FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

TDA Arbol Binario de Busqueda

Nombre y apellido	Lautaro De Nobili
Padrón	107394
Email	ldenobili@fi.uba.ar

Índice

Introducción al ABB

El TDA a implementar consiste en un ABB (definicion del mismo dada en la teoria a continuacion). Nuestro TDA dispone de una estructura abb_t, que es nuestro arbol, en donde tenemos una referencia al nodo raiz de nuestro arbol, un abb_comparador, que es una estructura definida para la comparacion de elementos de nuestro arbol lo cual nos sirve para conocer en donde ordenar nuestros elementos, y bueno, un size_t tamanio que nos aclara la cantidad de elementos que contiene nuestro arbol.

Adicionalmente, tenemos definidos nuestros nodo_abb_t, los cuales son nodos recursivos en el sentido en que referencian a otros nodo_abb_t, siendo estos nuestro hijo izquierdo e hijo derecho, ya que es binario nuestro arbol.

Teoría

- Que es un arbol?

  Es un tipo de dato aleatorio en el que disponemos de un nodo raiz, recordando que nuestros nodos almacenan referencias a nuestro elemento a almacenar en el arbol, y tantos nodos hijos como nos propongamos.

- Que es un arbol binario?

  Es un arbol en el cual cada nodo, incluida la raiz, tienen solo dos referencias a hijos, en nuestro caso llamados izquierda y derecha, y referencia al elemento.

- Que es un arbol binario de busqueda?

  Es un arbol binario que mantiene un orden definido no solo por cuando se inserto el elemento, sino que tambien se ordena de izquierda a derecha según nosotros querramos. En nuesto TDA, lo implementamos con elementos menores o iguales a la izquierda del elemento actual y mayores a la derecha.

- La diferencia…

  Basicamente estos distintos tipos de implementacion difieren en la complejidad algoritmica para cada operación, siendo esta diferencia mucho mayor en la busqueda en el ABB (justamente por eso es de busqueda), ya que se utilizaria una busqueda binaria impulsada por nuestro comparador.

Funciones sin manejo de nodos

- abb_crear:

  Reservo memoria inicializada en 0 para la estructura abb_t y asigno el comparador del usuario a la misma;

- abb_vacio:

  Devuelvo true si el abb_t es nulo o si el tamanio del mismo es 0;

- abb_tamanio:

  Devuelvo el tamanio de mi abb_t;

Funciones con manipulacion de nodos… abb_insertar

Llamo inicialmente a mi insercion de nodos recursiva enviandole mi nodo raiz, luego me muevo hacia el primer nulo recorriendo hacia izquierda o derecha con cada nodo según mi comparacion me lo dicte.

Cada llamada recursiva es con asignacion hacia su respectivo nodo al cual se le envia a la funcion recursiva, de manera tal que una vez llegado al nulo donde debe de ir mi nodo, devuelva mi nuevo nodo y ya quede asignado ordenadamente.

Una vez terminado todo el recorrido y posterior a la asignacion, la funcion devuelve el mismo nodo que le fue enviado, de manera tal que quede todo asignado correctamente.

Para sellar el trato, incremento en uno el tamanio del arbol.

abb_buscar

Utilizo una funcion auxiliar para encontrar el nodo que contiene mi elemento buscado, funcion la cual es recursiva y sus llamadas recursivas son en base al resultado de la comparacion.

Una vez encontrado el elemento devuelvo el nodo que lo contiene o, en caso de no haberlo encontrado, devuelvo NULL.

Finalizada la busqueda del nodo, comparo el elemento de mi nodo conseguido a mi elemento buscado. Si coinciden, devuelvo el elemento de mi nodo, si no coinciden, devuelvo NULL porque no esta en el arbol mi elemento.

abb_destruir

La funcion abb_destruir es basicamente la funcion abb_destruir_todo pero pasando NULL como destructor, asi que eso es como lo implemente.

La funcion abb_destruir_todo llama a una funcion destructora de nodos recursivos pasandole el destructor. Esa misma funcion se llama recursivamente tanto hacia la izquierda como para la derecha y aplica el destructor en caso de haber, luego libera el nodo actual.

Una vez finalizada toda la recursividad, es decir que todos los nodos fueron destruidos y los elementos aplicados su respectivo destructor, libero la memoria del arbol.

abb_con_cada_elemento

Mi implementacion consiste en llamar a distintas tres funciones de recorridos según el recorrido que me pidan. Adicionalmente, tengo un contador y un booleando de continuidad, los cuales seran modificados cada vez que sea aplicada la funcion en mis recorridos.

Por que opte por tener el contador y el booleano dentro del abb_con_cada_elemento? Basicamente porque no me funcionaba lo que hacia, segundamente, porque me parecio mas sencillo y entendible tener “globalmente” para mis llamadas recursivas el si continuar o no y el contador. De esta manera no solo no necesito hacerme la cabeza con que valores devolver y cuando devolver, sino que ademas funciona y esta, creo yo, lindo el codigo.

Finalmente devuelvo mi contador al usuario.

abb_recorrer

Al igual que con la funcion abb_con_cada_elemento, mi funcion consiste en llamar otras funciones según cual sea el recorrido especificado, estas funciones son nuevamente del tipo void y llevan un contador “global”.

Sin embargo, mi condicion de corte en estas no es un booleano de continuidad, sino que es la comparacion de si mi contador es menor al tamanio del array especificado y/o me haya quedado sin elementos en el arbol.

Algo que cabe destacar para ambos abb_recorrer y abb_con_cada_elemento es que en los inorder y postorder chequeo que mi condicion de corte no haya cambiado en las llamadas recursivas previas, para asi no aplicar mi procedimiento nuevamente, ya que una vez terminadas esas llamadas recursivas, mis funciones siguen haciendo lo suyo “sin haberse enterado de que tenian que parar”.

abb_quitar (sin hijos)

Si bien en mi codigo la implementacion del quitar sin hijos y con un hijo estan hechas juntas, mis explicaciones van a ser por separado.

La funcion consiste en llamar a otra auxiliar que va por todos los nodos y avanza a la izquierda o la derecha en funcion de como sea la comparacion. Esta funcion auxiliar devuelve un nodo, de forma tal que al quitar algun nodo, esta, recursivamente, devuelva su predecesor.

Adicionalmente, tiene un elemento_buscado, el cual se le cambia su valor una vez encontrado el elemento entre comparaciones, pasado por referencia de manera tal que se pueda almacenar fuera de la funcion sin necesidad de que esta lo devuelva en el return. Opte por esto por simplicidad y claridad, y porque me venia funcionando con el abb_recorrer y abb_con_cada_elemento.

En el caso de no tener hijos, mi funcion libera el nodo, obvio luego de almacenar el elemento encontrado, y devuelve NULL, ya que va a ser mi nuevo hijo (izquierda o derecha) para el nodo que haya llamado a esta funcion.

Terminada la funcion recursiva, comparo el elemento obtenido al buscado, si coinciden, devuelvo el elemento obtenido, si no, devuelvo NULL.

abb_quitar (con un hijo)

Para quitar, ya sea sin hijos, con un hijo, o con dos hijos, mi funcion hace lo mismo, compara hasta llegar al nodo que contenga mi elemento a quitar, solo cambia el curso de accion en base a cuantos hijos contenga mis nodos.

Para un solo hijo, me almaceno auxiliarmente el hijo, o un hijo izquierdo o hijo derecho, libero el nodo y lo devuelvo para que el padre de mi nodo que contenia el elemento lo herede. Heredar hijos? Quien soy yo para juzgar…

Por que mi quitar con un hijo y sin hijos son lo mismo? Porque si mi nodo no tiene hijos, es decir que es nodo hoja, “tiene” de hijos a dos NULL, y aplicando lo mismo que con un hijo, terminaria devolviendole NULL al padre de mi nodo contenedor.

abb_quitar (con dos hijos)

Si que fue complicado implementar esta funcion, no tanto el razonamiento detrás de este, pero si el codigo. Muchos loops infinitos en donde el padre de mi predecesor seguia con hijo a mi predecesor, muchos segmentation fault. Pero se pudo!

Al razonar el quitar un nodo con dos hijos pude contemplar ciertos casos:

- Este predecesor puede ser directamente el nodo izquierdo al que quiero borrar. En este caso solo asigno la derecha de mi nodo contenedor a la derecha del predecesor. Libero el contenedor y devuelvo el predecesor;

- Ahora, este predecesor puede no ser el izquierdo inmediato, en cuyo caso lo busco yendo a la izquierda y luego todo a la derecha, asi lo obtengo. No sirve solo asignarle la derecha de mi contenedor al predecesor, pues mi padre sigue referenciando a mi contenedor. LOOP! Asi que al buscar mi predecesor, me voy guardando su padre asi una vez llegado al que necesito, le cambio su nodo derecho a NULL. Me ahorro los loops, no los necesito.

abb_quitar (con dos hijos)

Adicionalmente a los casos anteriores, puede haber otra variacion. Que mi predecesor tenga hijos izquierdos.

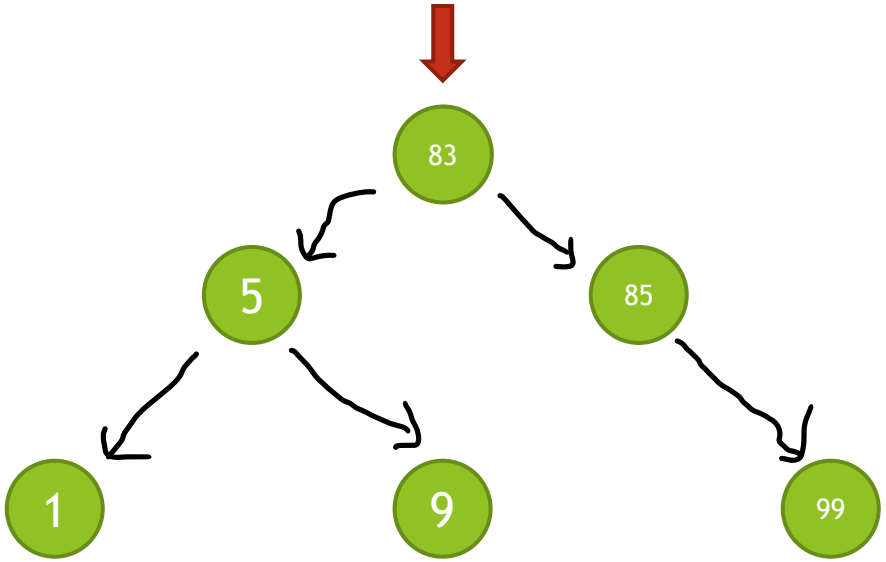
Si no los tiene, todo lindo, le asigno a la izquierda y la derecha sus respectivos del nodo contenedor, y que su padre apunte a NULL de ser necesario.

Ahora, si los tiene, la cosa cambia. Si mi nodo predecesor es el izquierdo inmediato al nodo contenedor, no importa si tenga izquierdo o no, solo le asigno el derecho. Si fuese a asignarle el izquierdo tambien, mi nodo entraria en loop consigo mismo.

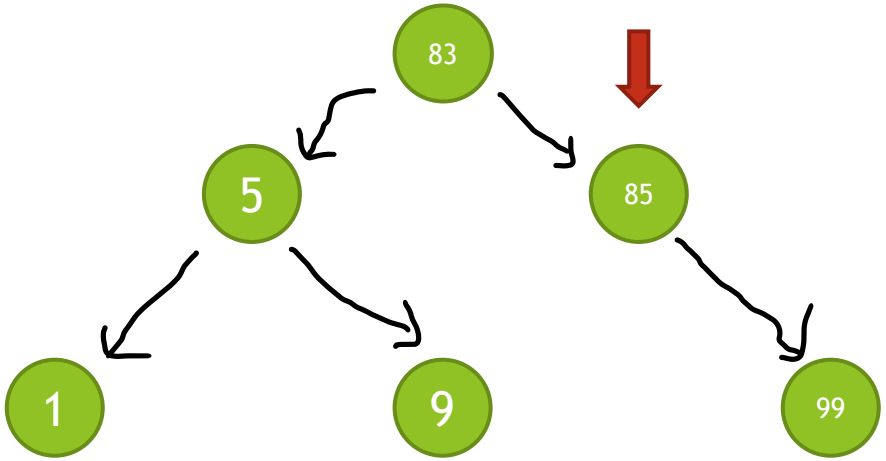
Si mi nodo predecesor no es el izquierdo inmediato y le asigno el derecho e izquierdo de mi contenedor, perderia la referencia a los N nodos a la izquierda del predecesor. En este caso, el mas complicado de avivarme de su existencia, se pueden optar por dos opciones:

- Puedo hacer que el padre de mi predecesor herede a la derecha los izquierdos de mi predecesor. Se mantiene el orden;
- Puedo hacer que el nodo mas izquierdo de mi predecesor herede a la izquierda el nodo izquierdo de mi nodo contenedor. Se sigue manteniendo el orden. Yo opte por esta misma.

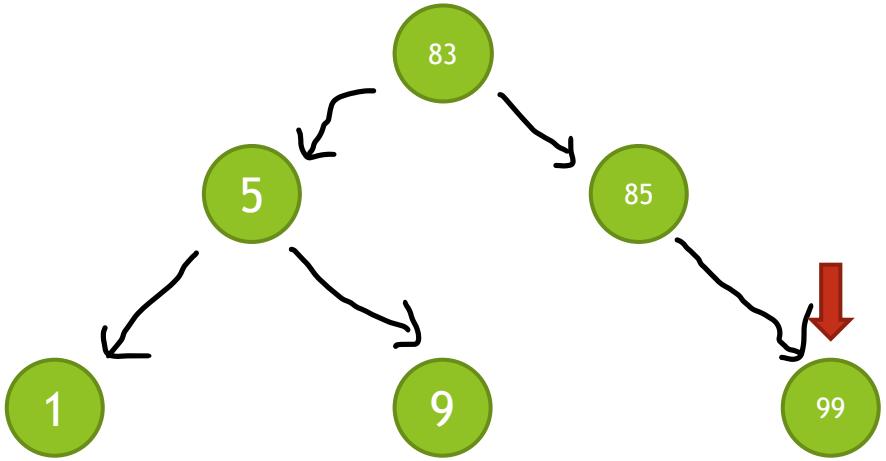
abb_quitar: elemento 99



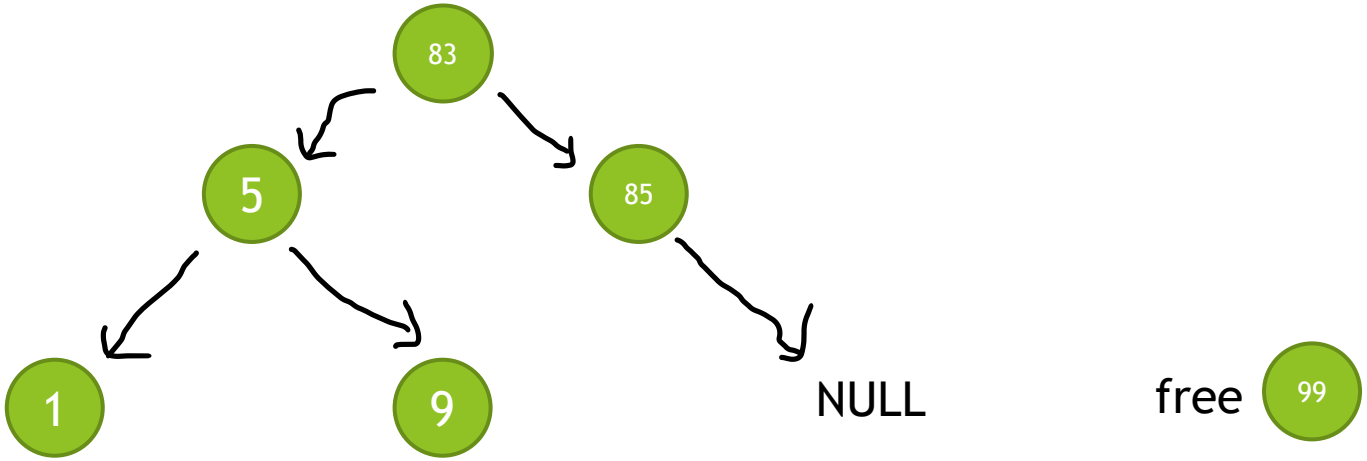
abb_quitar: elemento 99



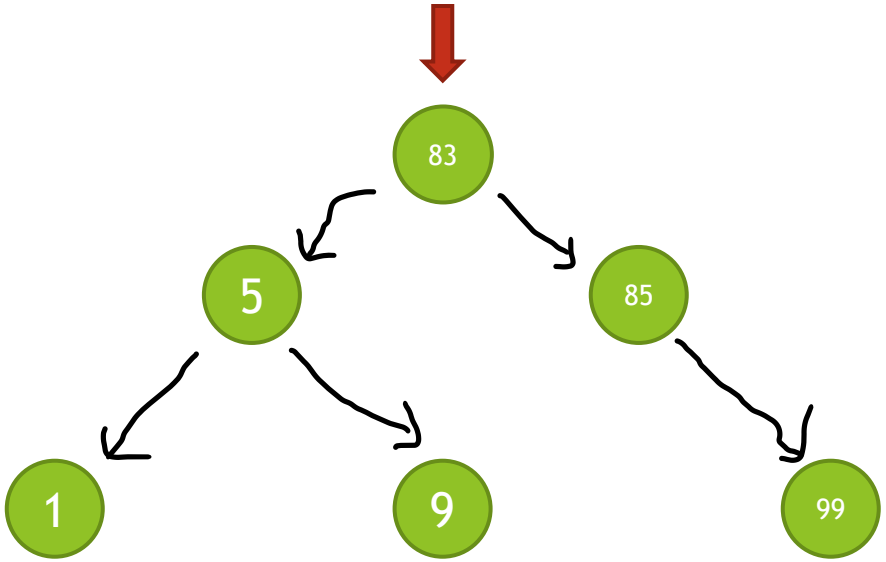
abb_quitar: elemento 99



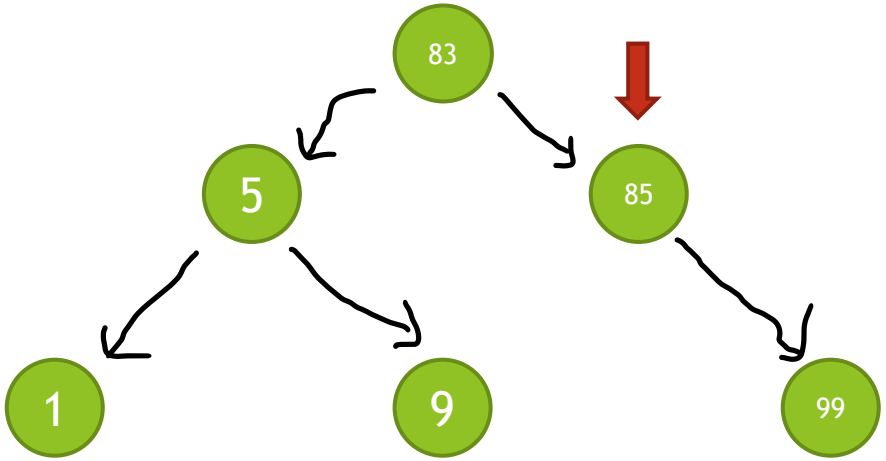
abb_quitar: elemento 99



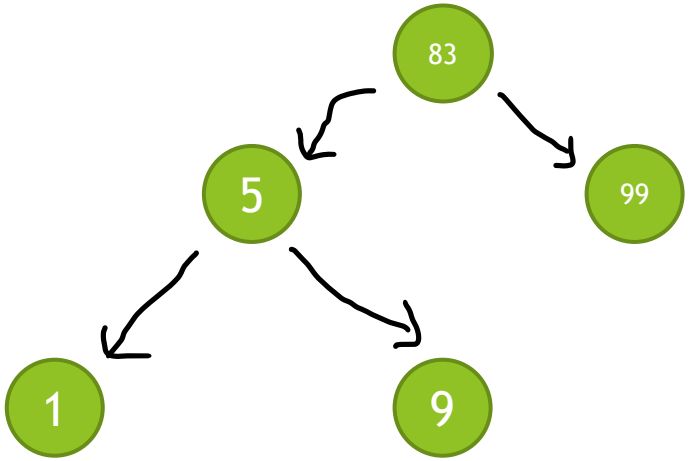
abb_quitar: elemento 85



abb_quitar: elemento 85

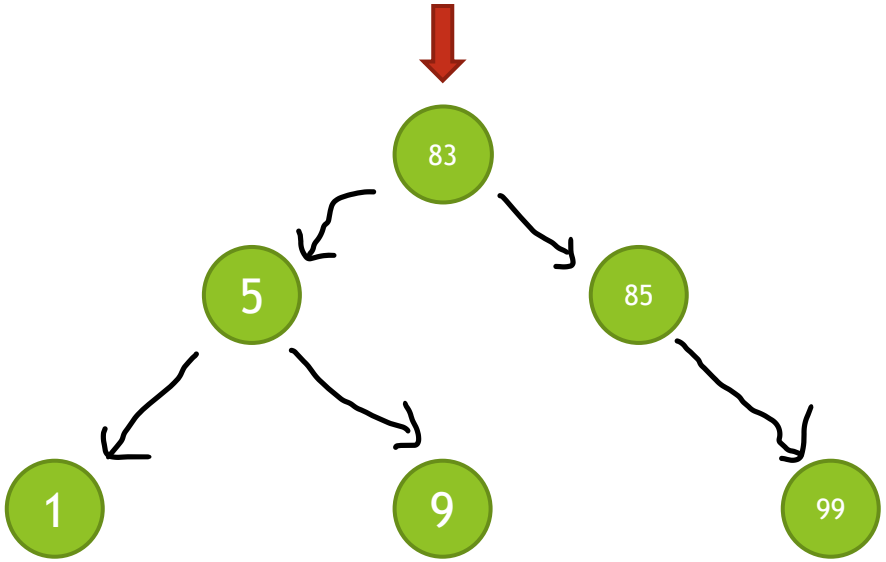


abb_quitar: elemento 85

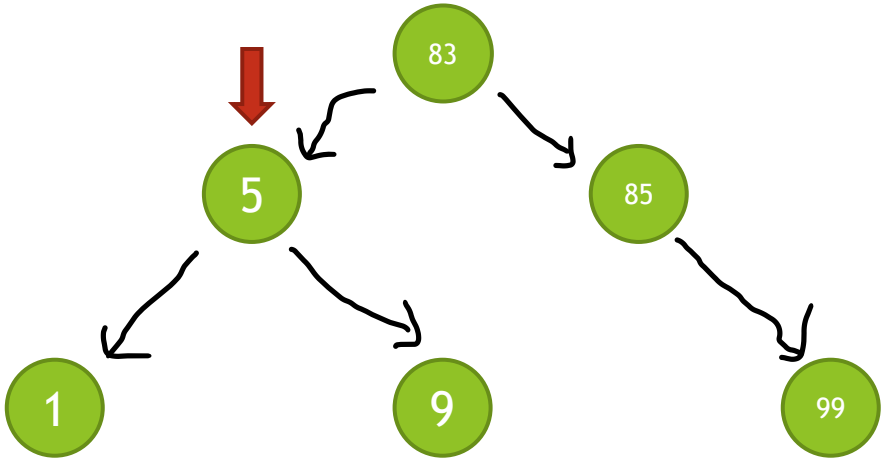


free 85

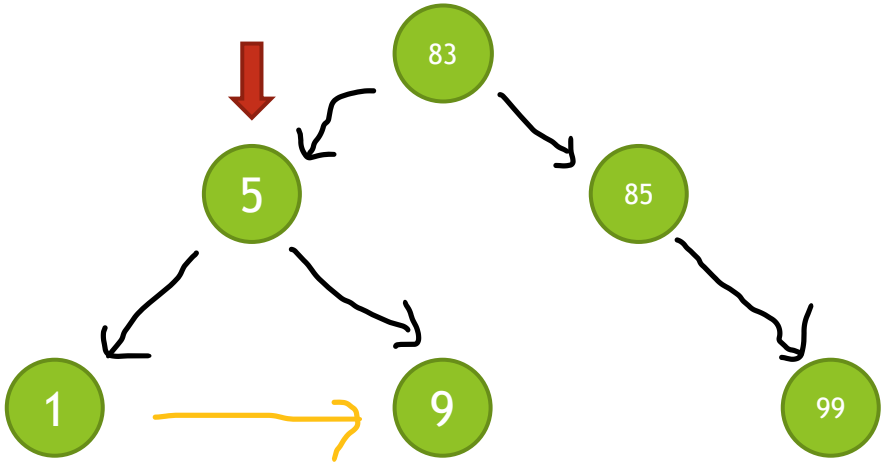
abb_quitar: elemento 5



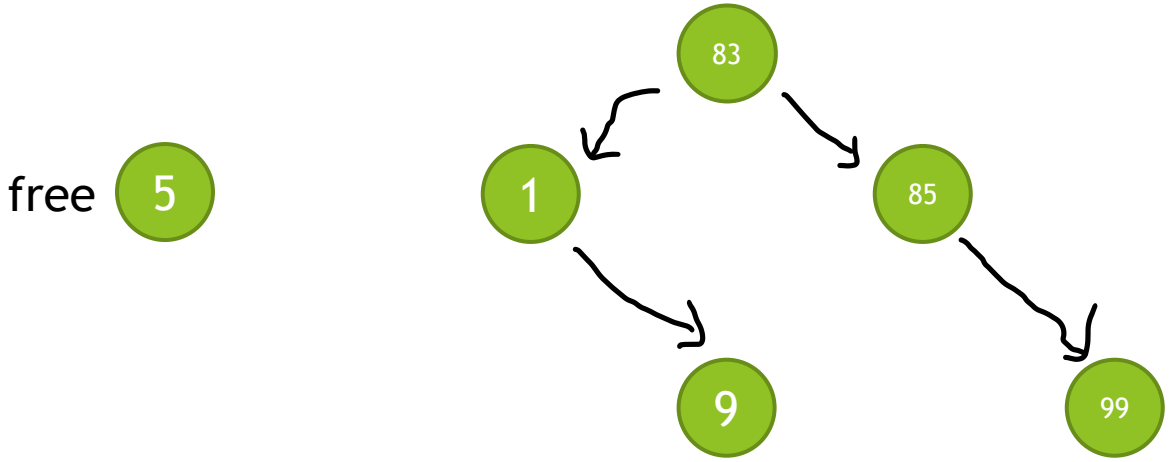
abb_quitar: elemento 5



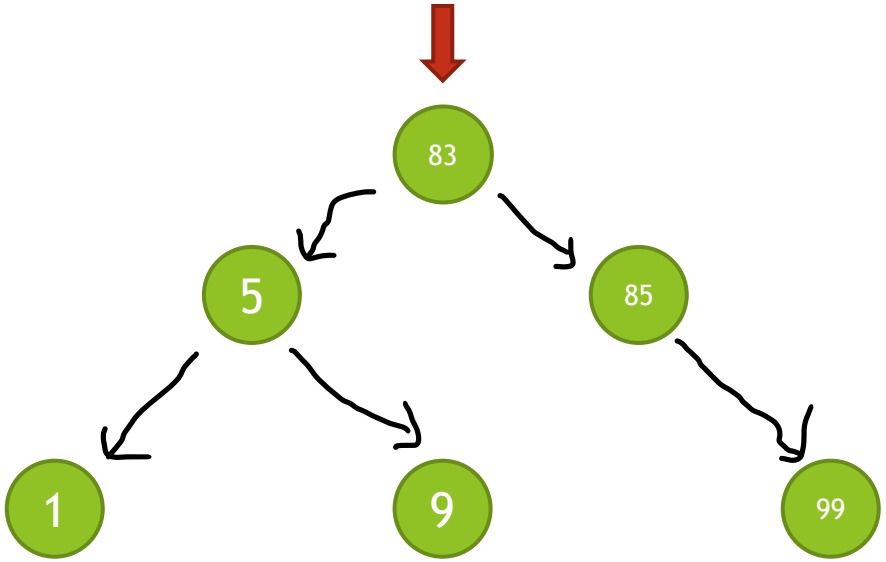
abb_quitar: elemento 5



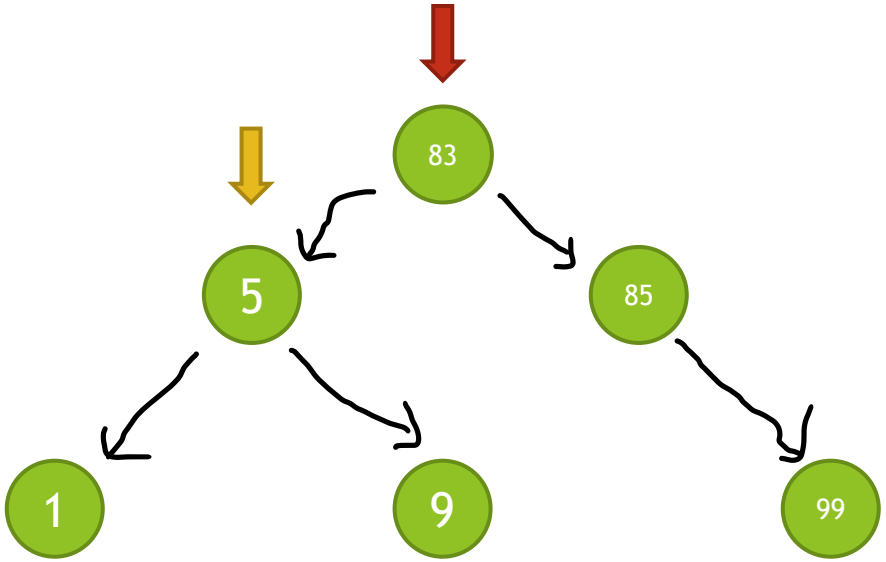
abb_quitar: elemento 5



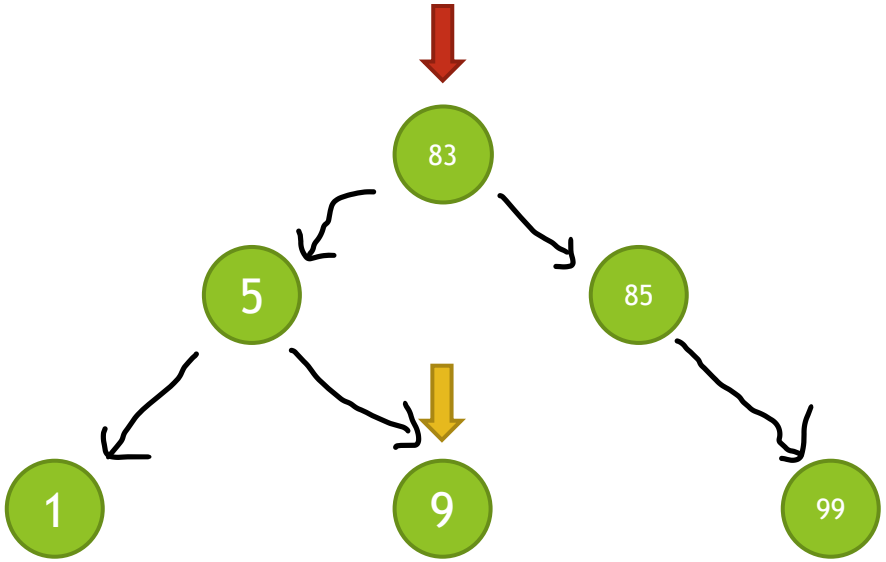
abb_quitar: elemento 83



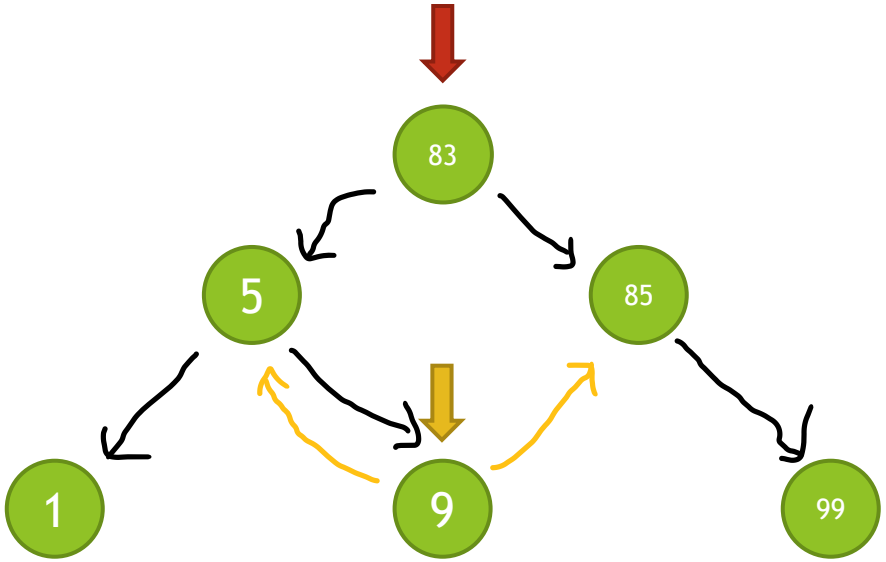
abb_quitar: elemento 83



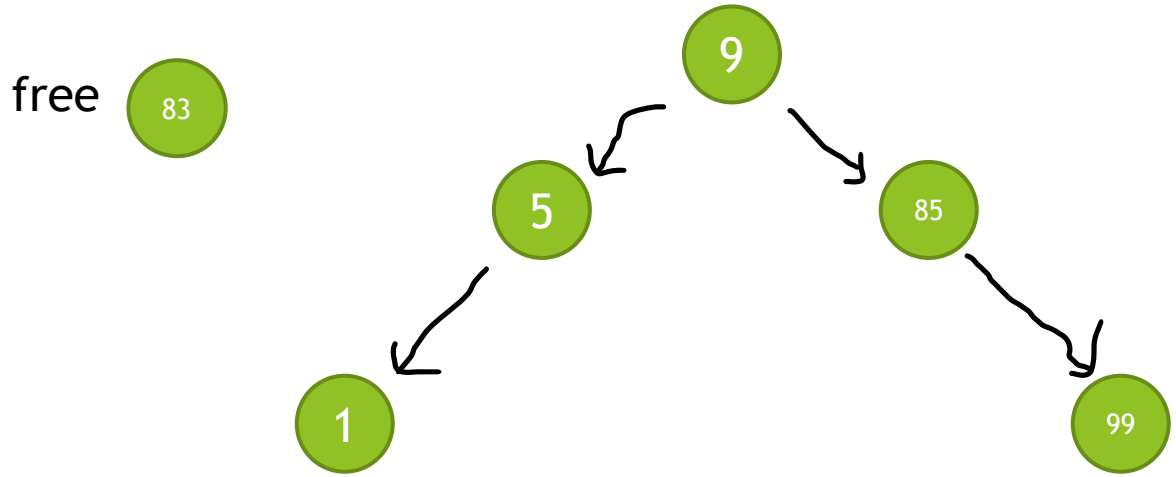
abb_quitar: elemento 83



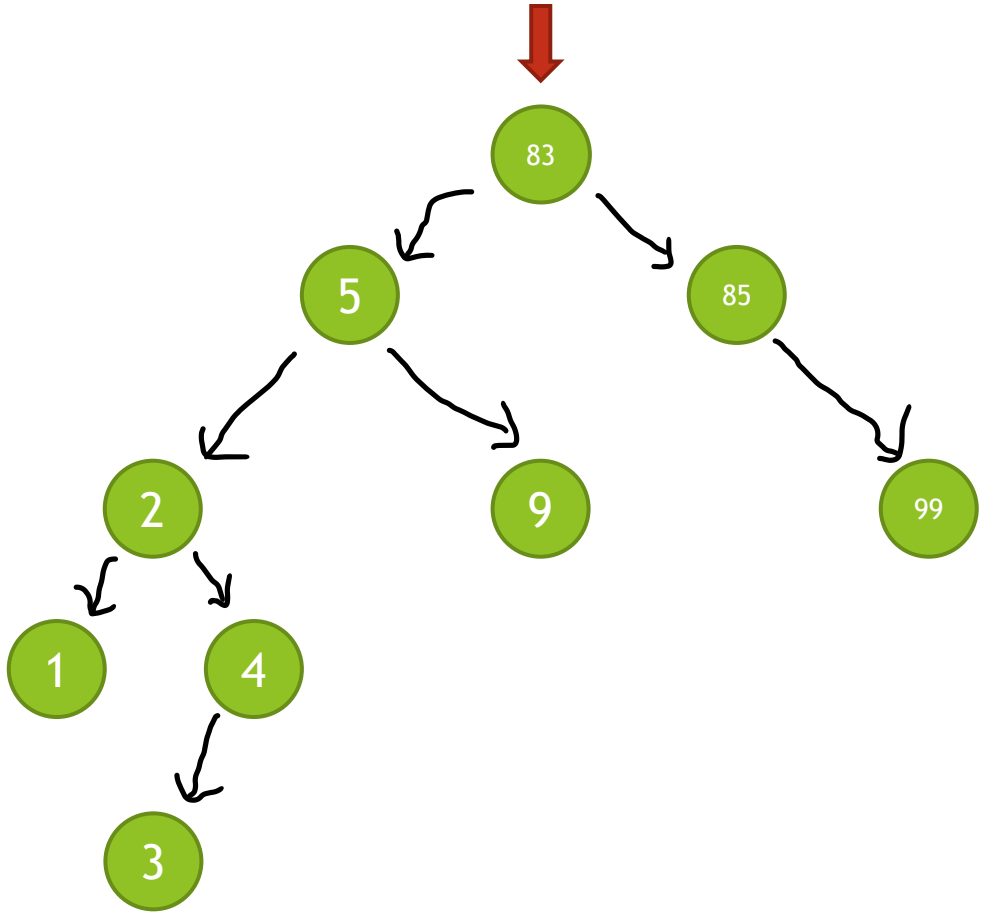
abb_quitar: elemento 83



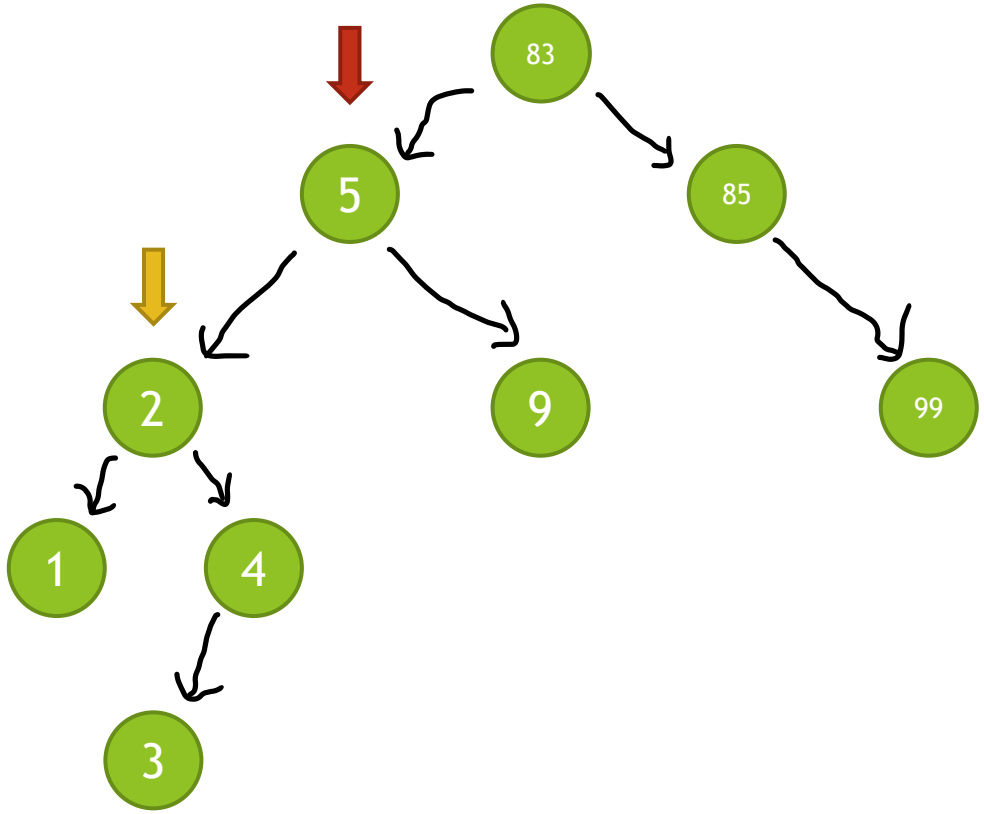
abb_quitar: elemento 83



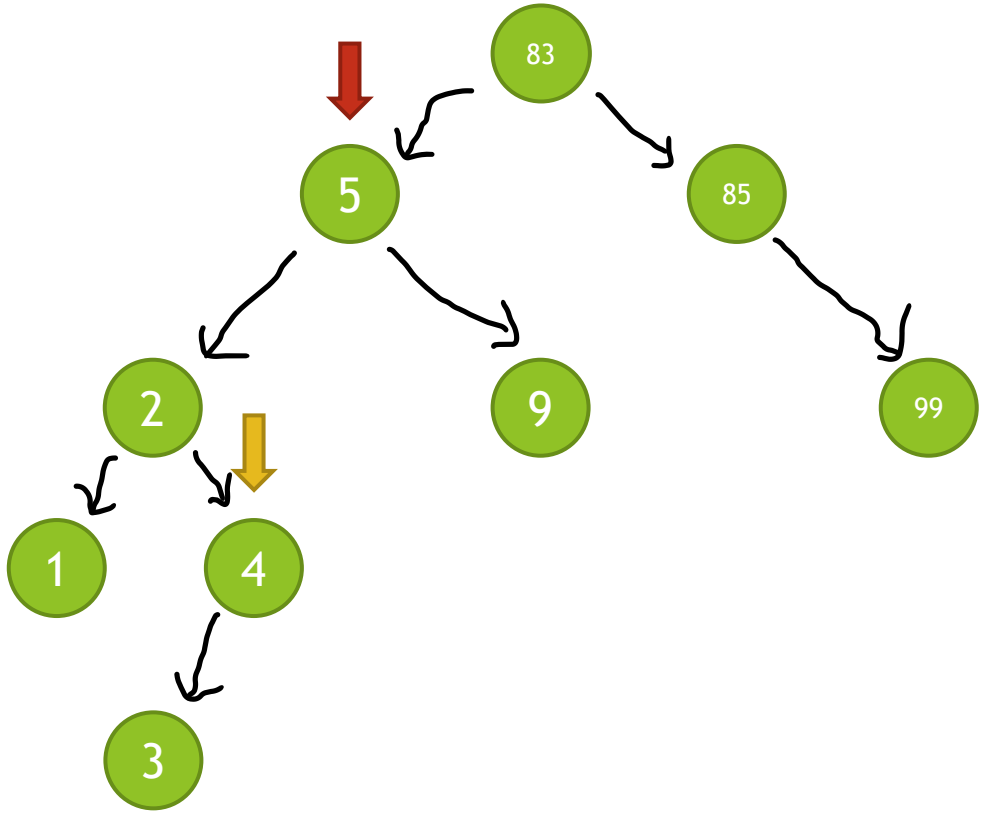
abb_quitar: elemento 5



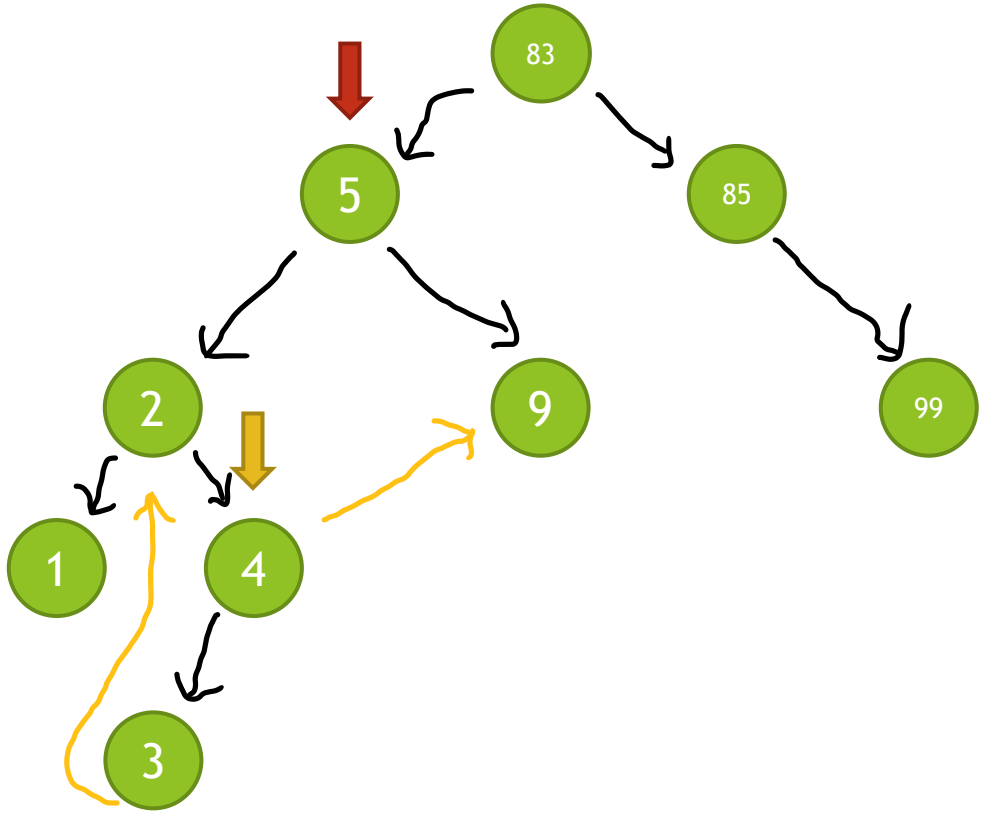
abb_quitar: elemento 5



abb_quitar: elemento 5



abb_quitar: elemento 5



abb_quitar: elemento 5