```
#define MAX 1000
//Modelo de nodo Arbol binario Busqueda.
struct nodo_arbol_binario
{
    int dato;
    struct nodo_arbol_binario* iz;
    struct nodo_arbol_binario* de;
    struct nodo_arbol_binario* padre;
};
typedef struct nodo_arbol_binario NABinario;
//Modelo de nodo Pila estatica
struct pila_estatica
{
    NABinario* entrada[MAX];
    int tamanio;
    int tope;
    int entri;
};
typedef struct pila_estatica PilaE;

NABinario* crear_nodo(int n,NABinario* Padre);
void insertar_ABB(NABinario*&arbol,int n,NABinario* Padre);
void mostrar_arbol(NABinario* arbol, int cont);
bool Busqueda_ABB(NABinario* arbol,int n);
NABinario*minimo(NABinario*arbol);
void destruir(NABinario*ndo);
void eliminar_nodo(NABinario*ndo_eliminar);
void reemplazar(NABinario* arbol,NABinario*nodo_2);
//Barridos Arboles Recursivos
void abinario_preorden_recursivo(NABinario *arbol);
void abinario_inorden_recursivo(NABinario *arbol);
void abinario_postorden_recursivo(NABinario *arbol);
void abinario_mostrar_recursivo (NABinario* arbol, int tabulado);
void abinario_mostrar_recursivo2 (NABinario* arbol, int n);
int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
//Crea un nodo para un arbol binario de busqueda
NABinario* crear_nodo(int n, NABinario *Padre){
    NABinario* nuevo_nodo=new NABinario();
    nuevo_nodo->dato=n;
    nuevo_nodo->de=NULL;
    nuevo_nodo->iz=NULL;
    nuevo_nodo->padre=Padre;
    return nuevo_nodo;
}
//Algoritmo para insertar nodos a un arbol binario de busqueda
void insertar_ABB(NABinario*&arbol,int n,NABinario* Padre){
    if(arbol==NULL)//si el arbol esta vacio
    {
        NABinario* nuevo_nodo=crear_nodo(n,Padre);
        arbol=nuevo_nodo;
    }
    else{//si el arbol tiene uno o mas nohttps://www.google.com/dos
        int valor_raiz=arbol->dato;//obtenemos el valor de la raiz
        //si el elemento es menor , se inserta a la izquierda
        if(n<valor_raiz)
        {   insertar_ABB(arbol->iz,n,arbol);}
```

```
        else{//si el elemento es mayor, se inserta a la derecha
            insertar_ABB(arbol->de,n,arbol);
        }
    }
}
//Mostrar Arbol Girado a la izquierda
    void mostrar_arbol(NABinario* arbol,int cont){
        if(arbol==NULL)return;
        else{
            mostrar_arbol(arbol->de,cont+1);
            for(int i=0;i<cont;i++)
            {
                cout<<"    ";//3 espacios para separar nodos
            }
            cout<<arbol->dato<<" "<<endl;
            mostrar_arbol(arbol->iz,cont+1);
        }
    }
//Buscar Nodo en un arbol
    bool Busqueda_ABB(NABinario* arbol,int n){
        //si el arbol esta vacio no hago nada
        if(arbol==NULL)return false;
        //si encuentro el dato , retorno true que indica que lo encontro en la
lista
        else if(arbol->dato==n)return true;
        //si no lo encontre , me voy bien por la izquierda
        else if(n<arbol->dato)return Busqueda_ABB(arbol->iz,n);
        //si no lo encontre, me voy bien por la derecha
        else return Busqueda_ABB(arbol->de,n);
    }

 //Algoritmos para hacer eliminacion de nodos en un arbol
//funcion determinar nodo mas izquierdo posible
NABinario*minimo(NABinario*arbol){
  if(arbol==NULL)return NULL;//arbol vacio
  if(arbol->iz)return minimo(arbol->iz);
  else return arbol;
 }
//Algoritmo para reemplazar un nodo por otro
void reemplazar(NABinario* arbol,NABinario*nodo_2){
   if(arbol->padre)
  {
   //di tiene padre procedemos a asignarle el nuevo hijo
   if(arbol->dato==arbol->padre->de->dato)arbol->padre->iz=nodo_2;
   else if (arbol->dato==arbol->padre->de->dato)arbol->padre->de=nodo_2;
  }
 //al hijo le decimos quien es su padre
 if(arbol)nodo_2->padre=arbol->padre;
}
//funcion destruir nodo
void destruir(NABinario*ndo){
   ndo->iz=NULL;
   ndo->de=NULL;
   delete ndo;
}
//Algoritmo para eliminar un nodo 3 casos vistos
void eliminar_nodo(NABinario*ndo_eliminar){
        if(ndo_eliminar->iz && ndo_eliminar->de)
        {int dato;
            NABinario*menor=minimo(ndo_eliminar->de);
            ndo_eliminar->dato=menor->dato;
```

```
            eliminar_nodo(menor);
        }
        else if(ndo_eliminar->iz){
            reemplazar(ndo_eliminar,ndo_eliminar->iz);
            destruir(ndo_eliminar);
        }
        else if(ndo_eliminar->de){
            reemplazar(ndo_eliminar,ndo_eliminar->de);
            destruir(ndo_eliminar);
        }
        else {
            reemplazar(ndo_eliminar,NULL);
            destruir(ndo_eliminar);
        }
}
//Algoritmo para eliminar un nodo dvoid abinario_inorden_recursivo(NABinario
*arbol)el arbol y acomodandolo
void eliminar_ABB(NABinario*arbol,int n){
    if(arbol==NULL)return;
    else if (n<arbol->dato)eliminar_ABB(arbol->iz,n);
    else if (n>arbol->dato)eliminar_ABB(arbol->de,n);
    else eliminar_nodo(arbol);
}
//Barridos de forma recursiva.
void abinario_preorden_recursivo(NABinario *arbol){
    if(arbol!=NULL){
        cout<<arbol->dato<<"-->";
        abinario_preorden_recursivo(arbol->iz);
        abinario_preorden_recursivo(arbol->de);
    }
}
//de este barrido salen ordenados
void abinario_inorden_recursivo(NABinario *arbol){
    if(arbol!=NULL){
        abinario_preorden_recursivo(arbol->iz);
        cout<<arbol->dato<<"-->";
        abinario_preorden_recursivo(arbol->de);
    }
}
void abinario_postorden_recursivo(NABinario *arbol){
    if(arbol!=NULL){
        abinario_preorden_recursivo(arbol->iz);
        abinario_preorden_recursivo(arbol->de);
        cout<<arbol->dato<<"-->";
    }
}
//Mostrvoid abinario_mostrar_recursivo2 (NABinario* arbol, int n)ar de forma
recursiva
void abinario_mostrar_recursivo (NABinario* arbol, int tabulado)
{
    if (arbol != NULL)
    {
        cout << string (tabulado, '\t');

        cout << "Nodo: " << arbol->dato << " | " << "Iz-> ";
        if (arbol->iz != NULL)
            cout << arbol->iz->dato;
        else
            cout << "NULL";
        cout << "  " << "De-> ";
        if (arbol->de != NULL)
```

```
            cout << arbol->de->dato;
        else
            cout << "NULL";
        cout << endl;

        tabulado++;
        abinario_mostrar_recursivo (arbol->iz, tabulado);
        abinario_mostrar_recursivo (arbol->de, tabulado);
    }
}

void abinario_mostrar_recursivo2 (NABinario* arbol, int n)
{
    if (arbol == NULL)
        return;

    abinario_mostrar_recursivo2 (arbol->de, n+1);
    cout << string (n, '\t') << arbol->dato << endl;
    abinario_mostrar_recursivo2(arbol->iz, n+1);
}
//Alta recursiva..
void abinariob_alta_recursivo (NABinario* &arbol, int nuevo_dato)
{
    if (arbol == NULL)
    {
        arbol = new (NABinario);
        arbol->iz = NULL; arbol->de = NULL;
        arbol->dato = nuevo_dato;
    }
    else if (nuevo_dato < arbol->dato)
        abinariob_alta_recursivo (arbol->iz, nuevo_dato);
    else if (nuevo_dato > arbol->dato)
        abinariob_alta_recursivo (arbol->de, nuevo_dato);
}
//implementacion de pila estatica para algoritmos iterativos
bool pila_vacia (PilaE *pila)
{
 bool Pvac=false;
 if(pila == NULL){
     Pvac=true;
 }
    return Pvac;
}

void pila_agregar (PilaE* &pila,int n)
{
    if(pila->tope<pila->tamanio){
        pila->entrada[pila->tope]->dato=n;
        pila->tope++;
    }else{
        cout<<"Error al agregar en pila verifique algoritmo"<<endl;
    }
}

int pila_sacar (PilaE* &pila)
{  if(pila->tope>0){
        pila->tope--;
    }
    else{
        cout<<"Error al sacar elementos de una pila"<<endl;
```

```
    }
    return(pila->entrada[pila->tope]->dato);
}
//Barridos iterativos
void abinario_preorden_iterativo (NABinario* arbol)
{
    NABinario* aux;
    PilaE* pila; pila->tamanio = MAX; pila->tope=0;

    if (arbol != NULL)
        pila_agregar (pila, arbol->dato);

    while (!pila_vacia (pila))
    {
        aux->dato = pila_sacar (pila);
        cout << aux->dato << "  ";

        if (aux->de != NULL)
            pila_agregar (pila, aux->de->dato);
        if (aux->iz != NULL)
            pila_agregar (pila, aux->iz->dato);
    }
}
//pila estatica pero para InOrden SACAR
NABinario* pila_sacarent(PilaE* &pila,int &entri) {
    if (pila->tope > 0)
    {
        pila->tope--;
        entri = pila->entrada[pila->tope]->dato;
        return pila->entrada[pila->tope];
    }
    else return NULL;
}
//Pila agregar
void pila_agregarent(PilaE* &pila,int &entri,NABinario* arbol)
{
    if(pila->tope<pila->tamanio){
        pila->entrada[pila->tope]->dato=arbol->dato;
        pila->tope++;
        pila->entri=entri;
    }else{
        cout<<"Error al agregar en pila verifique algoritmo"<<endl;
    }
}

//Recorrido arbol IN-Orden
void abinario_inorden_iterativo (NABinario* arbol)
{
    NABinario* aux;
    PilaE* pila; pila->tamanio = MAX; pila->tope = 0;
    int ent;
    int n=1;
    pila_agregarent(pila,n, arbol);
    while (!pila_vacia (pila))
    {
         aux = pila_sacarent(pila,ent);
         if (ent == 1 ) {
            pila_agregarent(pila,n,aux);
         if (aux->iz != NULL) pila_agregarent(pila,n, aux->iz);
           } else {
```

```
                    cout << aux->dato << " ";
                    if (aux->de != NULL)
                        pila_agregarent(pila,n, aux->de);
                      }
            }
}
//Recorrido Post-Orden
void abinario_postorden_iterativo (NABinario* arbol)
{
    NABinario* aux;
    PilaE* pila; pila->tamanio = MAX; pila->tope = 0;
    int ent;
    int n=1;
    pila_agregarent(pila,n, arbol);
    while (!pila_vacia (pila))
    {
        aux = pila_sacarent(pila,ent);
        switch(ent)
        {
            case 1:
                    pila_agregarent(pila,n,aux);
                    if (aux->iz != NULL)
                    pila_agregarent(pila,n, aux->iz);
                    break;
            case 2:
                    pila_agregarent(pila,n,aux);
                    if (aux->de != NULL)
                    pila_agregarent(pila,n, aux->de);
                    break;
            case 3:
                    cout << aux->dato << " ";
                    break;
          }
    }
}
//Mostrar todos los nodos Hojas guia nº3 ejercicio 4
void abinario_mostrarhoja_recursivo(NABinario *arbol){
    if(arbol==NULL){ cout<<"arbol vacio"<<endl;}
        if((arbol->iz)&&(arbol->de)==NULL){
            cout<<arbol->dato<<"-->";
        }
        abinario_preorden_recursivo(arbol->iz);
        abinario_preorden_recursivo(arbol->de);
    }

void eliminar_hoja(NABinario* &arbol,NABinario* padre){

    if(arbol==NULL)return;
    padre=arbol;
    if(arbol->iz!=NULL)eliminar_hoja(arbol->iz,padre);
    else if(arbol->de!=NULL)eliminar_hoja(arbol->de,padre);
    else if((arbol->de==NULL)&&(arbol->iz==NULL)){
        if(arbol->dato==padre->iz->dato)padre->iz=NULL;
        else padre->de=NULL;
        delete arbol;
    }
}
int abinario_altura_recursivo(NABinario *arbol){
    int alt_iz,alt_de;
    if(arbol==NULL)return -1;
    else
```

```
            alt_iz=abinario_altura_recursivo(arbol->iz);
            alt_de=abinario_altura_recursivo(arbol->de);
            if(alt_iz<alt_de)return alt_iz +1;
            else return alt_de+1;
}
```