Estructura de Dades

Pràctica 3: Arbres binaris

UNIVERSITAT DE BARCELONA

Blai Ras Jimenez i Albert Morales, Parella 5

Professor: Pere Hierro, Grup D

1. Exercici 1
    a. ***Objectius***

    L'exercici 1 és una introducció als arbres binaris per realitzar els exercicis que venen a continuació, és a dir, és demana la implementació d'un TAD d'un arbre de cerca binària amb els mètodes i funcions que el caracteritzen, per després poder reutilitzar-lo en l'exercici 2.

    b. ***Enunciat***

    Implementeu el TAD BinarySearchTree que representi un arbre binari de cerca amb una representació encadenada. Addicionalment, implementeu els nodes de l'arbre seguint l'especificació del TAD Position. Heu de tenir en compte que cada node de l'arbre estarà representat per una paraula i a més haurem de guardar-hi totes les línies i posicions de la paraula en el text.

    c. ***Així doncs, hem realitzat els següents mètodes en el TAD BST amb estructura de nodes Position:***

        i. Constructor(): inicialitza l'arbre. Seteja l'arrel a zero.

        ii. Bool empty(): booleà que retornarà cert o fals segons si l'arbre està buit o no.

        iii. Int size(Position *node): aquest mètode em retorna un enter corresponent a la mida del arbre, és a dir, quans nodes té. Ja no es fa amb una variable mida que s'incrementa, sinó que es calcula recursivament. Nosaltres em fet que es cridi aquesta funció per cada node dret i esquerre, i després s'obliga sumar 1 que correspon al node Arrel.

```
//Constructor. Inicialitza l'arrel del arbre a res (NULL)
BinarySearchTree::BinarySearchTree() {
    arrel = NULL;
}

//Metode que utilitzo per saber si l'arbre esta buit
bool BinarySearchTree::empty() {
    return arrel == NULL; //Condicio
}

//Metode que calcula recursivament la mida
int BinarySearchTree::size(Position *node) {
    if(node == NULL){ //Si el node que em pasen es null, la mida es zero
        return 0;
    } else {
        return size(node->left()) + size(node->right()) + 1; //Calcula tota la dreta, suma 1 de l'arrel i calcula tota l'esquerre
    }
}
```

iv. Bool search(string aBuscar, Position *arrelFalsa): booleà que em retornarà cert o fals segons si ha trobat o no l'element passat per paràmetre en l'arbre. Es demana, un altre cop, una implementació amb recursivitat, així que el que hem codificat ha sigut mirar si l'element a buscar era més petit que string de l'arrel, i llavors cridar un altre cop a la funció search() però a dreta i a esquerre, de manera que fins que no trobi l'element no pararà, a no ser que arribi a un node fulla i llavors retornarà fals:

```
//Metode que retorna si cert/fals segons si troba l'element passat en l'arbre
bool BinarySearchTree::search(string aBuscar, Position *arrelFalsa) {
    if (arrelFalsa != NULL) { //Si l'arbre no esta buit...
        if (aBuscar== arrelFalsa->getElement()) { //Miro primer si l'element esta en l'arrel
            return true;
        }

        //Miro si l'element es mes petit o més gran que l'arrel, per saber si buscar a dreta o e
        if (aBuscar<arrelFalsa->getElement()){
            return search(aBuscar,arrelFalsa->left());
        } else {
            return search(aBuscar, arrelFalsa->right());
        }

    } else { //Si no el trobo retorno fals
        return false;
    }
}
```

v. Position* root(): aquest mètode retorna un punter al node arrel. El fem servir per inicialitzar l'arbre creant la seva arrel.

vi. Int heigt(Position *node): aquest mètode calcula l'altura d'un arbre. S'entén per altura d'un arbre el màxim nivell d'ell, és a dir, si el nivell de l'arrel és 1, el seu fill estarà al nivell 2, etc. Es demana una implementació recursiva, així que el que hem fet ha sigut primer de tot fer una condició que serà el que faci que la recursivitat pari; i és que si el node passat per paràmetre és de tipus fulla o extern, que retorni 1. Després, gestionaré les 3 úniques possibilitats: que arribi a un node esquerre null, a un dret null o a un esquerre i dreta null. En cadascun dels casos aniré sumant 1 i cridaré un altre cop a la funció perquè continuï calculant pels següents nodes:

```
//Metode que em retorna l'arrel del arbre. Serveix per inicialitzar l'arrel
Position* BinarySearchTree::root(){
    return arrel;
}

//Metode que em retorna l'altura o alcada de l'arbre
int BinarySearchTree::height(Position *node){
    if (node->isExternal()){ //Si el node es extern (no ho sera en la primera crida, ja que passare l'arbre)
        return 1; //Acumula 1
    } else {
        if (node->right() == NULL){ //Si no, estableix la dreta com que ja l'has fet i..
            return 1 + height(node->left()); //Torna a cridar la funcio pel seguent node dret acumulant 1.
        }
        if (node->left() == NULL){ //Fes el mateix per l'esquerre
            return 1 + height(node->right());
        }
        if (node->left() != NULL && node->right() != NULL){ //Em el cas de coincidir dreta i esquerre, farem el mateix pero cridant a dreta i esquerre
            return 1 + max(height(node->left()), height(node->right()));
        }
    }
}
```

vii. Void insert(string clau, Position *node): aquest mètode insereix un string al arbre. L'argument que fa servir per inserir a un lloc o a un altre es comparant si l'element a inserir es mes gran o mes petit que un element d'un node ja afegit, de manera que anirà recorrent tot l'arbre amb aquesta condició. La funció de suport que fem servir per afegir un node és setLeft() o setRight(), que com el seu nom indica, estableix un nou node amb l'element passat a l'esquerre o a la dreta:

```
//Metode que insereix un element passat per parametre a l'arbre
void BinarySearchTree::insert(string clau, Position *node) {
    Position *nouNode = new Position(clau); //Node temporal

    //Estableixo el node com a dret, pare i esquerre
    nouNode->setLeft(NULL);
    nouNode->setRight(NULL);
    nouNode->setParent(node);

    //Comprovo si la mida es zero, i estableixo una arrel i un pare a nullptr
    if (size(node) == 0){
        arrel = nouNode;
        arrel->setParent(NULL);

    }else { //Si no,...
        if (node->isExternal()){ //Si es tracte d'un node fulla,
            if (clau < node->getElement()){ //Si l'element inserit d'aquest es mes petit que la cla
                node->setLeft(nouNode); //L'estableixo
            } else { //Si no, anira a la dreta, seguint la teoria dun BST
                node->setRight(nouNode);
            }

        }else { //Si no es un node fulla,
            if (clau < node->getElement()) { //Si la clau es mes petita...
                if (node->left() == NULL ) { //Estableixo un node nou a la esquerre amb l'element p
                    node->setLeft(nouNode);

                }else { //Si no, va a la dreta
                    insert(clau, node->left());
                }
            } else { //Si no es un node fulla...
                if (node->right() == NULL ) { //Estableix un nou node a la dreta, canviant el que h
                    node->setRight(nouNode);

                }else { //Si no, estableix un pero no canviis el que hi havia
                    insert(clau, node->right());
                }
            }

        }
    }

}
```

viii. Void printInorder(Position* node): un recorregut Inordre és aquell que visita un node quan ja s'han visitats els seus subarbres dret i esquerre, és a dir, els seus fills. Seguint la teoria, hem fet un codi que comprova si un node és esquerre o dreta per cridar després la mateixa funció (recursivitat) un altre cop cap a la dreta. És a dir, si estic en un node dret, i aquest té fills, cridaré la funció inOrder però amb aquest node dret com a paràmetre. Un cop hagi recorregut els drets, imprimiré per pantalla l'element que he anat trobant de cada node, tant dret com esquerre.

ix. Void printPreorder(Position* node): un recorregut Preordre és aquell en que un node es visita abans que els seus fills, és a dir, fa un recorregut de manera sistemàtica, de manera que imprimeix "de manera estructurada". La meva implementació per tant és idèntica a l'anterior però l'impresió per pantalla, el "cout", és fa al inici, perquè és visita abans que els seus descendents.

x. Void printPostorder(Position* node): un recorregut PostOrdre és aquell que es comença a visitar pels descendents, és a dir, semblant al inOrdre però un cop visitat els seus descendents. El que faig per tant és establir l'impressió (el "cout") al final de tot, un cop he visitat la descendència.

```
//Metode que imprimeix l'arbre seguint el recorregut Inorder
void BinarySearchTree::printInorder(Position* node) {
    if(node != NULL) { //Si el arbre no esta buit...
        if(node->left()){ //Si em trobo un node esquerre
            printInorder(node->left()); //El visito i vaig a pel seguent
        }
        cout << " " << node->getElement() << " "; //Imprimeix el que he trobat
        if(node->right()) { //Faig el mateix per la dreta
            printInorder(node->right());
        }
    } else { //Si esta buit, imprimeix una senyal
        cout << "[]";
    }
}

//Metode que imprimeix l'arbre seguint el recorregut Preorder
void BinarySearchTree::printPreorder(Position* node) {
     if(node != NULL) { //Si l'arbre no esta buit...
        cout << " " << node->getElement() << " "; //Imprimeix (res en la primera crida)
        if(node->left()){//Si em trobo un node esquerre
            printPreorder(node->left()); //El visito i vaig a pel seguent (s'imprimirà)
        }
        if(node->right()){ //Faig el mateix per la dreta
            printPreorder(node->right());
        }
    }else { //Si esta buit, imprimeix una senyal
        cout << "[]";
    }
}

//Metode que imprimeix l'arbre seguint el recorregut Postorder
void BinarySearchTree::printPostorder(Position* node) {
    if(node != NULL) {//Si l'arbre no esta buit...
        if(node->left()){ //Si em trobo un node esquerre
            printPostorder(node->left());//El visito i vaig a pel seguent
        }
        if(node->right()) { //Ara ho faig amb la dreta
            printPostorder(node->right());
        }
        //Llavors ja puc imprimir, ja ho he visitat tot
        cout << " " << node->getElement() << " ";
    }else {
        cout << "[]";
    }
}
```

   d. ***Cost de les funcions. Varien segons l'alçada de l'arbre.***
      i. Pitjor cas: alçada igual que el nombre de nodes
      ii. Arbre perfectament balancejat:
         1. $O(\log_2 n)$
      iii. Arbre totalment desbalancejat: l'arbre de cerca binària és una Ordered Linked List:
         1. O(n)

2. Exercici 2

   En l'exercici 2 es demana implementar un buscador de paraules. És a dir, utilitzant el nostre arbre binari, em de ser capaços d'implementar mètodes que facin últim aquest, i es que un cercador de paraules és un dels usos d'un arbre binari.

   Ens demanen, doncs, un algoritme que agafi un text i creï un arbre binari amb les seves paraules, de manera que podem buscar paraules concretes. Implementem, per tant, els mètodes següents:

   a. appendText(filename): és el mètode que insereix tot un text passat per paràmetre en l'arbre binari. És demana, per cada text, guardar cada paraula amb el seu número de línia i de mot, així que aquest mètode troba aquestes tres coses. A part, es demana que s'insereixi en minúscula.
      Per fer-ho, esta clar que tindrà estructura de while. La condició que implico és .eof() (end of file), és a dir, mentre no arribi al final del fitxer... Després, faig la gestió amb dos if's i un else, en el que gestiono:
      i. Si el caràcter és una coma, un punt, un parèntesis, etc.
      ii. Si hi ha un salt de línia
      iii. Si hi ha un espai
      iv. Si es majúscula

En el cas de complir la condició, inserto amb un mètode que veurem a continuació. Queda així doncs:

```
void BTSWordFinder::appendText(string filename){
    ifstream meu_fitxer(filename);
    string paraula = "";
    char caracter;
    int linia = 0;
    int posicio = 0;
    while (!meu_fitxer.eof()) {
        caracter = meu_fitxer.get();
        if(caracter == ',' or caracter == '.' or caracter == ':' or caracter == '('

        }else if(caracter == '\n'){
            linia++;
            posicio++;
            insertWord(paraula,linia,posicio);
            posicio = 0;
            paraula = "";
        }else if(caracter == ' '){
            posicio++;
            insertWord(paraula,linia,posicio);
            paraula = "";
        }else{
            if(caracter > 64 and caracter < 91){
                caracter = caracter + 32;
            }
            paraula = paraula + caracter;
        }
    }
    meu_fitxer.close();
}
```

b. insertWord(word, line, position): aquest mètode és el que com he dit insereix una paraula en l'arbre. Ja tenim un mètode que fa això, insert(), però com podem veure tenim els string línia i posició per paràmetre. El que fem, doncs, és una classe externa, Dades, que emmagatzema aquesta informació, mentre que l'insert de la paraula en sí es delega al insert del exercici 1. Aquí veiem la classe dades, un simple vector de dos posicions:

```
#include "Dades.h"

Dades::Dades(int line, int posicio) {
    dades[0] = line;
    dades[1] = posicio;
}

int* Dades::getDades(){
    return dades;
}

int Dades::getLine(){
    return dades[0];
}

int Dades::getPosicio(){
    return dades[1];
}
```

c. findOcurrences(word): mètode que com el seu nom indica troba coincidències entre l'arbre i la paraula passada. Ja tenim un mètode que fa això, search(), de l'entrega anterior. Es delega, per tant, amb aquest mètode, amb l'ajuda de la classe contenidora Dades:

```
list<Dades> BTSWordFinder::findOccurences(string word){
    list<Dades> llistaBuida;
    if (tree.search(word, tree.root())){
        return tree.searchNode(word, tree.root())->getDades();
    } else {
        return llistaBuida;
    }
}
```

d. viewIndex(): aquest mètode retorna la informació del arbre, és a dir, cada paraula en ell amb la informació de número de línia i de mot. Nosaltres no em fet un mètode en sí, sinó que em utilitzat el print Inorder() de l'entrega anterior amb el format que es demana. Així, ho retorna utilitzant les directrius d'aquest recorregut en ordre alfabètic:

```
//Metode que imprimeix l'arbre seguint el recorregut Preorder
void BinarySearchTree::printPreorder(Position* node) {
    if(node != NULL) { //Si l'arbre no esta buit...
        cout << " " << node->getElement() << " "; //Imprimeix (res en la primera
        if(node->left()){//Si em trobo un node esquerre
            printPreorder(node->left()); //El visito i vaig a pel seguent (s'imp
        }
        if(node->right()){ //Faig el mateix per la dreta
            printPreorder(node->right());
        }
    }else { //Si esta buit, imprimeix una senyal
        cout << "[]";
    }
}
```

L’algoritme final, és a dir, el main, queda així:

```
string nomFitxer;
string linia, word;
int altura;

cout << "Introdueix el nom del fitxer '.txt.' \n" << endl;
cin >> nomFitxer;
BTSWordFinder bts;
bts.appendText(nomFitxer);
ifstream myfile (nomFitxer);
ifstream diccionari("dictionary.txt");
list<Dades> list;

clock_t start = std::clock();
if (diccionari.fail() or myfile.fail()) {
    throw runtime_error("Error al obrir el document!");
}else {
    string aux1, aux2;

    while(getline(diccionari,aux1)) {

        istringstream streamWord(aux1);
        while(!streamWord.eof()) {

            streamWord>>aux2;
            if (aux2!="") {
                list = bts.findOccurences(aux2);

                if (list.size()==0) {


                } else {
                    cout << 8<< endl;
                    cout<<aux2<<": "<<endl;
                    for (int i = 0; i< list.size(); i++) {
                        cout << "Linia: " << list.back().getLine() << endl;
                        cout << "Posicio: " << list.back().getPosicio() << endl
                        list.pop_back();

                    }

                    cout <<endl;
                }
            }
        }
    }
```

3. Arbres Binaris Balancejats

   En la tercera entrega d'aquesta tercera pràctica es demana fer un TAD d'un arbre binari balancejat, és a dir, un arbre binari en les que les seves insercions són perfectes, de manera que la seva altura, tant per dreta com per esquerre mai és major de d'1.

   D'aquesta manera, cada vegada que es comet una inserció, s'analitza l'arbre per veure si hi ha un desequilibri. Si es així, es fan dos fins a quatre tipus de balancejaments:

   a. Rotació esquerre: tenim la branca dreta desbalancejada, així que ens movem cap a la esquerre. Si tenim tres nodes un sota l'altre, el del mig es queda com arrel amb els seus corresponents fills.
   b. Rotacio dreta: el mateix cas que l'anterior però amb la branca esquerre desequilibrada.
   c. Doble Rotació Dreta: tenim la branca esquerre del nostre arbre desbalancejat, i volem inserir en l'arbre dret del fill esquerre del node a re-balaceijar. S'efectua amb dos rotacions, la primera cap a l'esquerra i la segona cap a la dreta que acaba de balancejar les branques.
   d. Doble Rotació Esquerre: el mateix, però amb el desequilibri a la dreta.

   Així doncs, l'únic que modificarem del nostre Arbre Binari inicial es el nostre insert, que mitjançant una funció isBalanced() comprovarà l'existència de desbalancejaments mitjançant la condició:

A continuació, si s'ha detectat un desbalancejament, s'haurà de passar a gestionar aquesta, és a dir, ha veure el factor de desbalanceig per poder dur a terme la rotació adequada per cada desequilibri. Qui precisament s'encarrega d'això és balance(), una funció que mitjançant les condicions de factor de balancejament crida la rotació correcte:

```
void BinarySearchTree::balance(Position* node) {
    if (isBalanced(node)) {
        cout <<"it isn't balanced" << endl;
        if (balanceCondicio(node) < 0) { //Dreta
            cout << "balance left" << endl;
            if (balanceCondicio(node->right()) > 0) {
                rotatoRightLeft(node);
            } else {
                rotatoLeft(node);
            }
        } else {
            cout << "balance right" << endl;
            if(balanceCondicio(node->left()) < 0){
                rotatoLeftRight(node);
            } else {
                rotatoRight(node);
            }
        }
    } else{
        cout << "it's balanced" << endl;
    }

    if (!node->isRoot()){
        cout << node->getElement() << endl;
        cout << node->parent()->getElement() << endl;
        balance(node->parent());
    }

}

int BinarySearchTree::balanceCondicio(Position* node) {
    return height(node->left())- height(node->right());
}
```

Finalment, les rotacions, les realitzem fent modificacions en els nodes. Per exemple, si la rotació implica un canvi d'arrel, aquest node haure d'assignar-li un punter arrel null, fill dret i fill esquerre, etc. Per exemple, en la Rotació dreta:

Podem comprovar el correcte funcionament de les rotacions fent printPostordre o preordre, però no Inorder, ja que va per ordre "alfabètic":

4. Cercador de paraules amb arbres binaris balancejats

   No ens ha donat temps a implementar l'exercici 4, ens sap greu.

5. Avaluació de les estructures

No hem pogut fer el cercador de paraules, però podem intentar-ho igual. Analitzem el cost d'un arbre binari balancejat:

Una rotació en un AVL, sempre que usem un arbre binari amb encadenaments, és O(1). La seva funció search continua sen O(log n) en temps, ja que no es necessita reestructurar. El mateix succeeix amb Insert, de cost O(log n). En veritat, seria O(log n) + O(log n), que s'estima igual a O(log n).

Si inserim el mateix text (largeText) en els dos arbres, veiem que la cerca dels seus strings és de dos segons en el cas del BST, mentre que en l'AVL ens ha sortit d'aproximadament 1,5 segons, ja que el BST es molt poc probable que estigui balancejat (de fet, no ho esta), mentre que el fet de balancejar el nostre AVL i després fer la cerca provoca aquest decrement de mig segon.