PONTIFICIA UNIVERSIDAD JAVERIANA

TALLER 03: TALLER ARBOLES

JOHN CORREDOR

ESTRUCTURAS DE DATOS

NICOLAS ALGARRA

DANIEL GONZALEZ

JULIANA PACHECO

BOGOTA, COLOMBIA

2024

TALLER ARBOLES

I. ARBOL

TADS

TAD Nodo

- CrearNodo(dato): Crea un nodo con el dato proporcionado.
- ObtenerDato(nodo): Retorna el dato almacenado en el nodo.
- FijarDato(nodo, dato): Asigna un nuevo dato al nodo.
- AgregarHijo(nodo, hijo): Agrega un nodo hijo al nodo actual.
- EliminarHijo(nodo, hijo): Elimina un nodo hijo del nodo actual.
- ObtenerHijos(nodo): Retorna una lista de los nodos hijos.

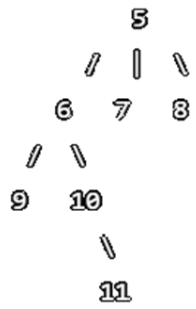
TAD Arbol

- CrearArbol(raiz): Crea un árbol con un nodo raíz proporcionado.
- EsVacio(arbol): Retorna true si el árbol está vacío, false de lo contrario.
- ObtenerRaiz(arbol): Retorna el nodo raíz del árbol.
- FijarRaiz(arbol, raiz): Asigna un nuevo nodo raíz al árbol.
- InsertarNodo(arbol, padre, dato): Inserta un nodo con el dato proporcionado como hijo del nodo padre.
- EliminarNodo(arbol, dato): Elimina un nodo con el dato especificado.
- BuscarNodo(arbol, dato): Busca un nodo con el dato especificado y lo retorna.
- PreOrden(arbol): Realiza un recorrido en preorden del árbol.
- InOrden(arbol): Realiza un recorrido en inorden del árbol.
- PosOrden(arbol): Realiza un recorrido en postorden del árbol.
- NivelOrden(arbol): Realiza un recorrido por niveles del árbol.

COMPILACION Y PRUEBAS

Prueba:

El arbol utilizado es el siguiente:



Salida esperada:

Preorder: 5 6 9 10 7 11 8

Salida optenida:

```
PS C:\Users\julia\Documents\Universidad\Estructuras de datos\Taller03\Taller03_Trees\Taller03_Trees\Arbol>
g++ prueba_arbol.cpp -o arbol
PS C:\Users\julia\Documents\Universidad\Estructuras de datos\Taller03\Taller03_Trees\Taller03_Trees\Arbol>
./arbol
        5
        6
        9
        10
        7
        11
        8
```

II. ARBOL AVL

TADS

1. TAD NodoBinarioAVL

Propiedades:

- dato: Valor almacenado en el nodo.
- hijoIzq: Puntero al hijo izquierdo.
- hijoDer: Puntero al hijo derecho.

Operaciones:

- NodoBinarioAVL(): Constructor por defecto que inicializa el nodo.
- ~NodoBinarioAVL(): Destructor del nodo.
- T& getDato(): Devuelve el dato almacenado en el nodo.
- void setDato(T&): Establece el dato almacenado en el nodo.
- NodoBinarioAVL<T>* getHijoIzq(): Devuelve el puntero al hijo izquierdo.

- NodoBinarioAVL<T>* getHijoDer(): Devuelve el puntero al hijo derecho.

- void setHijoIzq(NodoBinarioAVL<T>*): Establece el puntero al hijo izquierdo.

- void setHijoDer(NodoBinarioAVL<T>*): Establece el puntero al hijo derecho.

2. TAD ArbolBinarioAVL

Propiedades:

- raiz: Puntero al nodo raíz del árbol.

Operaciones:

- ArbolBinarioAVL(): Constructor por defecto que inicializa el árbol.

- ~ArbolBinarioAVL(): Destructor del árbol.

- void setRaiz(NodoBinarioAVL<T>*): Establece la raíz del árbol.

- NodoBinarioAVL<T>* getRaiz(): Devuelve la raíz del árbol.

- bool esVacio(): Verifica si el árbol está vacío.

- T& datoRaiz(): Devuelve el dato almacenado en la raíz.

- int altura(NodoBinarioAVL<T>*): Calcula la altura del árbol.

- int tamano(NodoBinarioAVL<T>*): Calcula el número de nodos en el árbol.

- bool insertar(T&): Inserta un valor en el árbol.

- bool eliminar(T&): Elimina un valor del árbol.

- bool buscar(T&): Busca un valor en el árbol.

- NodoBinarioAVL<T>* giroDerecha(NodoBinarioAVL<T>*&): Realiza una rotación simple a la derecha.

- NodoBinarioAVL<T>* giroIzquierda(NodoBinarioAVL<T>*&): Realiza una rotación simple a la izquierda.

- NodoBinarioAVL<T>* giroIzquierdaDerecha(NodoBinarioAVL<T>*&): Realiza una rotación doble izquierda-derecha.

- NodoBinarioAVL<T>* giroDerechaIzquierda(NodoBinarioAVL<T>*&): Realiza una rotación doble derecha-izquierda.

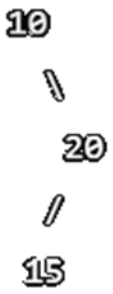
- void preOrden(NodoBinarioAVL<T>*): Recorre el árbol en preorden.

- void inOrden(NodoBinarioAVL<T>*): Recorre el árbol en inorden.
- void posOrden(NodoBinarioAVL<T>*): Recorre el árbol en posorden.
- void nivelOrden(NodoBinarioAVL<T>*): Recorre el árbol por niveles.

COMPILACION Y PRUEBAS

Prueba:

Arbol utilizado para la prueba:



En la terminar se pone:

A 10

A 5

A 20

A 15

E 5

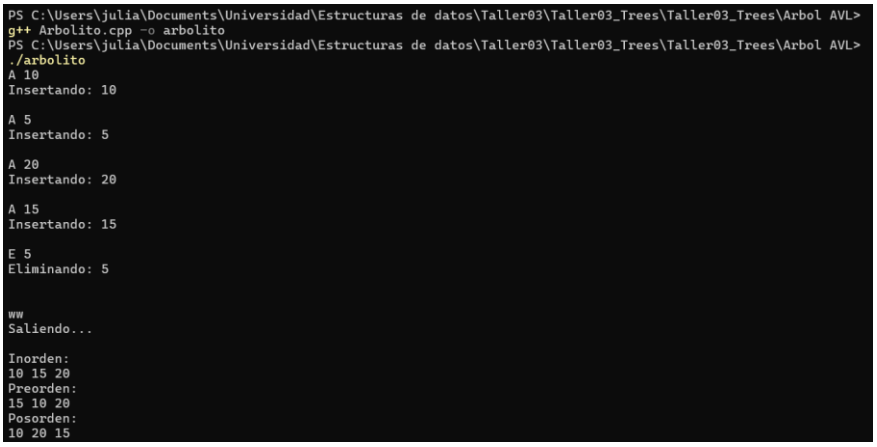
Salida esperada:

Inorden: 10 15 20

Preorden: 10 15 20

Posorden: 15 20 10

Salida optenida:



III. ARBOL BINARIO

TADS

1) Nodo:

   Datos mínimos:

   - Dato; entero; contiene el dato a guardar en el nodo
   - NodoIzquierda; apuntador de nodo; contiene la dirección de memoria de su nodo hijo a la izquierda
   - NodoDerecha; apuntador de nodo; contiene la dirección de memoria de su nodo hijo a la derecha

2) ArbolBinario

   Datos mínimos:

   - Raiz; apuntador de nodo; contiene la dirección de memoria del nodo raíz del árbol

   Operaciones:

   - ArbolBinario(numero), constructor que asigna un valor a la raíz dado por parámetro

   - busquedaInOrder(), imprime todos los elementos del árbol usando la búsqueda InOrder

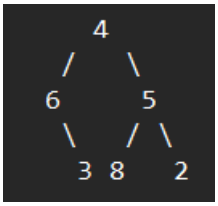
   - busquedaPostOrder(), imprime todos los elementos del árbol usando la búsqueda PostOrder

   - getRaiz(), devuelve la dirección de memoria de la raíz

   - busquedaPreOrder(), imprime todos los elementos del árbol usando la búsqueda PreOrder

COMPILACION Y PRUEBAS

Se construirá el siguiente árbol con los comandos mostrados después del árbol:




```
ArbolBinario arbol(4);

arbol.getRaiz()->nodoDerecha = new nodo(5);
arbol.getRaiz()->nodoIzquierda = new nodo(6);
arbol.getRaiz()->nodoDerecha->nodoIzquierda = new nodo(8);
arbol.getRaiz()->nodoDerecha->nodoDerecha = new nodo(2);
arbol.getRaiz()->nodoIzquierda->nodoDerecha = new nodo(3);
```

Para la verificación una creación efectiva y funcionamiento de los comandos se realizará un recorrido InOrder, PreOrder y PostOrder donde se esperan los siguientes resultados:

InOrder- 6 3 4 8 5 2

PreOrder – 4 6 3 5 8 2

PostOrder – 3 6 8 2 5 4

Resultados:

```
InOrder
 6 3 4 8 5 2
PreOrder
 4 6 3 5 8 2
PostOrder
 3 6 8 2 5 4
```

Los resultados son correctos y de acuerdo con el plan de pruebas

Para verificar si es posible modificar un nodo se va a realizar el debido comando y luego se generará un recorrido inorder, se va a modificar el nodo donde está el 5 por un 10. Donde se esperan los siguientes resultados:

InOrder - 6 3 4 8 10 2

Resultados:

```
InOrder al modificar un nodo
 6 3 4 8 10 2
```

Los resultados son correctos y de acuerdo con el plan de pruebas

Para verificar si es posible eliminar un nodo se va a realizar el debido comando y luego se generará un recorrido inorder, se va a eliminar el nodo con dato 2. Donde se esperan los siguientes resultados:

InOrder - 6 3 4 8 10

Resultados:

```
InOrder al eliminar un nodo
 6 3 4 8 10
```

Los resultados son correctos y de acuerdo con el plan de pruebas

IV. ARBOL BINARIO ORDENADO

TADS

1) Nodo:

   Datos mínimos:

   - Dato; entero; contiene el dato a guardar en el nodo
   - NodoIzquierda; apuntador de nodo; contiene la dirección de memoria de su nodo hijo a la izquierda
   - NodoDerecha; apuntador de nodo; contiene la dirección de memoria de su nodo hijo a la derecha

2) BST:
   Datos mínimos:
   - Raiz; apuntador de nodo; contiene la dirección de memoria del nodo raiz del arbol

   Operaciones:

   - BST(numero), constructor que asigna un valor a la raíz dado por parámetro

   - busquedaInOrder(), imprime todos los elementos del árbol usando la búsqueda InOrder

   - busquedaPostOrder(), imprime todos los elementos del árbol usando la búsqueda PostOrder

   - busquedaPreOrder(), imprime todos los elementos del árbol usando la búsqueda PreOrder

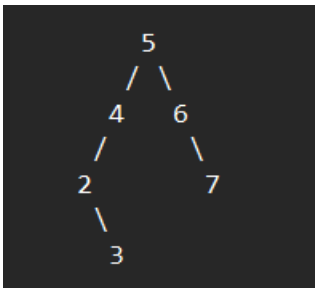
   - buscarNumero(numero), busca en el árbol si el numero dado por parámetro está en el arbol

   - agregarNumero(numero), agrega al árbol un nuevo nodo con el numero dado por parametro

- eliminarNumero(numero), elimina del árbol el nodo con el numero dado por parametro

COMPILACION Y PRUEBAS

Se construirá el siguiente árbol con los comandos del programa:



Para confirmar su creación efectiva se usará el comando buscarNumero() con los números 3, 7 y 1 donde se espera los siguientes resultados:

3 – True

7 – True

1 – False

Resultados:

```
Esta el 3?: 1
Esta el 7?: 1
Esta el 1?: 0
```

Los resultados son correctos y de acuerdo con el plan de pruebas

También se va a hacer uso de los diferentes recorridos de un árbol binario para ver su buena implementacion, para esto se va a hacer un recorrido InOrder, PreOrder y PostOrder donde se esperan los siguientes resultados:

InOrder – 2 3 4 5 6 7

PreOrder – 5 4 2 3 6 7

PostOrder – 3 2 4 7 6 5

Resultados:

```
InOrder
 2 3 4 5 6 7
PreOrder
 5 4 2 3 6 7
PostOrder
 3 2 4 7 6 5
```

Los resultados son correctos y de acuerdo con el plan de pruebas

Y últimamente se borrará un valor del árbol y se confirmará mostrando el recorrido inorder del nuevo árbol de búsqueda binario, el valor a borrar sera 3, y se espera el siguiente resultado:

2 4 5 6 7

Resultados:

```
InOrder despues eliminar el 3
 2 4 5 6 7
```

Los resultados son correctos y de acuerdo con el plan de pruebas

V. ARBOL EXPRESION

TADS

1. TAD: Nodo expresion

Atributos:

- dato (char): Almacena el valor del nodo, que puede ser un operador o un operando.
- operando (bool): Indica si el nodo es un operando (true) o un operador (false).
- hijoIzq (NodoExpresion*): Puntero al hijo izquierdo del nodo.
- hijoDer (NodoExpresion*): Puntero al hijo derecho del nodo.

Operaciones:

- NodoExpresion(): Constructor que inicializa un nodo sin hijos y sin datos.
- ~NodoExpresion(): Destructor del nodo.
- char getDato(): Retorna el dato almacenado en el nodo.
- void setDato(char val): Establece el valor del nodo.
- bool getOperando(): Retorna si el nodo es un operando.
- void setOperando(bool op): Establece si el nodo es un operando.
- NodoExpresion* getHijoIzq(): Retorna el puntero al hijo izquierdo.
- void setHijoIzq(NodoExpresion* izq): Establece el puntero al hijo izquierdo.
- NodoExpresion* getHijoDer(): Retorna el puntero al hijo derecho.
- void setHijoDer(NodoExpresion* der): Establece el puntero al hijo derecho.

2. TAD: Arbol expresion

Atributos:

- raiz (NodoExpresion*): Puntero al nodo raíz del árbol de expresión.

Operaciones:

- ArbolExpresion(): Constructor que inicializa un árbol vacío con la raíz en NULL.
- ~ArbolExpresion(): Destructor del árbol de expresión.
- NodoExpresion* getRaiz(): Retorna el puntero a la raíz del árbol.
- void setRaiz(NodoExpresion* nod): Establece el puntero a la raíz del árbol.
- void llenarDesdePrefija(string &expresion): Llena el árbol a partir de una expresión prefija.
- void llenarDesdePosfija(string &expresion): Llena el árbol a partir de una expresión posfija.
- void obtenerPrefija(NodoExpresion* inicio): Imprime la expresión en notación prefija.
- void obtenerInfija(NodoExpresion* inicio): Imprime la expresión en notación infija.
- void obtenerPosfija(NodoExpresion* inicio): Imprime la expresión en notación posfija.
- int evaluar(NodoExpresion* nodi): Evalúa la expresión representada por el árbol y devuelve su resultado.
- bool siOperando(char car): Retorna true si el carácter es un operador; de lo contrario, false.

COMPILACION Y PRUEBAS

Prueba:
Ejercicio 1

- Expresión Prefija: -*/5-7+113-+2+1*43*2-68
- Versión Posfija Impresa: 8 6 - 2 * 3 4 * 1 + 2 + - 3 1 1 + 7 - 5 / * -
- Resultado Evaluado: -8

Evaluación Ejercicio 1

1. Expresión Prefija: -*/5-7+113-+2+1*43*2-68
    a. Esta es una expresión prefija, y el orden de evaluación sería el siguiente:
        - (Raíz)
        * (Hijo izquierdo de la raíz)
        / (Hijo izquierdo de *)
        5 (Hijo izquierdo de /)

- (Hijo derecho de /)
7 (Hijo izquierdo de -)
+ (Hijo derecho de -)
.....

Al evaluar esta expresión, deberíamos obtener el resultado -8.

Ejercicio 2

- Expresión Posfija: 45+23+*6+87+/12+3*6+23+/*
- Versión Prefija Impresa: * / + * + 4 5 + 2 3 6 + 8 7 / + * + 1 2 3 6 + 2 3
- Resultado Evaluado: 9

Evaluación Ejercicio 2

1. Expresión Posfija: 45+23+*6+87+/12+3*6+23+/*
    a. Al evaluar manualmente la expresión, el orden sería:

        45+ → 9
        23+ → 5
        9 * 5 → 45
        6 + → 51
        .....

    b. Al evaluar esta expresión, deberíamos obtener el resultado -9.

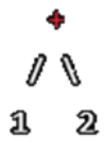
```
PS C:\Users\julia\Documents\Universidad\Estructuras de datos\Taller03\Taller03_Trees\Taller03_Trees\Arbol Expresion>
g++ main.cpp -o Expresion
PS C:\Users\julia\Documents\Universidad\Estructuras de datos\Taller03\Taller03_Trees\Taller03_Trees\Arbol Expresion>
./Expresion
EJERCICIO 1

1. Construir Arbol Expresion:
-*/5-7+113-+2+1*43*2-68
2. Imprimir Version Posfija=
8 6 - 2 * 3 4 * 1 + 2 + - 3 1 1 + 7 - 5 / * -
3. Imprimir Resultado=
-8
EJERCICIO 2

1. Construir Arbol Expresion:
45+23+*6+87+/12+3*6+23+/*
2. Imprimir Version Prefija=
* / + * + 4 5 + 2 3 6 + 8 7 / + * + 1 2 3 6 + 2 3
3. Imprimir Resultado=
9
```

Se realizo una segunda prueba con un árbol más pequeño para verificar la efectividad del código:

Se utilizo el siguiente arbol:



Modificamos el main:

```
cout << "EJERCICIO 1" << endl;
cout << endl;
cout << "1. Construir Arbol Expresion (Prefija): " << endl;
cout << "+12" << endl;  // Expresión prefija simple
ArbolExpresion* arbexp = new ArbolExpresion();
string exp = "+12";
arbexp->llenarDesdePrefija(exp);
```

```
cout << "EJERCICIO 2" << endl;
cout << endl;
cout << "1. Construir Arbol Expresion (Posfija): " << endl;
cout << "12+" << endl;  // Expresión posfija simple
ArbolExpresion* arbexp2 = new ArbolExpresion();
string exp2 = "12+";
arbexp2->llenarDesdePosfija(exp2);
```

Ejercicio 1: Expresión Prefija +12

+ es el operador.

1 y 2 son los operandos.

La expresión representa 1 + 2.

- Recorrido Posfijo (Posorden) del Árbol:

  Se recorre el hijo izquierdo, luego el hijo derecho, y finalmente la raíz.

- Resultado del recorrido: 1 2 +.

  Evaluación del Árbol:

  1 + 2 = 3.

- Resultados Esperados para el Ejercicio 1:

  Versión Posfija Impresa: 1 2 +

  Resultado Evaluado: 3

Ejercicio 2: Expresión Posfija 12+

1 y 2 son los operandos.

+ es el operador que suma los dos operandos.

La expresión representa 1 + 2.

- Recorrido Prefijo (Preorden) del Árbol:

  Se visita la raíz, luego el hijo izquierdo y finalmente el hijo derecho.

- Resultado del recorrido: + 1 2.

  1 + 2 = 3.

- Resultados Esperados para el Ejercicio 2:

  Versión Prefija Impresa: + 1 2

  Resultado Evaluado: 3

Resultados obtenidos:

```
PS C:\Users\julia\Documents\Universidad\Estructuras de datos\Taller03\Taller03_Trees\Taller03_Trees\Arbol Expresion>
g++ main.cpp -o main
PS C:\Users\julia\Documents\Universidad\Estructuras de datos\Taller03\Taller03_Trees\Taller03_Trees\Arbol Expresion>
./main
EJERCICIO 1

1. Construir Arbol Expresion (Prefija):
+12
2. Imprimir Version Posfija =
2 1 +
3. Imprimir Resultado =
3

EJERCICIO 2

1. Construir Arbol Expresion (Posfija):
12+
2. Imprimir Version Prefija =
+ 1 2
3. Imprimir Resultado =
3
```

VI. ARBOL KD-TREE

TADS

TAD: Node

Atributos:

- puntos, Un vector de coordenadas (puntos) en el espacio multidimensional.
- izquierda, Puntero al nodo hijo izquierdo.
- derecha, Puntero al nodo hijo derecho.

Operaciones:

- Node(), Crea un nodo con un vector de puntos.
- vector<double> getPuntos() const: Devuelve las coordenadas del nodo.
- void setPuntos(), Modifica las coordenadas del nodo.
- Node* getIzquierda() Devuelve el nodo hijo izquierdo.
- Node* getDerecha() Devuelve el nodo hijo derecho.

TAD: KDTree

Atributos:

- raiz, Puntero al nodo raíz del árbol.
- dimension, El número de dimensiones de los puntos almacenados en el árbol.

Operaciones:

- KDTree(), Inicializa el KDTree con una dimensión dada.
- KDTree(), Libera la memoria de todos los nodos al eliminar el árbol.
- void AgregarNodoAlArbol(), Agrega un nuevo nodo al árbol.
- Node* agregarNodo(), Inserta un nuevo nodo recursivamente dependiendo de la dimensión actual.
- bool buscarNodo(), Busca un nodo con coordenadas específicas.
- bool buscar(), Busca un nodo con los puntos especificados recursivamente.
- string preOrder(), Recorre el árbol en preorden.
- string inOrder(),Recorre el árbol en inorden.

- string postOrder(), Recorre el árbol en postorden.

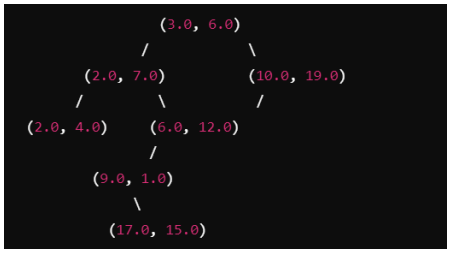
- void borrarNodos(), Elimina todos los nodos del árbol de forma recursiva.

- int getdimension(), Devuelve la dimensión del árbol.

- void setdimension(), Modifica la dimensión del árbol.

-Node* getraiz(), Devuelve la raíz del árbol. PRUEBA:

-Árbol utilizado:



Lo que se espera en cada caso:

```
Preorden: (10.0, 19.0) (3.0, 6.0) (2.0, 7.0) (9.0, 1.0) (6.0, 12.0) (17.0, 15.0) (2.0, 4.0)
Inorden: (9.0, 1.0) (2.0, 7.0) (3.0, 6.0) (6.0, 12.0) (10.0, 19.0) (2.0, 4.0) (17.0, 15.0)
Postorden: (9.0, 1.0) (2.0, 7.0) (6.0, 12.0) (3.0, 6.0) (2.0, 4.0) (17.0, 15.0) (10.0, 19.0)
```

Prueba:

```
~/repaso-2$ cd "KD-tree/KD-tree"
~/.../KD-tree/KD-tree$ g++ *.cpp -o kdtree
~/.../KD-tree/KD-tree$ ./kdtree
Hola
Buscando el punto 10, 19: Encontrado
Buscando el punto 5, 5: No encontrado
Recorrido en Preorden: (3.000000,6.000000)(2.000000,7.000000)(2.000000,4.000000)(17.000000,15.000000)(6.000000,12.000000)(9.000000,1.000000)(10.000000,19.000000)
Recorrido en Inorden: (2.000000,4.000000)(2.000000,7.000000)(3.000000,6.000000)(6.000000,12.000000)(9.000000,1.000000)(17.000000,15.000000)(10.000000,19.000000)
Recorrido en Postorden: (2.000000,4.000000)(2.000000,7.000000)(9.000000,1.000000)(6.000000,12.000000)(10.000000,19.000000)(17.000000,15.000000)(3.000000,6.000000)
~/.../KD-tree/KD-tree$
```

VII. ARBOL QUAD-TREE

TADS

1) Coordenada:

   Datos mínimos:

   - X, entero, posición en x
   - Y, entero, posición en y

   Operaciones:

   - Coordenada(_x,_y), constructor que asigna los valores dados como parametro a la coordenada

- Coordenada(), constructor que asigna la coordenada a 0,0 (default)

2) Nodo:

   Datos mínimos:

   - Pos, coordenada, contiene la posicion del nodo
   - Dato, entero, contiene el valor que guardara el nodo

   Operaciones:

   - Nodo(_pos, _nodo), Constructor que construye un nodo en base a los valores pasados por parametro
   - Nodo(), Constructor que crea un nodo vacío, sin posición y dato igual a 0

3) Quadtree:

   Datos mínimos:

   - **LimiteNO**, coordenada, contiene la coordenada más arriba a la izquierda del cuadrante (coordenada limite en el noroeste)
   - **LimiteSE**, coordenada, contiene la coordenada más abajo a la derecha del cuadrante (coordenada limite en el sureste)
   - **Node**, puntero de nodo, contiene el nodo que se encuentra en el cuadrante
   - **ArbolNO**, puntero de quadtree, contiene la dirección del cuadrante en el noroeste
   - **ArbolNE**, puntero de quadtree, contiene la dirección del cuadrante en el noreste
   - **ArbolSO**, puntero de quadtree, contiene la dirección del cuadrante en el suroeste
   - **ArbolSE**, puntero de quadtree, contiene la dirección del cuadrante en el sureste

   Operaciones:

   - **Quadtree(NO, SE),** Constructor que asigna los límites del cuadrante en base a los datos dados por parámetro
   - **Quadtree(),** Constructor predeterminado que no asigna un límite al cuadrante
   - **Insertar(nodo),** inserta un nodo dado por parámetro en el árbol
   - **Buscar(coordenada),** busca el nodo que esta está en la coordenada dada por parámetro
   - **Contiene(coordenada),** retorna si la coordenada se encuentra dentro de los límites del cuadrante

COMPILACION Y PRUEBAS

Para comprobar el funcionamiento del arbol se va a crear un arbol quadtree con un mapa 8x8 donde se encontrarán 3 puntos con datos, siendo los siguientes:

- Nodo a: con coordenadas 1,1 y el dato 1
- Nodo b: con coordenadas 2,5 y el dato 2
- Nodo c: con coordenadas 7,6 y el dato 3

Como primera forma de prueba se buscará en el árbol el dato que se encuentra en las 3 coordenadas de cada nodo y el de la coordenada 5,5 que no está en el árbol, se esperan los siguientes resultados:

- Nodo a: 1
- Nodo b: 2
- Nodo c: 3
- Nodo no existente: NULL

Resultados:

```
Nodo a: 1
Nodo b: 2
Nodo c: 3
Nodo no existente: 0
```

Los resultados son correctos y de acuerdo con el plan de pruebas

También se pondrá a prueba la posibilidad de modificar un dato de un nodo ya existente, para ello si usara el comando para modificar el nodo a y que su dato sea 5. Se espera los siguientes resultados:

- Nodo a: 5

Resultados:

```
Nodo a despues de modificacion: 5
```

Los resultados son correctos y de acuerdo con el plan de pruebas

VIII. ARBOL DECISION

TADS

1) TAD: Nodo

Atributos:

pregunta: Cadena que representa la pregunta o condición en el nodo.
si: Puntero al subárbol si la respuesta es "sí".
no: Puntero al subárbol si la respuesta es "no".
Operaciones:

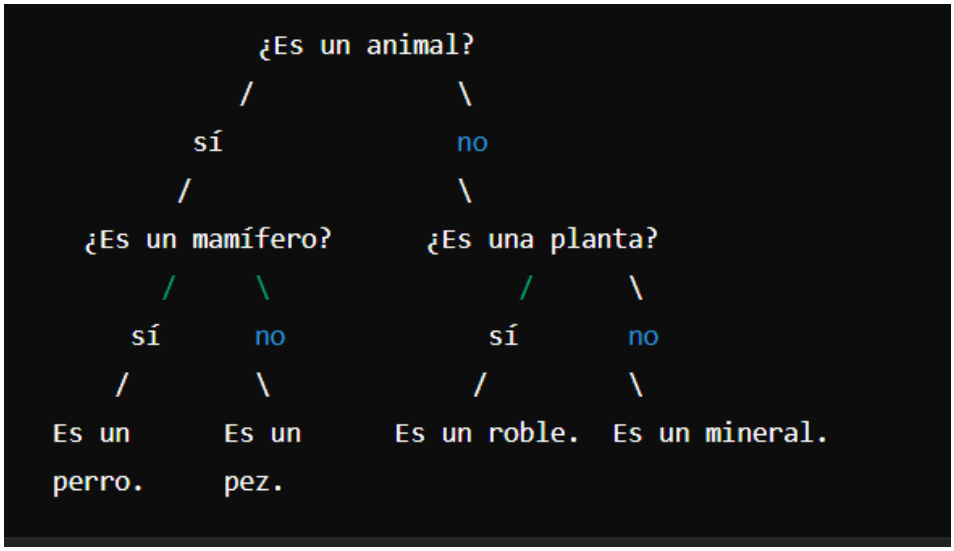
Nodo(): Crea un nodo con una pregunta y punteros nulos para si y no.
getPregunta() const: Devuelve la pregunta del nodo.
setPregunta(string nuevaPregunta): Modifica la pregunta del nodo.
getSi() const: Devuelve el puntero al subárbol "sí".
getNo() const: Devuelve el puntero al subárbol "no".

2) TAD: Árbol de Decisión
Atributos:

raiz: Puntero al nodo raíz del árbol de decisión.
Operaciones:

ArbolDecision(): Crea un árbol de decisión vacío.
liberarArbol(Nodo* nodo): Libera la memoria del árbol recursivamente.
establecerRaiz(Nodo* nodo): Establece el nodo raíz del árbol.
iniciarDecision(): Inicia el proceso de toma de decisiones en el árbol.
crearNodo(string pregunta): Crea un nuevo nodo con la pregunta especificada.

Arbol utilizado.




```
C:\Users\danie\Documents\UNIVERSIDAD\4 SEMESTRE\ESTRUCT-OF-DATES\CODIGOS\SEGUNDO_CORTE\Laboratorio_Arboles\ARBOL_DECISIO
N>g++ main.cpp -o main

C:\Users\danie\Documents\UNIVERSIDAD\4 SEMESTRE\ESTRUCT-OF-DATES\CODIGOS\SEGUNDO_CORTE\Laboratorio_Arboles\ARBOL_DECISIO
N>main.exe
¬Es un animal? (si/no): si
¬Es un mamÝfero? (si/no): no
Decisi¾n final: Es un pez.
```

IX. ARBOL RED BLACK

TADS

1) TAD Node

Atributos,

- data, Valor del nodo (tipo int).
- color, Color del nodo (tipo Color, donde Color puede ser RED o BLACK).
- left, Puntero al nodo hijo izquierdo (tipo Node*).
- right, Puntero al nodo hijo derecho (tipo Node*).
- parent, Puntero al nodo padre (tipo Node*).

Operaciones

- Node(), Crea un nodo con el valor dado y establece el color a RED.
- int getData(), Devuelve el valor del nodo.
- Color getColor(), Devuelve el color del nodo.
- void setColor(Color nuevoColor), Establece el color del nodo.
- Node* getLeft(), Devuelve el puntero al nodo hijo izquierdo.
- Node* getRight(), Devuelve el puntero al nodo hijo derecho.
- Node* getParent(), Devuelve el puntero al nodo padre.
-

2) TAD RedBlackTree

Atributos,

- root, Puntero al nodo raíz del árbol (tipo Node*).

Operaciones:

- RedBlackTree(), Crea un árbol Red-Black vacío (raíz en NULL).
- void insert(int data), Inserta un nuevo valor en el árbol Red-Black, asegurando las propiedades del árbol.
- void inorder(), Inicia el recorrido en orden del árbol.
- void inorderHelper(Node* node), Ayuda a realizar el recorrido en orden desde un nodo dado.
- void rotateLeft(Node*& root, Node*& pt), Realiza una rotación a la izquierda en el nodo dado.

- void rotateRight(Node*& root, Node*& pt), Realiza una rotación a la derecha en el nodo dado.
- void fixViolation(Node*& root, Node*& pt), Arregla las violaciones de las propiedades del árbol después de una inserción.

Árbol utilizado:

Prueba:

```
~/.../ARBOLES/Arbol_RedBlack$ g++ redblack.cpp -o redblack
~/.../ARBOLES/Arbol_RedBlack$ ./redblack
Recorrido en orden del árbol Red-Black:
10 20 25 30 40 50 ~/.../ARBOLES/Arbol_RedBlack$
```