Arboles Binarios.

Consideraciones.

- Un árbol binario está compuesto por un objeto denominado la raíz. Este objeto puede ser vacío lo que produce un árbol binario vacío. En caso de no ser vacío tendrá dos hijos T_i (subárbol izquierda) y T_d (subárbol derecha) .
- Un nodo dentro de un AB (árbol binario) apunta a una zona de memoria compuesta por una estructura de cuatro campos, que la llamaremos info_nodo. Estos campos son:
  - et: de tipo T será la etiqueta del nodo
  - padre: puntero al nodo padre
  - hizq: puntero al nodo hijo a la izquierda
  - hder: puntero al nodo hijo a la derecha
- Un nodo dentro de un árbol binario actua como un posición dentro del árbol binario.
- Un árbol puede ser recorrido en preorden, postorden e inorden. Dependiendo del recorrido escogido será determinado el siguiente a un nodo (operator++ de los iteradores). Esto nos indica que deberemos definir tres tipos de iteradores asociado a los tres tipos de recorridos.

Paso 1: Cómo es un nodo

Un nodo apunta a una zona de información que contiene una etiqueta, un puntero a la información del nodo padre, a la del hijo a la izquierda y a la del hijo a la derecha. Esta información la hemos denominado info_nodo. Y **nodo** no es más que un posicionamiento (iterador) de esa información.

```
1. template <class T>
2. struct info_nodo{
3.     T et;
4.     info_nodo * padre;
5.    info_nodo * hizq;
6.    info_nodo * hder;
7.    info_nodo(){ padre=hizq=hder=0; }
8.   info_nodo(const T & e){ et = e; padre=hizq=hder=0;}
9. };
10. //funciones auxiliares
11. template <class T>
12. void CopiarInfo(info_nodo<T> * &dest,const info_nodo<T>*&source){
    1. if (source!=0){
        1. dest=new info_nodo<T>(source->et);
        2. CopiarInfo(dest->hizq,source->hizq);
        3. if (dest->hizq!=0){
```

```
            1. dest->hizq->padre= dest;
        4. }
        5. CopiarInfo(dest->hder,source->hder);
        6. if (dest->hder!=0) dest->hder->padre=dest;
    2. }
    3. else dest=0;
    4. }
13. template <class T>
14. void BorrarInfo(info_nodo<T> *&d){
    1. if (d!=0){
        1. BorrarInfo(d->hizq);
        2. BorrarInfo(d->hder);
        3. delete d;
    2.  }
15. template <class T>
16. int numero_nodos(info_nodo<T>*d)const {
    1. if (d==0) return 0;
    2. else return 1+numero_nodos(d->hizq)+numero_nodos(d->hder);
17. }
18. template <class T>
19. bool iguales(info_nodo<T>*s1,info_nodo<T>*s2){
    1. if (s1==0 && s2==0) return true;
    2. else
    3. if (s1==0 || s2==0) return false
    4. if (s1->et!=s2->et) return false
       else return iguales(s1->hizq,s2->hizq) && iguales(s1->hder,s2->hder);
    5. }
20. template <class T>
21. void InsertarHijoIzquierda(info_nodo<T> * n,info_nodo<T> *sub){
    1. if (sub!=0){
        1. BorrarInfo(n->hizq);
        2. sub->padre=n;
        3. n->hizq=sub;
    2. }
    3. }
```

```
22. template <class T>
23. void InsertarHijoIzquierda(info_nodo<T> * n,const T & e){
    1. info_nodo *nuevo=new info_nodo<T>(e);
    2. InsertarHijoIzquierda(n,nuevo);
24. }
25. template <class T>
26. void InsertarHijoDerecha(info_nodo<T>* n,info_nodo<T> *sub){
    1. if (sub!=0){
        1. BorrarInfo(n->hder);
        2. sub->padre=n;
        3. n->hder=sub;
    2. }
    3. }
27. template <class T>
28. void InsertarHijoDerecha(info_nodo<T> * n,const T & e){
    1. info_nodo *nuevo=new info_nodo<T>(e);
    2. InsertarHijoDerecha(n,nuevo);
29. }
30. template <class T>
31. void PodarHijoIzquierda(info_nodo<T> * n){
    1. if (n->hizq!=0){
        1. info_nodo * aux=n->hizq;
        2. n->hizq=0;
        3. BorrarInfo(aux);
    2. }
32. }
33. template <class T>
34. void PodarHijoDerecha(info_nodo<T> * n){
    1. if (n->hder!=0){
        1. info_nodo * aux=n->hder;
        2. n->hder=0;
        3. BorrarInfo(aux);
    2. }
35. }
```

//Aqui podemos añadir funciones recursivas para hacer el listado en preorden,inorden o postorden de un subarbol con raiz n.

Ahora definimos la clase nodo. Actúa como un posicionamiento de info_nodo.

```
template <class T>
class nodo{
    private:
        info_nodo * n;
    public:
        //no duplicamos memoria
        nodo(info_nodo <T>*i){
            n=i;
        }
        nodo(){
            n=0;
        }
        nodo (const nodo <T> &s){
            n = s.n;
        }
        const T & operator*()const{
            return n->et;
        }
        T & operator*(){
            return n->et;
        }
        nodo padre() const{
            nodo father(n->padre);
            return  father;
        }
        nodo hizq() const{
            nodo hijo_izq(n->hizq);
            return hijo_izq;
        }
        nodo hder() const{
            nodo hijo_der(n->hder);
            return hijo_der;
        }
        bool null()const{
            return n==0;
```

19. }
20. bool operator!=(const nodo & s){
    1. return n !=s.n;
21. }
22. bool operator==(const nodo &s){
    1. return n==s.n;
23. }

No hemos implementado el operator ++ ni el operator – porque no sabemos en que orden (inorden, preorden, postorden) vamos a mover el nodo.

Paso 2: Cómo es el arbolbinario

El árbol binario estará representado como un objeto “raiz” de tipo info_nodo.

```
template<class T>
class arbolbinario{
        private:
        info_nodo * raiz;
        public:
          arbolbinario(){ raiz=0;}
          arbolbinario(const T & e){ raiz = new info_nodo(e);}
          arbolbinario(const arbolbinario<T> & ab){
                if (ab.raiz==0) raiz=0;
                else
                        CopiarInfo(raiz,ab.raiz);
-         }
           arbolbinario<T> operator=(const arbolbinario<T> &ab){
                if (this!=&ab){
                        BorrarInfo();
                        CopiarInfo(ab.raiz);
                }
        }
        typename arbolbinario<T>::nodo getraiz(){
                nodo nuevo(raiz);
                return nuevo;
        }
        typename arbolbinario<T>::nodo Insertar_hi(typename arbolbinario<T>::nodo pos, const
```

```
<T>&e){
        InsertarHijoIzquierda(pos.n,e);
       return typename arbolbinario<T>::nodo(pos.n->hi);
 }
typename arbolbinario<T>::nodo Insertar_hi(typename arbolbinario<T>::nodo pos,
arbolbinario    <T>& tree){
        InsertarHijoIzquierda(pos.n,tree.raiz);
       return typename arbolbinario<T>::nodo(pos.n->hi);
 }


typename arbolbinario<T>::nodo Insertar_hd(typename arbolbinario<T>::nodo pos, const
<T>&e){
        InsertarHijoIDerecha(pos.n,e);
       return typename arbolbinario<T>::nodo(pos.n->hi);
 }
typename arbolbinario<T>::nodo Insertar_hd(typename arbolbinario<T>::nodo pos,
arbolbinario    <T>& tree){
        InsertarHijoDerecha(pos.n,tree.raiz);
       return typename arbolbinario<T>::nodo(pos.n->hi);
 }
void Poda_hd(typename arbolbinario<T>::nodo pos){
    PodarHijoDerecha(pos.n);
}
void Poda_hi(typename arbolbinario<T>::nodo pos){
     PodarHijoIzquierda(pos.n);
}
void Sustituye_Subarbol( typename arbolbinario<T>::nodo pos_this,arbolbinario<T>&a,
typename arbolbinario<T>::nodo pos_a){
                if (this!=&a){
                        if (pos_this.n==raiz){
                                BorrarInfo(pos_this.n);
                                CopiarInfo(pos_this.n,pos_a.n);
                                if (raiz!=0) raiz->padre=0;
                        }
                        else {
                                info_nodo * padre = pos_this.n->padre;
                               if (padre->hizq==pos_this.n){ //si es el hijo a la izquierda
```

```
                                        Borrar(padre->hizq);
                                        Copiar(padre->hizq,pos_a->n);
                                        padre->hizq->padre=padre;
                                }
                                else { //es el hijo a la derecha
                                        Borrar(padre->hder);
                                        Copiar(padre->hder,pos_a->n);
                                        padre->hder->padre=padre;
                                }
                        }
                }

        }
        void clear(){
                BorrarInfo(raiz);
                raiz=0;
        }
        int size()const{
                return numero_nodos(raiz);
        }
        bool empty()const{
                return raiz==0;
        }
        bool operator==(const arbolbinario<T>&a){
                return iguales(raiz,a.raiz);
        }
        bool operator!=(const arbolbinario<T>&a){
                return !(*this==a);
        }

        …...

};//en arbolbinario
```

Para que todo funcione debemos introducir en arbolbinario la estructura info_nodo y la clase nodo, quedando así:

```
template<class T>
class arbolbinario{
    private:
        struct info_nodo{
            T et;
            info_nodo * padre;
            info_nodo * hizq;
            info_nodo * hder;
            info_nodo(){ padre=hizq=hder=0; }
          info_nodo(const T & e){ et = e; padre=hizq=hder=0;}
        };
        //Funciones asociadas a info_nodo

        void CopiarInfo(info_nodo<T> * &dest,const info_nodo<T>*&source);
        void BorrarInfo(info_nodo<T> *&d);
        int numero_nodos(info_nodo<T>*d);
        bool iguales(info_nodo<T>*s1,info_nodo<T>*s2);
        void InsertarHijoIzquierda(info_nodo * n,info_nodo *sub);
        void InsertarHijoIzquierda(info_nodo * n,const T & e);
        void InsertarHijoDerecha(info_nodo * n,info_nodo *sub);
        void InsertarHijoDerecha(info_nodo * n,const T & e);
        void PodarHijoIzquierda(info_nodo * n);
        void PodarHijoDerecha(info_nodo * n);
         void RecorridoPreorden(ostream &os,info_nodo *n);
         void RecorridoPostOrden(ostream &os,info_nodo *n);
         void RecorridoInOrden(ostream & os,info_nodo *n);
         void RecorridoNiveles(ostream & os,info_nodo *n);


      info_nodo * raiz;
       public:
          arbolbinario();
          arbolbinario(const T & e);
          arbolbinario(const arbolbinario<T> & ab)
        arbolbinario<T> operator=(const arbolbinario<T> &ab);
        typename arbolbinario<T>::nodo getraiz();
        typename arbolbinario<T>::nodo Insertar_hi(typename arbolbinario<T>::nodo pos, const
        <T>&e);
        typename arbolbinario<T>::nodo Insertar_hi(typename arbolbinario<T>::nodo pos,
       arbolbinario   <T>& tree);
       typename arbolbinario<T>::nodo Insertar_hd(typename arbolbinario<T>::nodo pos, const
      <T>&e);
      typename arbolbinario<T>::nodo Insertar_hd(typename arbolbinario<T>::nodo pos,
      arbolbinario    <T>& tree);
      void Poda_hd(typename arbolbinario<T>::nodo pos)
```

```
void Poda_hi(typename arbolbinario<T>::nodo pos);
void Sustituye_Subarbol( typename arbolbinario<T>::nodo pos_this,arbolbinario<T>&a,
typename arbolbinario<T>::nodo pos_a);
void clear();
int size()const;
bool empty()const;
```

```
class nodo{
    private:
        info_nodo * n;
    public:
            nodo(info_nodo <T>*i);
        nodo();
        nodo (const nodo <T> &s);
        const T & operator*()const
        T & operator*();
        nodo padre() const;
        nodo hizq() const;
        nodo hder() const;
        bool null()const;
        bool operator!=(const nodo & s);
        bool operator==(const nodo &s);

        friend class arbolbinario;
        friend class preorder_iterator;
        friend class inorder_iterator;
        friend class postorder_iterator;
        };
```

};//en arbolbinario

Como podéis observar hemos añadido dentro de la clase *nodo* "***friend class arbolbinario***" para que la clase arbolbinario pueda acceder a la representación de nodo.

Si sois meticulosos os daréis cuenta que *nodo*, a pesar de que hemos dicho que tiene el comportamiento de un iterador dentro de arbolbinario, no tiene el operador de preincremento que me permitiría recorrer el árbol. Esto es así porque para definir el operador++ deberíamos definir si el recorrido lo hacemos en preorden, postorden, o en inorden. Para dar mayor flexibilidad a nuestro arbolbinario vamos a dar la posibilidad de hacer el recorrido de las tres formas. Para ello vamos a crear tres clases más, que son iteradores y que se distinguen por la forma de hacer el ++ correspondiendo al recorrido preorden, inorden o postorden.

PASO 3: RECORRIDOS DEL ARBOL BINARIO

Clase Preorden_Iterador para realizar Recorrido en preorden

```
class preorder_iterator
 {
  private:

     info_nodo *p;
     preorder_iterator( info_nodo * data):p(data){}
__public:
    preorder_iterator():p(0){};

    // para posibilitar la siguiente operación preorden_iterator tiene que ser amiga de nodo
    preorder_iterator( typename arbolbinario<T>::nodo & pos):p(pos.p){}
    bool operator!=(const preorder_iterator & i) const{
          return (p!=i.p);
    }
    bool operator==(const preorder_iterator & i) const{
          return (p==i.p);
    T& operator*(){
          return (p->et);
    }
    preorder_iterator & operator++( ){
          if (p==0)
             return *this;
          if (p->hizq!=0) //no hemos listado  aún el hijo a la izquierda
             p = p->hizq;
          else if (p->hder!=0) //no hemos listado el hijo a la derecha
              p = p->hder;
           else {
            //vamos subiendo mientras sea  el hijo a la derecha
             while (( p->padre!=0) &&
                (p->padre->hder == p ||
                 p->padre->hder==0))
                 p=p->padre;

             if (p->padre==0) //si es la raiz ya hemos terminado
               p = 0;
             else
              p = p->padre->hder;
          }
          return *this;
     }
      friend class arbolbinario;

};
```

Ahora en la clase arbolbinario debemos incorporar las funciones correspondientes begin y end que las llamaremos para distinguirla de los otros recorridos beginpreorder y endpreorder . Estas funciones deben implementarse dentro de la clase arbolbinario en la parte pública :

```
..
typename arbolbinario<T>::preorden_iterator  beginpreorder(){
        typename arbolbinario<T>::preorden_iterator it(raiz);
        return it;
}

typename arbolbinario<T>::preorden_iterator  endpreorder(){
        typename arbolbinario<T>::preorden_iterator it;
        return it;
}
…...
```

Clase Inorden_Iterador para realizar Recorrido en inorden

```
class inorder_iterator
 {
  private:
    info_nodo * p;
    inorder_iterator( info_nodo* n):p(n);
  public:
    inorder_iterator():p(0){}
    inorder_iterator( typename arbolbinario<T>::nodo & pos):p(pos.n);
    bool operator!=(const inorder_iterator & i) const{  return p!=i.p;}
    bool operator==(const inorder_iterator & i) const{return p==i.p;}
    T& operator*(){ return p->et;}
    inorder_iterator & operator++( ){
           if (p==0)
              return *this;
           if (p->hder!=0) {
               p = p->hder;
              while (p->hizq!=0)
                 p = p->hizq;
             }
             else {
                 while (p->padre!=0 &&
                    p->padre->hder == p)
                  p = p->padre;
                 // Si (padre de p es nulo), hemos terminado En caso contrario,
                  //el siguiente a p es el padre
                 p  = p->padre;

          }
      return *this;
     }
    friend class arbolbinario;
};
```

Y en arbolbinario las funciones begin y end asociadas

```
..
typename arbolbinario<T>::inorden_iterator  begininorder(){
        typename arbolbinario<T>::inorden_iterator it(raiz);
        return it;
}

typename arbolbinario<T>::inorden_iterator  endinorder(){
        typename arbolbinario<T>::inorden_iterator it;
        return it;
}
…...
```

Clase Postorder_Iterador para realizar Recorrido en Postorden

```
class postorder_iterator
 {
  private:
    info_nodo * p;
    postorder_iterator( info_nodo* n):p(n);
  public:
    postorder_iterator():p(0){}
    postorder_iterator( typename arbolbinario<T>::nodo & pos):p(pos.n);
    bool operator!=(const postorder_iterator & i) const{  return p!=i.p;}
    bool operator==(const postorder_iterator & i) const{return p==i.p;}
    T& operator*(){ return p->et;}
    postorder_iterator & operator++( ){
          if (p==0)
             return *this;
          if (p->padre==0) {
                        p=0;
             else{
               if (p->padre->hizq==p){
                     If (p->padre->hder!=0){
                                p=p->padre->hder;
                                do{
                                        while (p->hizq!=0) p=p->hizq;
                                        If (p->hder!=0) p=p->hder;
                                }while(p->hizq!=0 || p->hder!=0);
                     }
                     else p=p->padre;
              }
               else p=p->padre;

      return *this;
     }
    friend class arbolbinario;
};
```

Y en arbolbinario las funciones begin y end asociadas

```
..
typename arbolbinario<T>:postorder_iterator  begininpostorder(){
        typename arbolbinario<T>::inorden_iterator it(raiz);
        return it;
}

typename arbolbinario<T>::inorden_iterator  endinorder(){
        typename arbolbinario<T>::inorden_iterator it;
        return it;
}
…..
```