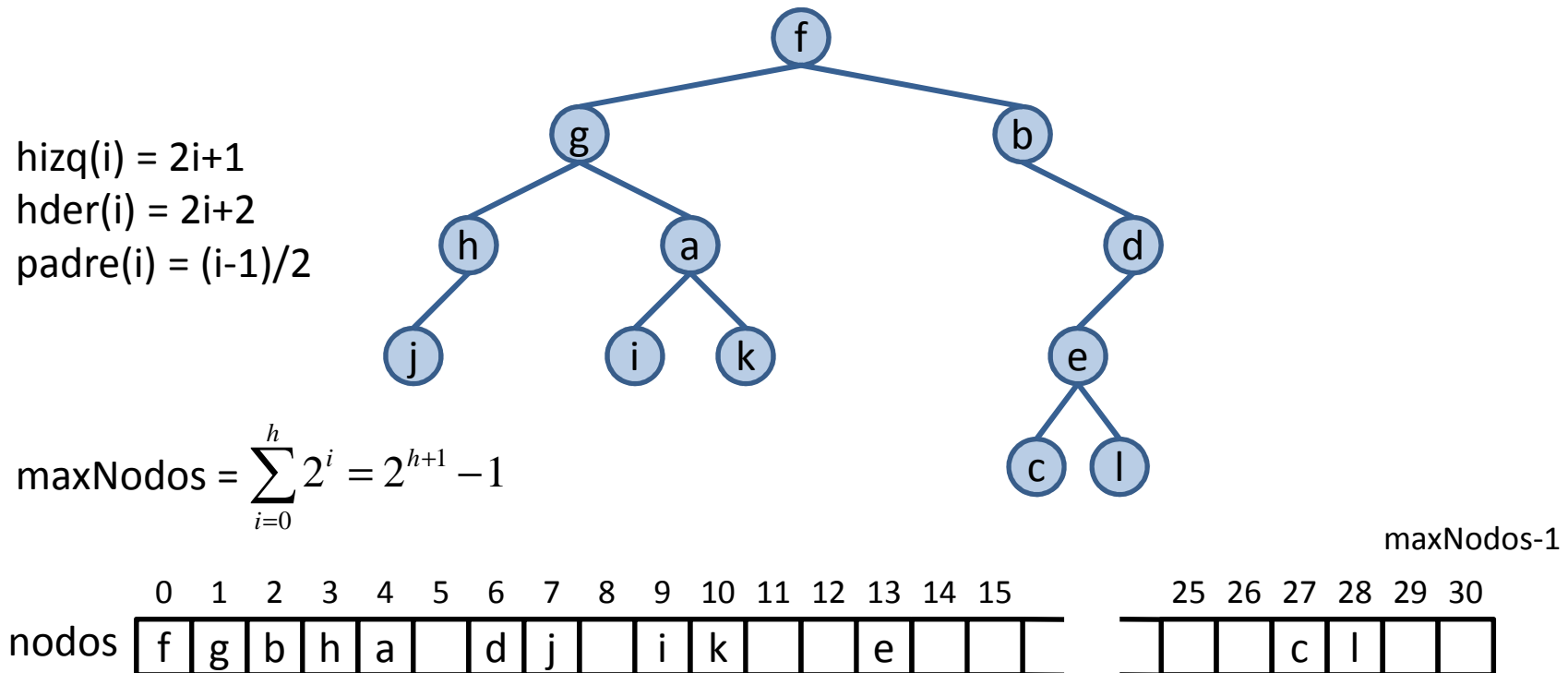


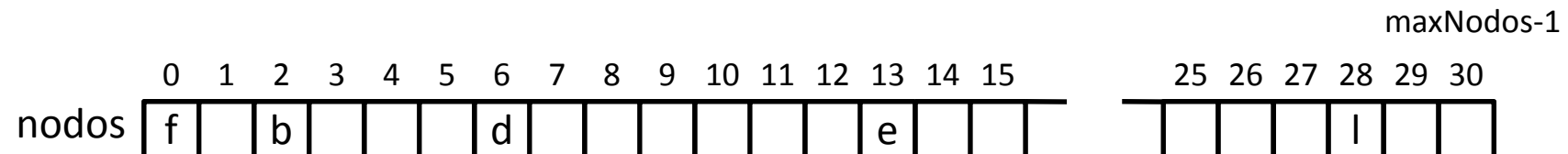
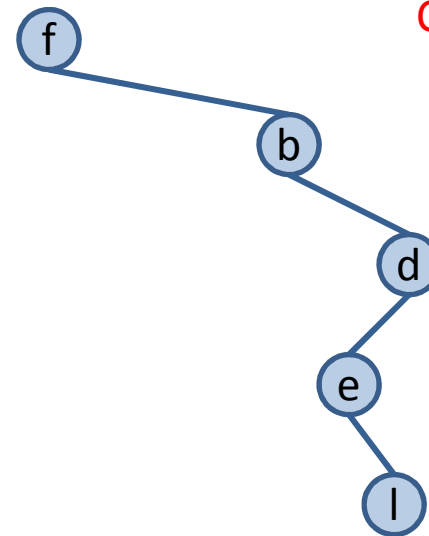
Implementación de un árbol binario mediante un vector de posiciones relativas



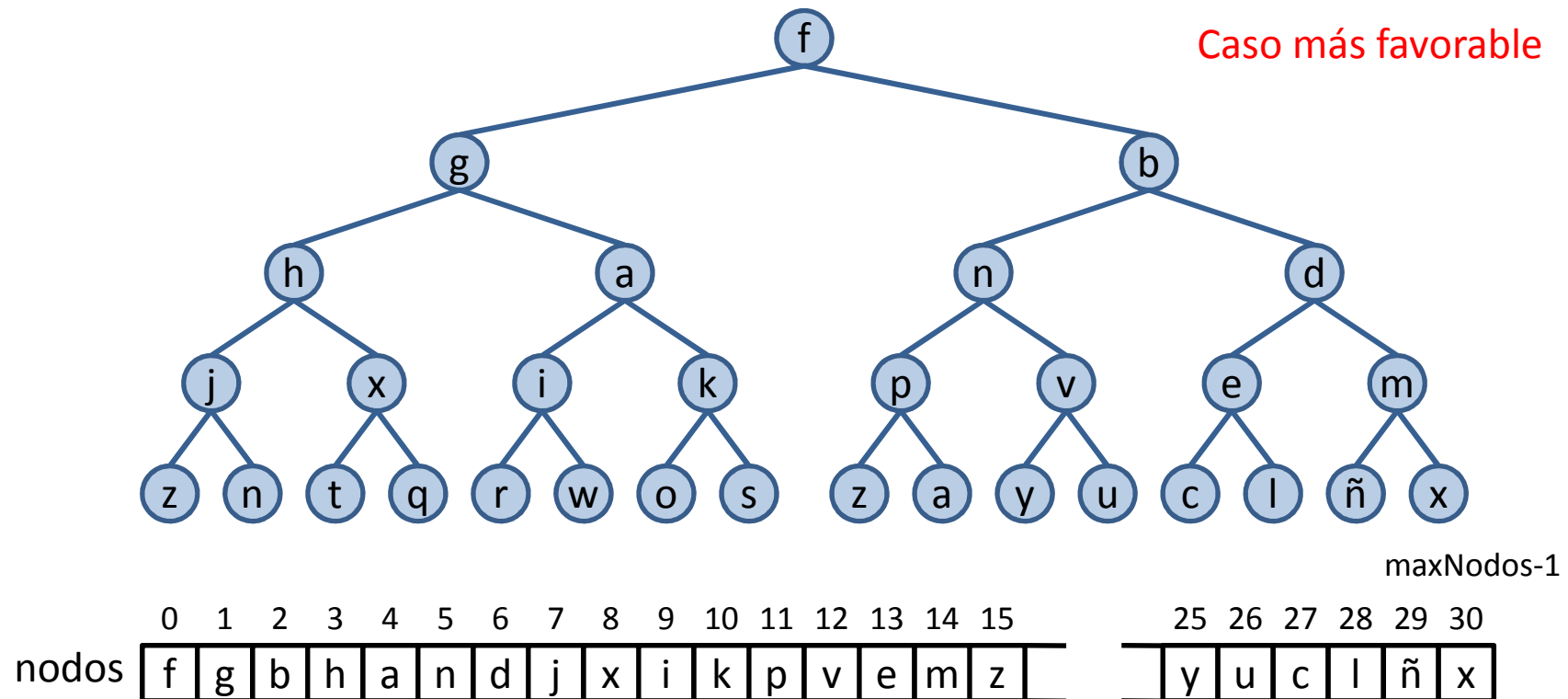
Implementación de un árbol binario mediante un vector de posiciones relativas

Sea un árbol de altura máxima h .
La ausencia de un nodo en el nivel $n \leq h$ provocará $2^{h-n+1}-1$ posiciones libres en el vector.

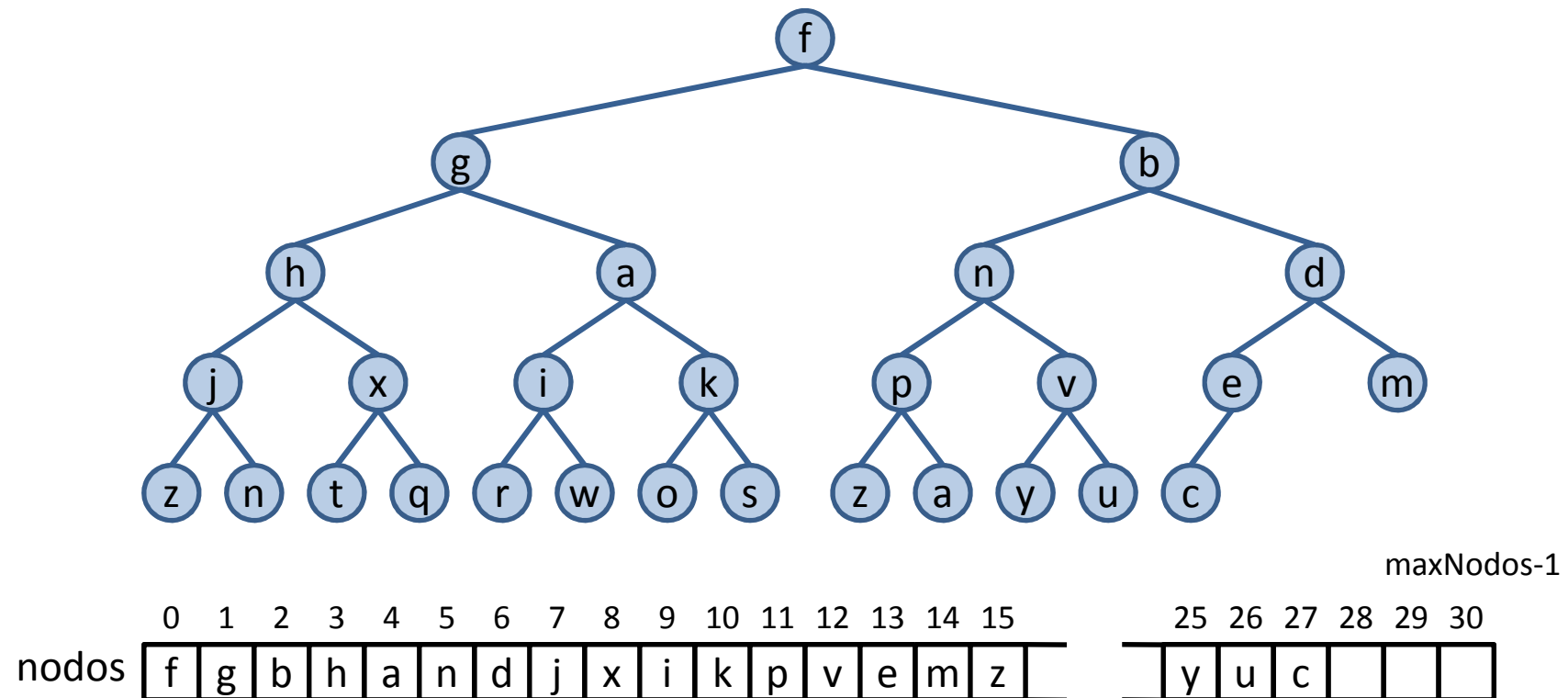
Caso más desfavorable



Implementación de un árbol binario mediante un vector de posiciones relativas

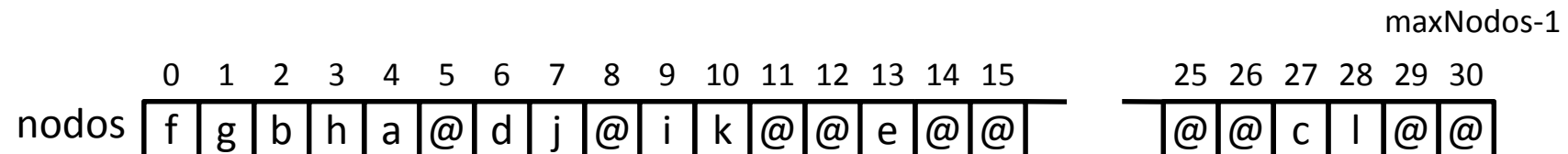
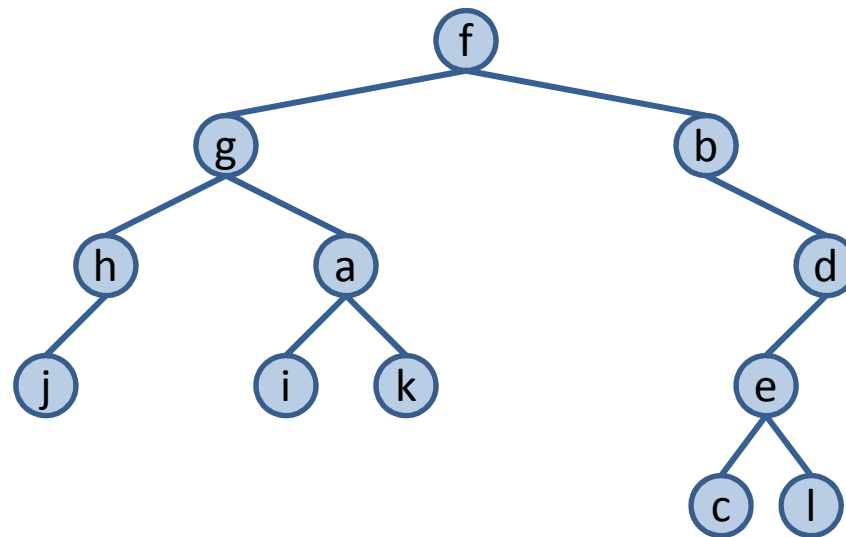


Implementación de un árbol binario mediante un vector de posiciones relativas



La eficiencia espacial será mayor cuanto más lleno esté el árbol, es decir cuantos menos nodos falten y, por tanto, más bajos sean los niveles en que falten.

Implementación de un árbol binario mediante un vector de posiciones relativas



Un valor del tipo de los elementos del árbol, no significativo en la aplicación, se utilizará para marcar las posiciones libres del vector.

```
#ifndef ABIN_VEC1_H
#define ABIN_VEC1_H
#include <cassert>

template <typename T> class Abin {
public:
    typedef int nodo; // índice del vector
                        // entre 0 y maxNodos-1

    static const nodo NODO_NULO;
    explicit Abin(size_t maxNodos, const T& e_nulo = T()); //ctor
    void insertarRaizB(const T& e);
    void insertarHijoIzqdoB(nodo n, const T& e);
    void insertarHijoDrchoB(nodo n, const T& e);
    void eliminarHijoIzqdoB(nodo n);
    void eliminarHijoDrchoB(nodo n);
    ~Abin(); // destructor
    bool arbolVacioB() const;
    void eliminarRaizB();
```

```

    const T& elemento(nodo n) const; // acceso a elto, lectura
    T& elemento(nodo n); // acceso a elto, lectura/escritura
    nodo raizB() const;
    nodo padreB(nodo n) const;
    nodo hijoIzqdoB(nodo n) const;
    nodo hijoDrchoB(nodo n) const;
    Abin(const Abin<T>& a); // ctor. de copia
    Abin<T>& operator =(const Abin<T>& a); //asignación de árboles
private:
    T* nodos; // vector de nodos
    int maxNodos; // tamaño del vector
    T ELTO_NULO; // marca celdas vacías
};

```

```

/* Definición del nodo nulo */
template <typename T>
const typename Abin<T>::nodo Abin<T>::NODO_NULO(-1);

```

```

template <typename T>
Abin<T>::Abin(size_t maxNodos, const T& e_nulo) :
    nodos(new T[maxNodos]),
    maxNodos(maxNodos),
    ELTO_NULO(e_nulo)
{
    // marcar todas las celdas libres
    for (nodo n = 0; n <= maxNodos-1; n++)
        nodos[n] = ELTO_NULO;
}

```

```

template <typename T>
inline void Abin<T>::insertarRaizB(const T& e)
{
    assert(nodos[0] == ELTO_NULO);    // árbol vacío

    nodos[0] = e;
}

```



```

template <typename T> inline
void Abin<T>::insertarHijoIzqdoB(Abin<T>::nodo n,const T& e)
{
    assert(n >= 0 && n <= maxNodos-1); // nodo válido
    assert(nodos[n] != ELTO_NULO);      // nodo del árbol
    assert(2*n+1 < maxNodos);           // hijo izqdo. cabe en el árbol
    assert(nodos[2*n+1] == ELTO_NULO); // n no tiene hijo izqdo.

    nodos[2*n+1] = e;
}

```

```

template <typename T> inline
void Abin<T>::insertarHijoDrchoB(Abin<T>::nodo n,const T& e)
{
    assert(n >= 0 && n < maxNodos-1); // nodo válido
    assert(nodos[n] != ELTO_NULO);      // nodo del árbol
    assert(2*n+2 < maxNodos);           // hijo drcho. cabe en el árbol
    assert(nodos[2*n+2] == ELTO_NULO); // n no tiene hijo drcho.

    nodos[2*n+2] = e;
}

```

```
template <typename T> inline
void Abin<T>::eliminarHijoIzqdoB(Abin<T>::nodo n)
{
    assert(n >= 0 && n <= maxNodos-1);    // nodo válido
    assert(nodos[n] != ELTO_NULO);         // nodo del árbol
    assert(2*n+1 < maxNodos);              // hijo izqdo. cabe en el árbol
    assert(nodos[2*n+1] != ELTO_NULO);     // n tiene hijo izqdo.
    if (4*n+4 < maxNodos) // caben los hijos del hijo izqdo. de n
        assert(nodos[4*n+3] == ELTO_NULO && // hijo izqdo. de
                nodos[4*n+4] == ELTO_NULO); // n es hoja
    else if (4*n+3 < maxNodos) //sólo cabe h. izq. de h. izq. de n
        assert(nodos[4*n+3] == ELTO_NULO); //hijo izq. de n es hoja

    nodos[2*n+1] = ELTO_NULO;
}
```

```

template <typename T> inline
void Abin<T>::eliminarHijoDrchoB(Abin<T>::nodo n)
{
    assert(n >= 0 && n <= maxNodos-1);    // nodo válido
    assert(nodos[n] != ELTO_NULO);        // nodo del árbol
    assert(2*n+2 < maxNodos);    // hijo drcho. cabe en el árbol
    assert(nodos[2*n+2] != ELTO_NULO);    // n tiene hijo drcho.
    if (4*n+6 < maxNodos)    // caben los hijos del hijo drcho. de n
        assert(nodos[4*n+5] == ELTO_NULO &&    // hijo drcho. de
                nodos[4*n+6] == ELTO_NULO);    // n es hoja
    else if (4*n+5 < maxNodos) //sólo cabe h. izq. de h. drch de n
        assert(nodos[4*n+5] == ELTO_NULO); //hijo drch de n es hoja

    nodos[2*n+2] = ELTO_NULO;
}

```

```
template <typename T>
inline void Abin<T>::eliminarRaizB()
{
    assert(nodos[0] != ELTO_NULO);    // árbol no vacío
    assert(nodos[1] == ELTO_NULO &&
           nodos[2] == ELTO_NULO);    // la raíz es hoja

    nodos[0] = ELTO_NULO;
}
```

```
template <typename T>
inline Abin<T>::~~Abin()
{
    delete[] nodos;
}
```

```
template <typename T>
inline bool Abin<T>::arbolVacioB() const
{
    return (nodos[0] == ELTO_NULO);
}
```

```
template <typename T>
inline const T& Abin<T>::elemento(Abin<T>::nodo n) const
{
    assert(n >= 0 && n <= maxNodos-1);    // nodo válido
    assert(nodos[n] != ELTO_NULO);         // nodo del árbol
    return nodos[n];
}
```

```
template <typename T>
inline T& Abin<T>::elemento(Abin<T>::nodo n)
{
    assert(n >= 0 && n <= maxNodos-1);    // nodo válido
    assert(nodos[n] != ELTO_NULO);         // nodo del árbol
    return nodos[n];
}
```

```
template <typename T>
inline typename Abin<T>::nodo Abin<T>::raizB() const
{
    return (nodos[0] == ELTO_NULO) ? NODO_NULO : 0;
}
```

```

template <typename T> inline
typename Abin<T>::nodo Abin<T>::padreB(Abin<T>::nodo n) const
{
    assert(n >= 0 && n <= maxNodos-1);    // nodo válido
    assert(nodos[n] != ELTO_NULO);         // nodo del árbol

    return (n == 0) ? NODO_NULO : (n-1)/2;
}

```

```

template <typename T> inline
typename Abin<T>::nodo Abin<T>::hijoIzqdoB(Abin<T>::nodo n) const
{
    assert(n >= 0 && n <= maxNodos-1);    // nodo válido
    assert(nodos[n] != ELTO_NULO);         // nodo del árbol

    return (2*n+1 >= maxNodos || nodos[2*n+1] == ELTO_NULO) ?
        NODO_NULO : 2*n+1;
}

```

```

template <typename T> inline
typename Abin<T>::nodo Abin<T>::hijoDrchoB(Abin<T>::nodo n) const
{
    assert(n >= 0 && n <= maxNodos-1); // nodo válido
    assert(nodos[n] != ELTO_NULO);      // nodo del árbol

    return (2*n+2 >= maxNodos || nodos[2*n+2] == ELTO_NULO) ?
        NODO_NULO : 2*n+2;
}

```

```

template <typename T>
Abin<T>::Abin(const Abin<T>& a) :
    nodos(new T[a.maxNodos]),
    maxNodos(a.maxNodos),
    ELTO_NULO(a.ELTO_NULO)
{
    // copiar el vector
    for (nodo n = 0; n <= maxNodos-1; n++)
        nodos[n] = a.nodos[n];
}

```

```

template <typename T>
Abin<T>& Abin<T>::operator =(const Abin<T>& a)
{
    if (this != &a)    // evitar autoasignación
    {
        // Destruir el vector y crear uno nuevo si es necesario
        if (maxNodos != a.maxNodos)
        {
            delete[] nodos;
            maxNodos = a.maxNodos;
            nodos = new T[maxNodos];
        }
        ELTO_NULO = a.ELTO_NULO;
        // Copiar el vector
        for (nodo n = 0; n <= maxNodos-1; n++)
            nodos[n] = a.nodos[n];
    }
    return *this;
}

#endif // ABIN_VEC1_H

```