Universidad de Chile
Departamento de Ciencias de la Computación
CC-3001-Algoritmos y Estructuras de datos

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

Tarea 4: Arbol 2-3

Autor: Vanessa Gaete
Correo: naan.u.285@gmail.com

Profesor: Jeremy Barbay
Auxiliares: Cristóbal Muñoz
Daniela Campos
Sven Reisenegger
Bernardo Subercaseaux
Curso: CC3001

Fecha de entrega: 26 de Noviembre de 2018
Santiago, Chile

Índice de Contenidos

1. Introducción

El problema a resolver en esta tarea consiste en crear un programa en Java capaz de implementar la estructura y distintas operaciones del TDA diccionario. Particularmente, dada un serie de instrucciones sobre árboles 2-3, recibidas en el input, el programa debe imprimir en el output (especificamente en una línea) el resultado de la aplicación de cada una de ellas.

El arbol 2-3 a trabajar contendrá dos tipos de datos que estarán asociados entre sí: claves correspondientes a enteros y elementos del tipo "*char*". Las claves cumplen la condición de unicidad.

Las instrucciones que debe reconocer y realizar el programa corresponden a: inserción de clave y elemento, obtención del elemento asociado a una clave dada, cálculo de la altura del árbol, impresión de la notación infix del arbol.

A grandes rasgos, la solución implementada corresponde a una clase Arbol23 y otra clase Main. La primera contiene los métodos de árboles 2-3 requeridos y la estructura de árbol 2-3 que se compone de dos ArrayLists para los datos, una referencia al árbol padre, y 4 referencias a árboles hijos. La segunda ocupa los métodos de la clase Arbol23 para poder identificar las instrucciones y ejecutarlas.

2. Diseño de la solución

En un principio, la estructura de árbol 2-3 se hizo creando otra clase llamada "*sección*" que contenía dos valores, un "*char*" y un "*int*", los cuales correpondían al elemento y a la clave respectivamente. Sin embargo, esta opción se desechó luego de implementar unos cuantos métodos cuando se descubrió que el código podría ser más eficiente si se trabajaba con "*ArraysLists*". Esto porque en estas estructuras se puede, directamente, acceder a un elemento o encontrar su largo, en cambio en la estructura nodo que se había creado, se debía recorrer todos los elementos del nodo para acceder a los parámetros recién mencionados. Así, se siguió más bien la estructura de un árbol de un solo dato en el nodo, con la diferencia de que en este caso se amplió, dando 2 valores en vez de 1 y 4 árboles en vez de 2. Estos árboles serían, un arbol izquierdo, uno medio, uno derecho y uno extra. Si bien el arbol nunca tendrá 4 hijos no nulos, el extra se añade cuando un nodo queda con sobrecarga (arreglo igual a 3) y debe hacer split. Se hicieron muchos constructores, que varían según los subarboles requeridos.

En la clase Arbol23 se implementaron los métodos necesarios para después poder aplicarlos de manera simple en la clase Main y cumplir con las instrucciones solicitadas. Por esta razón solo se detallarán los algoritmos de la clase Arbol23. Los métodos y sus algoritmos corresponden, a grandes rasgos, a:

*Métodos que añaden datos a cierto nodo. Reciben un par (k,e) y árboles(la cantidad depende del método). Los métodos varían según la cantidad de datos que ya posee el nodo, además de los árboles hijos y padre que se les desea adjudicar o mantener como estaban. El algoritmo funciona en base a condicionales para chequear la posición en el arreglo en la que se deben insertar la clave y el elemento. No utiliza recursión, pues lo único que hacen estos métodos es como ya se dijo, agregar un elemento a un arreglo y modificar las componentes del árbol (que tampoco es recursivo). El invariante en este caso, es que el nodo estará siempre ordenado de menor a mayor de izquierda a derecha, pues todos los métodos que agregan datos a un nodo, chequean su posición correspondiente (ver Figura 1).

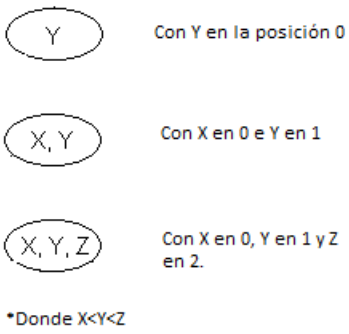

Figura 2.1: Invariante 1.Apunte cc3001 por Patricio Poblete

*Método "*insertar*". Recibe un par (k,e) y un árbol (que sería la referencia a la raíz) y lo inserta en el árbol. El algoritmo de este método funciona con un bucle determinado por un "*while*" y que se repite siempre que el árbol actual sea distinto de null, si el árbol entra al bucle, se hacen comparaciones de las claves con condicionales para determinar cuál será el nuevo árbol a chequear: der, mid o izq, aquí se vuelve comprobar la condición del bucle pero para el árbol al que se decidió pasar. Cuando se rompe el bucle, es porque se llego a un nodo externo y el par (k,e) debe ser insertado, se vuelven a utilizar condicionales para saber qué método de adición usar. Se llama al método "*split*" si el nodo tenía dos datos antes de hacer la inserción. El invariante de esta función es que cada nodo se mantiene ordenado de menor a mayor debido a los métodos de adición que mantienen esta condición; y que todos los árboles hijos están ordenados tal como se de en la Figura 2. El hecho de que la inserción se haga siempre ordenada, permite decir que el árbol está siempre

ordenado y muchos métodos siguientes basan su algoritmo en esta condición.

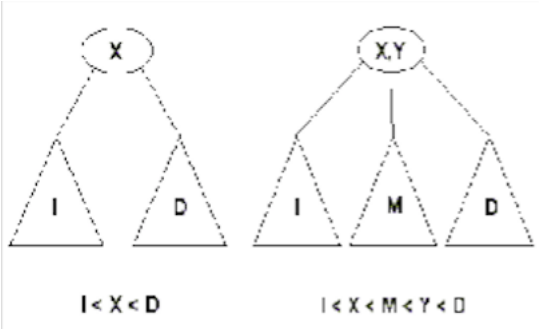

Figura 2.2: Invariante 2. Apunte cc3001 por Patricio Poblete

*Método "*buscar*". Recibe una clave y permite saber si ya existe en el árbol. Este algoritmo también consta de un bucle que permite ir recorriendo el árbol de manera correcta mediante condicionales, es decir, comparando la clave ingresada con las del árbol actual para determinar el árbol al que se debe pasar y que volverá a entrar al while. El bucle se detiene cuando el árbol que intenta entrar es un null, donde se retorna false. El invariante de este algoritmo, es que al estar el árbol siempre ordenado, la búsqueda se hará por el camino correcto (debido a las condiciones que se aplican para determinar a que arbol se pasa), esto permite asegurar que si no se encontró el elemento en el camino recorrido, no estará en ningún otro (sin la necesidad de recorrer el árbol completo). El tiempo de este algoritmo en el peor caso es igual a la altura del árbol, es decir O(lg(n)).

*Método "*split*". Es utilizado dentro de "*insertar*". Hace todas las modificaciones requeridas cuando un nodo hace split, es decir, separa el nodo en dos y luego chequea las condiciones que cumple el nodo padre para saber si se debe volver a hacer split o no. Funciona en base a condicionales y en los casos de que un nodo quede con 3 datos después hacer split, se usa recursión. Además este método también posee un bucle ("*while*") para, cuando ya se hicieron todos los cambios necesarios, devolverse hasta la raíz del árbol y retornar esa referencia. Su invariante es que debido a que el árbol está siempre ordenado, solo se debe modificar la rama en que se hace "*split*", y luego de hacer esta operación todos los árboles hijos de este estarán ordenados, pues la operación se hace hacia arriba. Su tiempo es O(lg(n)), pues en el peor caso, la rama más larga deberá hacer "*split*" en cada nodo.

*Método "*obtener*". Retorna el elemento asociado a una clave. EL algoritme es el mismo que para buscar, solo que retorna un "*char*" en vez de un "*boolean*". El invariante es el mismo que buscar. El tiempo que demora es también, en el peor caso, igual a la altura del árbol, es decir O(lg(n)).

*Método "*altura*". Devuelve la altura del árbol. También funciona con un bucle, mediante el cual se recorre toda la rama izquierda y cuenta la cantidad de nodos que pasó mediante un contador. Su invariante es que al ser un Arbol23, por su estructura, la rama izquierda siempre será la más larga (por ejemplo pueden haber nodos con árbol "*izq*" pero sin "*der*"), con esto se asegura que se entrega el largo de la raíz a la hoja más lejana (altura). El tiempo de operación es siempre igual a la altura del árbol, O(lg(n)).

*Método "$infix$". Devuelve el árbol en notación ínfix. Es un algoritmo recursivo. El tiempo que demora es igual a la cantidad de nodos del árbol, O(n).

Para realizar este código se supuso que el usuario ingresaría siempre un input válido, es decir, que cada instrucción estuviese separada por un espacio de la siquiente, que la instrucción ingresada existe y está bien escrita,y que los elementos y claves son solo caracteres y enteros respectivamente.

3. Implementación

Se presentarán los códigos de la mayoría de los métodos, no se mostrarán los de los constructores, pues son muchos.

*Método de "*buscar*":

```
1   public boolean buscar(int k) {// busca la clave x en el arbol AB.
2       Arbol23 a= this;
3       while(a!=null){
4           if (a.esVacio()) {
5               return false;
6           }else if(a.knodo.size()==1){
7               if(a.knodo.get(0)==k){
8                   return true;
9               }else if(a.knodo.get(0)<k){
10                  a=a.mid;
11              }else {
12                  a=a.izq;
13              }
14          }else{
15              if(a.knodo.get(1)==k){
16                  return true;
17              }else if(a.knodo.get(0)==k){
18                  return true;
19              }else if(a.knodo.get(0)>k){
20                  a=a.izq;
21
22              }else if(a.knodo.get(1)>k){
23                  a=a.mid;
24              }else{
25                  a=a.der;
26              }
27          }
28      }return false;
29  }
```

Este método recorre el árbol con un while. Si el árbol es vacío, no hay elementos con lo que no está la clave buscada. Si el nodo tiene largo uno se revisa si la unica clave que posee es igual, si es asi se retorna true, sino se chequean condiciones para saber a que árbol hijo dirigirse, si la clave buscada es menor al elemento se pasa al arbol izq y sino al medio. Luego se hace algo similar para cuando el nodo tiene dos datos

El método de "*obtener*" no se incluye porque funciona de la misma forma que "*buscar*" solo que en vez de retornar un bool cuando se encuentra la clave, retorna el elemento asociado.

El método "*altura*" es el siguiente:

```
1   public int altura() {// para determinar la altura del arbol solo se recorrerán las ramas
     izquierdas,
2                                   //pues no habra subarboles con mas altura que los izq,
    pero si mas cortos.
3       Arbol23 a= this;
4       int count = 0;
5       if (a.esVacio()) {
6           return count;
7       }while (a != null){
8           a=a.izq;
9           count+=1;
10      }
11
12
13      return count;
14  }
```

Si es vacio se retorna cero. Mientras "*a*" no sea una hoja, se mueve hacia el árbol de la izq y se agrega uno al contador. Cuando sale del loop se retorna ese contador.

El método de "*infix*" es:

```
public String infix(Arbol23 AB){
    if(AB == null || AB.esVacio()){
        return "[]";
    }else if(AB.knodo.size()==1){
        return "(" + infix(AB.izq) + AB.knodo.get(0) + infix(AB.mid) + ")";
    }else if (AB.knodo.size()==2){
        return "(" + infix(AB.izq) + AB.knodo.get(0)  + infix(AB.mid) + AB.knodo.get(1)
+ infix(AB.der) + ")";
    }else{
        return "(" + infix(AB.izq) + AB.knodo.get(0)  + infix(AB.mid) + AB.knodo.get(1)
+ infix(AB.der) + AB.knodo.get(2) + infix(AB.extra) + ")";
    }
}
```

Cada árbol null se imprime con []. Se imprimen los árboles hijos intercalados con los nodos mediante recursión. Se diferencian los casos para cuando hay uno o dos datos en el nodo.

El código de "*insertar*" es muy extenso, por lo que solo se incluyen fragementos

```
        while (a.izq != null) {
            if (a.knodo.size() == 1) {
                if (k < a.knodo.get(0)) {
                    p=a;
                    a = a.izq;

                } else if (a.knodo.get(0) < k) {
                    p=a;
                    a = a.mid;
                }
            } else {
```

Primero se hace una variable vacia llamada "*p*" que va actualizando el padre a medida que se avanza. Se comienza con un while que chequea distintas condiciones, primero para cuando el nodo tiene 1 elemento (las que se ven en el fragmento, y luego otras para cuando tiene largo 2 (estas últimas no se incluyeron porque siguen la misma idea y es muy extenso). Estas condiciones permiten ir recorriendo el árbol de forma muy parecida buscar, hasta que se llega a un nodo externo, momento en el que se sale del while y se aplican nuevas condiciones para saber la operación que se debe hacer para añadir el par (k,e) al nodo, estas condiciones son las siguientes:

```
        if (a.esVacio()) {
            a.Arbol1(p,k, e);
        } else if (a.knodo.size() == 1) {
            Arbol2(a,p ,k, e);
        } else { //(a.knodo.size()==2)
            Arbol3(a, p,k, e);
            a = split(a);  //Si el nodo tenia dos elementos, al agregar queda con 3 y hay
que hacer split.
        }
```

Luego de esto se usa un while para devolverse a la raíz y entregar ese puntero cuando se necesite.
El código de "*split*" tampoco se incluirá completo, pues es el más extenso de todos.

```
public Arbol23 split(Arbol23 AB) {
    int a=AB.knodo.get(1);  //Se extrae el la clave y el elemento del medio del nodo.
    char b=AB.enodo.get(1);
    //Dependiendo de la posicion del nodo, se crea o se modifica el arbol que sea
necesario. Se actualiza el
    //padre en cada caso y se hace split recursivamente cuando sea necesario.
    if (AB.padre==null){
        AB= new Arbol23(AB.knodo.get(1),AB.enodo.get(1),AB.padre,AB.Arbolsplitizq(AB),AB
.Arbolsplitder(AB));
    }else if(AB==AB.padre.izq) {
```

```
9             if (AB.padre.knodo.size() == 1) {
10                Arbol2(AB.padre, AB.padre.padre,Arbolsplitizq(AB), Arbolsplitder(AB), AB.
    padre.mid, a, b);
11
12            }
13            else if (AB.padre.knodo.size() == 2) {  //Si el largo es dos, tiene 3 hijos.
14                Arbol3(AB.padre,AB.padre.padre,Arbolsplitizq(AB), Arbolsplitder(AB),AB.padre
    .mid,AB.padre.der,a,b);
15                split(AB.padre);
16
17            }
18        }else if (AB==AB.padre.der) { //Si el nodo que se analiza es un hijo derecho, el
    padre tiene 2 elementos.
```

Primero se obtienen la clave y el elemento del medio del nodo. Se chequean distintas condiciones para saber la operación que se debe usar para agregar al nodo padre. Si el nodo tenía solo un dato, se agrega sin hacer split. En todos los casos, la clave y elemento "a" y "b" van subiendo en el árbol, esto se hace hasta que se ingresan a un nodo que no debe hacer split cuando son agregados. Así, se chequean distintas condiciones, como por ejemplo la cantidad de datos del nodo padre, si tiene una o ninguna, simplemente se agrega "a" y "b" y se detiene split, en cambio si tenía dos datos, quedará con tres, por lo que se llama a la función para que haga split. Este método además va chequeando la posición del nodo que hizo split, pues los árboles hijos del nodo padre al que hizo split no se distribuirán de la misma forma, así se hacen muchos casos para determinar cuáles serán estos árboles hijos. Al final del código se ocupa un while para devolverse hasta la raíz y retornar ese puntero. El fragmento de código mostrado solo muestra las condiciones para el caso en que el nodo que hace split es hijo izquierdo, sin embargo después se hace algo similar para cuando era el derecho (esto fue omitido, se puede ver en el código fuente).

Método de "*Arbolsplitizq*":

```
1     public Arbol23 Arbolsplitizq(Arbol23 AB) {
2         int k=AB.knodo.get(0);
3         char e=AB.enodo.get(0);
4         return new Arbol23(k, e,AB.padre,AB.izq,AB.mid);
5     }
```

Este método se ocupa para dividir un nodo al momento de hacer split. En este caso cuando un nodo tiene 3 datos, toma el dato de la izquierda, el arbol izquierdo y el medio, y crea un nuevo árbol con el dato como padre, el árbol izq como hijo izq y el medio como hijo medio. Arbolsplitder hace lo mismo solo que se saca el dato derecho y sus hijos serán el árbol derecho (como hijo izquierdo) y el arbol extra (como hijo medio).

4. Resultados y Conclusiones

Se probaron los casos entregados en la tarea:

CASO 1	RESULTADO
+1a h +2b h +3c h	1 1 2

CASO 2	RESULTADO
+1a +2b +3c +4d ?1 ?2 ?3 ?4 ?5	a b c d Error

CASO 3	RESULTADO
+1a +2b +3c ?2 h	b 2

CASO 4	RESULTADO
+1a +2b +2c +3d ?2 2 ?2	Error b b

CASO 5	RESULTADO
+25a +10b +32c p	(([]10[])25([]32[]))

CASO 6	RESULTADO
+25a +10b +32c +57x +74y +48z p	(([]10[])25([]32[]48[])57([]74[]))

En esta tarea se pudo llevar a la práctica la implementación de árboles23 permitiendo estudiar a fondo su estructura y sus diferentes propiedades como por ejemplo la inserción. Esto contribuye mucho al aprendizaje, pues no es lo mismo leer y ver ejemplos sobre cómo funciona la inserción en árboles23 a implementarlo por uno mismo, pues en este último se va más allá de entender unos cuantos casos, se aprende el caso general. También, se pudo comprobar en la práctica el tiempo de operación de cada método, coincidiendo con los teóricos, pero nuevamente, el haberlo implementado ayuda a recordar el algoritmo propio y entender más fácilmente los resultados vistos en clases.

5. Código Fuente

El siguiente es el código completo que se implementó para resolver el problema de la tarea.

```
import java.util.*;



public class Main {
    public static void A23(String[] line) {
        Arbol23 AB= new Arbol23();  //Arbol vacio al que se aplicaran operaciones.
        String r="";                //Arreglo vacio en el que se guardaran los resultados de las lineas

        //Por cada elemento del arreglo, que seria una instruccion, se guarda en la variable instruc,
        // se le calcula el largo (n) y se le aplican operaciones.
        for (int i = 0; i < line.length; i++) {
            String instruc=line[i];
            int n=instruc.length();

    //Si la letra inicial es '+', tiene dos datos, el numero y el elemento, el ultimo corresponde al ultimo caracter.
            if(instruc.charAt(0)=='+'){
                char e=instruc.charAt(n-1);
                int k= Integer.parseInt(instruc.substring(1,n-1));
                if(AB.buscar(k)){                //Primero se ve si la clave esta en el arbol, si es asi, retorna error.
                    r= r+" "+"Error";
                }else{
                    AB=AB.insertar(k,e);         //Si no está, se inserta.
                }

            //Si la primera letra es '?' se compone solo de la clave, que corresponde a la cadena sin el primer caracter.
            }else if(instruc.charAt(0)=='?'){
                int k= Integer.parseInt(instruc.substring(1,n));
                if(AB.obtener(AB,k)=='0'){
                    r= r+" "+"Error";
                }else{
                    r=r+" "+AB.obtener(AB,k);
                }

            //Si el primer caracter es 'h' no se compone de nada mas y se retorna la altura.
            }else if(instruc.charAt(0)=='h'){
                r=r+" "+AB.altura();

            //Si el primer caracter es 'p' no se compone de nada mas y se retorna la notacion infix del arbol.
            }else if(instruc.charAt(0)=='p'){
            r=r+" "+AB.infix(AB);
            }
        }
        if(r==""){
            System.out.println(r);
        }else{
            System.out.println(r.substring(1)); //Se printea r sin el primer caracter, que es un espacio.
        }
    }



    public static void main(String[] args) {
        // Se crea el objeto scanner que nos permite leer el input del usuario.
        Scanner in= new Scanner(System.in);
```

```
56          while (in.hasNextLine()) {
57              String linea = in.nextLine();
58              // Se crea un arreglo de strings que guarde cada elemento entregado en el input
    separado por los espacios.
59              String[] arreglo = linea.split(" ");
60              A23(arreglo);
61          }
62      }
63  }
64
65
66  class Arbol23{
67  // El arbol  se compone de 2 arreglos que contendran los datos del nodo, es decir, un
    arreglo para las claves(int) y
68  // otro para los elementos(char), la clave en la posicion 'i' de su arreglo, se corresponde
    con el elemento de la
69  // posicion 'i' del arreglo de elementos.
70  // Ademas se compone de 5 arboles2-3, referencia al padre, y 4 hijos (los 3 que se pueden
    tener mas uno extra para
71  // el caso en que se necesite hacer split.
72
73      ArrayList<Integer> knodo;
74      ArrayList<Character> enodo;
75
76
77      Arbol23 padre;
78      Arbol23 izq;
79      Arbol23 mid;
80      Arbol23 der;
81      Arbol23 extra;
82
83      public Arbol23() { //Crea un arbol vacio.
84          this.knodo = new ArrayList<>();
85          this.enodo = new ArrayList<>();
86          this.padre=null;
87          this.izq=null;
88          this.mid=null;
89          this.der=null;
90      }
91
92      //Crea un arbol al que se le ingresan arbol23 padre, izq y der, y par (k,e)
93      public Arbol23(int k, char e,Arbol23 p,Arbol23 i,Arbol23 m) {
94
95          this.padre=p;
96          this.knodo= new ArrayList<>();
97          this.enodo= new ArrayList<>();
98          this.knodo.add(k);
99          this.enodo.add(e);
100         this.izq=i;
101         this.mid=m;
102
103     }
104
105     //Este metodo se aplica cuando el nodo tiene una variable y se le agrega otra.
106     public void Arbol2(Arbol23 AB, Arbol23 p,Arbol23 i, Arbol23 m,Arbol23 d, int k, char e)
    {
107
108         AB.izq=i;
109         AB.mid=m;
110         AB.der=d;
111         AB.padre=p;
112         if(AB.knodo.get(0) < k){
113             AB.knodo.add(k);
114             AB.enodo.add(e);
115         }else {
116             AB.knodo.add(0,k);
```

```
117             AB.enodo.add(0,e);
118         }
119     }
120 
121     //Este metodo se aplica para agregar un par (k,e) a arbol vacio, sin hijos y con padre.
122     public void Arbol1(Arbol23 p,int k, char e) {
123         this.padre=p;
124         this.knodo= new ArrayList<>();
125         this.enodo= new ArrayList<>();
126         this.knodo.add(k);
127         this.enodo.add(e);
128         this.izq=null;
129         this.mid=null;
130     }
131 
132     //Este metodo se aplica cuando el nodo tiene una variable y se le agrega otra.
133     public void Arbol2(Arbol23 AB,Arbol23 p ,int k, char e) {
134         AB.der=null;
135         AB.padre=p;
136         if(AB.knodo.get(0) < k){
137             AB.knodo.add(k);
138             AB.enodo.add(e);
139         }else {
140             AB.knodo.add(0,k);
141             AB.enodo.add(0,e);
142         }
143     }
144 
145     public void Arbol2(Arbol23 AB,Arbol23 p ,Arbol23 m,Arbol23 d, int k, char e) {
146         //Este metodo se aplica cuando el nodo tiene una variable y se le agrega otra.
147         AB.mid=m;
148         AB.der=d;
149         AB.padre=p;
150         if(AB.knodo.get(0) < k){
151             AB.knodo.add(k);
152             AB.enodo.add(e);
153         }else {
154             AB.knodo.add(0,k);
155             AB.enodo.add(0,e);
156         }
157     }
158 
159 
160     //Este metodo se aplica cuando el nodo tiene dos variables y se le agrega otra
161     public void Arbol3(Arbol23 AB,Arbol23 p ,int k, char e) {
162         AB.extra=null;
163         AB.padre=p;
164         if(AB.knodo.get(0) > k){
165             AB.knodo.add(0,k);
166             AB.enodo.add(0,e);
167         }else if (k>AB.knodo.get(1)){
168             AB.knodo.add(2,k);
169             AB.enodo.add(2,e);
170         }else{
171             AB.knodo.add(1,k);
172             AB.enodo.add(1,e);
173         }
174     }
175 
176     public void Arbol3(Arbol23 AB,Arbol23 p,Arbol23 i, Arbol23 m,Arbol23 d,Arbol23 ex,int k,
     char e) {
177         AB.extra=ex;
178         AB.der=d;
179         AB.izq=i;
180         AB.mid=m;
181         AB.padre=p;
```

```
182         if(AB.knodo.get(0) > k){
183             AB.knodo.add(0,k);
184             AB.enodo.add(0,e);
185         }else if (k>AB.knodo.get(1)){
186             AB.knodo.add(k);
187             AB.enodo.add(e);
188         }else{
189             AB.knodo.add(1,k);
190             AB.enodo.add(1,e);
191         }
192     }
193 
194     public void Arbol3(Arbol23 AB,Arbol23 p,Arbol23 m, Arbol23 d,Arbol23 ex,int k, char e) {
195         AB.extra=ex;
196         AB.mid=m;
197         AB.der=d;
198         AB.padre=p;
199         if(AB.knodo.get(0) > k){
200             AB.knodo.add(0,k);
201             AB.enodo.add(0,e);
202         }else if (k>AB.knodo.get(1)){
203             AB.knodo.add(k);
204             AB.enodo.add(e);
205         }else{
206             AB.knodo.add(1,k);
207             AB.enodo.add(1,e);
208         }
209     }
210 
211     public void Arbol3(Arbol23 AB, Arbol23 p,Arbol23 d,Arbol23 ex,int k, char e) {
212         AB.extra=ex;
213         AB.der=d;
214         AB.padre=p;
215         if(AB.knodo.get(0) > k){
216             AB.knodo.add(0,k);
217             AB.enodo.add(0,e);
218         }else if (k>AB.knodo.get(1)){
219             AB.knodo.add(k);
220             AB.enodo.add(e);
221         }else{
222             AB.knodo.add(1,k);
223             AB.enodo.add(1,e);
224         }
225     }
226 
227     //Sirve para hacer split. Retorna el arbol que se genera de la parte izquierda al
    separar el nodo.
228     public Arbol23 Arbolsplitizq(Arbol23 AB) {
229         int k=AB.knodo.get(0);
230         char e=AB.enodo.get(0);
231         return new Arbol23(k, e,AB.padre,AB.izq,AB.mid);
232     }
233 
234     //Sirve para hacer split. Retorna el arbol que se genera de la parte derecha al separar
    el nodo.
235     public Arbol23 Arbolsplitder(Arbol23 AB) {
236         int k = AB.knodo.get(2);
237         char e = AB.enodo.get(2);
238         return new Arbol23(k, e, AB.padre, AB.der, AB.extra);
239     }
240 
241     public boolean buscar(int k) {// busca la clave x en el arbol AB.
242         Arbol23 a= this;
243         while(a!=null){
244             if (a.esVacio()) {
245                 return false;
```

```
246             }else if(a.knodo.size()==1){
247                 if(a.knodo.get(0)==k){
248                     return true;
249                 }else if(a.knodo.get(0)<k){
250                     a=a.mid;
251                 }else {
252                     a=a.izq;
253                 }
254             }else{
255                 if(a.knodo.get(1)==k){
256                     return true;
257                 }else if(a.knodo.get(0)==k){
258                     return true;
259                 }else if(a.knodo.get(0)>k){
260                     a=a.izq;
261 
262                 }else if(a.knodo.get(1)>k){
263                     a=a.mid;
264                 }else{
265                     a=a.der;
266                 }
267             }
268         }return false;
269     }
270 
271     //inserta par (k,e) al arbol23
272     public Arbol23 insertar(int k,char e) {
273         Arbol23 a = this;
274         Arbol23 p=null;
275         //Con el while se recorre el arbol hasta que se llega hasta un nodo externo. Se
    actualiza el padre en cada caso.
276         while (a.izq != null) {
277             if (a.knodo.size() == 1) {
278                 if (k < a.knodo.get(0)) {
279                     p=a;
280                     a = a.izq;
281 
282                 } else if (a.knodo.get(0) < k) {
283                     p=a;
284                     a = a.mid;
285                 }
286             } else {
287                 if (k < a.knodo.get(0)) {
288                     p=a;
289                     a = a.izq;
290                 } else if (k < a.knodo.get(1)) {
291                     p=a;
292                     a = a.mid;
293                 } else {
294                     p=a;
295                     a = a.der;
296                 }
297             }
298         }//cuando se llega a nodo externo, se inserta dependiendo de la cantidad de
    elementos que tenga.
299         if (a.esVacio()) {
300             a.Arbol1(p,k, e);
301         } else if (a.knodo.size() == 1) {
302             Arbol2(a,p ,k, e);
303         } else { //(a.knodo.size()==2)
304             Arbol3(a, p,k, e);
305             a = split(a);  //Si el nodo tenia dos elementos, al agregar queda con 3 y hay
    que hacer split.
306         }
307         //Se devuelve hasta la raiz y se retorna ese arbol
308         while(a.padre!=null){
```

```
309             a=a.padre;
310         }
311         return a;
312     }
313 
314 
315 
316     public Arbol23 split(Arbol23 AB) {
317         int a=AB.knodo.get(1);  //Se extrae el la clave y el elemento del medio del nodo.
318         char b=AB.enodo.get(1);
319         //Dependiendo de la posicion del nodo, se crea o se modifica el arbol que sea necesario. Se actualiza el
320         //padre en cada caso y se hace split recursivamente cuando sea necesario.
321         if (AB.padre==null){
322             AB= new Arbol23(AB.knodo.get(1),AB.enodo.get(1),AB.padre,AB.Arbolsplitizq(AB),AB.Arbolsplitder(AB));
323         }else if(AB==AB.padre.izq) {
324             if (AB.padre.knodo.size() == 1) {
325                 Arbol2(AB.padre, AB.padre.padre,Arbolsplitizq(AB), Arbolsplitder(AB), AB.padre.mid, a, b);
326 
327             }
328             else if (AB.padre.knodo.size() == 2) {  //Si el largo es dos, tiene 3 hijos.
329                 Arbol3(AB.padre,AB.padre.padre,Arbolsplitizq(AB), Arbolsplitder(AB),AB.padre.mid,AB.padre.der,a,b);
330                 split(AB.padre);
331 
332             }
333         }else if (AB==AB.padre.der) { //Si el nodo que se analiza es un hijo derecho, el padre tiene 2 elementos.
334             Arbol3(AB.padre, AB.padre.padre,Arbolsplitizq(AB),Arbolsplitder(AB),a, b);
335             split(AB.padre);
336 
337         }else {
338             if (AB.padre.knodo.size() == 1) {
339                 Arbol2(AB.padre, AB.padre.padre,Arbolsplitizq(AB),Arbolsplitder(AB), a, b);
340 
341             }else if (AB.padre.knodo.size() == 2) {  //Si el largo es dos, tiene 3 hijos.
342                 Arbol3(AB.padre,AB.padre.padre,Arbolsplitizq(AB), Arbolsplitder(AB),AB.padre.der,a,b);
343                 split(AB.padre);
344             }
345         }while(AB.padre!=null){
346            AB=AB.padre;
347        }return AB;
348     }
349 
350     public int altura() {// para determinar la altura del arbol solo se recorrerán las ramas izquierdas,
351                                     //pues no habra subarboles con mas altura que los izq, pero si mas cortos.
352         Arbol23 a= this;
353         int count = 0;
354         if (a.esVacio()) {
355             return count;
356         }while (a != null){
357             a=a.izq;
358             count+=1;
359         }
360 
361 
362         return count;
363     }
364 
365     //Se le da un arbol y se obtiene el elemento asociado a la clave.
366     public char obtener(Arbol23 AB,int k){
```

```
367         Arbol23 a=AB;
368         while(a!=null){
369             if (a.esVacio()) {
370                 return '0';
371             }else if(a.knodo.size()==1){
372                 if(a.knodo.get(0)==k){
373                     return a.enodo.get(0);
374                 }else if(a.knodo.get(0)<k){
375                     a=a.mid;
376                 }else {
377                     a=a.izq;
378                 }
379             }else{
380                 if(a.knodo.get(1)==k){
381                     return a.enodo.get(1);
382                 }else if(a.knodo.get(0)==k){
383                     return a.enodo.get(0);
384                 }else if(a.knodo.get(0)>k){
385                     a=a.izq;
386 
387                 }else if(a.knodo.get(1)>k){
388                     a=a.mid;
389                 }else{
390                     a=a.der;
391                 }
392             }
393         }return '0';
394     }
395 
396     //aplica la notacion infix.
397     public String infix(Arbol23 AB){
398         if(AB == null || AB.esVacio()){
399             return "[]";
400         }else if(AB.knodo.size()==1){
401             return "(" + infix(AB.izq) + AB.knodo.get(0) + infix(AB.mid) + ")";
402         }else if (AB.knodo.size()==2){
403             return "(" + infix(AB.izq) + AB.knodo.get(0)  + infix(AB.mid) + AB.knodo.get(1)
    + infix(AB.der) + ")";
404         }else{
405             return "(" + infix(AB.izq) + AB.knodo.get(0)  + infix(AB.mid) + AB.knodo.get(1)
    + infix(AB.der) + AB.knodo.get(2) + infix(AB.extra) + ")";
406         }
407     }
408 
409     //Detecta si un arbol es vacio
410     public boolean esVacio(){
411         return (this.knodo.size()==0);
412     }
413 }
```