Algoritmos y Estructura de datos

Clase 14
Arboles parte 3

Profesor: Carlos Diaz

Contenido

- Operaciones en árboles binarios de búsqueda
- Contar nodos de un árbol
- Profundidad de un árbol
- Recorrido en anchura
- Función recursiva verNivel()
- Insertar un nodo
- Búsqueda de un nodo
- Borrar un nodo
- Practica calificada 4

Operaciones en árboles binarios de búsqueda

- Los árboles binarios de búsqueda, tienen naturaleza recursiva y, en consecuencia, las operaciones sobre los árboles son recursivas, si bien siempre se tiene la opción de realizarlas de forma iterativa.
- Estas operaciones son:

1. Contar nodos del árbol.
2. Profundidad de un árbol.
3. Recorrido de un árbol: En profundidad(preorden, enorden y postorden) y en anchura.
4. Insertar un nodo: Crea un nodo con su dato asociado y lo añade, en orden, al árbol.
5. Búsqueda de un nodo: Devuelve la referencia al nodo del árbol, si no lo encuentra devuelve NULL.
6. Borrado de un nodo. Busca el nodo del árbol que contiene un dato y lo quita del árbol. El árbol debe seguir siendo de búsqueda.

Contar nodos de un árbol

- Para contar nodos empezamos de la raíz y el contador n se inicializa a 1.
- Luego recorre analizando primero las ramas izquierda hasta encontrar NULL y luego prosigue con la rama derecha.

```
//Cuenta nodos del arbol
int arbolBB::contarNodos(Nodo* raiz)
{
    int n=1;
    if (raiz->izq!=NULL)
        n=n+contarNodos(raiz->izq);
    if (raiz->der!=NULL)
        n=n+contarNodos(raiz->der);
    return n;
}
```

Profundidad de un árbol

- La profundidad de un árbol con solo raíz es 1, por eso se suma 1 a la respuesta final.
- El programa recorre todas las ramas hasta llegar a las hojas y luego al regresar va sumando 1 y compara para obtener el máximo recorrido.

```
//Altura del nodo
int arbolBB::encontrarAltura(Nodo* raiz)
{
    if(raiz!=NULL)
        return max(encontrarAltura(raiz->izq),encontrarAltura(raiz->der))+1;
}
```

Recorrido de un árbol en anchura

- Para recorrer un árbol por anchura primero debemos conocer sus niveles.
- La función porAnchura(), llama a la función encontrarAltura() y con ello calcula los niveles, luego imprime cada nivel empezando desde 0 mediante la función recursiva verNivel().

```
//Recorrer árbol por anchura (por nivel)
void arbolBB::porAnchura(arbolBB a)
{
        int nivel=0,niveles;
        niveles=a.encontrarAltura(a.raiz)-1;
        while(nivel<=niveles)
        {
                verNivel(a.raiz,nivel);
                nivel++;
                cout<<endl;
        }
}
```

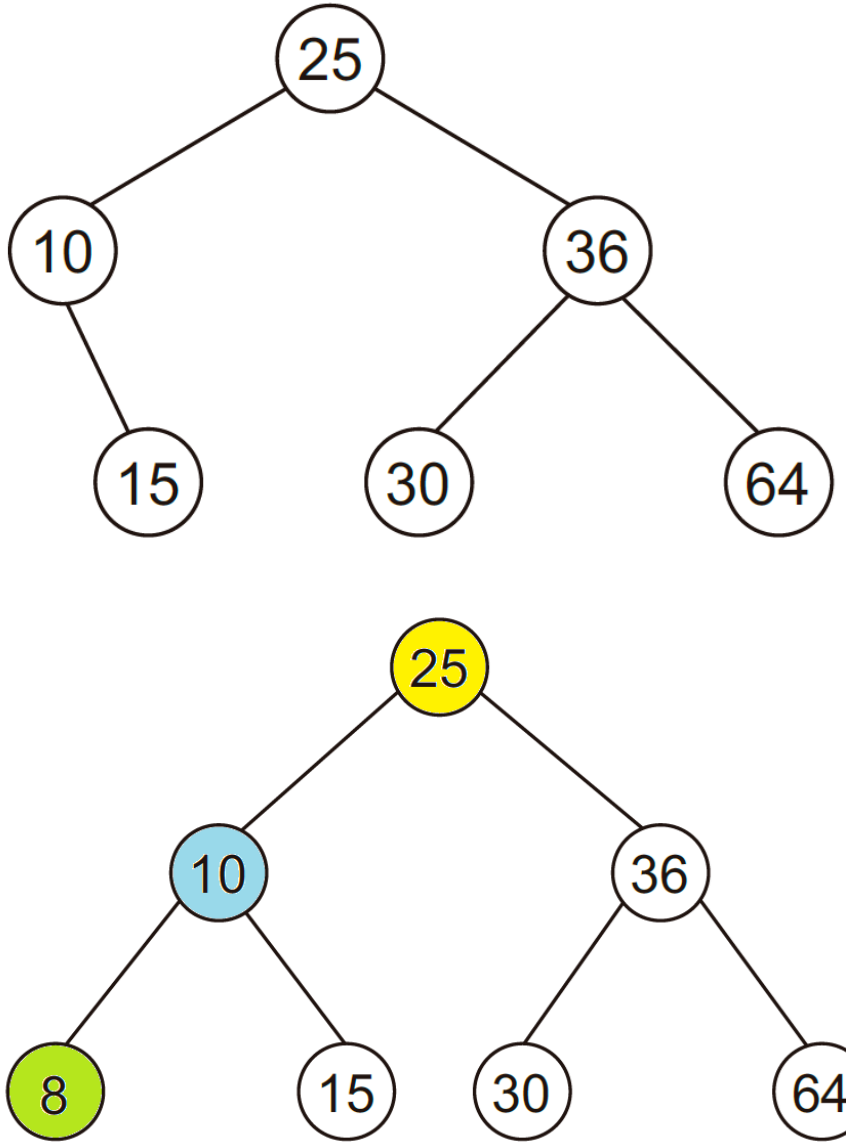
La función recursiva verNivel()

- La función recursiva verifica si la raíz es NULL.
- Si no lo es, verifica si ya se llego al nivel que deseamos mostrar.
- Se inicia desde la raíz del árbol.
- Y si deseamos ver el nivel n, entonces la función va bajando al siguiente subárbol (siguiente nivel), cambiando la raíz y le resta 1 a n.
- Cuando n llega a cero, significa que llego al nivel solicitado e imprime el dato de esa raíz del subárbol.

```
//Funcion recursiva para ver por nivel
void arbolBB::verNivel(Nodo* raiz,int n)
{
    if(raiz!=NULL)
    {
        if(n==0)
        {
            cout<<raiz->dato<<" ";
        }
        else
        {
            verNivel(raiz->izq,n-1);
            verNivel(raiz->der,n-1);
        }
    }
}
```

Insertar un nodo

- Para añadir un nodo al árbol se sigue el camino de búsqueda, y al final del camino se enlaza el nuevo nodo, por consiguiente, siempre se inserta como hoja del árbol.
- El árbol que resulta después de insertar sigue siendo siempre de búsqueda.
- Por ejemplo, al árbol de la figura se le va a añadir el nodo 8.
- El proceso describe un *camino de búsqueda* que comienza en el raíz 25, el nodo 8 debe estar en el subárbol izquierdo de 25 (8 < 25).
- El nodo 10 es el raíz del subárbol actual, el nodo 8 debe estar en el subárbol izquierdo (8 < 10), que está actualmente vacío y, por tanto, ha terminado el *camino de búsqueda*.
- El nodo 8 se enlaza como hijo izquierdo del nodo 10.
- Esto ya esta programado en el método crear().



Búsqueda de un nodo

- El programa comienza inicializando r a la raíz.
- Si es r no es NULL, comprueba si tiene el dato buscado, si no lo es, el siguiente paso es preguntarse si es menor a mayor a la raíz para reasignar a r el nodo izquierdo o derecho de la raíz respectivamente.
- Tan pronto se encuentre el nodo el bucle while finaliza con un break. Si no lo encuentra devuelve un NULL.

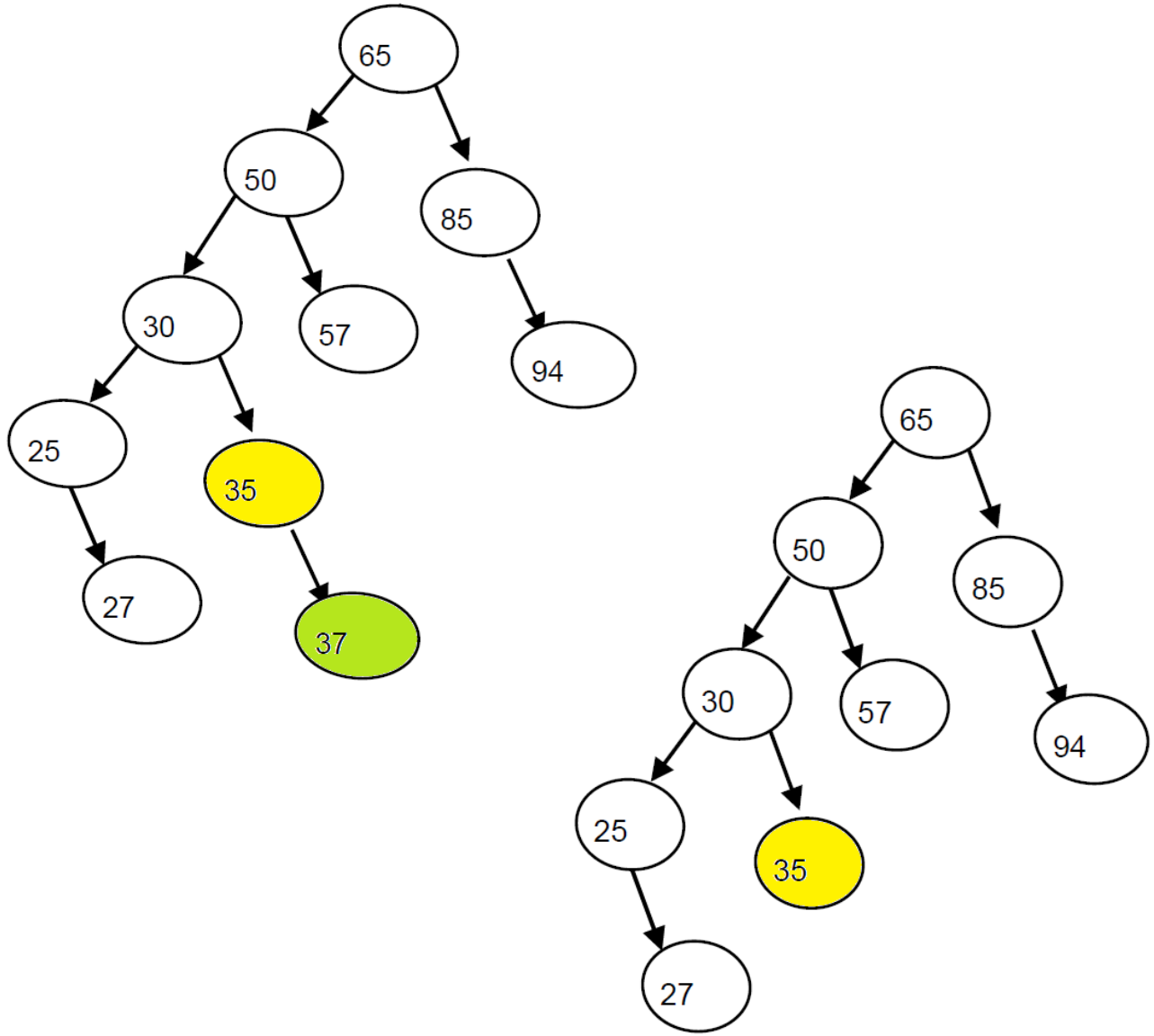
```
//Busqueda de un nodo
Nodo* arbolBB::busqueda(Nodo* raiz, int datoBuscado)
{
        Nodo* r;
        Nodo* rpta=NULL;
        r=raiz; //Inicializa la raiz
        while (r!=NULL)
        {
                if (r->dato==datoBuscado)
                {
                        rpta=r;
                        break;
                }
                if (datoBuscado < r->dato)
                        r=r->izq;
                else
                        r=r->der;
        }
        return rpta;
}
```

Borrar un nodo

- Durante la eliminación de un nodo, se debe tomar en cuenta los siguientes casos: (denominaremos destino al nodo que eliminaremos)

1. El nodo destino no tiene ningún nodo hijo (nodo hoja).
2. El nodo destino tiene solo un hijo (puede ser izquierdo o derecho).
3. El nodo destino tiene ambos hijos.

Caso 1: El nodo destino no tiene ningún nodo hijo (nodo hoja): Necesitamos determinar al padre del nodo destino. Una vez que se conoce al padre, simplemente se hace que el puntero hijo de este padre sea NULL.



Código Caso 1: Nodo hoja

```
//Borrar nodo hoja
void arbolBB::borrarNodo (int datoBorrar)
{
	Nodo*r,*padre=NULL,*temp;
	r=raiz;
	while(r!=NULL) //Busca el dato a borrar
	{
		if(r->dato==datoBorrar)
			break;
		padre=r;
		if (datoBorrar < r->dato)
			r=r->izq;
		else
			r=r->der;
	} // Fin de la busqueda
	if(r==NULL)
	{
		cout<<"El dato no esta en el arbol"<<endl;
		return;
	}
```

```
	else if (padre->dato < r->dato) //Si el nodo objetivo esta a la derecha
	{
		if ((r->izq==NULL) &&(r->der==NULL)) // si r es una hoja
		{
			temp=r;
			padre->der=NULL;
			delete temp;
		}
	}
	else if (padre->dato > r->dato) //Si el nodo objetivo esta a la izquierda
	{
		if ((r->izq==NULL) &&(r->der==NULL)) // si r es una hoja
		{
			temp=r;
			padre->izq=NULL;
			delete temp;
		}
	}
}
```

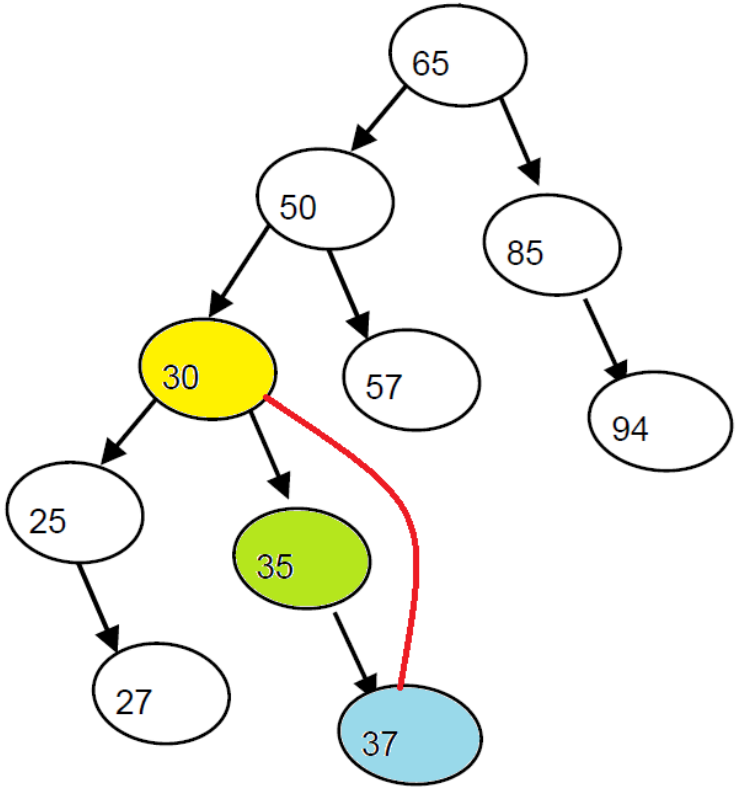
Borrar un nodo

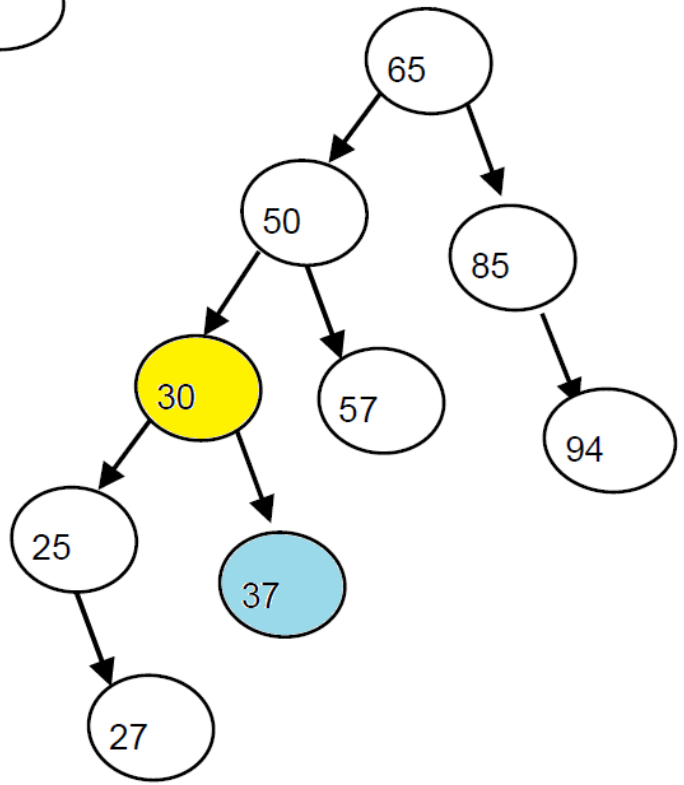
Caso 2: El nodo destino tiene solo un hijo (puede ser izquierdo o derecho): En este caso, necesitamos determinar el nodo padre del nodo destino.

Una vez que el nodo padre es determinado, tenemos que averiguar si el nodo destino es el izquierdo o el derecho.

Conecte el padre del nodo destino al hijo del nodo destino.

Al programa anterior agregamos más condicionales que analicen este caso:





Código Caso 2: Solo un hijo

```
//Borrar nodo
void arbolBB::borrarNodo (int datoBorrar)
{
	Nodo*r,*padre=NULL,*temp;
	r=raiz;
	while(r!=NULL) //Busca el dato a borrar
	{
		if(r->dato==datoBorrar)
			break;
		padre=r;
		if (datoBorrar < r->dato)
			r=r->izq;
		else
			r=r->der;
	} //Fin de la busqueda
	if(r==NULL)
	{
		cout<<" El dato no esta en el arbol"<<endl;
		return;
	}
	else if (padre->dato < r->dato) //Si el nodo objetivo esta a la derecha
	{
		if ((r->der==NULL) && (r->izq!=NULL)) // Si r solo tiene hijo izquierdo
		{
			temp=r;
			padre->der=r->izq;
			delete temp;
		}
		else if ((r->der!=NULL) && (r->izq==NULL)) // Si r solo tiene hijo derecho
		{
			temp=r;
			padre->der=r->der;
			delete temp;
		}
		else if ((r->izq==NULL) &&(r->der==NULL)) // si r es una hoja
		{
			temp=r;
			padre->der=NULL;
			delete temp;
		}
	}

	else if (padre->dato > r->dato) //Si el nodo objetivo esta a la izquierda
	{
		if ((r->der==NULL) && (r->izq!=NULL)) // Si r solo tiene hijo izquierdo
		{
			temp=r;
			padre->izq=r->izq;
			delete temp;
		}
		else if ((r->der!=NULL) && (r->izq==NULL)) // Si r solo tiene hijo derecho
		{
			temp=r;
			padre->izq=r->der;
			delete temp;
		}
		else if ((r->izq==NULL) &&(r->der==NULL)) // si r es una hoja
		{
			temp=r;
			padre->izq=NULL;
			delete temp;
		}
	}
}
```

Calificada 4

- Viernes 05/02/2021, 3 pm – 6pm
- Tema árboles binarios de búsqueda

FIN