void alta\_nodo(notoT \*raiz, dato){

//crea e Inserta un nuevo nodo con info = dato, en un árbol binario de búsqueda

//apegarse a pseudocódigo de libro, Algoritmo 6.8

// entonces puede imprimir mensaje dentro de función de nodo repetido;

}

nodoT \* crear\_arbol\_busqueda( int arre[], ?) { //nueva versión,

raiz = new nodoT()

raiz->dato = arre[0];

raiz -> izq = NULL;

raiz -> der = NULL ;

for ( , , cont++ ){

alta\_nodo(raiz, arre[ cont ])

}

return raiz;

}

Void busqueda\_binaria(raiz, dato){

//apegarse al algoritmo 6.6. Del libro; entonces puede imprimir mensajes dentro de función;

}

void imprime\_Indice(raiz, nivel ){

/\* tipo indice , es recorrido preorden, pero con tabuladores( un tabulador equivalente por 2 o 3 espacios) (a mayor nivel mas tabuladores), y saltos línea por cada nodo,

Para nodos terminales no es necesario imprimir los dos hijos NULL, pero si el nodo tiene un sólo hijo, entonces indicar con NULL el faltante;

Ejemplo, 25 tiene hijos izq a 12 y derecho a 37 ; 12 no tiene hijo izq, pero derecho a 15;

25

12

NULL

15

37

\*/

}

void libera\_memoria(raiz){

//Pista: sólo uno de los tres recorridos de árbol servirá de apoyo en liberar memoria,¿cuál será?

}

void main(){

arre= {120, 87, 43, ...} //capturar arreglo del ejemplo 6.12 , página 207, del libro, visto en clase , esto para que podamos validar en clases; pero debe funcionar para cualquier arreglo;

raiz= crear\_arbol\_busqueda( arre, tam )

imprime\_preorden(raiz); //de tarea anterior, para comparar en autoevaluación

imprime\_inorden(raiz); //de tarea anterior, para comparar en autoevaluación

imprime\_Indice(raiz, 1);

//buscar valores aleatorios

busqueda\_binaria(raiz, 56) // debe imprimir , sí se encuentra en el árbol;

busqueda\_binaria(raiz, 57) // debe imprimir , no se encuentra en el árbol;

libera\_memoria(raiz)

}

Void borrar\_nodo(raiz, dato)

ap= buscar\_nodo(raiz, dato) //ap apuntador al nodo por borrar

Caso 1 ap es terminal ?

borrar\_terminal(..)

Return

Caso 2

Ap tiene un sólo hijo?

Borrar\_nodo\_un\_hijo (..)

Return

Caso 3 ap tiene dos hijos?

Borrar\_nodo\_dos\_hijo (ap ,)

}

Void Borrar\_nodo\_dos\_hijo ( ap , ){

Aux = ap -> izq //Ir al hijo izq de ap,

// recursión Irse toda derecha; preferente función

Aux = recursion\_derecha( Aux )

// sencillo, sólo copiar info/dato de aux en ap ;

ap->dato = aux ->dato;

//Hijo izq de aux sube al lugar de aux, ojo no es simple copia de dato!

Padre de aux // pendiente por resolver, modificar esta rutina para saber padre de aux,

Padre\_aux -> der = aux -> izq ; //nos saltamos a aux

free aux;

}

main ( ){

//.. Aquí va Tarea anterior..

raiz= crear\_arbol (..)

//..

//Borrando …

arre\_borrar= [ , , ] // valores del ejemplo 6.14 del libro

ciclo en arre\_borrar {

Print ‘borrando dato’ , arre\_borrar[i]

borrar\_nodo(raiz, arre\_borrar[i] )

Imprimir\_in\_orden(..)

}

Imprimir\_indice (raiz)

}

//para caso 3, ¿por qué no sólo subir algún hijo? , ¿porqué complicar con recursion\_derecha()?