UNIVERSITAT DE BARCELONA

FACULTAT DE MATEMÀTIQUES I INFORMÀTICA

MEMÒRIA DEL TERCER LLIURAMENT DE PRÀCTIQUES:

ARBRES BINARIS

ESTRUCTURA DE DADES

Gabriel Marín Sánchez, NIUB 16617215
Martí Pedemonte Bernat, NIUB 16638016

Professora: Maria Salamó
Grup C - Parella 4

13 de maig de 2018

Comentaris de la pràctica

Exercici 1 - Arbre binari de cerca

Observacions i decisions preses

En aquest exercici hem decidit no fer un menú i testejar els mètodes un a un de manera consecutiva, amb un arbre arbitrari. No obstant, aquest procediment l'hem dut a terme per més d'un arbre, per assegurar-nos que tots els mètodes funcionaven correctament.

Qüestions

- **Justifiqueu el cost computacional teòric de les funcions del TAD BinarySearchTree i del TAD NodeTree. Si implementeu altres mètodes necessaris pel desenvolupament de la pràctica, justifiqueu-ne el seu ús i el seu cost computacional teòric.**
  Els mètodes auxiliars que hem afegit els utilitzem per implementar el mètode de manera recursiva. Per exemple en el mètode insert, tenim un mètode auxiliar que rep com a paràmetre un node. Aleshores comprova si el nou node hauria d'anar a l'esquerra o a la dreta i torna a cridar el mètode per al node fill que correspongui.

  - **Constructors:**
    - BinarySearchTree(): Només assigna valors, així que té cost constant, O(1).
    - BinarySearchTree(const **BinarySearchTree**& orig): El seu cost és el mateix del seu mètode auxiliar constructor_copia, ja que simplement crida aquest mètode.
  - **Destructor:**
    - *virtual* ∼BinarySearchTree(): El seu cost és el mateix del seu mètode auxiliar postDelete, ja que simplement crida aquest mètode.
  - **Consultors:**
    - *int* size(): El seu cost és el mateix del seu mètode auxiliar size, ja que simplement crida aquest mètode.
    - *bool* isEmpty(): Només comprova si **pRoot** és nul, per tant té cost constant, O(1).
    - **NodeTree**<**Type**>* root(): Simplement retorna el node arrel, així que té cost constant, O(1).
    - *bool* search(const **Type**& element): El seu cost és el mateix del seu mètode auxiliar search, ja que simplement crida aquest mètode.
    - *void* printInorder() const: El seu cost és el mateix del seu mètode auxiliar printInorder, ja que simplement crida aquest mètode.
    - *void* printPreorder() const: El seu cost és el mateix del seu mètode auxiliar printPreorder, ja que simplement crida aquest mètode.
    - *void* printPostorder() const: El seu cost és el mateix del seu mètode auxiliar printPostorder, ja que simplement crida aquest mètode.

- ▷ *int* getHeight(): El seu cost és el mateix del seu mètode auxiliar getHeight, ja que simplement crida aquest mètode.

– **Modificadors:**

- ▷ *void* insert(const **Type**& element): El seu cost és el mateix del seu mètode auxiliar insert, ja que simplement crida aquest mètode.

– **Mirall:**

- ▷ **BinarySearchTree**<**Type**>* mirror(): El seu cost és el mateix del seu mètode auxiliar constructor_mirall, ja que simplement crida aquest mètode.

– **Mètodes privats interns. Són mètodes auxiliars:**

- ▷ *void* postDelete(**NodeTree**<**Type**>* p): Mètode recursiu que recorre l'arbre en post ordre i n'elimina els nodes. Com que recorre tots els nodes el seu cost és O(n).
- ▷ *int* size(**NodeTree**<**Type**>* p) const: Com abans, es recorren tots els nodes. Cost O(n).
- ▷ *void* printPreorder(**NodeTree**<**Type**>* p) const: Mateix cas, O(n).
- ▷ *void* printPostorder(**NodeTree**<**Type**>* p) const: Mateix cas, O(n).
- ▷ *void* printInorder(**NodeTree**<**Type**>* p) const: Mateix cas, O(n).
- ▷ *int* getHeight(**NodeTree**<**Type**>* p): Mateix cas, O(n).
- ▷ *void* insert(**NodeTree**<**Type**>* p, const **Type**& element): Mètode recursiu auxiliar per a insertar un node. En general recorrerà un nombre de nodes igual a l'alçada (log n), però pot ser que els hagi de recorrer tots, per tal com està construit l'arbre binari. Per tant el cost és O(n).
- ▷ **NodeTree**<**Type**>* search(**NodeTree**<**Type**>* p, const **Type**& element): Mateix cas que abans. Cost O(n).
- ▷ **NodeTree**<**Type**>* constructor_copia(**NodeTree**<**Type**>* from): Mètode recursiu auxiliar per al constructor còpia. Recorre tot els nodes de l'arbre del qual es vol fer la copia, així que té cost lineal O(n).
- ▷ **NodeTree**<**Type**>* constructor_mirall(**NodeTree**<**Type**>* from): Similar al d'abans. Cost O(n).

Exercici 2 - Cercador de pel · lícules amb arbres de cerca binària

Observacions i decisions preses

Per a dur a terme aquesta part de l'entrega hem agafat les classes programades a l'exercici 1 i hem afegit les classes **Movie** i **BSTMovieFinder**. La segona contindrà l'arbre binari, format per nodes amb pel · lícules. Hem adaptat les classes **BinarySearchTree** per tal que disposin d'una clau i fer més simple l'ús de la classe **Movie**. Per a realitzar la funció de lectura de fitxers, com que cada 40 elements volem preguntar a l'usuari si segueix interessat en llegir-ne més, hem afegit un comptador a la classe **BinarySearchTree** que es va actualitzant cada vegada que s'imprimeix un element, és a dir, que també hem modificat lleugerament el mètode printInorder.

Exercici 3 - Arbres binaris balancejats

Observacions i decisions preses

Respecte els exercicis anteriors essencialment hem canviat tan sols el mètode insert i n'hem afegit de complementaris per a realitzar les rotacions. Inicialment hem seguit l'estructura utilitzada anteriorment, però degut a que obteniem errors hem decidit canviar el mètode i fer que aquesta vegada retornès un node, modificant lleugerament com estava plantejat el mètode insert original.
Respecte les funcions de saber el títol més llarg i la puntuació més baixa [alta] i les pel · lícules amb aquesta puntuació, hem modificat el mètode toString de **Movie** perquè retornés el títol i la puntuació en un mateix *string*. D'aquesta manera, com que la puntuació té sempre la mateixa longitud, no influia en saber el títol més llarg. Pel que fa a les puntuacions, primer hem fet mètodes per buscar recursivament dins l'arbre la millor [pitjor] puntuació, i un cop la teniem hem creat un mètode recursiu que imprimia per pantalla les pel · lícules amb aquella mateixa puntuació.
L'alternativa a aquesta implementació era deixar d'implementar el TAD amb template, cosa que ens ha semblat innecessari i no gaire estètic, ja que volíem que els nostres TAD's tinguessin la major generalitat possible i es poguéssin utilitzar amb classes diferents.

Qüestions

- **Expliqueu les similituds i diferències en la implementació d'aquest TAD BalancedBST respecte al TAD BinarySearchTree. Detalleu quin és el cost computacional teòric de cadascuna de les operacions del TAD.**
  El funcionament del TAD és exactament el mateix, només canvia el mètode insert. En aquest, un cop afegit el nou node, fem comprovacions per a veure si l'arbre està balancejat, i en cas negatiu realitzem les rotacions pertinents. D'aquesta manera ens assegurem que l'alçada de l'arbre sigui sempre $\log n$, on n és el nombre de nodes de l'arbre.
  Per tant, els mètodes insert i search tindran ara un cost $O(\log n)$, inferior al lineal.
  Per altra banda, els mètodes de les rotacions tenen un cost constant, ja que simplement reassignen punters.
  Finalment, els mètodes afegits per a obtenir el títol de pel · lícula més llarg i les millors i pitjors puntuacions (amb les pel · lícules corresponents), donat que recorren l'arbre sencer, tindran un cost lineal O(n).

Exercici 4 - Cercador de pel · lícules amb arbres binaris balancejats

Observacions i decisions preses

Per aquest exercici hem agafat el número dos i hem canviat la classe **BinarySearchTree** per la **BalancedBST**. Com es tracten d'arbres amb les mateixes funcionalitats no hem hagut de canviar res més, tan sols la classe de l'atribut arbre de la classe **BSTMovieFinder**, la resta és totalment compatible.

Exercici 5 - Avaluació de les estructures

Observacions i decisions preses

En aquest exercici hem decidit no fer un menú, ja que volíem que el test el fes l'ordinador tot sol. És per això que es fan els tests en sèrie, és a dir, primer amb el fitxer petit i després amb el gran. Hem modificat el fitxer de cerca per que sigui més gran i augmentar el temps per veure diferències; tot i això no n'hem pogut trobar (com expliquem més endavant).

Qüestions

- **Raoneu els resultats de temps obtinguts en la comparació dels TADs `BSTMovieFinder` i `BalancedBSTMovieFinder`.**
  Pel fitxer petit no es poden apreciar diferències en cap de les accions (creació o cerca). Les creacions (procés de llegir el fitxer) són molt ràpides, menys d'un mil · lisegon, ja que no s'aprecia diferència entre el temps inicial i el final. Malgrat els dos arbres tenir 100 elements, el binari té una alçada de 13, mentre que el balancejat en té 8 només (recordem que serà aproximadament $\lceil \log_2 n \rceil + 1$). La busca tampoc té diferències significatives.
  En canvi, quan fem la prova al fitxer gran, hi ha una diferència molt gran a l'hora de construir els arbres. L'arbre binari ho fa molt de pressa, però el balancejat triga molt més, ja que ha de rotar alguns nodes en algunes ocasions, i això, malgrat ser només reassignacions de punters, el fa endarrerir molt. Recordem que fer una rotació és un cost lineal, però ho fa entre 0 i dues vegades per node, de mitjana unes 10000 vegades.
  Analitzant la cerca veiem que tampoc hi ha una diferència apreciable de temps. Suposem que per a un nombre de dades més elevat es notaria molt més (l'arbre binari té 31 nivells i el balancejat 16, diferència gairebé inapreciable al fer una cerca amb un processador decent).

- **Indiqueu quin és el cost computacional teòric de les operacions d'inserció i cerca en els dos arbres implementats (exercici 1 i exercici 3).**
  El cost computacional teòric per a la inserció de l'exercici 1 és logarítmic de mitjana, però el pitjor dels casos és lineal, O(n). La cerca és el mateix, ja que la inserció és una cerca i assignacions a un node.
  En el cas de l'exercici 3, al ser balancejat el cost teòric mitjà i el pitjor coincideix, i és $O(\log n)$. La cerca té el mateix cost, ja que una inserció consta d'una cerca i assignacions de punters (aquí en poden ser més, d'assignacions, ja que es poden fer rotacions dels nodes. Malgrat això, al tenir un cost lineal no alteren el comportament logarítmic de l'algorisme).