UNIVERSIDAD NEBRIJA

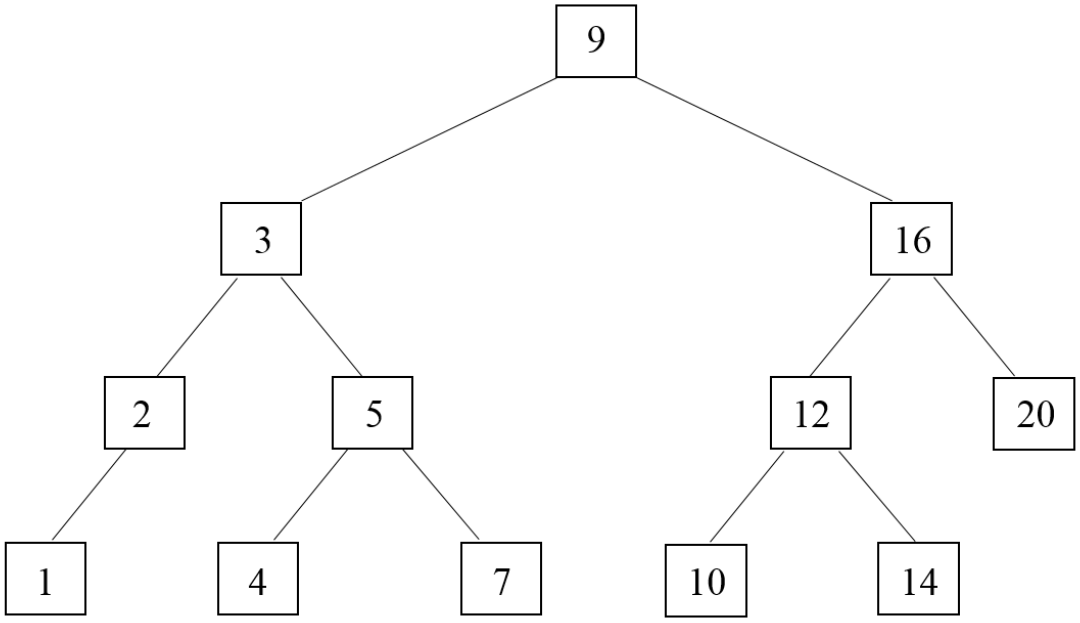
Escuela Politécnica Superior

Estructuras de datos y algoritmos

Práctica 4

Los árboles AVL son árboles binarios de búsqueda autobalanceados. Fueron desarrollados por Adelson-Velskii y Landis en 1962.

Un árbol AVL es un árbol binario de búsqueda que cumple que la diferencia entre las alturas de los subárboles de cada uno de sus nodos es, como máximo 1. Los árboles AVL siempre están balanceados, por lo que la complejidad de la búsqueda en un árbol AVL es $O(log\ n)$.



Un árbol AVL se define:

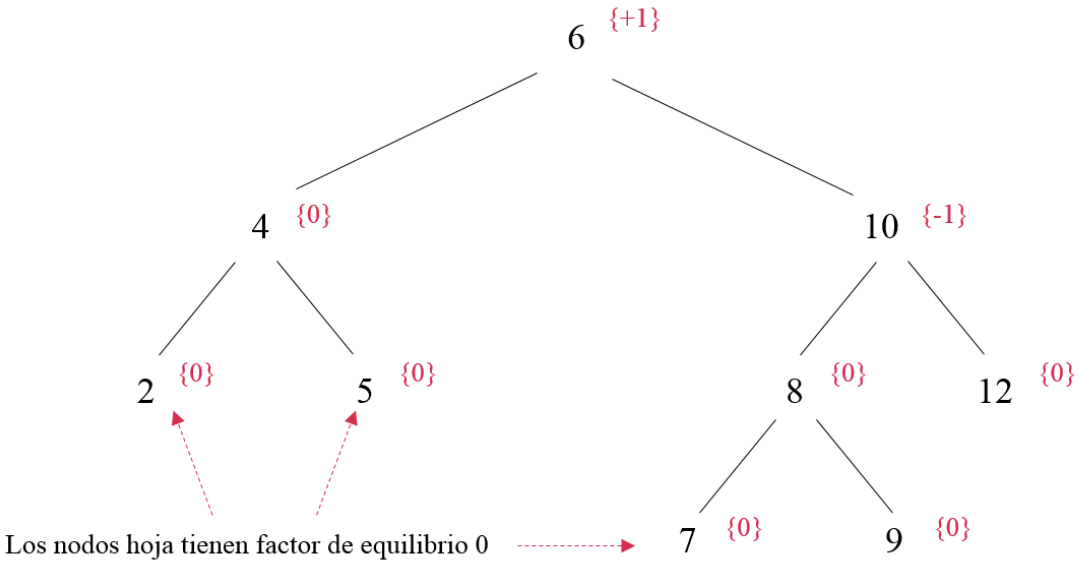
- Un árbol vacío es un árbol AVL
- Si *T* es un árbol binario no vacío en el que *Tizquierdo* y *Tderecho* son sus ramas izquierda y derecha, *T* es un árbol AVL si y solo si se cumple:

  Tizquierdo es un árbol AVL

  Tderecho es un árbol AVL

  $|altura(Tderecho) - altura(Tizquierdo)| \leq 1$

El factor de equilibrio de un nodo es la altura de su rama derecha menos la altura de su rama izquierda. El factor de equilibrio de los nodos de un árbol AVL balanceado toma los valores $+1, 0, -1$.

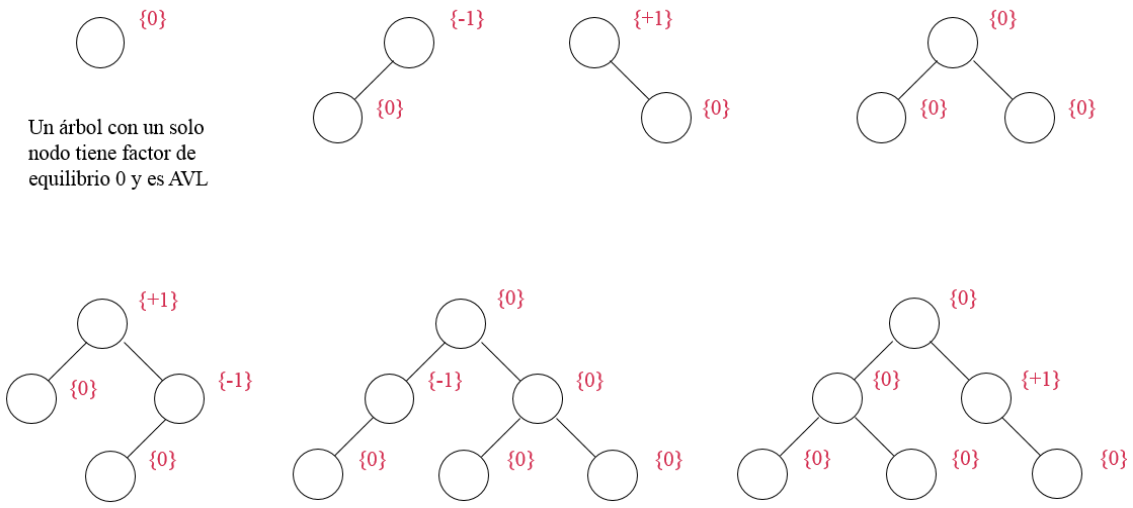


El factor de equilibrio del nodo raíz es +1, la rama derecha tiene altura 3 y la rama izquierda altura 2

Para todos los nodos de un árbol AVL balanceado se cumple:

- Si la altura de la rama derecha de un nodo es mayor que la altura de la rama izquierda, el factor de equilibrio es $+1$.
- Si el nodo es una hoja o sus ramas derecha e izquierda tienen la misma altura, el factor de equilibrio es 0.
- Si la altura de la rama derecha de un nodo es menor que la altura de la rama izquierda, el factor de equilibrio es -1.

Si el factor de equilibrio de un nodo es mayor que |1| es necesario rotar los nodos para balancear el árbol.

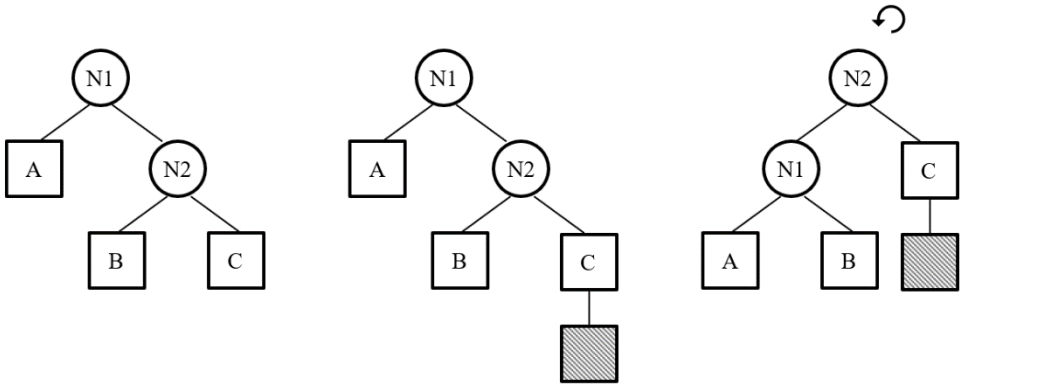


Cuando se inserta o se elimina un nodo de un árbol AVL balanceado se puede perder la propiedad de equilibrio. Para mantener la propiedad de equilibrio de los nodos del árbol AVL se realizan rotaciones de nodos.

- Rotación simple a la izquierda
- Rotación simple a la derecha
- Rotación doble izquierda, derecha
- Rotación doble derecha, izquierda

Las operaciones de rotación de nodos mantienen el orden de los valores almacenados en los nodos del árbol. Después de realizar una rotación, el árbol resultante es un árbol binario de búsqueda.

Rotación simple a la izquierda.

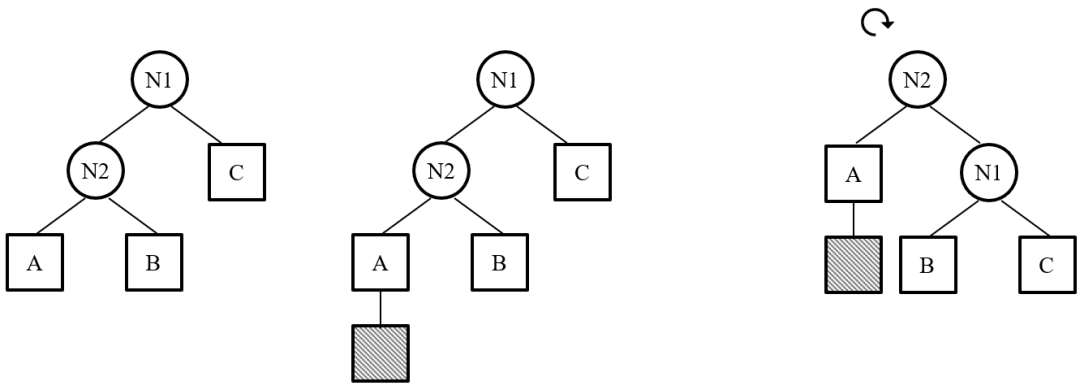


Árbol balanceado antes de insertar el nuevo nodo

Árbol desbalanceado después de insertar un nodo en la rama C

El árbol se balancea con la rotación simple a la izquierda de los nodos N1 y N2

Rotación simple a la derecha.

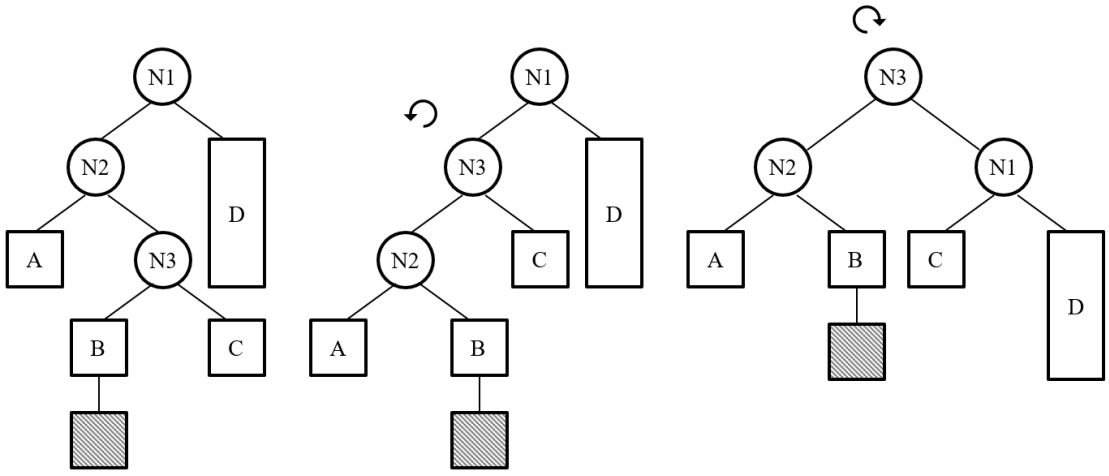


Árbol balanceado antes de insertar el nuevo nodo

Árbol desbalanceado después de insertar un nodo en la rama A

El árbol se balancea con la rotación simple a la derecha de los nodos N1 y N2

Rotación doble izquierda-derecha.

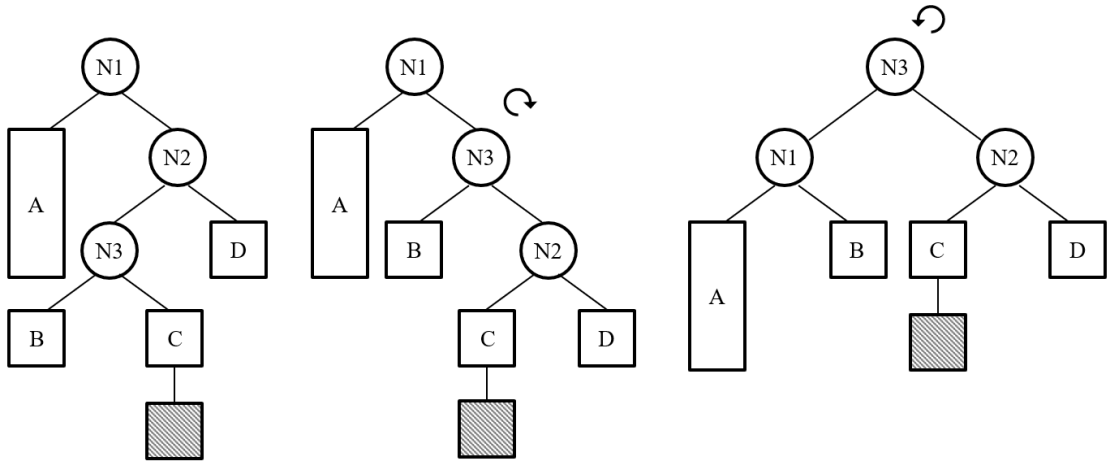


Árbol desbalanceado

Árbol después de rotar a la izquierda los nodos N2 y N3

Árbol balanceado después de rotar a la derecha los nodos N3 y N1

Rotación doble derecha-izquierda.



Árbol desbalanceado

Árbol después de rotar a la derecha los nodos N2 y N3

Árbol balanceado después de rotar a la izquierda los nodos N3 y N1

Si un nodo *T* tiene factor de equilibrio +2, la altura de la rama derecha es mayor que la altura de la rama izquierda. Para balancear el nodo *T* se realiza una rotación simple a la izquierda.

Si un nodo *T* tiene factor de equilibrio -2, la altura de la rama izquierda es mayor que la altura de la rama derecha. Para balancear el nodo *T* se realiza una rotación simple a la derecha.

La interfaz del TAD árbol balanceado AVL.

```
inicializa(R) Inicializa R a un árbol vacío

  Precondición:  Ninguna
  Postcondición: Un árbol binario de búsqueda vacío

vacio(R) Devuelve verdadero si el árbol R está vacío y falso en
         cualquier otro caso

  Precondición:  Ninguna
  Postcondición: true si el árbol R está vacío y false e.o.c.

inserta(R, v) Inserta un nodo con el valor v

  Precondición:  Ninguna
  Postcondición: El nodo con el valor v ocupa la posición que
                 le corresponde en el árbol R

busca(R, v) Busca el valor v en el árbol R

  Precondición:  Ninguna
  Postcondición: Puntero al nodo con valor v o null


elimina(R, v) Elimina el nodo de R que almacena el valor v

  Precondición:  Ninguna
  Postcondición: El árbol R ya no tiene un nodo con el valor v

preorder(R) Recorre los nodos del árbol R en “preorder”

  Precondición:  Ninguna
  Postcondición: Ninguna

inorder(R) Recorre los nodos del árbol R en “inorder”

  Precondición:  Ninguna
  Postcondición: Ninguna

postorder(R) Recorre los nodos del árbol R en “postorder”

  Precondición:  Ninguna
  Postcondición: Ninguna
```

Implemente el TAD Árbol AVL utilizando el siguiente código.

```
class NodoArbolAVL {
public:

   int dato, altura, equilibrio;
   NodoArbolAVL *izquierdo, *derecho;
};

class ArbolAVL {
public:

   ArbolAVL();
   ~ArbolAVL();

   bool vacio();
   void inserta(int v);
   void elimina(int v);
   NodoArbolAVL* busca(int v);
   std::string imprime();
   std::string preorder();
   std::string inorder();
   std::string postorder();
   int altura();

private:

   NodoArbolAVL* raiz;

   void inserta(NodoArbolAVL*& r, int v);
   NodoArbolAVL* insertaNodo(int v);
   void elimina(NodoArbolAVL*& r, int v);
   int eliminaMinimo(NodoArbolAVL*& r);
   NodoArbolAVL* busca(NodoArbolAVL* r, int v);

   std::string preorder(NodoArbolAVL* r);
   std::string inorder(NodoArbolAVL* r);
   std::string postorder(NodoArbolAVL* r);
   int max(int a, int b);
   int altura(NodoArbolAVL* r);
   int equilibrio(NodoArbolAVL* r);

   void rotacionIzquierda(NodoArbolAVL*& r);
   void rotacionDerecha(NodoArbolAVL*& r);
   void autobalance(NodoArbolAVL*& r);

   NodoArbolAVL* minimo(NodoArbolAVL* r);
   NodoArbolAVL* maximo(NodoArbolAVL* r);

};
```

```
void ArbolAVL::inserta(NodoArbolAVL*& r, int v) {
   if (r == nullptr)
      r = insertaNodo(v);
   else {
      if (v < r->dato) {
         if (r->izquierdo == nullptr)
            r->izquierdo = insertaNodo(v);
         else
            inserta(r->izquierdo, v);
      }
      else {
         if (v > r->dato)
            if (r->derecho == nullptr)
               r->derecho = insertaNodo(v);
            else
               inserta(r->derecho, v);
      }

      r->altura = altura(r);
      r->equilibrio = equilibrio(r);

      if (abs(r->equilibrio) > 1)   // autobalance del árbol
         autobalance(r);
   }
}

void ArbolAVL::rotacionIzquierda(NodoArbolAVL*& r) {

   // rotación a la izquierda

}

void ArbolAVL::rotacionDerecha(NodoArbolAVL*& r) {

   // rotación simple a la derecha

}

void ArbolAVL::autobalance(NodoArbolAVL*& r) {
   // si el factor de equilibrio es +2, rotación a la izquierda
   // si el factor de equilibrio es -2, rotación a la derecha

   if (r->equilibrio == 2) {
      rotacionIzquierda(r);
   }
   else if (r->equilibrio == -2) {
      rotacionDerecha(r);
   }
}
```

El programa de prueba del TAD Árbol AVL.

```
int main() {

   std::cout << "Arbol balanceado AVL \n";

   //
   // caso 1. rotacion simple a la izquierda
   //

   ArbolAVL avl1 = ArbolAVL();

   avl1.inserta(8);
   avl1.inserta(1);
   avl1.inserta(12);
   avl1.inserta(10);
   avl1.inserta(15);

   std::cout << "\n";
   std::cout << "\n- caso 1. rotacion simple a la izquierda";
   std::cout << "\n";

   std::cout << avl1.imprime();
   avl1.inserta(20);
   std::cout << "\n- caso 1. rotacion después de insertar el 20 \n";
   std::cout << avl1.imprime();

   //
   // caso 2. rotacion simple a la derecha
   //

   ArbolAVL avl2 = ArbolAVL();

   avl2.inserta(15);
   avl2.inserta(10);
   avl2.inserta(20);
   avl2.inserta(5);
   avl2.inserta(12);

   std::cout << "\n";
   std::cout << "\n- caso 2. rotacion simple a la derecha";
   std::cout << "\n";

   std::cout << avl2.imprime();
   avl2.inserta(2);
   std::cout << "\n- caso 2. rotacion despues de insertar el 2 \n";
   std::cout << avl2.imprime();
```

```
    //
    // caso 3. rotacion doble izquierda-derecha
    //

    ArbolAVL avl3 = ArbolAVL();

    avl3.inserta(15);
    avl3.inserta(2);
    avl3.inserta(20);
    avl3.inserta(1);
    avl3.inserta(6);
    avl3.inserta(25);
    avl3.inserta(5);
    avl3.inserta(8);

    std::cout << "\n";
    std::cout << "\n- caso 3. rotacion doble izquierda-derecha";
    std::cout << "\n";

    std::cout << avl3.imprime();
    avl3.inserta(3);
    std::cout << "\n- caso 3. rotacion después de insertar el 3 \n";
    std::cout << avl3.imprime();

    //
    // caso 4. rotacion doble derecha-izquierda
    //

    ArbolAVL avl4 = ArbolAVL();

    avl4.inserta(5);
    avl4.inserta(3);
    avl4.inserta(15);
    avl4.inserta(1);
    avl4.inserta(8);
    avl4.inserta(20);
    avl4.inserta(6);
    avl4.inserta(9);

    std::cout << "\n";
    std::cout << "\n- caso 4. rotacion doble derecha-izquierda";
    std::cout << "\n";

    std::cout << avl4.imprime();
    avl4.inserta(10);
    std::cout << "\n- caso 4. rotacion despues de insertar el 10 \n";
    std::cout << avl4.imprime();
}
```

La eliminación de un nodo de un árbol AVL se realiza de la misma forma que en un árbol binario de búsqueda.

- El nodo n es una hoja
- El hijo izquierdo del nodo n es nulo
- El hijo derecho del nodo n es nulo
- Los dos hijos del nodo n son distintos de nulo

Después de borrar el nodo se comprueba si es necesario balancear el árbol.

Desarrolle las siguientes funciones para eliminar nodos de un árbol AVL.

```
int ArbolAVL::eliminaMinimo(NodoArbolAVL*& r) {

   // elimina el valor mínimo del árbol con raíz r

}

void ArbolAVL::elimina(int v) {
   elimina(raiz, v);
}

void ArbolAVL::elimina(NodoArbolAVL*& r, int v) {

   // elimina el valor v del árbol

}
```

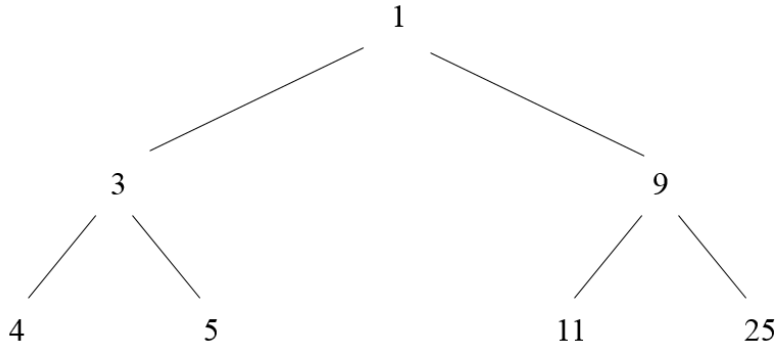
El programa de prueba del TAD Árbol AVL.

```
ArbolAVL avl5 = ArbolAVL();

avl5.inserta(2);
avl5.inserta(1);
avl5.inserta(8);
avl5.inserta(0);
avl5.inserta(5);
avl5.inserta(9);
avl5.inserta(4);
avl5.inserta(10);

std::cout << "\n";
std::cout << "\n- eliminacion de nodos";
std::cout << "\n";

std::cout << avl5.imprime();

avl5.elimina(4);
std::cout << "\n- elimina nodo 4 \n";
std::cout << avl5.imprime();

avl5.elimina(9);
std::cout << "\n- elimina nodo 9 \n";
std::cout << avl5.imprime();

avl5.elimina(2);
std::cout << "\n- elimina nodo 2 \n";
std::cout << avl5.imprime();

avl5.elimina(0);
std::cout << "\n- elimina nodo 0 \n";
std::cout << avl5.imprime();

avl5.elimina(1);
std::cout << "\n- elimina nodo 1 \n";
std::cout << avl5.imprime();
```

Un montículo (heap) es un árbol binario completo parcialmente ordenado que permite seleccionar entre un conjunto de valores, el máximo o el mínimo.

En un montículo de mínimos, los nodos padre almacenan un valor menor que el de sus hijos. En un montículo de máximos, los nodos padre almacenan un valor mayor que el de sus hijos.

Un montículo es un árbol binario completo. Dado que los nodos intermedios de un árbol binario completo tienen 2 hijos, un montículo se puede representar utilizando un array.



	Nodo	
Raíz	0	1
	1	3
	2	9
	3	4
	4	5
	5	11
	6	25

El nodo almacenado en la posición *i* de la tabla tiene su hijo izquierdo en la posición $2*i + 1$ y su hijo derecho en la posición $2*i + 2$

Para insertar un elemento en un montículo, el nuevo elemento se inserta en la siguiente posición disponible y se comprueba si se cumple el orden de un montículo de mínimos. Si no se cumple el orden del montículo, se intercambia el valor del nodo con el de su antecesor hasta llegar al nodo raíz.

Cuando se elimina un elemento, siempre se elimina el valor del nodo raíz, que almacena el valor mínimo o máximo del árbol, dependiendo del orden del montículo. Al eliminar un elemento, el nodo raíz toma el valor del último elemento del montículo. Si no se cumple el orden del montículo, se intercambia el valor del nodo raíz con sus descendientes de la rama que almacena el valor menor, hasta que el montículo queda ordenado.

La interfaz del TAD Montículo de mínimos.

```
inicializa(M) Inicializa el montículo M a vacío

vacio(M) Devuelve verdadero si el montículo está vacía y falso en
         cualquier otro caso

inserta(M, p) Inserta un elemento con prioridad p en el montículo

eliminaMínimo (M, min) Elimina el elemento mínimo del montículo
```

Implemente el TAD Montículo de mínimos utilizando el siguiente código.

```
class MonticuloMinimos {

public:

   MonticuloMinimos(int c);
   ~MonticuloMinimos();

   bool vacio();
   void inserta(int p);
   int eliminaMinimo();
   std::string imprime();

private:

   bool minimo(int &min);
   void intercambio(int i, int j);
   void intercambioAscendente(int padre, int hijo);
   void intercambioDescendente(int padre);

   int ultimo;
   int capacidad;
   int *prioridad;

};
```

```
MonticuloMinimos::MonticuloMinimos(int c) {
   if (c <= 0)
      throw std::runtime_error("Tamaño de montículo no válido");

   ultimo = -1;
   capacidad = c;
   prioridad = new int[capacidad];
}

MonticuloMinimos::~MonticuloMinimos() {
   delete prioridad;
}

bool MonticuloMinimos::vacio() {
   return (ultimo < 0);
}

void MonticuloMinimos::inserta(int p) {
   if (ultimo == capacidad - 1)
      throw std::runtime_error("Monticulo lleno");

   ultimo++;
   prioridad[ultimo] = p;
   intercambioAscendente((ultimo - 1) / 2, ultimo);
}

int MonticuloMinimos::eliminaMinimo() {
   if (vacio())
      throw std::runtime_error("Monticulo vacío");

   int min = prioridad[0];
   prioridad[0] = prioridad[ultimo--];
   intercambioDescendente(0);

   return min;
}

std::string MonticuloMinimos::imprime() {
   std::stringstream ss;

   ss << "{";

   for (int i = 0; i <= ultimo; i++)
      ss << prioridad[i] << ", ";

   ss << "} \n";

   return ss.str();
}
```

```
bool MonticuloMinimos::minimo(int &min) {
   if (vacio())
      return false;
   else
      min = prioridad[0];
   return true;
}

void MonticuloMinimos::intercambio(int i, int j) {
   int v = prioridad[i];
   prioridad[i] = prioridad[j];
   prioridad[j] = v;
}

void MonticuloMinimos::intercambioAscendente(int padre, int hijo) {
   if (prioridad[hijo] < prioridad[padre]) {
      intercambio(padre, hijo);

      if (padre != 0)
         intercambioAscendente((padre - 1) / 2, padre);
   }
}

void MonticuloMinimos::intercambioDescendente(int padre) {

   // intercambia los valores de los nodos, desde la raíz
   // hasta una hoja para mantener el orden del montículo

}
```

El programa de prueba del TAD Montículo de mínimos.

```
int main() {

   std::cout << "Monticulo de minimos \n";

   MonticuloMinimos m = MonticuloMinimos(50);

   m.inserta(4);
   m.inserta(5);
   m.inserta(6);
   m.inserta(3);
   m.inserta(11);
   m.inserta(7);
   m.inserta(9);
   m.inserta(14);
   m.inserta(10);
   m.inserta(8);

   std::cout << m.imprime();
   std::cout << "inserta 2 \n";
   m.inserta(2);
   std::cout << m.imprime();

   int min;

   min = m.eliminaMinimo();

   std::cout << "elimina minimo=" << min << "\n";

   std::cout << m.imprime();

   min = m.eliminaMinimo();

   std::cout << "elimina minimo=" << min << "\n";

   std::cout << m.imprime();
}
```