Estructures De Dades

Pràctica 3.Arbres Binaris

Nombre	NIUB
Ling Zhu	18088210
Jonny Muñez Cano	18088081

Ejercicio1

Problema planteado: Implementar el TAD BinarySearchTree que represente un árbol binario de búsqueda con una representación encadenada. Implementar los nodos del árbol siguiendo la especiación del TAD NodeTree.

Decisiones tomadas

1) Hemos decidido implementar las clases BinarySearchTree y NodeTree como Templates para poder guardar distintos tipos de elementos con una sola implementación, favoreciendo así la re-utilización del código.

2) En la clase BinarySearchTree, se declaran y se implementan los atributos y operaciones básicas:

   Atributos privados:

   - int heightB: altura de un árbol
   - NodeTree<E>* rootB: el nodo que guarda la información

   Operaciones:

   - Constructor de BinarySearchTree: BinarySearchTree(). Construye el árbol vacío. Ponemos rootB a cero (un nodo vació) y heightB a uno (altura del árbol es 1 ya que el árbol tiene un nodo vació).

   - Destructor de BinarySearchTree: ~BinarySearchTree(); llama a la función DesBinarySearchTree(NodeTree<E>* subArbre) pasando como subArbre la raíz del árbol. ~BinarySearchTree(..) se llama recursivament pasando como parámetro el hijo izquierdo y el hijo derecho y eliminando el nodo recibido como parámetro.

   Métodos consultores:

   - int size(): devuelve el número de nodos que hay en el árbol binario
   - int height(NodeTree<E>& node): devuelve la altura del árbol binario
   - bool empty(): devuelve true si el árbol esta vació.

   - bool search(const E& element): busca un elemento en el árbol binario de búsqueda y devuelve true si lo ha encontrado.

   - void printPreorder(const NodeTree<E>* elem): muestra el contenido del árbol siguiendo un recorrido preorden.

- void printPostorder (const NodeTree<E>* elem): muestra el contenido del árbol siguiendo un recorrido postorden.

- void printInorder(const NodeTree<E>* elem) : muestra el contenido del árbol siguiendo un recorrido inorden.

Métodos modificadores:

- void insert(const E& element): añade un nuevo nodo al árbol binario de búsqueda.

Respuesta a las preguntas

Justificación de la implementación del TAD NodeTree

- En la clase NodeTree se declaran y se implementan los atributos y operaciones básicas:

➔ Atributos privados:

- NodeTree<E>* leftN: apuntador al hijo izquierda
- NodeTree<E>*rightN: apuntador al hijo derecha
- NodeTree<E>* parentN : apuntador al padre
- E elementN: guarda el elemento de un nodo

➔ Operaciones:

➔ NodeTree(const E& element, NodeTree<E> * const parentAux): iniciamos el hijo izquierda, derecha y el padre. También se inicializa element.
➔ const E getElement() const: el método devuelve el elemento guardado.
➔ NodeTree<E> * const parent() const: devuelve el nodo padre.
➔ NodeTree<E> * const right() const: devuelve el node derecho.
➔ NodeTree<E> * const left() const: devuelve el node izquierdo.
➔ bool isRoot() const: devuelve true si el node es el arrel.
➔ bool isExternal() const: devuelve true si el nodo es un nodo externo.

➔ Métodos modificadores de los punteros de hijo izquierdo, derecho, el padre y el elemento que hay guardado en el node:
-void setElement(NodeTree<E> * const Element);
-void setRight(NodeTree<E> * const rightAux);
- void setLeft(NodeTree<E> * const leftAux);

➔ void setParent(NodeTree<E> * const parentAux);
➔ int height(NodeTree<E>*const Element); devuelve la altura del árbol

Coste computacional teórico de las funciones del TAD BinarySearchTree

- BinarySearchTree coste O(1) – la instrucción se ejecuta una vez y no depende del tamaño.
- ~BinarySearchTree y DesBinarySearchTree coste O(n) – recorre todo el árbol para destruir todos los nodos-.
- size; coste O(n) – se tiene que recorrer todo el árbol
- empty; coste O(1) – la instrucción se ejecuta una vez y no depende del tamaño.
- root; coste O(1) – la instrucción se ejecuta una vez y no depende del tamaño.
- Search –usando el Teorema Master- coste O(Logn) en el caso promedio, pero O(n) en el peor caso.
- printPreorder coste O(n) – recorre todo el árbol.
- printPostorder coste O(n) – recorre todo el árbol.
- printInorder coste O(n) – recorre todo el árbol.
- height –usando el Teorema Master- coste O(log n) en el caso promedio, pero O(n) en l peor caso.
- Insert –usando el Teorema Master- coste O(log n) en el caso promedio, pero O(n) en el peor caso.
-

Métodos adicionales: Se ha añadido un método auxiliar para el destructor DesBinarySearchTree(NodeTree<E>* element); para poder destruir el árbol recursivamente, ya que ~BinarySearchTree() no puede recibir parámetros. Su coste computacional es O(n) porque destruye todo el árbol.

Ejercicio2

Problema planteado: Implementar una clase de contactos Contacte, un buscador de contactos BSTContactFinder. utilizando un árbol(BinarySearchTree y NodeTree). En este BinarySearchTree los nodos contendrán elementos que consisten en una "key " y un contacto. Se re-definirá el método getElement en NodeTree para que devuelve la "key· y se creará un nuevo método getValues que devuelve valores asociados.

Decisiones tomadas:

1) En la clase NodeTree, hemos añadido:

    - Un atributo privado String Key: que representa la clave y luego otro atributo E values: que representa los valores asociados.
    - Métodos getValues y setValues: método getter y setter del atributo Values.
    - Métodos getKey y setKey: método getter y setter del atributo key.
    - También hemos añadido más parámetros en el constructor de NodeTree.

2) En la clase BinarySearchTree:

- Hemos añadido un atributo “index” de tipo int para poder tener el numero de los nodos que hay en el árbol.

- Hemos cambiado el método printInorder donde añadimos un if-else y un atributo “resposta” de tipo string. En el primer if hemos puesto como a condición si el index==40, entonces mostramos una mensaje preguntado al usuario si quiere seguir leyendo.

3) En la clase BSTContactFinder, se declaran y se implementan los atributos y operaciones básicas:

- Atributos privados: BinarySearchTree<Contacte> bst: el bst es un árbol binario de tipo contacte.

- Operaciones:

  - BSTContactFinder(): constructor de BSTContactFinder.
  - ~BSTContactFinder(): destructor de BSTContactFinder.
  - void appendContactList(const string filename): Este método recibe un parámetro filename y guarda su contenido en un árbol.
  - void insertContact(const string nom, const string cognom, const string adreca,const string ciutat,string telefon,const string email): Este método recibe los parámetros de un contacto y guarda el contacto en el árbol.
  - const string showContact(string cognom) const: este método muestra los contactos guardados en el árbol.
  - Contacte findContact(string cognom): este método busca un contacto mediante el apellido de un contacto.
  - void viewIndex() const: este método llama el método printInorder de la clase BinarySearchTree.
  - const int height() const: este método devuelve un entero que representa la altura del árbol.

<u>Problemas que hemos obtenidos durante la realización del ejercicio:</u>

1) Hemos tenido problema en la parte de leer un fichero .

2) No nos sale la parte de hacer un contador que pide la confirmación en cada 40 contactos. Hemos intentado cambiar el método printInorder para que cuando llama a este método en el main, pide al usuario cada 40 contactos si quiere seguir leyendo o no. Pero e¡no nos han salido esta parte.

Ejercicio3

Problema planteado: Implementar el TAD BalancedBST que representa un árbol balanceado- La funcionalidad de este TAD tiene que ser la misma que el TAD BinarySearchTree del ejercicio 1.

1) No hemos cambiado nada la clase NodeTree y la clase Contacte.
2) En la clase BalancedBST, se declaran y se implementan los atributos y operaciones básicas:
   -Atributos privados:
   - NodeTree<E>* rootB: el atributo rootB representa un nodo

   -Métodos privados: estos métodos son llamados en los métodos públicos de la misma clase
   - const int size(NodeTree<E>* subArbre) const: devuelve un entero que representa el tamaño del árbol balanceado.
   - void printPreorder(const NodeTree<E>* elem) const: muestra por la pantalla el árbol mediante un recorrido preorden.
   - void printPostorder(const NodeTree<E>* elem) const: muestra por la pantalla el árbol mediante un recorrido postorden.
   - void printInorder(const NodeTree<E>* elem) const: muestra por la pantalla el árbol mediante un recorrido inorden.
   - const int height (NodeTree<E>* node) const: el método devuelve un entero que representa la altura del nodo
   - NodeTree<E>* nodoDesbalanceado(NodeTree<E>* nodeInserted) const
   - void rebalancear(NodeTree<E>* ArbolaRebalancear): Calcula el banlance de un nodo
   - void leftRotation(NodeTree<E>* NodeaRebalancear): realiza una rotación simple a la izquierda.
   - void rightRotation(NodeTree<E>* NodeaRebalancear): realiza una rotación simple a la derecha.

Respuesta de la pregunta

- Explicar las similitudes y diferencias en la implementación de este TAD BalancedBST respecto al TAD BinarySearchTree.

-Todos los métodos consultores (height, search, empty, size, printPreorder, printPostorder, printInorder), el constructor y el destructor son tiene casi la misma implementación.

- En la clase BalancedBST hemos implementado más métodos: NodeTree<E>* nodoDesbalanceado(NodeTree<E>* nodeInserted) const; void rebalancear(NodeTree<E>* ArbolaRebalancear)o; void leftRotation(NodeTree<E>* NodeaRebalancear); void rightRotation(NodeTree<E>* NodeaRebalancear) que sirven para balancear el árbol.

Detallar cuál es el coste computacional teórico de cada una de las operaciones del TAD.

- BalancedBST coste computacional es de O(1) – la instrucción se ejecuta una vez y no depende del tamaño.
- ~BalancedBST DesBalancedBST coste computacional es de O(n) – recorre y destruye los n nodos del árbol.
- Size coste computacional es de O(n) – recorre todo el árbol.
- Height coste O(n) – recorre todo el árbol.
- Empty coste O(1) – la instrucción se ejecuta una vez y no depende del tamaño..
- Root coste O(1) – la instrucción se ejecuta una vez y no depende del tamaño..
- El coste de insertar es log(n)*log(n) ~ log(n)
- PrintPreorder coste O(n) – recorre todo el árbol.
- PrintPostorder coste O(n) – recorre todo el árbol.
- PrintInorder coste O(n) – recorre todo el árbol.
- Search coste –aplicando el Teorema Master- O(log n) –la altura del árbol-Height coste –aplicando el Teorema Master- O(log n)
- rebalancear: coste O(log n)
- nodoDesbalanceado : – log(n)*log(n) ~log(n)-.
- leftRotation coste O(1) son asignaciones –constante: no depende del tamaño-.
- RightRotation coste O(1) son asignaciones –constante: no depende del tamaño-.

Ejercicio4

Problema planteado: Implementar una nueva versión del cercador de contacto ContactFinder con un árbol balanceado.

- En este ejercicio hemos podido reutilizar las clases NodeTree y Contacte del ejercicio2.
  La clase BalancedBST del ejercicio3.

- La clase ContactFinder que tenemos que implementar en este ejercicio es casi la misma que en el ejercicio2, simplemente que ahora hace uso de un árbol AVL(el método de insertar corresponde al modelo AVL).

Ejercicio5

Problema planteado: Haced una evaluación del rendimiento experimentado de las dos implementaciones anteriores(árboles de búsqueda binaria y árboles binarios balanceados), y razonad las diferencias: contad los tiempo de acceso para dos textos de diferentes tamaño; contad el tiempo de generación de la estructura para dos textos de diferente tamaño.

Árbol	Acción	Texto pequeño	Texto grande
árbol de búsqueda binaria	Inserción	1.8213ms	2.3529ms
árbol binario balanceado	Inserción	12.8343ms	13.1698ms
árbol de búsqueda binaria	Búsqueda	0.4028ms	0.4193ms
árbol binario balanceado	Búsqueda	1.4267ms	1.3682ms

Árbol	Inserción	Búsqueda
Árbol binario	O(logn)o O(n)	O(logn)o O(n)
AVL	O(logn)	O(logn)