de l'estructura, busca només paraules.

Per altra banda, hi ha un punt important a afegir, hem optat per la representació de les ocurrències mitjançant parells de ints representats a C++ amb el TAD Pair<class E, class N>. D'aquesta manera exprimim al màxim els recursos de C++.

Exercici 3

Els mètodes públic que es troben en l'arbre balancejat i l'arbre binari de cerca, són els mateixos, bàsicament no varia la implementació de cap mètode a excepció de 2, la inserció i el height. Tot i que el nostre codi estigui modularitzat i el insert pugui semblar igual, s'afegeixen 5 mètodes privat més que seran usats durant la inserció per rebalancejar l'arbre en cas que l'alçada entre el fill dret i esquerre d'un node sigui 2 (o més, encara que no és possibles si es rebalanceja a cada inserció). Aquests mètodes consisteixen en buscar (i en cas que hi hagi trobar) si hi ha o no algun node que calgui rebalancejar, analitzar quin tipus de rebalanceig cal fer i executar-lo. Aquesta modificació fa que l'arbre tingui la mínima alçada possible (sigui complert o quasi complert) però continuï mantenint el seu ordre.

Per altre banda, en el height varia el fet que, en guardar en cada node quina alçada té en l'arbre, només cal agafar l'alçada del root per trobar l'alçada total de l'arbre. Això millora substancialment el rendiment d'aquest mètode

Cost computacional:

- const int size() const – Cost O(n) Recursivament ha de visitar tots els nodes.
- const int height() const – O(1) Retorna l'alçada guardada en el root.
- bool empty() const – O(1) Comprova únicament si existeix arrel.
- const Position<E>* root() const – O(1) Retorna un punter
- void insert (const E& element) – Sempre O(log n) Ja que el arbre sempre és troba complert o quasi complert (no hi ha arbres com llistes).
- bool search (const E& element) const – Sempre O(log n) Ja que el arbre sempre és troba complert o quasi complert (no hi ha arbres com llistes).
- void printPreorder() O(n).
- void printPostorder() O(n).

- void printInorder() O(n).

Exercici 4:

La implementació d'aquest exercici és pràcticament la mateixa que en l'exercici 2 ja que els mètodes públics que usa el TAD BSTWordFinder són els mateixos que podrà usar el TAD BalancedTreeWordFinder, per tant, no calrà variar gaire cosa.

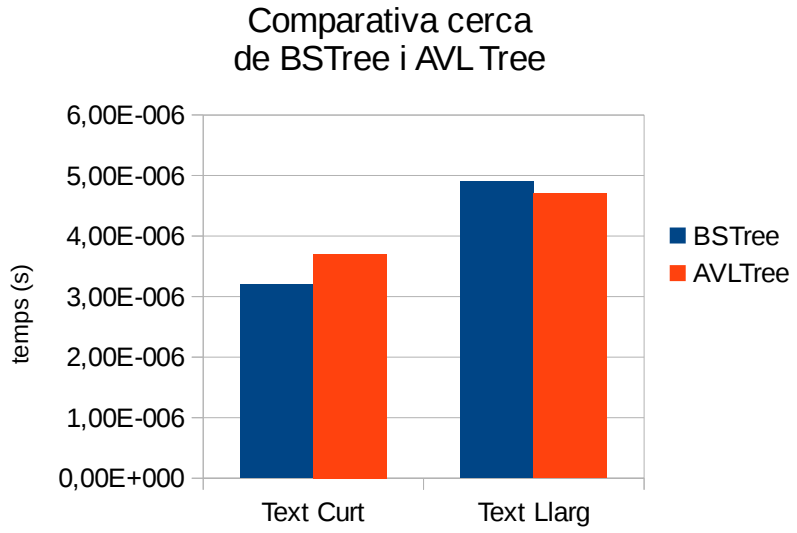
Exercici 5

Hem fet 10 proves de cada cas per obtenir uns resultats el màxim de fiables (les proves es van fer en el mateix ordinador i en execucions del programa diferents). Les proves (tant de cerca com de creació) es fan sobre els dos tipus d'arbre i sobre un el text llarg i el curt donats.

Tests en la cerca:

Cerca	BSTree		AVLTree	
	Text Curt	Text Llarg	Text Curt	Text Llarg
Test 1	4,00E-006	5,00E-006	4,00E-006	5,00E-006
Test 2	3,00E-006	5,00E-006	3,00E-006	5,00E-006
Test 3	4,00E-006	5,00E-006	3,00E-006	4,00E-006
Test 4	3,00E-006	5,00E-006	4,00E-006	5,00E-006
Test 5	2,00E-006	5,00E-006	4,00E-006	5,00E-006
Test 6	3,00E-006	5,00E-006	4,00E-006	4,00E-006
Test 7	4,00E-006	5,00E-006	3,00E-006	5,00E-006
Test 8	3,00E-006	5,00E-006	4,00E-006	5,00E-006
Test 9	4,00E-006	5,00E-006	4,00E-006	4,00E-006
Test 10	2,00E-006	4,00E-006	4,00E-006	5,00E-006
Resultats	3,20E-006	4,90E-006	3,70E-006	4,70E-006

Gràfic comparatiu:

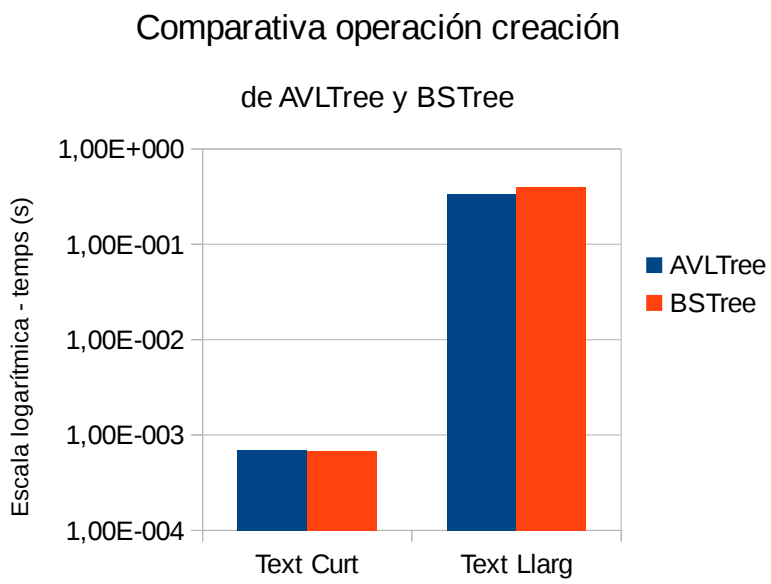


Aquest gràfic mostra com, en textos més llargs el temps computacional de l'arbre AVL tarda una mica menys però, en textos curts el BST va funcionar millor. Cal recordar per a explicar aquest gràfic que el cost teòric dels mètodes en els dos TAD és O(log n) (el mateix), però també cal tenir present que el cost O(log n) en l'arbre BST es dona només en el cas promig, ja que en el pitjor cas es té un cost O(n), això pot haver influït en el fet que la cerca en textos més llargs sigui pitjor.

A més, també influeix la posició de la paraula buscada (en tots els casos va ser «such»).
Test de creació:

Creació	AVLTree		BSTree	
	Text Curt	Text Llarg	Text Curt	Text Llarg
Test 1	6,93E-004	3,42E-001	7,38E-004	4,03E-001
Test 2	6,44E-004	3,41E-001	6,38E-004	4,05E-001
Test 3	6,50E-004	3,40E-001	6,83E-004	4,03E-001
Test 4	6,45E-004	3,39E-001	7,04E-004	4,04E-001
Test 5	7,45E-004	3,39E-001	6,95E-004	4,05E-001
Test 6	7,28E-004	3,40E-001	6,59E-004	4,02E-001
Test 7	6,64E-004	3,39E-001	6,69E-004	4,04E-001
Test 8	7,75E-004	3,39E-001	6,67E-004	4,03E-001
Test 9	6,73E-004	3,40E-001	6,72E-004	4,03E-001
Test 10	6,69E-004	3,39E-001	6,34E-004	4,05E-001
Resultats	6,89E-004	3,40E-001	6,76E-004	4,04E-001

Gràfic comparatiu:



A simple vista ja es pot veure que la creació és un procés més costós que la cerca. El temps en tots dos casos (comparativament en el gràfic) és molt semblant. Tot i això, és pot veure una petita variació en textos llargs, on el BST triga més atès que, la inserció d'un sol element pot ser major que

O(log n) (fins O(n)), i en l'AVL sempre és O(log n).

En textos curts és pot donar el procés invers ja que el cost del balanceig (encara que sigui O(1)), pot influir en la inserció (de forma petita però perceptible per textos curts).

El cost teòric del procés de creació de l'arbre (en tots dos casos) és O(nlog n), ja que s'han d'inserir n elements de cost O(log n) cadascun.