Asignatura: Algoritmia y Estructura de Datos
Profesor: D. Sc. Gerardo García Gil
2021-B
Alumno: José Rafael Ruiz Gudiño
Ingeniería en Desarrollo de Software
Centro de Enseñanza Técnica Industrial (CETI)

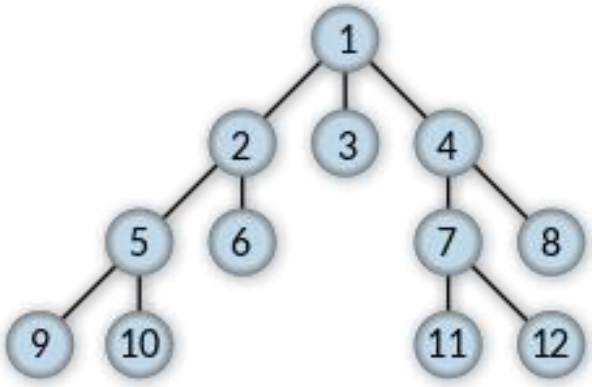
-Presentación

Los árboles son una de las estructuras de datos no lineales, empleadas en informática, tanto para resolver problemas de hardware como de software. Los árboles de directorios son organizaciones bastante empleadas por cualquier usuario o programador de una computadora. De igual manera cumplen un buen papel en la toma de decisiones, valido como árbol de decisiones.

Un algoritmo de búsqueda es un conjunto de instrucciones que están diseñadas para localizar un elemento con ciertas propiedades dentro de una estructura de datos; por ejemplo, ubicar el registro correspondiente a cierta persona en una base de datos, o el mejor movimiento en una partida de ajedrez.

-Introducción

En Ciencias de la Computación, Búsqueda en anchura (en inglés BFS - Breadth First Search) es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo (usado frecuentemente sobre árboles). Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.

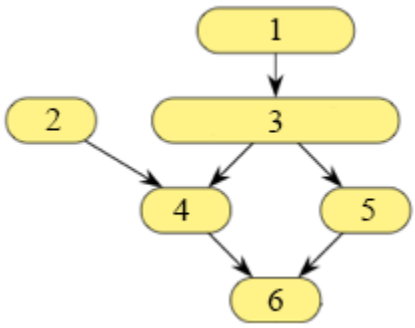


Formalmente, BFS es un algoritmo de búsqueda sin información, que expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución. El algoritmo no usa ninguna estrategia heurística.

Si las aristas tienen pesos negativos aplicaremos el algoritmo de Bellman-Ford en alguna de sus dos versiones.

-Desarrollo

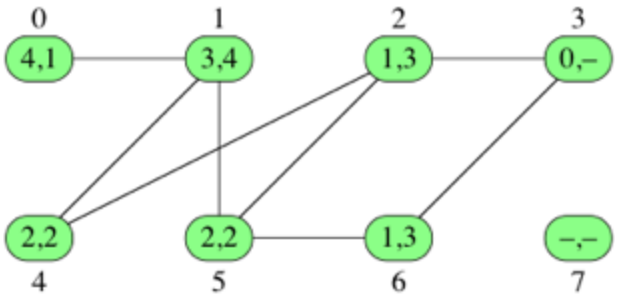
La búsqueda en anchura le asigna dos valores a cada vértice vvv:



- Una distancia, que da el número mínimo de aristas en cualquier camino del vértice de origen al vértice *v*.
- El vértice predecesor de *v* a lo largo de algún camino más corto del vértice de origen. El predecesor del vértice de origen es algún valor especial, como null, que indica que no tiene predecesor.

Si no hay un camino del vértice de origen al vértice *v*, entonces la distancia de *v* es infinita y su predecesor tiene el mismo valor especial que el predecesor del vértice de origen.

Por ejemplo, aquí hay un grafo no dirigido con ocho vértices, numerados del 0 al 7, con los números de los vértices mostrados arriba o abajo de los vértices. Dentro de cada vértice hay dos números: su distancia del origen, que es el vértice 3, seguido de su predecesor en el camino más corto desde el vértice 3. Un guion indica null:



En BFS, inicialmente hacemos que la distancia y el predecesor de cada vértice sea el valor especial (null). Empezamos la búsqueda en el origen y le asignamos una distancia de 0. Después visitamos a todos los vecinos del origen y a cada vecino le damos una distancia de 1 y hacemos su predecesor igual al origen. Después visitamos a todos los vecinos de los vértices cuya distancia es 1 *y* que no han sido visitados

anteriormente, y le damos a cada uno de estos vértices una distancia de 2 y hacemos que su predecesor sea el vértice a partir del cual lo visitamos. Seguimos avanzando hasta que todos los vértices alcanzables desde el vértice de origen han sido visitados, siempre visitando los vértices a una distancia *k* del origen antes de visitar cualquier vértice a una distancia *k*+1.

Complejidad computacional

La complejidad computacional del algoritmo se puede expresar como O(|V|+|E|, donde |V| es el número de vértices y |E| es el número de aristas. El razonamiento es porque en el peor caso, cada vértice y cada arista será visitado por el algoritmo.

Complejidad en memoria

Cuando se sabe anteriormente el número de vértices en el grafo, y se pueden usar otras estructuras de data para determinar qué vértices han sido añadidos a la cola, la complejidad en memoria es O(|V|), donde |V| es el número de vértices. Éste es en adición al espacio requerido para el grafo mismo, que depende en la representación del grafo.

Factor de ramificación

Especialmente con grafos muy grandes (posiblemente infinitos), puede ser más práctico describir la complejidad del algoritmo en términos del factor de ramificación. Para encontrar los nodos que están en una distancia d de la raíz (distancia medida por el número de aristas que tiene que usar), Búsqueda en anchura requiere O(b^(d+1)) en términos computacionales y de memoria, donde b es el factor de ramificación del grafo.

-Implementación

Algoritmo:

Básicamente, hay dos funciones los cuales usan recursividad en este método. Uno es imprimir todos los nodos en un nivel dado (nivelActual), y otro es imprimir el orden de nivel transversal del árbol (nivelOrden). nivelOrden hace uso de nivelActual para imprimir nodos en todos los niveles uno por uno comenzando desde la raíz.

Complejidad de tiempo: O(n ^ 2) en el peor de los casos. Para un árbol sesgado, nivelActual () toma O (n) tiempo donde n es el número de nodos en el árbol sesgado. Entonces, la complejidad de tiempo de printLevelOrder () es O(n) + O(n-1) + O(n-2) + .. + O(1) que es O (n ^ 2).

Complejidad espacial: O(n) en el peor de los casos. Para un árbol sesgado, nivelActual () usa el espacio O(n) para la pila de llamadas. Para un árbol equilibrado, la pila de llamadas utiliza el espacio O(log n), (es decir, la altura del árbol equilibrado).

Código en C

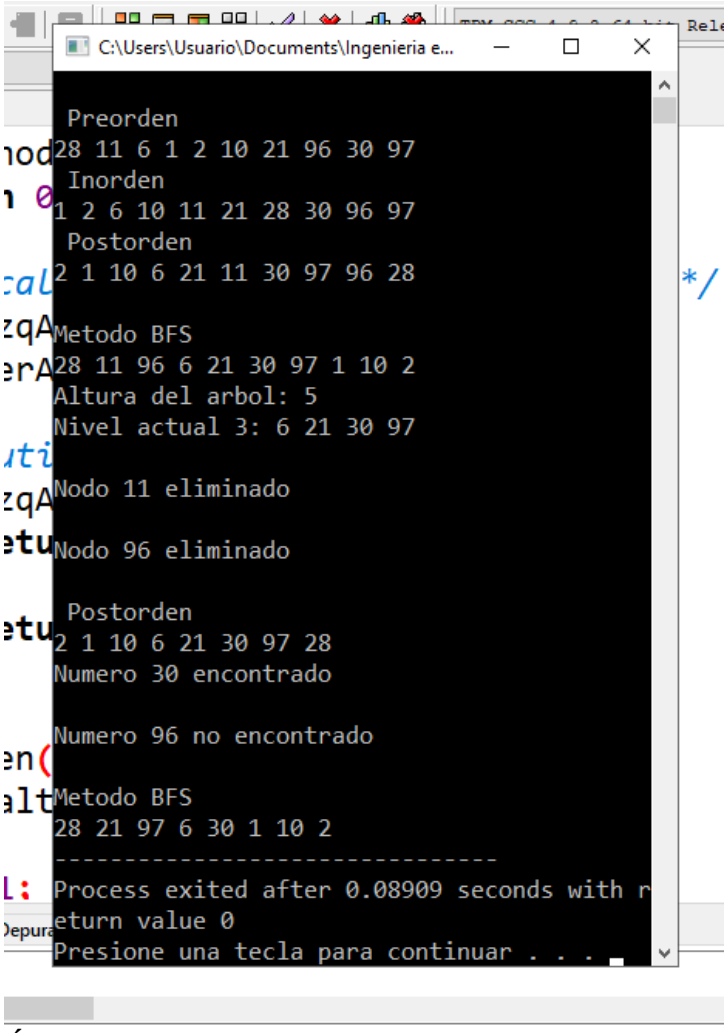
```
#include <stdio.h>
#include <stdlib.h>
struct Nodo{
        int dato;
        struct Nodo *izq;
        struct Nodo *der;
};
//////////////////////creacion de un nodo/////////////////
//     size_t  es un tipo de dato sin signo , que es nmayor a 0

struct Nodo *nuevonodo(int dato);
void insertar(struct Nodo *nodo,int dato);
void preorden(struct Nodo *nodo);
void inorden(struct Nodo *nodo);
void postorden(struct Nodo *nodo);
void Buscar(struct Nodo* a, int dat);
void Borrar(struct Nodo* arbol, int dat);
int Vacio(struct Nodo* r);
int EsHoja(struct Nodo* r);
//Funciones para el Metodo BFS
void nivelActual(struct Nodo* raiz, int nivel);
int altura(struct Nodo* nodo);
void nivelOrden(struct Nodo* raiz);

int main(int argc, char *argv[]) {

        struct Nodo *raiz=nuevonodo(28);
        insertar(raiz, 11);
        insertar(raiz, 96);
        insertar(raiz, 21);
        insertar(raiz, 6);
        insertar(raiz, 97);
        insertar(raiz, 1);
        insertar(raiz, 30);
         insertar(raiz, 10);
         insertar(raiz, 2);
   printf("\n Preorden\n");
   preorden(raiz);
   printf("\n Inorden\n");
   inorden(raiz);
   printf("\n Postorden\n");
   postorden(raiz);
   //Uso del metodo de busqueda en anchura
   printf("\nMetodo BFS\n");
   nivelOrden(raiz);
   printf("\nAltura del arbol: %d\n", altura(raiz));
   printf("Nivel actual 3: ");
   nivelActual(raiz,3);
```

```
   Borrar(raiz,11);
   Borrar(raiz,96);
   printf("\n Postorden\n");
   postorden(raiz);
   Buscar(raiz,30);
   Buscar(raiz,96);
   printf("\nMetodo BFS\n");
   nivelOrden(raiz);
        return 0;
}
struct Nodo *nuevonodo(int dato){
          //se pide el espacio en memoria para el nuevo nodo
        size_t tamaNodo=sizeof(struct Nodo);
        struct Nodo *nodo=(struct Nodo *)malloc(tamaNodo);
                // Asignar el dato al iniciar
        nodo->dato=dato;
        nodo->izq=nodo->der=NULL;
        return nodo;

}
void insertar(struct Nodo *nodo,int dato){

        //si es mayor va a la derecha
        if(dato>nodo->dato){
                if(nodo->der==NULL){
                        nodo->der=nuevonodo(dato);
                }
                else{
                        insertar(nodo->der,dato);
                }
        }       else{
                        // sino a la izquiera
                        if(nodo->izq==NULL){

                                nodo->izq=nuevonodo(dato);
                        }else{
                                insertar(nodo->izq,dato);
                        }}}
void preorden(struct Nodo *nodo){
        if(nodo!=NULL){
                printf("%d ",nodo->dato);
                preorden(nodo->izq);
                preorden(nodo->der);
                }}
void inorden(struct Nodo *nodo){
        if(nodo!=NULL){
          inorden(nodo->izq);
          printf("%d ",nodo->dato);
          inorden(nodo->der);
                }}
```

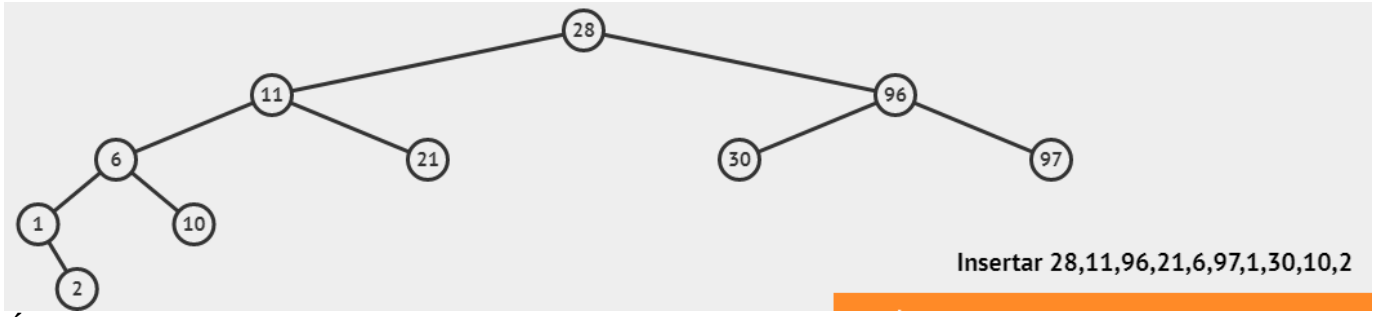
```
void postorden(struct Nodo *nodo){
          if(nodo!=NULL){
                    postorden(nodo->izq);
                    postorden(nodo->der);
             printf("%d ",nodo->dato);
}}
void Borrar(struct Nodo *arbol, int dat) {
  struct Nodo* padre = NULL; /* (1) */
  struct Nodo* actual;
  struct Nodo* nodo;
  int aux;
  actual = arbol;
  while(!Vacio(actual)) { /* Búsqueda else implícito */
    if(dat == actual->dato) {
      if(EsHoja(actual)) {
        if(padre)
          if(padre->der == actual) padre->der = NULL;
          else if(padre->izq == actual) padre->izq = NULL;
        free(actual);
        actual = NULL;
        printf("\nNodo %d eliminado\n",dat);
        return;
      }
      else {
        /* Buscar nodo */
        padre = actual;
        if(actual->der) {
          nodo = actual->der;
          while(nodo->izq) {
            padre = nodo;
            nodo = nodo->izq;
          }
        }
        else {
          nodo = actual->izq;
          while(nodo->der) {
            padre = nodo;
            nodo = nodo->der;
          }
        }
        /* Intercambio */
        aux = actual->dato;
        actual->dato = nodo->dato;
        nodo->dato = aux;
        actual = nodo;
      }
    }
    else {
      padre = actual;
      if(dat > actual->dato) actual = actual->der;
      else if(dat < actual->dato) actual = actual->izq;
```

```
    }
  }

}
void Buscar(struct Nodo* a, int dat) {
  struct Nodo* actual = a;
  while(!Vacio(actual)) {
    if(dat == actual->dato){
          printf("\nNumero %d encontrado\n",dat);
          return; /* dato encontrado  */
    }else if(dat < actual->dato) actual = actual->izq;
    else if(dat > actual->dato) actual = actual->der;
  }
  printf("\nNumero %d no encontrado\n",dat); /* No está en árbol*/
}
int Vacio(struct Nodo* r) {
  return r==NULL;
}
int EsHoja(struct Nodo* r) {
  return !r->der && !r->izq;
}
void nivelActual(struct Nodo* raiz, int nivel){
          if (!raiz)
          return;
    if (nivel == 1)
          printf("%d ", raiz->dato);
    else if (nivel > 1) {
          nivelActual(raiz->izq, nivel - 1);
          nivelActual(raiz->der, nivel - 1);
    }
}
int altura(struct Nodo* nodo){
          if (!nodo)
          return 0;
    else {
          /*Se calcula la altura de cada subarbol*/
          int izqAltura = altura(nodo->izq);
          int derAltura = altura(nodo->der);

          /*Se utiliza el mas largo */
          if (izqAltura > derAltura)
                return (izqAltura + 1);
          else
                return (derAltura + 1);
    }}
void nivelOrden(struct Nodo* raiz){
    int al = altura(raiz);
    int i;
    for (i = 1; i <= al; i++)
          nivelActual(raiz, i);
}
```

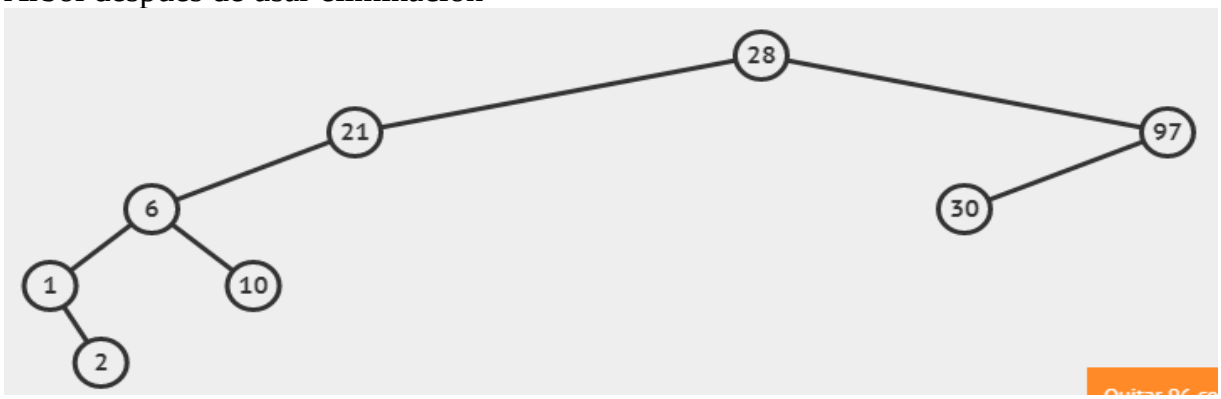
-Resultados



Árbol antes de usar eliminación



Árbol despúes de usar eliminación



Como podemos observar el método de búsqueda por anchura se realiza de manera óptima porque como podemos apreciar ya que imprime los nodos por niveles empezando desde la raíz para despúes seguir con los nodos que tiene enlazados para continuar haciendolo asi sucesivamente hasta llegar a lo más profundo del árbol, haciendo esto de manera de niveles. También podemos ver que el programa nos muestra la altura del arbol que en este caso tiene una altura de 5 niveles. Y tambien podemos imprimir solamente un nivel del árbol, como en este caso que quisimos imprimir sólo el nivel 3 el cual muestra exactamente como debe de ser: 6,21,30,97. Despúes tenemos los métodos de borrar y buscar los cuales ya hicimos con anterioridad, y para verificar que si funciona el metodo de busqueda en anchura, los volvemos a invocar al final y efectivamente nos muestra los datos como debe der ser.

-Conclusión

El método de búsqueda en anchura es una form de mostrar los nodos de un árbol pero tal manera que se muestren por niveles en el cual se visitan los hijos de cada rama para despúes regresar y mostrar los hijos del otro nodo del mismo nivel y asi sucesivamente. Este método primeramente tuve que comprender como funcionaba el algoritmo teóricamente, para despues al momento de escribir el código poder comprender que realiza cada línea. Me pareció interesante que los tres métodos que se usan en el algoritmo se pueden usar independientemente para mostar otros datos del árbol, como saber que tan profundo es el árbol con el método altura(), poder imprimir solamente un nivel del árbol con el método nivelActual(),y con estos dos métodos anteriores hacer uso de ellos para asi imprimir todo el árbol completo.

-Referencias

Búsqueda en anchura y búsqueda en profundidad para árboles. (2021, 15 noviembre). Estructura de Datos. Recuperado 14 de noviembre de 2021, de https://odilodif2.blogspot.com/2014/03/busqueda-en-anchura-y-busqueda-en.html

C con Clase. (s. f.). *C Con Clase | Estructuras de datos (cap7)*. Recuperado 14 de noviembre de 2021, de http://conclase.net/c/edd/cap7

GeeksforGeeks. (2021, 4 octubre). *Level Order Binary Tree Traversal*. https://www.geeksforgeeks.org/level-order-tree-traversal/