```
85    }
86    return res;
87  }
88  /*************************************************************/
89
90  string Expresion::Expresion_Prefijo()const{
91    ArbolBinario<string>::preorden_iterador pre=datos.beginpreorden();
92    string salida="";
93    for(;pre!=datos.endpreorden();++pre){
94       salida=salida+*pre+ " ";
95    }
96    return salida;
97
98  }
99  /*************************************************************/
100 string Expresion::Expresion_Postfijo()const{
101   ArbolBinario<string>::postorden_iterador post=datos.beginpostorden();
102   string salida="";
103   for(;post!=datos.endpostorden();++post){
104      salida=salida +*post+ " ";
105   }
106   return salida;
107
108 }
```

Con respecto a las funciones Expresion_Prefijo y Expresion_Posfijo, simplemente hace falta usar un iterador y recorrer el árbol en el sentido de dicho iterador.

6.4 Arboles generales

Para representar un árbol general, cada nodo contendrá su etiqueta y punteros al padre, al hijo a la izquierda y al hermano a la derecha:

```
1  template <class T>
2  struct info_nodo {
3        T et;
4        info_nodo<T> * padre, * hijoizq, * hermanodcha;
5        info_nodo() {
6              padre = hijoizq = hermanodcha = 0;
7        }
8        infonodo(const T & e) {
9              et = e;
10             padre = hijoizq = hermanodcha = 0;
11       }
12 }
```

En la figura 6.16 se puede ver a la derecha como se representa el árbol que se muestra a la izquierda. Como se puede ver un nodo ahora tiene tres punteros: al padre , hijo mas a la izquierda y hermano a la derecha del nodo.

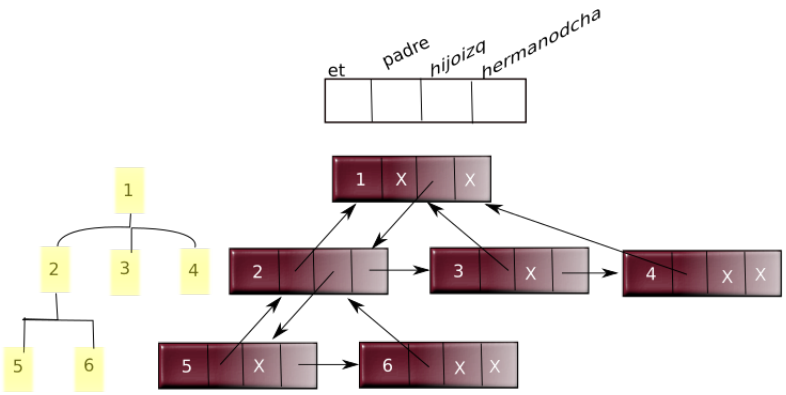


Figura 6.16: **Ejemplo de un árbol general y como se representa**

De esta forma la representación de Arbol General sería de la siguiente forma:

```
typename <class T>
class ArbolGeneral{
 private:
    info_nodo<T> *raiz;

    //aqui todas las funciones privadas

  public:
    //aqui la interfaz de ArbolGeneral


    ...
};
```

Veámos a continuación las funciones asociadas a la parte privada de la clase Arbol General. En primer lugar vamos a implementar la función Copiar, que copia las etiquetas de un conjunto de nodos en un nuevo conjunto de nodos.

```
template <class T>
void ArbolGeneral<T>::Copiar(info_nodo<T>* s, info_nodo<T>* &d) {
      if (s==0)
            d = 0;

      else {
            d = new info_nodo<T>(s->et);
            Copiar(s->hijoizq,d->hijoizq);
```

```
9               Copiar(s->hermanodcha,d->hermanodcha);
10              // le asignamos a los nodos que hemos copiado su padre
11              if (d->hijoizq != 0){
12                d->hijoizq->padre=d;
13                for (info_nodo<T> aux = d->hijoizq->hermanodcha;
14                     aux!=0;aux= aux->hermanodcha)
15                     aux->padre= d;
16              }
17        }
18  }
```

De esta forma el conjunto de nodos origen se indica por el puntero que tiene la variable s. Y el conjunto de nodos destino se identifican desde la variable d. En primero lugar si s no apunta a nada (es decir 0) entonces d también lo hará. En otro caso se solicita nueva memoria para un info_nodo con igual etiqueta que la etiqueta que contiene el nodo al que apunta s. Recursivamente se aplica el procedimiento de copiar para el hijo más a la izquierda y el hermano a la derecha. Un vez realizada la copia hace falta ajustar los padres de los hijos del nodo al que apunta d. Para ello se ejecutan las lineas 11-15.
Otro método privado que se define es Destruir que permite liberar toda la memoria que cuelga desde un nodo dado.

```
1   template<class T>
2   void ArbolGeneral<T>::Destruir (info_nodo<T>* t) {
3     // Debemos empezar con el hermano a la derecha del ultimo nodo hoja
4     // del arbol, si no lo hacemos en este orden, perdemos los enlaces.
5     // Es decir, para destruir el arbol tenemos que hacerlo en recorrido
6     // postorden
7     if (t != 0) {
8       Destruir(t->hijoizq);          // cada hijo resuelve su destruccion
9       Destruir(t->hermanodcha);      // antes de hacer el delete
10      delete t;
11    }
12    // cuando t es cero no entra al if, vuelve a la llamada recursiva
13    // y hace el siguiente paso.
14  }
```

Este método libera la memoria en primer lugar de todos lo hijos y a continuación la memoria de los hermanos a la derecha. La función Copiar se usará para implementar el constructor de copia como el operador de asignación. Destruir se usará en la implementación del operador de asignación y el destructor. De esta forma tenemos que:

```
1   template<class T>
2   ArbolGeneral<T>::ArbolGeneral(const ArbolGeneral<T> & ag){
3     Copiar(ag.raiz,raiz);
4   }
5
```

```
template<class T>
ArbolGeneral<T>::~ArbolGeneral(){
  Destruir(raiz);
}


template<class T>
ArbolGeneral<T> & ArbolGeneral<T>::operator=(const ArbolGeneral<T> & ag){
  if (this!=&ag){
    Destruir(raiz);
    Copiar(ag.raiz,raiz);
 }
 return *this;
}
```

A continuación presentamos dos métodos que hacen crecer en numero de nodos del árbol. El primer método permite insertar un nuevo hijo a la izquierda a un nodo. El actual hijo más a la izquierda del nodo pasa a ser el hermano a la derecha del nuevo hijo a la izquierda.

```
template <class T>
void ArbolGeneral<T>::InsertarHijoIzquierda (info_nodo<T>* n, info_nodo<T>* &t2) {
  // El hijo a la izquierda de n pasaria a ser el hermano a la derecha
  // de t2
  if (t2 != 0) {
        t2->hermanodcha = n->hijoizq;
        t2->padre=n;
        n->hijoizq=t2;
        t2=0;

  }
}
```


t2 no se destruye?

El segundo método, *InsertarHermanoDerecha* inserta un nuevo hermano a la derecha de un nodo. El hermano derecha actual pasa a ser el hermano derecha del nuevo nodo.

```
template <class T>
void ArbolGeneral<T>::InsertarHermanoDerecha (info_nodo<T>* n,
                                              info_nodo<T>* &t2) {
  if (t2 != 0) {
        t2->hermanodcha = n->hermanodcha;
        t2->padre = n;
        n->hermanodcha = t2;
        t2 = 0;
  }
}
```

Como contraposición a los anteriores métodos ahora vemos dos funciones que hacen decrecer el numero de nodos del árbol. La primera función elimina del árbol todo lo que cuelga a partir del hijo más a la izquierda de un nodo dado. Para realizar el proceso de forma correcta el nodo adoptará como nuevo hijo más a la izquierda el hermano del hijo a la izquierda que se quita. Adicionalmente el subarbol hijo más a la izquierda que se quita se devuelve como un árbol.

```
template <class T>
info_nodo<T>* ArbolGeneral<T>::PodarHijoIzquierda (info_nodo<T>* n) {
  info_nodo<T>* res = 0;  // creamos un nodo auxiliar
  if (n->hijoizq != 0) {
    // res apunta al subarbol hijo izquierda
    res = n->hijoizq;
    // el hijo a la izquierda del padre
    // pasa a ser el hermano a la derecha
    // del que era hijo a la izquierda
    n->hijoizq = res->hermanodcha;
    // y el hijo a la izquierda queda como
    // la raiz del arbol a devolver
    res->padre = res->hermanodcha=0;

  }
  return res;
}
```

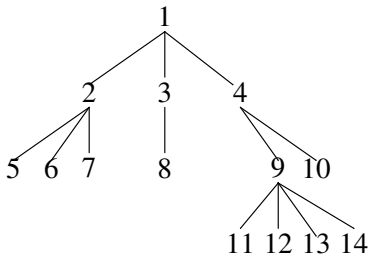
De la misma forma podemos quitar el subarbol hermano a la derecha de un nodo dado. Para ello tendremos que poner como nuevo subárbol a la derecha el nodo que fuese hermano a la derecha del que quitamos.
Finalmente el siguiente método obtiene la altura de un árbol. Recordad que la altura se define como el camino más largo que va desde la raíz a una hoja.

```
template <class T>
int ArbolGeneral<T>::altura (info_nodo<T>* t) {
  // las hojas tendran altura cero y la raiz la altura maxima
  if (t == 0)
        return -1;

  else {
        int max = -1;
        info_nodo<T>* aux;
        // recorremos los hijos del nodo
        for (aux=t->hijoizq;aux!=0;aux=aux->hermanodcha) {
          // comprobamos si la altura de los hijos es mayor a la maxima
          // que tenemos ya calculada

          //altura del nodo
```

```
            int alturahijo = altura(aux);

            if (alturahijo > max)
                  max = alturahijo;
         }
          // la altura de los hijos mas 1 por el padre
         return max+1;
   }
}
```

6.4.1 Recorridos en árboles generales

Los distintos recorridos del siguiente árbol serían:



Preorden: 1 2 5 6 7 3 8 4 9 11 12 13 14 10
Inorden: 5 2 6 7 1 8 3 11 9 12 13 14 4 10
Postorden: 5 6 7 2 8 3 11 12 13 14 9 10 4 1
Niveles: 1 2 3 4 5 6 7 8 9 10 11 12 13 14

En C++ la implementación de los recorridos serían:

```
template <class T>
void ListarPreorden (info_nodo<T>* t) {
      if (t != 0) {
            cout << t->et << ' ';    // primero listamos la raiz
            info_ndo<T>* aux;        // y luego sus hijos
            for (aux=t->hijoizq;aux!=0;aux=aux->hermanodcha)
                  ListarPreorden(aux);
      }
}

template <class T>
void ListarInorden (info_nodo<T>* n) {
      if (n != 0) {
            ListarInorden (n->hijoizq); // listamos el hijo a la izquierda
            cout << n->et << ' ';       // despues la raiz
            info_nodo<T>* aux=n->hijoizq;
            if (aux != 0) {
                  aux = aux->hermanodcha; // y luego los hijos a la
                  while (aux!=0) {        // derecha
                        ListarInorden(aux);
                        aux = aux->hermanodcha;
```

```
                    }
                }
        }
}

template <class T>
void ListarPostorden (info_nodo<T>* n) {
        if (n != 0) {
                info_nodo<T>* aux;
                for (aux = n->hijoizq; aux != 0; aux = aux->hermanodcha)
                        ListarPostorden (aux);

                cout << n->et << ' ';
        }
}

// para la siguiente funcion debemos haber hecho en la cabecera un
// #include <queue>
template <class T>
void ListarNiveles (info_nodo<T>* n) {
        // imprimimos un nodo y despues guardamos en la cola a sus hijos
        if (n != 0) {
                queue<info_nodo<T>* > c;
                c.push(n);
                while (!c.empty()) {
                        info_nodo<T>* aux = c.front();
                        c.pop();
                        cout << aux->et << ' ';
                        for (aux=aux->hijoizq;aux!=0;aux=aux->hermanodcha)
                                c.push(aux);
                } // cuando la cola quede vacia
                  //se termina el listado por niveles
        }
}
```

Otras funciones interesantes son:

- *size:* devuelve el numero de nodos del árbol.
- *iguales:* devuelve true si dos árboles son iguales o false en caso contrario

```
template <class T>
int size (info_nodo<T>* n) {
        if (n == 0)
                return 0;

        else {
```

```
7               int nt = 1; // al menos hay un nodo
8               info_nodo<T>* aux;
9               for (aux=n->hijoizq;aux!=0;aux=aux->hermanodcha)
10                    nt += size(aux);
11
12              return nt;
13        }
14  }
15  template <class T>
16  bool iguales (info_nodo<T>* t1, info_nodo<T>* t2) {
17    if (t1==0 && t2==0)
18          return true; // ambos son arboles vacios
19
20    else {
21          if (t1 == 0 || t2 == 0)
22                return false; // uno es vacio y el otro no
23
24          else {
25            if (t1->et != t2->et)
26                  return false;
27
28            else {
29                  info_nodo<T>* aux1, *aux2;
30                  bool igual = true;
31                  for (aux1=t1->hijoizq;
32                  aux2=t2->hijoizq;igual && aux1!=0 && aux2!=0;
33                        aux1=aux->hermanodcha;aux2=aux2->hermanodcha) {
34
35                        igual = iguales(aux1,aux2);
36                  }
37                  // ahora bien, puede ser que un arbol este contenido en otro,
38                  // es decir, su tamanio sea diferente, por lo que comprobamos
39                  // si los dos han terminado
40                  return igual && aux1==0 && aux2==0;
41            }
42          }
43    }
44  }
```

Ejemplo 6.4.1
Suponiendo que tenemos la clase ArbolGeneral implementar dentro de esta la clase preorden_iterator.

```
1   class preorden_iterator{
2    private:
3      info_nodo<T>* p;
4    public:
5       //Constructor por defecto
6       preorden_iterator(){}
7
8       //Constructor con parametro
9       preorden_iterator(const nodo<T> & n): p(n.p){}
10
11      // se define const porque no modifica p
12      T & operator * () const{
13         return *p;
14      }
15
16      bool operator ==(const preorden_iterator & pre)const {
17        return p==pre.p;
18      }
19      bool operator !=(const preorden_iterator & pre)const{
20        return p!=pre.p;
21      }
22
23      preorden_iterator & operator ++(){
24       if (p==0) // si no apunta a nada
25        return *this;
26       else {
27         if (p->hijoizq!=0)
28           //el siguiente es el hijo mas  a la izquierda
29           p= p->hijoizq;
30
31         else{
32
33          if (p->hermanodcha!=0)
34             // el siguiente sera el hermano a la derecha
35             p = p->hermanodcha;
36          else{
37            // no tiene hijo mas a la izquierda ni
38            //hermano a la derecha
39            //submos por los ascentros
40            //hasta encontrar un nodo que tenga hermano a la derecha
41            while (p->padre!=0 && p->padre->hermanodcha==0)
42              p=p->padre;
43
```

```
44              if (p->padre!=0)
45               p= p->padre->hermanodcha;//el siguiente es el
46                                        //hermano a la derecha
47              else
48               p=0;//no hay mas nodos
49
50              }
51             }
52            }
53            return *this;
54          }
55
56
57     }
58     // para poder implementar el begin y end
59     friend class ArbolGeneral<T>;
60     //para poder iniciar un preorden con un nodo
61     friend class nodo<T>;
62    };
```

Dentro de la clase ArbolGeneral debemos definir las funciones begin_preorden y end_preorden

```
1  preorden_iterator begin_preorden(){
2    preorden_iterator it;
3    it.p = raiz;
4    return it;
5  }
6  preorden_iterator end_preorden(){
7    preorden_iterator it;
8    it.p.=0;
9    return it;
10 }
```

□

6.5 Arboles parcialmente ordenados (APO)

Son árboles binarios con la condición de que la etiqueta de cada nodo es menor o igual que la etiqueta de sus hijos y además, es un árbol completo, es decir, tiene todos los niveles completos excepto el último donde los huecos están a la derecha.

Ejemplo 6.5.1

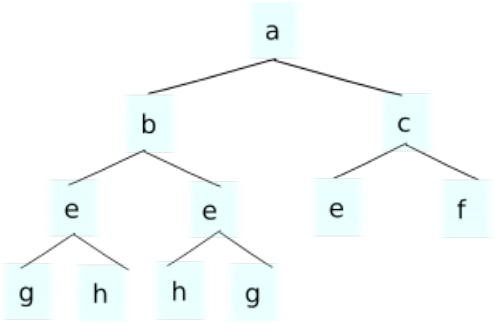


Figura 6.17: **Ejemplo de un APO**

□

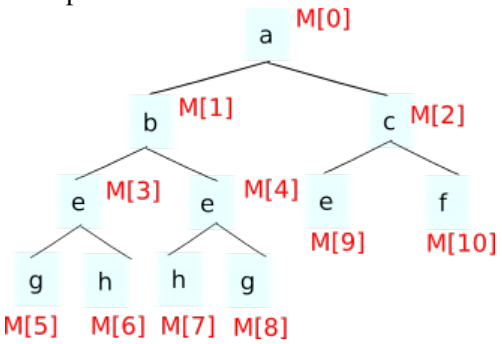
Las funciones típicas de un APO son:

1. Insertar un elemento manteniendo la condición de APO.
2. Borrar el mínimo. El mínimo siempre se encuentra en la raíz del APO.

Su representación óptima es un vector. Aunque lo visualicemos como un árbol binario, para representarlo usaremos un vector con una serie de restrincciones. Esta representación junto con las restricciones que vamos a detallar a continuación se le denomina un montón o heap en inglés. El APO se almacena en el vector por niveles. Así supongamos un heap M, entonces debe cumplirse que:

1. M[0] es la raíz
2. M[1] es el hijo a la izquierda
3. M[2] es el hijo a la derecha
4. En general, el nodo k estará en M[k]
5. Sus hijos, si existen, son los elementos M[2k+1] y M[2k+2]
6. Y su padre, teniendo en cuenta que $k = 2n+1$ y n es la posición que ocupa el padre, $n = \frac{k-1}{2}$ (hacemos división entera). Si fuese $k = 2n+2$ obtendríamos $n = \frac{k-2}{2}$

La representación en un vector del árbol del ejemplo en la figura 6.17sería:



a	b	c	e	e	e	f	g	h	h	g

6.5.1 Insertar un elemento en un APO

Los pasos para insertar un elemento en un APO son:

1. Insertamos el elemento en el hueco que haya en el último nivel, si no lo hay creamos un nivel nuevo.

2. Intercambiamos el nodo con el padre hasta que se cumpla la condición de APO.

La eficiencia es de $O(log_2(n))$, ya que en cada cambio dejamos sin analizar la mitad del subárbol.

Ejemplo 6.5.2

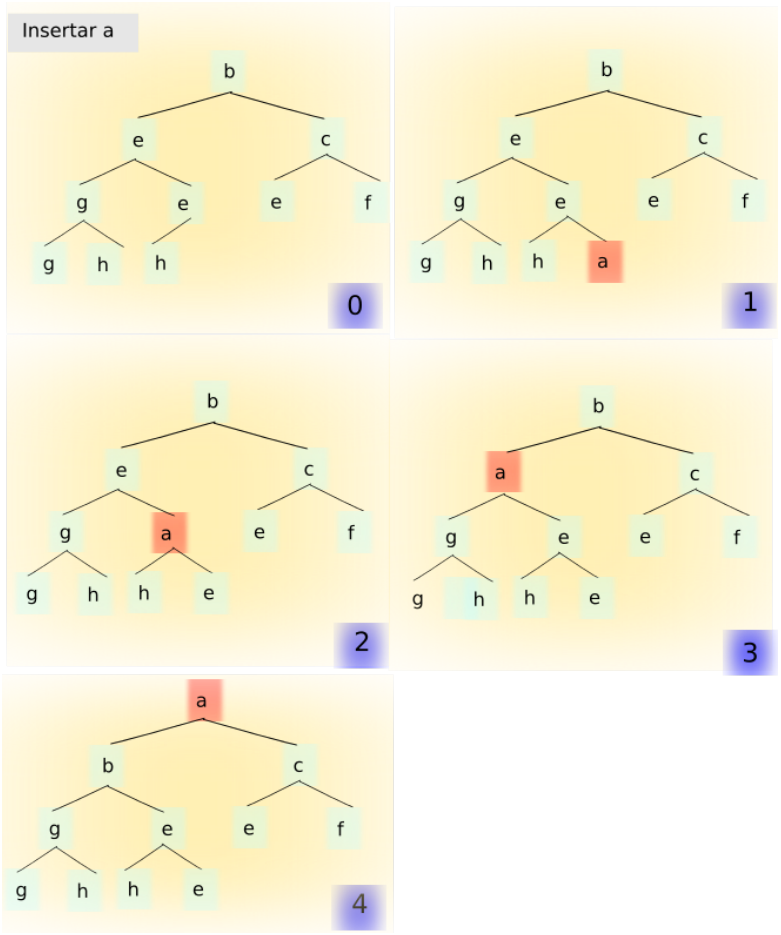


Figura 6.18: **Pasos para insertar un elemento en un APO.**

En la figura 6.18 se muestra los pasos para insertar un nuevo elemento en un APO. En la viñeta 0 de la figura tenemos el APO original. Como se puede observar todos los nodos cumplen que tiene una etiqueta mayor o igual que la de su padre. Supongamos que queremos insertar el carácter 'a'. Como se puede observar en la viñeta 1 se inserta en el ultimo nivel en el primer hueco que nos encontramos a la izquierda. Pero el árbol que se forma ya no cumple la condición de APO pues el padre tiene etiqueta 'e' que es mayor que 'a'. Por lo tanto procedemos a intercambiar la etiqueta 'e' por la etiqueta 'a', dando lugar a la viñeta 2. De nuevo no se cumple la condición de APO y para lograrlo realizamos un conjunto de intercambios hasta que 'a' alcanza la raíz. Estos

intercambios se puede ver desde la viñeta 1 a la viñeta 4. □

Suponiendo que la representación de nuestro APO es:

```
template <class T>
class APO{
private:
  T *datos;       // vector en el que guardamos los nodos
  int nelementos; // tamanio del arbol
  int reservados; //espacio de memoria reservado en datos
  ...
```

El algoritmo de inserción podría ser el siguiente:

```
template <class T>
void APO<T>::Insertar(const T & x){
 //si no tenemos espacio
 if (nelementos ==reservados)
    resize(reservados*2); //ampliamos la memooria

 // insertamos un elemento en la ultima casilla del vector
  datos[nelementos] = x;
 // incrementamos el numero de elementos del vector
  nelementos++;

 // Y ahora empezamos a comparar con el nodo padre hasta que
 // se cumpla la condicion de APO
 int pos = nelementos-1;
 while (pos > 0 && datos[pos] < datos[(pos-1)/2]) {
      swap (datos[pos],datos[(pos-1)/2]);
      pos = (pos-1)/2;
 }
}
```

Podemos también hacer uso de vector de la STL y todo se vuelve más fácil.

```
template <class T>
class APO{
private:
  vecto<T> datos;       // vector en el que guardamos los nodos
  ...
```

Con esta representación el algoritmo de inserción sería:

```
template <class T>
void APO<T>::Insertar(const T & x){
 // insertamos un elemento en la ultima casilla del vector
  datos.push_back(x);
 // Y ahora empezamos a comparar con el nodo padre hasta que
```

```
6    // se cumpla la condicion de APO
7    int pos = datos.size()-1;
8    while (pos > 0 && datos[pos] < datos[(pos-1)/2]) {
9         swap (datos[pos],datos[(pos-1)/2]);
10        pos = (pos-1)/2;
11   }
12  }
```

6.5.2 Borrar el mínimo (la raíz)

Cuando borramos un elemento en un APO siempre vamos a borrar la raíz, que es donde está situado el elemento mínimo de entre los que se encuentra en el APO. Los pasos para borrar la raíz de un APO son:

1. El elemento que se pone en la raíz es el que está en el último nivel más a la derecha.
2. El más pequeño de los dos hijos de la raíz se intercambia con ésta, así hasta que se obtenga la condición de APO.

Ejemplo 6.5.3

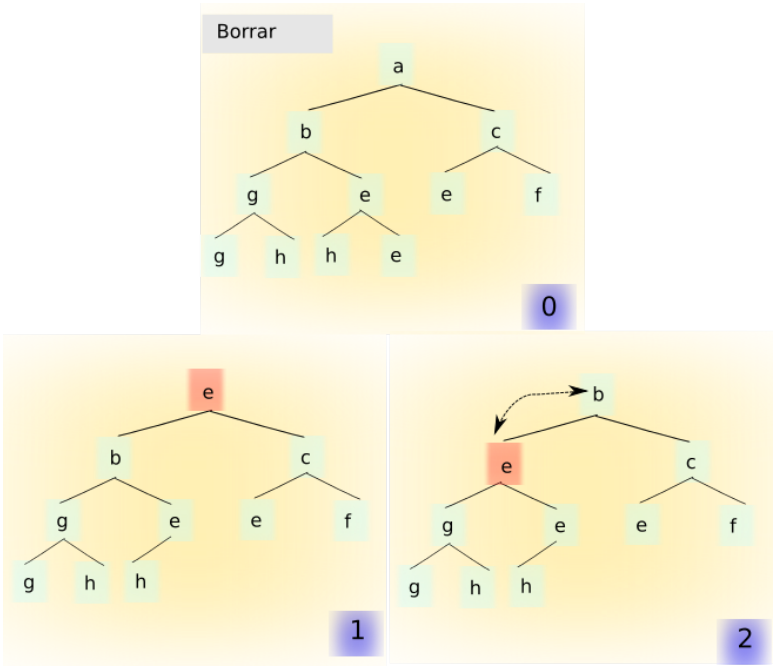


Figura 6.19: **Pasos para borrar en un APO.**

En la figura 6.19 se muestra los pasos para borrar el elemento mínimo (que se situa en la raíz) en un APO. Al borrar la raíz, esta se debe sustituir por el elemento que está situado en el último

nivel mas a la derecha (en la estructura heap el último elemento del vector). Así en el ejemplo que se muestra en la figura 6.19 al borrar 'a' se sitituye por 'e'. Este nuevo árbol que se muestra en la viñeta 1 no cumple la condición de APO. Por eso lo que se hace es intercambiar 'e' por el menor de sus hijos que es 'b'. El árbol resultante se muestra en la viñeta 2. Además el árbol de la viñeta 2 ya es un APO, por lo tanto el procedimiento de borrado aqui se detiene. □

La eficiencia de borrar es $O(log_2(n))$, ya que cada vez efectuamos un intercambio estamos eliminando en el análisis la mitad de nodos.

Los pasos fundamentales de este algoritmo en C++ sería:

```
 // colocamos en la raiz el ultimo nodo del arbol
datos[0] = datos[datos.size()-1];
datos.pop_back();// y lo eliminamos
int ultimo = datos.size()-1;
int pos = 0;
bool acabar = false;

while (pos <= (ultimo-1)/2 && !acabar) {
  int pos_min;
  // este primer if-else sirve para saber
   //cual de los dos hijos es menor y asi
   //saber cual intercambiar con el nodo superior
  if ((pos*2)+1 == ultimo)
        // si es el hijo a la izquierda y el unico hijo
        pos_min = ultimo;

  else {
        if (datos[2*pos+1] < datos[2*pos+2]) // si tenemos mas hijos
        // si el hijo a la izquierda es menor
              pos_min = (2*pos)+1;

        else // el hijo a la derecha es menor
              pos_min = (2*pos)+2;
  }

  // una vez calculado el hijo menor, si es menor que su padre los
  // intercambiamos
  if (datos[pos] > datos[pos_min]) {
        swap(datos[pos], datos[pos_min]);
        pos= pos_min; // actualizamos el valor de pos min para la
                      // siguiente iteracion
  }

  else // ya se cumple la condicion de APO
        acabar = true;
```

```
36 }
```

6.6 Arboles binarios de búsqueda (ABB)

ABB: Es un árbol binario con las etiquetas ordenadas de forma que el elemento situado en un nodo es mayor que todos los elementos que se encuentran en el subárbol izquierdo y menor que los que se encuentran en el subárbol derecho. En general, si nos fijamos en un nodo, su hijo izquierdo es menor y el derecho mayor. Aplicándolo recursivamente llegamos a que todas las etiquetas del subárbol izquierdo son menores y las etiquetas del subárbol derecho son mayores.

Ejemplo 6.6.1
El árbol binario que se muestra es un árbol binario de búsqueda. Como se puede observar por ejemplo el hijo a la izquierda de 8 es 4 que es menor que 8, y además todo el subarbol izquierdo tiene valores menores que 8. Por otro lado el hijo a la derecha es 12 y además todo el subárbol derecho tiene etiquetas mayores que 8.

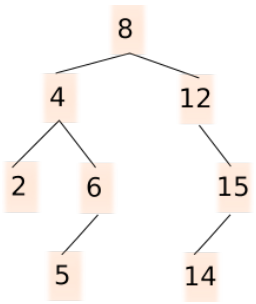


Figura 6.20: **Ejemplo de ABB**

Como detalle a analizar en un ABB el menor de los elementos será o bien el nodo mas a la izquierda que sea hoja o que como mucho tenga un hijo a la derecha. De la misma forma el elemento mayor almacenado en un ABB será aquel que se situe más a la derecha teniendo como mucho un hijo a la izquierda o siendo hoja. En el ABB que muestra en la figura 6.20 el mínimo es 2 (el elemento que se encuentra más a la izquierda y que en este caso es hoja). Y el mayor elemento es 15 que aunque no es una hoja solamente tiene un hijo a la izquierda.

□

Ejemplo 6.6.2
Si nos dan las siguientes etiquetas:

$$\{10,5,14,7,12,3,8,6\}$$

¿cómo podemos obtener el ABB? Los pasos a seguir son los siguientes:

1. El primer elemento del conjunto de etiquetas, 10, es la raíz. Se construye un árbol con un solo nodo como se puede ver en la figura 6.21 viñeta 1.

2. A continuación nos dan la etiqueta 5, ya que es menor que 10 esta se coloca como hijo izquierdo de 10, el hijo derecho será 14 (ver viñeta 2).
3. Para insertar 7 en primer lugar se compara con 10 que es menor, por lo tanto redirijimos nuestro proceso de inserción por el subárbol izquierdo. Ahora se compara con 5 al ser mayor y 5 no tener hijo a la derecha, el 7 pasa a ser el hijo a la derecha de 5.
4. Trás hacer la inserción de 12 se obtiene el ABB que se muestra en la viñeta 3.
5. Así, para insertar el 6 en el árbol debemos ir nodo por nodo viendo si tirar para la derecha o la izquierda. Los pasos serían:
   a) $6 < 10 \longrightarrow$ tiramos a la izquierda
   b) $6 > 5 \longrightarrow$ tiramos al subárbol derecho
   c) $6 < 7 \longrightarrow$ como no tiene hijo izquierdo, ponemos a 6 como hijo izquierdo.

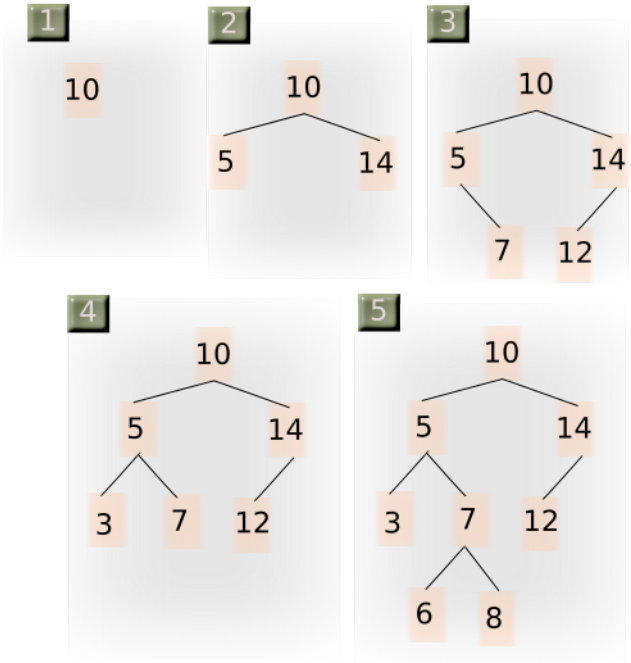


Figura 6.21: **Ejemplo de construcción ABB**

El ABB resultante se puede ver en la figura 6.21 viñeta 5.

□

En promedio, es decir en un conjunto de búsquedas, la mayoría va a tener una eficiencia de $log_2(n)$. Esto es así ya que cada vez que realizamos una comparación, en promedio, no tendremos que comparar con la mitad de los restantes valores. Pero existe un caso donde la búsqueda de un elemento tiene eficiencia $O(n)$. Esta es la situación que ocurre cuando las claves se disponen en una sola rama.
El recorrido inorden ordena las etiquetas de menor a mayor. Por ejemplo, el recorrido inorden el árbol calculado en el ejemplo 6.6.2 sería:

3 5 6 7 8 10 12 14

El tipo set de la STL está implementado con un árbol binario de búsqueda.

6.6.1 Implementación de un ABB

En cuanto a la implementación de un ABB en C++ podemos tomar como implementación base la vista para Arbol Binario en la sección 6.3.2. A diferencia de esta tenemos que añadir una función para buscar,insertar y borrar un elemento en el ABB. Por otro lado el único iterador que nos interesa es el inorden para dado el ABB obtener una ordenación de las claves que almacena. Por lo tanto se puede mantener simplemente la clase nodo del árbol binario (ver sección 6.3.2) y sobrecargar en este el operador ++ para pasar al siguiente nodo en inorden y respecto al operador −− igual.
La representación de un ABB en C++ sería:

```
template <class T>
struct info_nodo {
      T et;
      info_nodo<T> * padre, * hijoizq, * hijoder;
};
//funcion que busca en un ABB una etiqueta, si no esta devuelve 0
template <class T>
info_nodo<T> * Buscar (info_nodo<T> * n, T x) {
      if (n != 0) {
            if (n->et == x)
                  return n;

            else {
                  if (n->et > x)
                        return Buscar (n->hijoizq, x);

                  else
                        return Buscar (n->hijoder, x);
            }
      }

      else // la etiqueta no esta en el arbol
            return 0;
}
// misma funcion pero de forma iterativa
info_nodo<T> * Buscar (info_nodo<T> * n, T x) {
      if (n == 0)
            return 0;

      else {
            info_nodo<T> * p = n; // variable para recorrer el arbol
            while (p != 0) {
```

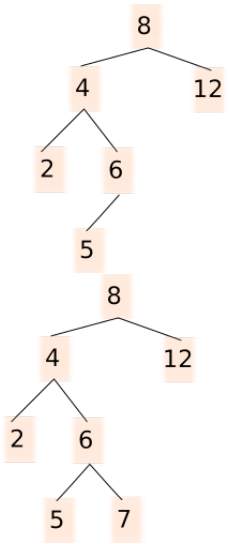
```
33                    if (p->et == x)
34                          return p;
35
36                    else {
37                          if (p->et > x)
38                                p = p->hijoizq;
39
40                          else
41                                p = p->hijoder;
42                    }
43              }
44
45              return 0; // salimos del while sin haberlo encontrado
46        }
47  }
48  /* La version iterativa es mas rapida que la recursiva
49  ya que la recursiva guarda contextos de las llamadas
50  y por tanto es mucho mas costosa */
```

El algoritmo de inserción consiste en buscar la posición donde poner el elemento que queremos insertar e insertarlo ahí.

Ejemplo 6.6.3

Insertar $x = 7$ en el siguiente árbol sigue el siguiente razonamiento lógico:

1. $7 < 8 \longrightarrow$ voy al subárbol izquierdo
2. $7 > 4 \longrightarrow$ voy al subárbol derecho
3. $7 > 6 \longrightarrow$ voy al subárbol derecho que está vacío, como está vacío, inserto el 7 aquí.



El resultado sería

□

La implementación en C++ sería:

```
1  // Devuelve true o false si se ha podido insertar x o no
2  template <class T>
3  bool Insertar (info_nodo<T> * & n, T x) {
4        bool res = false;
```

```
5         if (n == 0) {
6               n = new info_nodo<T> (x);
7               return true;
8         }
9
10        else {
11              if (n->et < x) {
12                    res = Insertar (n->hijoder,x);
13                    if (res)
14                          n->hijoder->padre = n;
15
16                    return res;
17              }
18
19              else {
20                    if (n->et > x) {
21                          res = Insertar(n->hijoizq,x);
22                          if (res)
23                                n->hijoizq->padre = n;
24
25                          return res;
26                    }
27                    else // la etiqueta es x, no se puede insertar
28                          return false;
29              }
30        }
31  }
```

Para el algoritmo de borrado, tenemos tres posibilidades:

Primera posibilidad : el `info_nodo` de x es una hoja. En cuyo caso, simplemente eliminamos dicho nodo. El código correspondiente sería:

```
info_nodo<T>* aux = n;
// suponiendo que n apunta a x
if (aux->padre != 0) {
      if (aux->padre->hijoder == n)
            aux->padre->hijoder = 0;

      else
            aux->padre->hijoizq = 0;
}

delete aux;
```

Segunda posibilidad : el nodo no es hoja. En este caso, se subdivide en otros tres casos:

1. **Que sólo tenga hijo a la derecha**, en cuyo caso se pondría en el lugar de n, su hijo a

la derecha:

```
info_nodo<T> * padre = n->padre;
if (padre != 0) {
      if (padre->hijoder == n) {
            padre->hijoder = n->hijoder;
            padre->hijoder->padre = padre;
      }

      else {
            if (padre->hijoizq == n) {
                  padre->hijoizq = n->hijoder;
                  padre->hijoizq->padre = padre;
            }
      }
}

info_nodo<T> * aux = n;
n = n->hijoder;
delete aux;
```

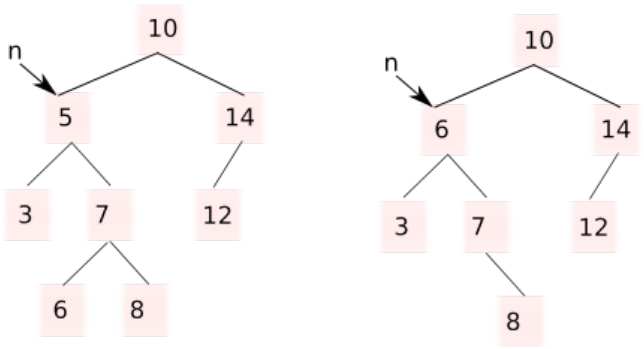
2. **Que sólo tenga hijo a la izquierda**, en cuyo caso se pondría en el lugar de n, su hijo a la izquierda:

```
info_nodo<T> * padre = n->padre;
if (padre != 0) {
      if (padre->hijoizq == n) {
            padre->hijoizq = n->hijoizq;
            padre->hijoizq = padre->padre;
      }

      else {
            if (padre->hijoder == n) {
                  padre->hijoder = n->hijoizq;
                  padre->hijoder->padre = padre;
            }
      }
}

info_nodo<T> * aux = n;
n = n->hijoizq;
delete aux;
```

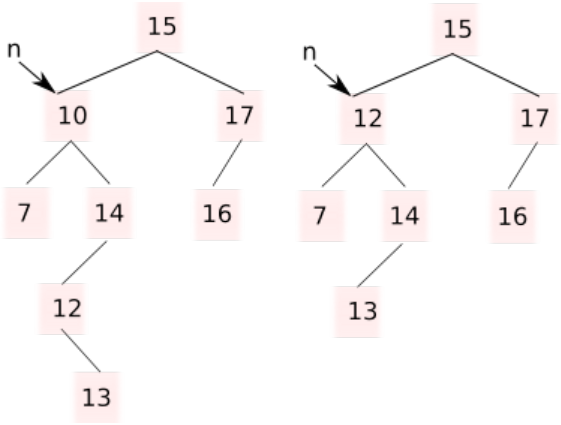
3. **Que tenga dos hijos**, en cuyo caso tenemos que sustituir el nodo por su "siguiente", es decir, el siguiente nodo con más valor. Para obtener dicho "siguiente" debemos seguir los siguientes pasos:
   a) Nos movemos hacia el hijo a la derecha de *n*.

b) Después, nos movemos hacia el hijo a la izquierda del hijo a la derecha de *n*
c) Vamos moviéndonos por todos los hijos a la izquierda hasta llegar a un nodo que no tenga más hijos a la izquierda. Puede ser una hoja (no tener hijos) o tener hijo a la derecha.
d) Cuando llegamos a dicho nodo, cambiamos el valor de la etiqueta de *n* por el del nodo y borramos el nodo siguiendo la posibilidad 1 (no tener hijos) o la posibilidad 2, caso 1 (tener hijo derecho).

Por ejemplo, queremos borrar el nodo con etiqueta 5. El árbol de la izquierda representa el antes y el de la derecha el después:



Otro ejemplo, en el que ahora el nodo con el que vamos a hacer el cambio no es una hoja, sino que sólo tiene un hijo a la derecha. El resultado final sería sustituir *n* por el nodo sin hijo izquierdo que hemos encontrado y al borrar éste nodo, dejamos como hijo izquierdo de su padre, a su hijo derecho:



Así, el algoritmo de borrar final, y las funciones auxiliares que necesita, quedarían:

```
1  // funcion que enlaza un hijo con su padre
2  template <class T>
3  void PutHijo_Padre(info_nodo *n, info_nodo *nuevo){
4        if (n->padre!=0){
5              if (nuevo!=0)// no tiene padre
6                    nuevo->padre = n->padre;
7              // el padre de n tendra como hijo a n
```

```
8               if (n->padre->hder==n)
9                     // su hijo a la derecha
10                    n->padre->hder = nuevo;
11
12              else  // el padre de n tendra como hijo
13                    // a su hijo a la izquierda
14                    n->padre->hizq=nuevo;
15
16        }
17  }
18
19  // En esta funcion es donde encapsulamos cada uno de los casos que
20  // hemos visto por separado y donde se borra el nodo.
21  template <class T>
22  void EliminarRaiz(info_nodo *&n){
23    if (n->hizq==0 && n->hder==0){ // Posibilidad 1: n no tiene hijos
24        PutHijo_Padre(n,0);// establecemos su padre a 0
25        delete n; n=0;     // lo borramos
26
27    }
28
29    else if (n->hizq==0){// Posibilidad 2, CASO 1:
30                         //n tiene hijo derecho
31        info_nodo *aux=n;
32        PutHijo_Padre(n,n->hder);// hijo a la derecha de n
33        if (n->padre==0)// si n es la raiz
34              n= n->hder;// La raiz del arbol
35                        //ahora es su hijo derecho
36        delete aux;aux=0;// Eliminamos n
37
38    }
39    else if (n->hder==0){// Posibilidad 2, CASO 2:
40                         //n tiene hijo izquierdo
41        info_nodo *aux=n;
42        PutHijo_Padre(n,n->hizq);
43        if (n->padre==0)
44              n = n->hizq;
45        delete aux; aux=0;
46
47    }
48    else{ // Posibilidad 2, CASO 3: n tiene dos hijos
49          // Buscamos el siguiente:
50        info_nodo *aux=n->hder;// Nos movemos al hijo a la derecha
```

```
51        while (aux->hizq!=0)// Avanzamos hasta que llegamos a un nodo
52              aux=aux->hizq;// que no tiene hijo a la izquierda
53        // OJO: no tiene por que ser una hoja
54        n->et=aux->et;// y cambiamos la etiqueta de n por la del nodo
55        // y por ultimo, borramos este nodo.
56        Borrar(n->hder,aux->et);
57     }
58
59  }
60
61  // Funcion que llama a EliminarRaiz para borrar alguna componente del arbol
62  template <class T>
63  void Borrar(info_nodo * &n,const T &e){
64        if (n!=0){
65          if (n->et==e)  // Si encontramos la raiz del subarbol que
66                EliminarRaiz(n);// queremos borrar, lo borramos
67          else if (n->et<e) // si el nodo que queremos borrar es mayor
68                Borrar(n->hder,e);// o menor que la raiz del subarbol actual,
69          else // nos movemos a la izquierda o a la derecha
70                Borrar(n->hizq,e);// recursivamente hasta encontrarlo.
71        }
72  }
```

Ejemplo 6.6.4
Construir un programa que dado un conjunto de claves de tipo char las muestre, por la salida estándar, de forma ordenada.

```
1  #include <ABB.h>
2  #include <iostream>
3   typename <class T>
4   void ImprimirAbb(ABB<T> &A){
5     typename ABB<T>::nodo n;
6     //el operator ++ de nodo hace avanzar a nodo en inorden
7     for (n=A.begin(); n!=A.end();++n){
8       std::cout<<*n<<std::endl;
9     }
10  }
11  typename <class T>
12  void LeerDatos(ABB<T> & A){
13    T dato;
14    while (std::cin>>dato){
15      A.Insertar(dato);
16    }
```

```
17   }
18
19   int main(){
20     ABB<char >A;
21     LeerDatos(A);
22     ImprimirAbb(A);
23   }
```

Ejemplo 6.6.5
Almacenar la información de un conjunto de alumnos (dni, nombre, apellidos, email) para que se encuentre ordenada por dni.

```
1  #include <ABB.h>
2  #include <iostream>
3  #include <string>
4  struct alumno{
5   string dni;
6   string nombre;
7   string apellidos;
8   string email;
9  };
10 bool operator<(const alumno &a1, const alumno &a2)const{
11    return a1.dni<a2.dni;
12 }
13 //suponemos que los campos de cada alumno se encuentra
14 //en una linea
15 std::istream & operator>>(std::istream &is,alumno & a){
16    is.getline(a.dni);
17    is.getline(a.nombre);
18    is.getline(a.apellidos);
19    is.getline(a.email);
20    return is;
21 }
22 typename <class T>
23 void LeerDatos(ABB<T> & A){
24    T dato;
25    while (std::cin>>dato){
26      A.Insertar(dato);
27    }
28 }
29
30  int main(){
31    ABB<alumno>A;
```

```
32      LeerDatos(A);
33      //hacemos algo con los datos
34      ...
35  }
```

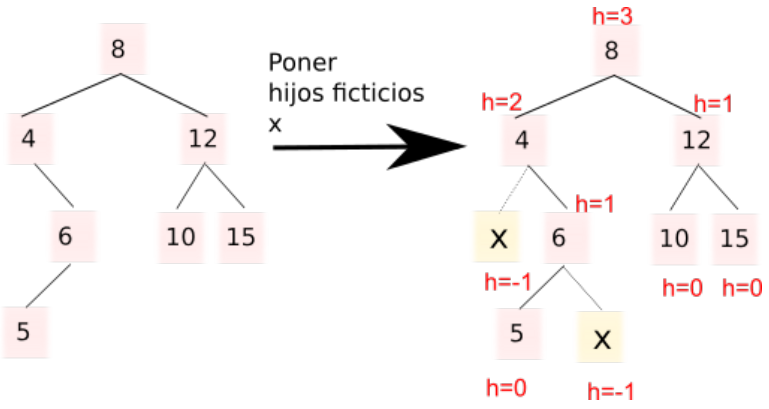
□

6.7 Arboles binarios de búsqueda AVL (Adelson-Velskii y Landis)

Son árboles de búsqueda equilibrados. Si un ABB está muy desequilibrado, los tiempos de búsqueda no son $log_2(n)$, en el peor de los casos podría ser $O(n)$. Lo ideal sería que ambas partes tuvieran más o menos el mismo número de nodos para que en cada iteración, descartar la mitad de nodos del árbol y por tanto, de verdad tener un tiempo de búsqueda $log_2(n)$.
Se dice que un ABB es AVL si la diferencia de altura de los subárboles izquierdo (T_i) y derecho (T_d) que cuelgan de un nodo es como mucho 1. Es decir, si T_i tiene altura h, T_d puede tener como máximo altura $h+1$ y viceversa.
Si no recuerdas lo que era la altura, repasa la seccion 6.2.1.

Ejemplo 6.7.1

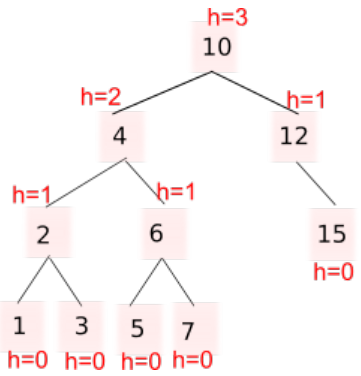
Dado el árbol a la izquierda vamos a obtener su altura. Pero antes vamos a transformarlo en el árbol de la derecha. Este árbol se obtiene añadiendo al árbol de la izquierda el hijo que le falta cuando un nodo tiene un sólo hijo. A este hijo ficticio le hemos puesto etiqueta x.



1. Los nodos que no existen (x) tienen altura $h = -1$
2. Las hoja tienen altura $h = 0$
3. Por ejemplo el nodo 6 tiene altura 1, ya que sería la altura máxima de sus hijos más 1.
4. Tenemos un desequilibrio en el 4, ya que sus hijos tienen $h(x) = -1$ y $h(2) = 1$, la altura de ambos difiere en más de 1, por lo que no es AVL.

□

Ejemplo 6.7.2



Este árbol si está equilibrado, aunque no tengamos el mismo número de nodos en T_i y T_d ya que se cumple la definición de AVL. □
La representación en C++ sería:

```
template <class T>
struct info_nodo_AVL {
      T et;
      info_nodo_AVL<T> * hijoizq, * hijoder, * padre;
      int altura;
};

// El proceso de busqueda es identico al ABB normal.
```

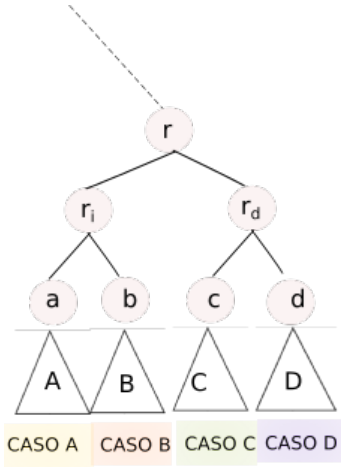
Cada nodo del árbol almacena su altura. Al realizar una inserción en el árbol la altura del nodo puede verse afectada.

6.7.1 Inserción en un AVL

En el proceso de inserción podemos desequilibrar el árbol, por tanto, debemos volver a hacer que esté equilibrado. Por tanto los pasos a seguir para insertar un elemento en un AVL serían:

1. Buscar dónde insertar el nuevo elemento
2. Insertarlo
3. Equilibrar el árbol

Como se puede ver en la siguiente figura el desequilibrio ocurre en el nodo r. Pero puede ocurrir porque se haya realizado la nueva inserción en el subárbol A, B, C o D.

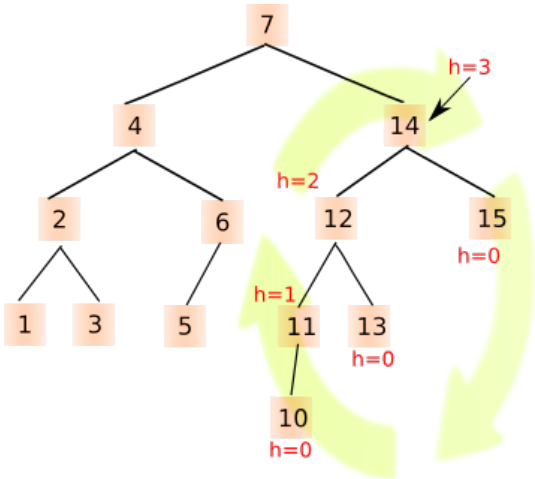


El procedimiento para volver a equilibrar el árbol depende de dónde se haya hecho la nueva inserción. Para lograr el equilibrio se aplicará rotaciones simples (ocurren cuando la nueva inserción se ha hecho en el subárbol A o D), o rotaciones dobles que ocurren cuando se hace la nueva inserción en los subárboles B y C. Veamos en mayor detalle estos casos.

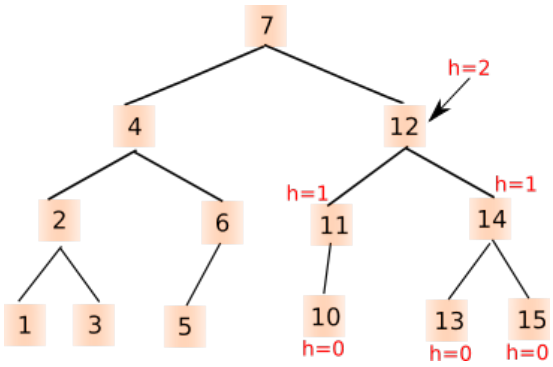
6.7.2 Rotaciones simples

CASO A : El desequilibrio se produce al insertar un nuevo elemento en la parte más a la izquierda del árbol (subárbol A). Para equilibrarlo de nuevo, se hace una *rotación simple a la derecha*.

Ejemplo 6.7.3



Supongamos que el árbol estaba equilibrado y se inserta la clave 10 dando lugar al árbol que se ve en la imagen anterior. En este caso, el desequilibrio está en 14, pues la altura de 12 es $h = 2$ y la de 15, $h = 0$. Al hacer la rotación simple a la derecha, el árbol queda ya equilibrado:



□

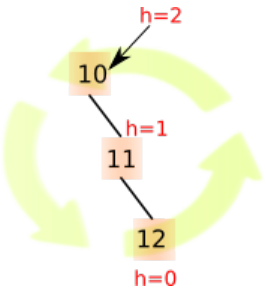
El algoritmo asociado a la rotación simple a derecha sería.:

```
template <class T>
void SimpleDerecha (info_nodo_AVL<T> * & n) { // n = 14 en el ejemplo
        info_nodo_AVL<T> * aux = n->hijoizq; // 12 en el ejemplo
        info_nodo_AVL<T> * padre = n->padre; // 7
        // a 14 le ponemos como hijo izquierdo 13
        n->hijoizq = aux->hijoder;
        if (n->hijoizq != 0)
          // el padre de 13 pasa a ser 14
            n->hijoizq->padre = n;
        n->padre = aux;//el padre de 14 pasa a ser 12
        aux->padre = padre;// el padre de 12 es 7
        aux->hder = n;// 12 tiene como hijo derecho a 14
        n = aux;
        ActualizarAltura(n->hder);
}

// Esta funcion Actualiza el campo n->altura
template <class T>
void ActualizarAltura (info_nodo_AVL<T> * & n) {
    if (n != 0) {
        n->altura = std::max(Altura(n->hijoizq), Altura(n->hijoder))+1;
        // La funcion Altura devuelve n->altura si es
        // distinta de 0 y -1 si es 0
        ActualizarAltura(n->padre);
    }
}
```

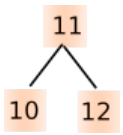
Como se puede analizar en la función ActualizarAltura se debe modificar si es necesario la altura de toda la rama hasta llegar al nodo raiz.

CASO D : El nodo que produce el desequilibrio se inserta en el subárbol D, para volver a equilibrarlo se hace una *rotación simple a la izquierda*.

Ejemplo 6.7.4

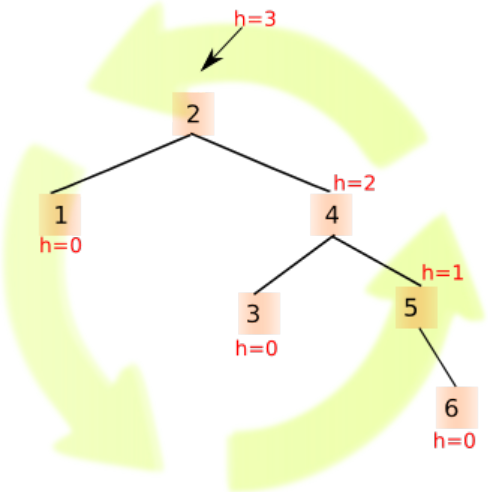


En este caso, el desequilibrio se encuentra en 10. Hay que recordad que si falta el hijo a la izquierda de 10, creamos un nodo ficticio que tiene altura -1 (hermano de 11) por lo tanto la diferencia es 2 en altura. El árbol, una vez equilibrado haciendo rotación simple a la izquierda quedaría así:

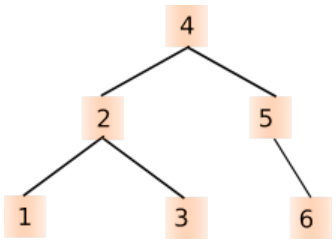


□

Ejemplo 6.7.5



En este caso el desequilibrio estaría en el 2, ya que su hijo derecho tiene altura $h = 2$ y el izquierdo, $h = 0$. Se resolvería haciendo una rotación simple a la izquierda:



El 3 se pone como hijo a la derecha de 2 para mantener la condición de AVL. □
La implementación en C++ sería:

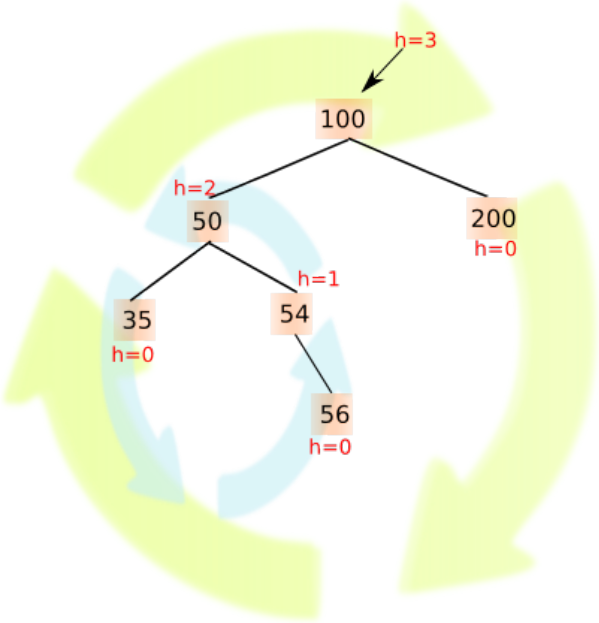
```
template <class T>
void SimpleIzquierda (info_nodo_AVL<T> * & n) { // n = 2
        info_nodo_AVL<T> * aux = n->hijoder;        // 4 en el ejemplo
        info_nodo_AVL<T> * padre = n->padre;        // nulo
        n->hder = aux->hijoizq; // a 2 se le pone a 3 como hijo derecho
        if (n->hder!=0)
          n->hder->padre = n;// el padre de 3 pasa a ser 2

        n->padre = aux;     // el padre de 2 es 4
        aux->padre = padre; // el padre de 4 es nulo, porque es la raiz
        aux->hijoizq = n;  // el hijo izquierdo de 4 es 2
        n = aux;
        ActualizarAltura(n->hijoizq);
}
```

6.7.3 Rotaciones dobles

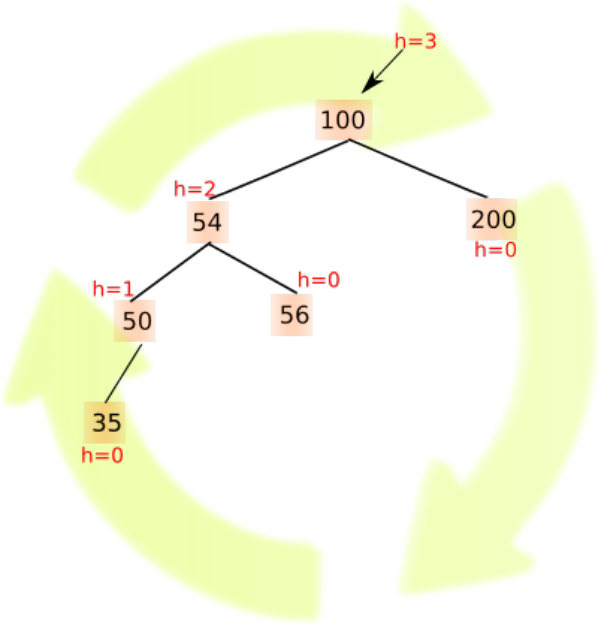
CASO B : el desequilibrio se produce al insertar un nodo en el subárbol B. Para equilibrar el árbol debemos seguir dos pasos:

1. Hacer una rotación simple a la izquierda sobre el hijo izquierdo del nodo donde se produzca el desequilibrio
2. Hacer una rotación simple a la derecha sobre el nodo donde surge el desequilibrio.

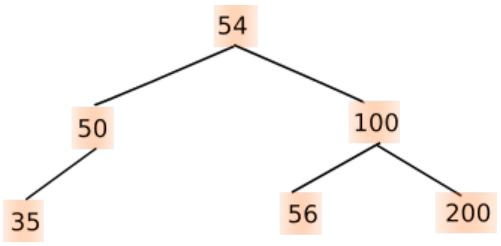
Ejemplo 6.7.6



El desequilibrio se da en el nodo con etiqueta 100, en este caso, para equilibrarlo debemos dar dos pasos. En primer lugar debemos hacer una rotación simple a la izquierda en el nodo de etiqueta 50:



Pero, el árbol aún no está equilibrado, falta el último paso que sería hacer una rotación simple a la derecha sobre 100:



□

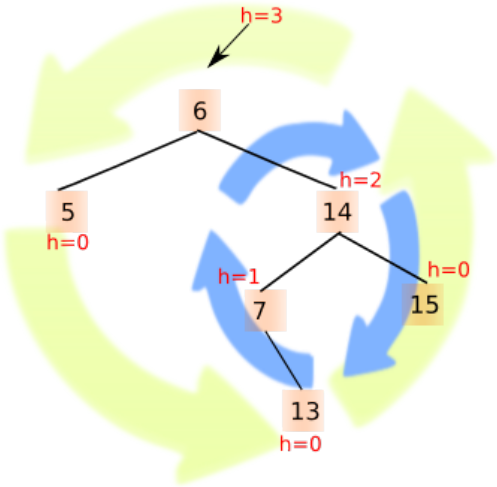
La rotación doble consistiría, por tanto, en llamar a las funciones de rotación simples pasando como argumento los nodos correspondientes:

```
template <class T>
void Doble_IzquierdaDerecha (info_nodo_AVL<T> * & n) {
        SimpleIzquierda (n->hijoizq);
        SimpleDerecha(n);
}
```

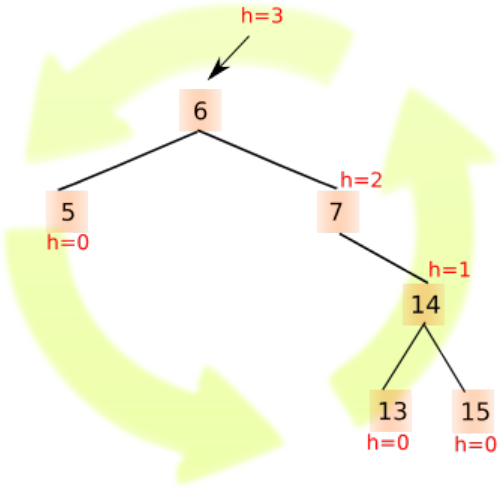
CASO C: es equivalente pero hay que hacerlo al contrario, es decir, los pasos a seguir serían:

1. Hacer una rotación simple a la derecha sobre el hijo derecho del nodo que tenga desequilibrio
2. Y hacer una rotación simple a la izquierda sobre el dicho nodo.

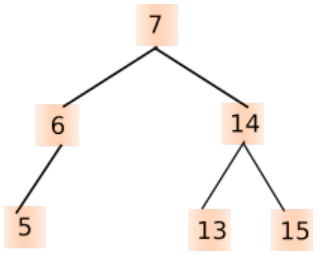
Ejemplo 6.7.7



El desequilibrio está en el nodo de etiqueta 6, para equilibrar el árbol debemos hacerlo en dos pasos. En primer lugar, haremos una rotación simple a la derecha sobre 14:



Y por último, hacemos una rotación simple a la izquierda sobre el nodo desequilibrado, 6:



□

El código implementado en C++ sería:

```
template <class T>
void Doble_DerechaIzquierda (info_nodo_AVL<T> * & n) {
        SimpleDerecha(n->hijoder);
        SimpleIzquierda (n);
}
```

Ejemplo 6.7.8
Dada una lista de números crear un árbol binario de búsqueda AVL:

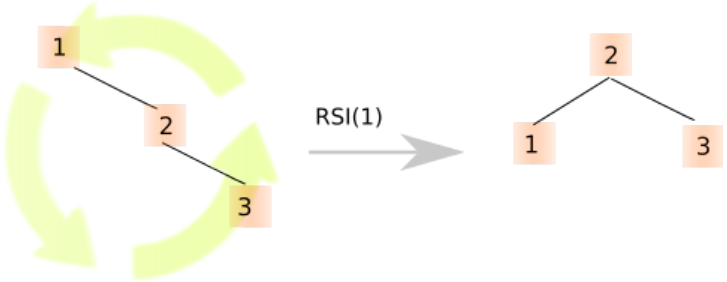
$$\{1,2,3,4,5,6,7,15,14,13,12,11,10,9\}$$

Los pasos para resolver este ejercicio son, elemento a elemento:

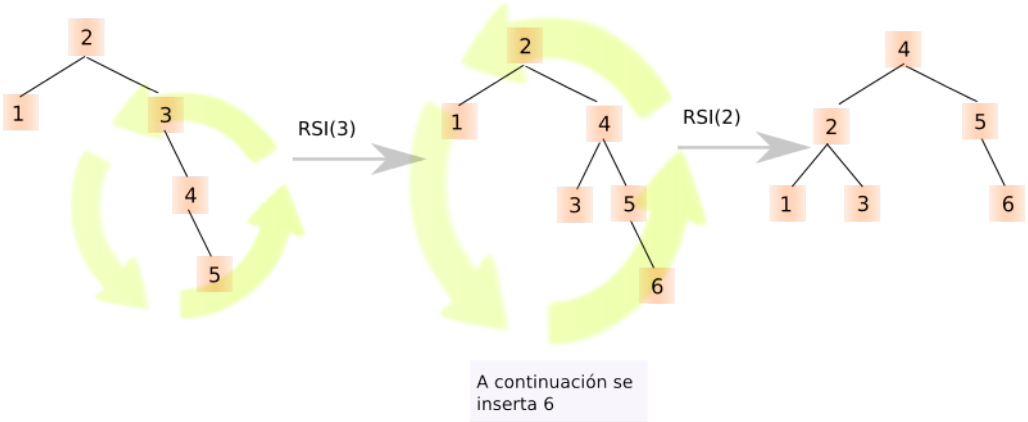
1. Insertar el elemento que corresponda.
2. Comprobar que el árbol está equilibrado.
   a) Si lo está, seguimos insertando elementos donde corresponda
   b) Si no lo está, lo equilibramos antes de seguir insertando.

Entonces, empezaríamos insertando el 1 como raíz del árbol y al ser el único nodo no quedaría desequilibrado. Después, insertaríamos el 2 como hijo a la derecha y seguiría estando equilibrado,

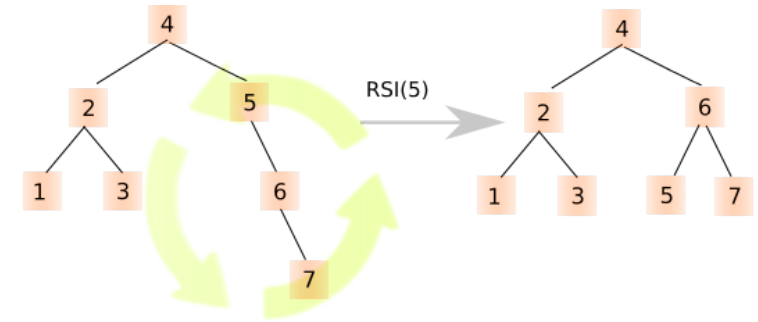
así que por último, insertamos el 3 como hijo a la derecha de 2 y como resultado, tenemos un desequilibrio **caso D**, por lo que lo equilibramos haciendo una rotación simple a la izquierda sobre 1:



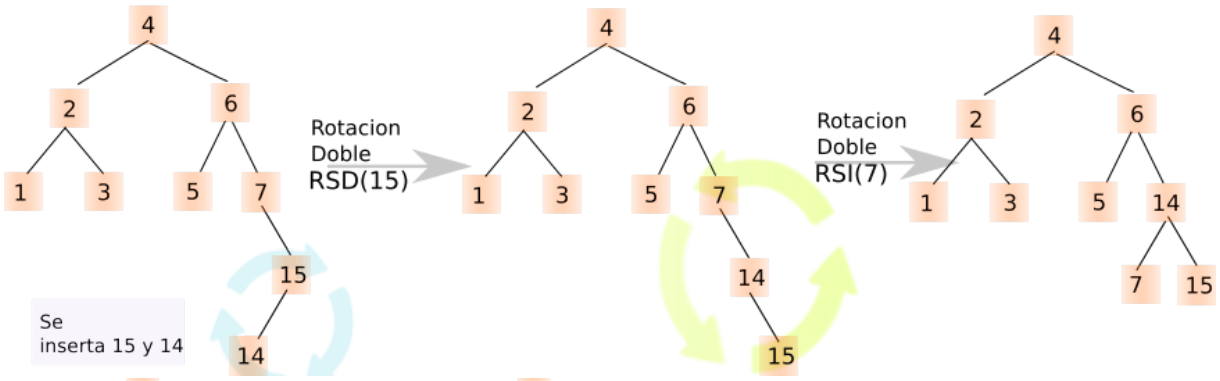
A continuación se hace la inserción de las claves 4, 5, y 6. Cuando se hace la inserción de la clave 5 se produce un desequilibrio en 3. A continuación se introduce la clave 6 y se produce un desequilibrio en 2.



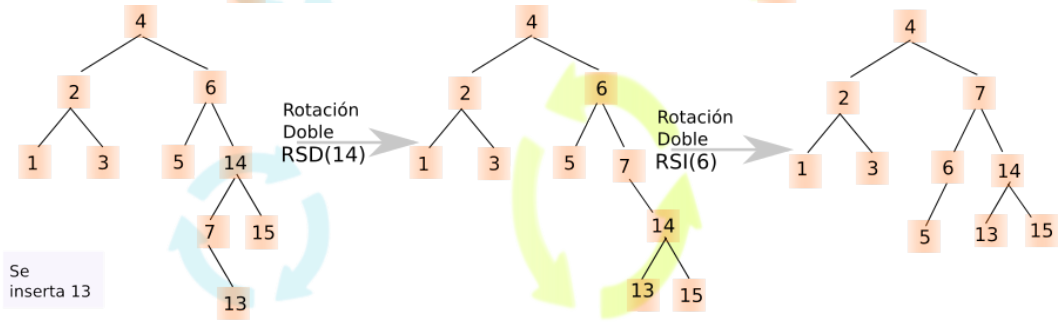
Ahora se inserta 7 y se aplica una rotación simple a izquierda. Al insertar 15 y 14 se debe aplicar una rotación doble compuesta de una rotación simple a derecha sobre 15 y un rotación simple a izquierda sobre 7. Y así seguiríamos paso por paso con cada número. Como se puede observar a continuación las rotaciones dobles deben hacerse en dos pasos:



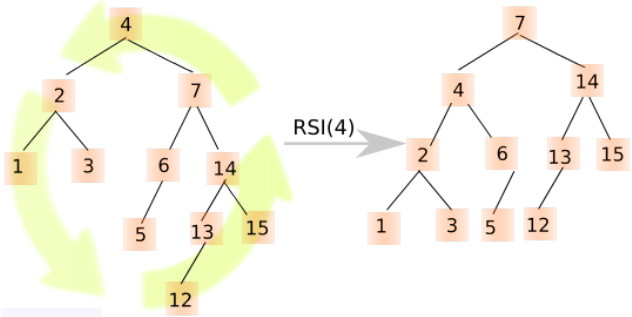
Se inserta 7



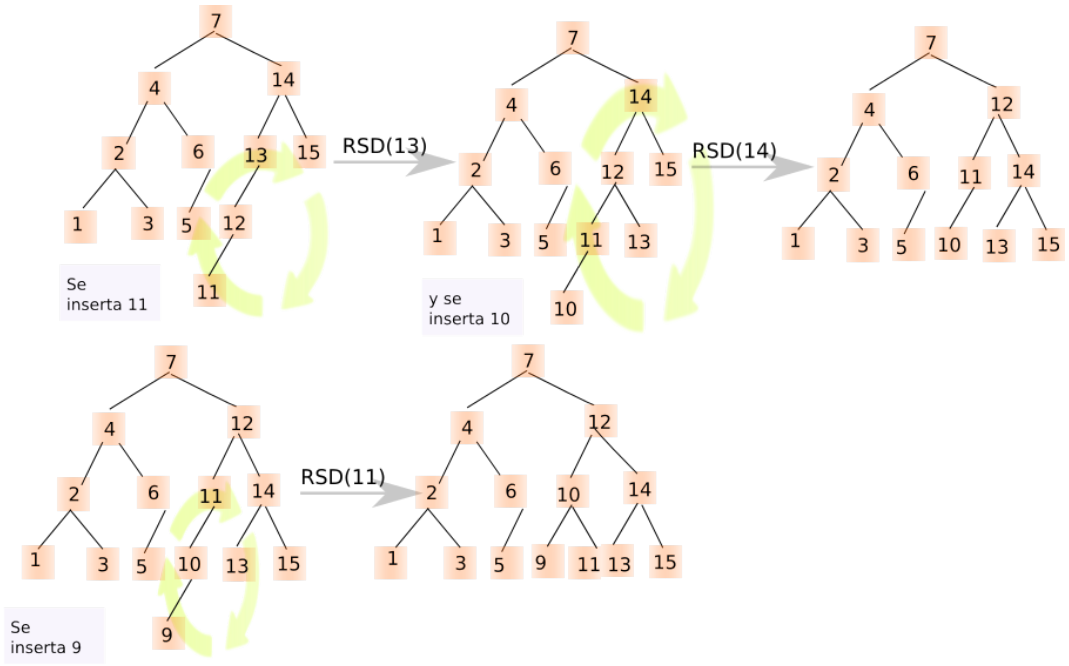
Se inserta 15 y 14



Se inserta 13



Se inserta 12



□

Por tanto, el algoritmo de inserción en C++ sería:

```
53 template <class T>
54 bool InsertarAVL (info_nodo_AVL<T> * & raiz, T x) {
55   if (raiz == 0) {
56     // constructor: crea un nodo vacio con et=x
57     raiz = new info_nodo_AVL(x);
58     raiz->altura = 0;
59     return true; // el nodo ha podido insertarse con exito
60   }
61   else {
62     if (x < raiz->et) {
63       if (raiz->hijoder != 0)
64       raiz->hijoder->padre = raiz;
65       if (InsertarAVL(raiz->hijoizq, x)) { // el arbol ha crecido,
66       // vemos la diferencia de altura entre ambos hijos
67       switch (Altura(raiz->hijoizq) - Altura(raiz->hijoder)) {
68       // los valores del switch deben ser 0, 1 o 2, si son mayores
69       // en algun momento no hemos insertado un elemento bien y
70       // nuestro arbol no esta equilibrado.
71       case 0:
72       return false; // el arbol no ha crecido
73
74       case 1: // ha crecido por la izquierda, sumamos 1 a la altura
75       raiz->altura++;   // de la raiz
```

```
76          return true;
77
78          case 2:
79          /* CASO A*/
80          if (Altura(raiz->hijoizq->hijoizq) >
81          Altura(raiz->hijoizq->hijoder))
82          SimpleDerecha(raiz);
83
84          /*CASO B*/
85          else
86          Doble_IzquierdaDerecha(raiz);
87
88          return false; // la altura no crece porque hemos
89                // equilibrado el arbol
90          }
91          }
92        }
93
94        else { // x es mayor que la etiqueta
95          if (raiz->hijoizq != 0)
96          raiz->hijoizq->padre = raiz;
97
98          if (InsertarAVL(raiz->hijoder, x)) {
99          switch (Altura(raiz->hijoder) - Altura(raiz->hijoizq)) {
100         case 0:
101         return false; // el arbol no ha crecido
102
103         case 1: // ha crecido por la izquierda, sumamos 1 a la altura
104         raiz->altura++;   // de la raiz
105         return true;
106
107         case 2:
108         /* CASO D */
109         if (Altura(raiz->hijoder->hijoder) >
110         Altura(raiz->hijoder->hijoizq))
111         SimpleIzquierda(raiz);
112
113         /* CASO C */
114         else
115         Doble_DerechoIzquierda(raiz);
116
117         return false;
118         }
```

```
119             }
120         }
121     }
122 }
```