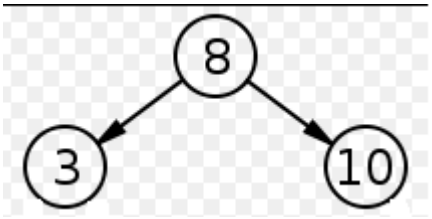
Arboles de Búsqueda y Barridos

Definición / Idea Básica

Una Árbol de búsqueda también llamado BST(por acrónimo binary search tree) es un tipo de árbol binario donde se cumple para todos sus subárboles que los menores van a la izquierda de la raíz y los mayores van a la derecha.

Representación:



Dicho de otra manera es un árbol binario con una política especial para las altas y bajas. (Los menores de acuerdo a la clave van a la izquierda y los mayores van a la derecha).

Definición de Nodos

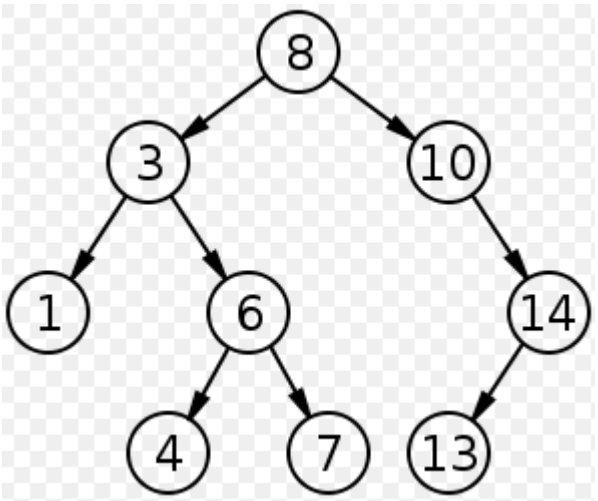
struct NodoArbol

información del nodo // dato	
linkIzquierdo	linkDerechao

En C++

```
struct nodo_arbol_binario
{
      struct informacion del nodo….; // dato
      struct nodoArboll * iz;
      struct nodoArboll * de;
};
typedef struct nodo_arbol_binario NABinario;
```

Ejemplo



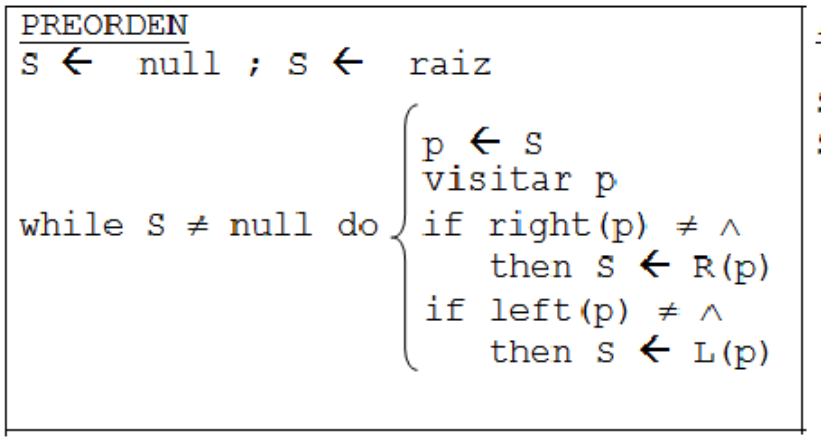
Barrido PreOrden

Se caracteriza por las iniciales RID (por su orden de mostrado), **R**aiz , hijo **I**zquierdo, hijo **D**erecho.
Para el ejemplo anterior: 8,3,1,6,4,7,10,14,13

preOrden Recursivo

```
1       void abinario_postorden_recursivo (NABinario* arbol)  {
2           if (arbol == NULL) return;
3           cout << "-Tratamiento nodo " << arbol->dato << endl;
4           abinario_postorden_recursivo (arbol->iz);
5           abinario_postorden_recursivo (arbol->de);
6       }
```

Pseudo código genérico



Pseudocódigo Base C++

Para implementar este algoritmo se utiliza la estructura auxiliar pila_estatica, definida en clases.

PreOrden ITERATIVO

```
void abinario_preorden_iterativo (NABinario* arbol)
{
        NABinario* aux;
        PilaE pila; pila.tamanio = MAX; pila.tope = 0;

        pila_agregar(pila, arbol);

        while (!pila_vacia (pila))     {
                aux = pila_sacar (pila);

                cout << aux->dato << "  ";

                if (aux->iz != NULL)          pila_agregar (pila, aux->iz);

                if (aux->de != NULL)          pila_agregar (pila, aux->de);
}
```

Barrido inOrden

También llamado Simétrico, se caracteriza por las iniciales IRD (por su orden de mostrado), hijo **I**zquierdo, **R**aiz, hijo **D**erecho.
Tiene la característica que si se aplica a un árbol de búsqueda o lexicográfico, el resultado del recorrido es la información ordenada por su clave de búsqueda.
Para el ejemplo anterior: 1, 3, 4, 6, 7, 8, 10, 13, 14.

inOrden Recursivo

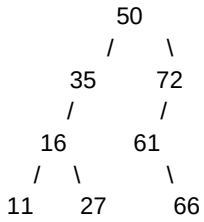
```
1       void recorrido_inorden(NABinario* &arbol,int llamada) {
2               if (arbol == NULL)  return;
3               if (arbol->iz != NULL)          recorrido_inorden(arbol->iz,llamada);  //<<Ver Nota>>
4               cout << "-Tratamiento nodo " << arbol->dato << endl;
5               if (arbol->de != NULL)          recorrido_inorden(arbol->de,llamada);  //<<Ver Nota>>
6       }
```

Nota: Los if de control no son estrictamente necesarios, fueron agregados para realizar un seguimiento mas claro de la dinámica que se realiza en el stack de ejecución.

Para analizar su comportamiento expandimos el código anterior (fuente color rosado), agregando instrucciones que muestren la secuencia de invocación de las distintas recursiones de una ejecución.

```
void recorrido_inorden(NABinario* &arbol,int llamada) {

    if (llamada == 0 ) {
            cout << "#llamada-Operacion                          DcionNodo valorNodo" << endl;
            cout << "-------------------------------------------------------------------------" << endl;
    }
    llamada++;
   for(int i=0;i<llamada-1;i++) cout << "       ";
   cout << llamada << "-Inicio          ";
   for(int i=0;i<5-llamada;i++) cout << "       ";
   cout <<  arbol << "       "  << (arbol?arbol->dato:0)  <<  endl;

   if (arbol == NULL) return;

    for(int i=0;i<llamada-1;i++) cout << "       ";
    cout << llamada << "-Invocar izq          " << endl;
    if (arbol->iz != NULL)   recorrido_inorden(arbol->iz,llamada);

    for(int i=0;i<llamada-1;i++) cout << "       ";
   cout << llamada << "-Tratamiento nodo " << arbol->dato << endl;

   for(int i=0;i<llamada-1;i++) cout << "       ";
   cout << llamada << "-Invocar der          " << endl;
    if (arbol->de != NULL)          recorrido_inorden(arbol->de,llamada);

    for(int i=0;i<llamada-1;i++) cout << "       ";
    cout << llamada << "-Fin             ";
    for(int i=0;i<5-llamada;i++) cout << "       ";
    cout  <<  arbol  << "       "    << (arbol?arbol->dato:-1)  << endl;
}
```

Ejecutamos la función de barrido expandida con el siguiente árbol en memoria:



El resultado que se obtiene es:

```
#llamada-Operacion                                          DcionNodo valorNodo
-------------------------------------------------------------------------------
1-Inicio                                                    0x955c60         50
1-Invocar izq
         2-Inicio                                           0x955c80         35
         2-Invocar izq
                  3-Inicio                                  0x9563f0         16
                  3-Invocar izq
                           4-Inicio                         0x956430         11
                           4-Invocar izq
                           4-Tratamiento nodo 11
                           4-Invocar der
                           4-Fin                            0x956430         11
                  3-Tratamiento nodo 16
                  3-Invocar der
                           4-Inicio                         0x956410         27
                           4-Invocar izq
                           4-Tratamiento nodo 27
                           4-Invocar der
                           4-Fin                            0x956410         27
                  3-Fin                                     0x9563f0         16
         2-Tratamiento nodo 35
         2-Invocar der
         2-Fin                                              0x955c80         35
1-Tratamiento nodo 50
1-Invocar der
         2-Inicio                                           0x955ca0         72
         2-Invocar izq
                  3-Inicio                                  0x9563b0         61
                  3-Invocar izq
                  3-Tratamiento nodo 61
                  3-Invocar der
                           4-Inicio                         0x9563d0         66
                           4-Invocar izq
                           4-Tratamiento nodo 66
                           4-Invocar der
                           4-Fin                            0x9563d0         66
                  3-Fin                                     0x9563b0         61
         2-Tratamiento nodo 72
         2-Invocar der
         2-Fin                                              0x955ca0         72
1-Fin                                                       0x955c60         50
```

inOrden Iterativo

Pseudo codigo genérico

```
SIMETRICO
S ← null ; S ← (raiz,1)
while S≠∧ do (p,i) ← S
              if i = 1
                then S ← (p,2)
                  if L(p) ≠ ∧
                    then S←(L(p),1)
                  endif
                else visitar p
                  if R(p) ≠ ∧
                    then S← (R(p),1)
                  endif
              endif

Nota: S es un STACK
```

Pseudocódigo Base C++

Para implementar este algoritmo se utiliza la estructura auxiliar pila_estatica, usada en preorden. Se agregó un elemento de información a esta, de tipo entero llamado entrada. Este elemento se usará en los algoritmos iterativos inOrden y PostOrden para administrar las veces que el algoritmo trata cada nodo a fin de cumplir con el recorrido deseado.

```
struct pila_estatica
{
        NABinario* dato[MAX];
        int     entrada[MAX];
        int tamanio;
        int tope;
};
typedef struct pila_estatica PilaE;
```

Por otro lado se modificaron las funciones pila_agregar y pila_sacar para que incluyan estos datos.
Un aspecto que merece tratarse es que en pila_sacar se paso el parametro entrada por referencia para recuperar su valor.

```
NABinario* pila_sacarent(PilaE &pila,int &entri)  {
        if (pila.tope > 0)
        {
                pila.tope--;
                entri   = pila.entrada[pila.tope];
                return pila.dato[pila.tope];
        }
        else
                return NULL;
}
```

Algoritmo

```
void abinario_inorden_iterativo (NABinario* arbol)
{
        NABinario* aux;
        PilaE pila; pila.tamanio = MAX; pila.tope = 0;
          int ent;

        pila_agregarent(pila,1, arbol);

        while (!pila_vacia (pila))
        {
                aux = pila_sacarent(pila,ent);

                if (ent == 1 ) {
                        pila_agregarent(pila,2,aux);
                        if (aux->iz != NULL)        pila_agregarent(pila,1, aux->iz);
                } else {
                        cout << aux->dato << "  ";
                        if (aux->de != NULL)        pila_agregarent(pila,1, aux->de);
                }
        }
}
```

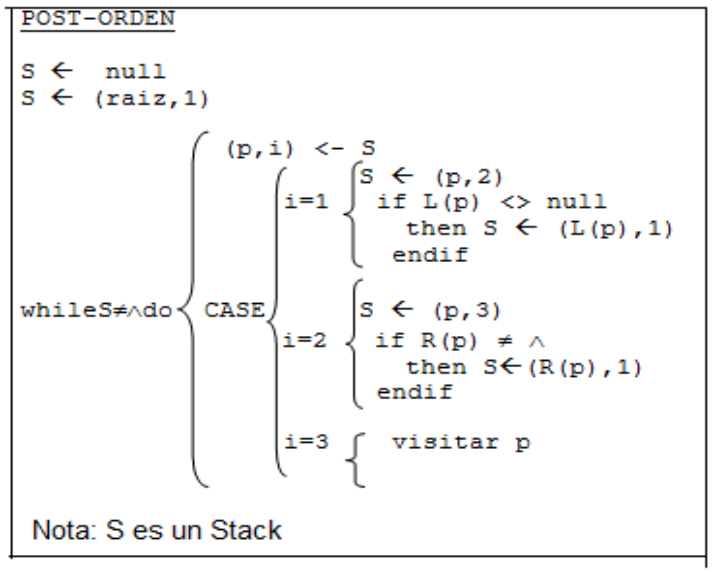
Barrido PostOrden

Se caracteriza por las iniciales IDR (por su orden de mostrado), hijo **I**zquierdo, hijo **D**erecho, **R**aiz..
Para el ejemplo anterior: 1, 4, 7, 6, 3, 13, 14. 10

postOrden Recursivo

```
1       void abinario_postorden_recursivo (NABinario* arbol)  {
2           if (arbol == NULL) return;
3           if (arbol->iz)      abinario_postorden_recursivo (arbol->iz);
4           if (arbol->de)    abinario_postorden_recursivo (arbol->de);
5           cout << "-Tratamiento nodo " << arbol->dato << endl;
6       }
```

Pseudo código genérico



Pseudocódigo Base C++

Para implementar este algoritmo se utiliza la estructura auxiliar pila_estatica, usada en preorden. Se agregó un elemento de información a esta, de tipo entero llamado entrada. Este elemento se usará en los algoritmos iterativos inOrden y PostOrden para administrar las veces que el algoritmo trata cada nodo a fin de cumplir con el recorrido deseado.

PostOrden ITERATIVO

```
void abinario_postorden_iterativo (NABinario* arbol)
{
        NABinario* aux;
        PilaE pila; pila.tamanio = MAX; pila.tope = 0;
         int ent;

        pila_agregarent(pila,1, arbol);

        while (!pila_vacia (pila))     {
                aux = pila_sacarent(pila,ent);

                switch(ent)                     {
                case 1:
                        pila_agregarent(pila,2,aux);
                        if (aux->iz != NULL)          pila_agregarent(pila,1, aux->iz);
                        break;
                case 2:
                        pila_agregarent(pila,3,aux);
                        if (aux->de != NULL)          pila_agregarent(pila,1, aux->de);
                        break;
                case 3:
                        cout << aux->dato << "  ";
                        break;
        }
        }
}
```