TDA — ABB

[7541/9515] Algoritmos y Programación II

Segundo cuatrimestre de 2021

Alumno:	Sanchez Fernandez de la Vega, Manuel
Número de padrón:	107951
Email:	msanchezf@fi.uba.ar

1. Introducción

El objetivo del TDA es implementar un arbol binario de busqueda, tambien conocido como ABB. Es un tipo de dato abstracto eficiente a la hora de realizar operaciones como busqueda, insercion o eliminacion.

2. Teoría

1. Un arbol es un tipo de dato abstracto conformado por una coleccion de nodos, los cuales estan enlazados en forma de arbol. Cada nodo tiene Hay muchos tipos de arboles, algunos mas eficientes que otros, distintas cantidades de hijos e implementaciones, etc. Por ejemplo, AVL, ABB, arbol n-ario, arbol rojo y negro, arbol B, B+ y B*, entre otros. Un tipo de arbol son los arboles binarios, cuyos nodos unicamente tienen 2 hijos. Este tipo de dato no esta ordenado de ninguna forma y simplemente consta de una coleccion de nodos conectados arbitrariamente. La idea de arbol binario se puede expandir un poco mas, al agregar la condicion de que los elementos mayores al elemento de cada nodo deben ir a la derecha, y los menores a la izquierda (esto es una convencion, tranquilamente se pueden poner los menores a la derecha y mayores a la izquierda). Esta condicion significa que se puede optimizar en gran forma las operaciones que se pueden realizar. De esta forma, obtenemos un ABB o arbol binario de busqueda. Al ser binario, tiene propiedades similares a la busqueda binaria. El orden de complejidad de algunas operaciones como buscar o insertar es O(log(n)) por lo que se puede considerar como eficiente (siempre y cuando el arbol este balanceado). Esto se debe a que aprovecha el principio de divide y conquista para optimizar las operaciones. Por ejemplo, al buscar un elemento, cada vez que se pasa por un nodo (si el arbol esta balanceado), se compara para

ver si el elemento buscado es mayor, menor o igual al actual y se descarta la mitad de los mismos.

2. La explicacion de la implementacion se encuentra en la seccion de detalles de implementacion.

3. Detalles de implementación

A la hora de implementar el arbol binario de busqueda, utilice un conjunto de nodos enlazados. Estos nodos contienen 3 datos importantes:

- Un puntero a el hijo izquierdo
- Un puntero a el hijo derecho
- Un puntero al elemento

El hijo derecho de cada nodo siempre tiene un elemento mayor al del nodo, y el hijo izquierdo siempre tiene un elemento menor. Como consecuencia, queda armada una especie de arbol de nodos en la que los elementos mas grandes se encuentran por la derecha y los elementos mas chicos, por la izquierda. Este orden se mantiene siempre. En otras palabras, cuando se insertan o quitan elementos, hay que ser extremadamente cuidadosos de no quebrantar estas reglas. Sobre este arbol de nodos se pueden realizar operaciones similares a las de una lista pero con una complejidad superior. Como ya fue mencionado en las seccion teorica, se utiliza el principio de division y conquista para lograr esta optimizacion.

Luego, el padre de todos los nodos, es decir el nodo raiz, es referenciado en una nueva estructura que definimos como arbol, cuyos campos son:

- Un puntero al nodo raiz
- La cantidad de elementos
- Una funcion para comparar los elementos

Esta ultima funcion es necesaria debido a que es posible guardar absolutamente cualquier tipo de dato en los nodos. Por ello, y como ya fue explicado previamente, el arbol realiza comparaciones constantemente. Como resultado, es necesario definir una funcion que sirva para realizar comparaciones entre 2 elementos del tipo de dato suminstrado al arbol.

En mi implementacion, hice uso de la recursividad para el desarrollo de las funciones. Esto se debe a que permite desplazarse a lo largo del arbol de una forma mucho mas sencilla e intuitiva. Sin embargo, para poder hacer que las funciones sean recursivas, es necesario que en los parametros exista uno que represente al nodo actual. En consecuencia, no queda otra alternativa que crear funciones auxiliares que en vez de tener un parametro de arbol, tenga un parametro que sea el nodo actual. Esto tiene como fin el poder hacer una llamada recursiva pero usando `nodo->izquierda` o `nodo->derecha` dependiendo que queremos hacer.

1. `abb_con_cada_elemento()` y `abb_con_cada_elemento_aux()` `abb_con_cada_elemento()` es una funcion que permite recorrer el arbol aplicando una funcion pasada por parametro (que devuelva un bool) hasta que esta devuelva `false`. Para lograr esto, implemente una funcion auxiliar recursiva llamada `abb_con_cada_elemento_aux()`. Esta funcion nos da la opcion de recorrer el arbol usando 3 recorridos posibles:

- PREORDEN: Primero se aplica la funcion al nodo, luego al hijo izquierdo y finalmente al derecho.
- INORDEN: Primero se aplica la funcion al hijo izquierdo, luego al nodo y finalmente al hijo derecho.
- POSTORDEN: Primero se aplica la funcion al hijo izquierdo, luego al hijo derecho y por ultimo, al nodo.

  Despues de aplicar la funcion, hay que verificar el valor de retorno de la misma. Si esta devuelve `true`, se sigue aplicando en otros nodos. No obstante, si esta devuelve false, hay que cortar inmediatamente y dejar de recorrer el arbol. Es por ello que agregue un parametro llamado `estado`, el cual me permite saber si la funcion fallo o no desde otras llamadas recursivas. En otras palabras, despues de hacer cada llamado de funcion, verifico el estado para saber si fallo en alguna otra llamada recursiva. Si falló, directamente retorno y dejo de recorrer el arbol.

2. `abb_recorrer()` y `agregar_a_vector()`

   A la hora de implementar la funcion para recorrer el arbol, opte por reusar la funcion `abb_con_cada_elemento()` pero pasandole por parametro una funcion que agregue elementos a un vector y como recorrido le paso INORDEN. El gran problema con el que me encontre es que esta funcion solo me permite pasar tan solo 1 parametro extra, mientras que yo necesito pasar 3: el vector, la cantidad actual de elementos y la cantidad maxima. Es por eso que decidi crear un registro llamado vector, el cual tiene la siguiente estructura:

   ```
   typedef struct vector{
       void** elementos;
       size_t tamanio_maximo;
       size_t tamanio_actual;
   } vector_t;
   ```

   Gracias a esta estructura, logre pasar toda la informacion necesaria para poder crear una funcion que agregue elementos al vector reutilizando `abb_con_cada_elemento()`. Esta funcion devuelve `true` siempre y cuando el vector tenga espacio para agregar mas elementos. Esto me vino perfecto ya que usando `abb_con_cada_elemento()`, puedo dejar de recorrer el arbol una vez que el vector este lleno.

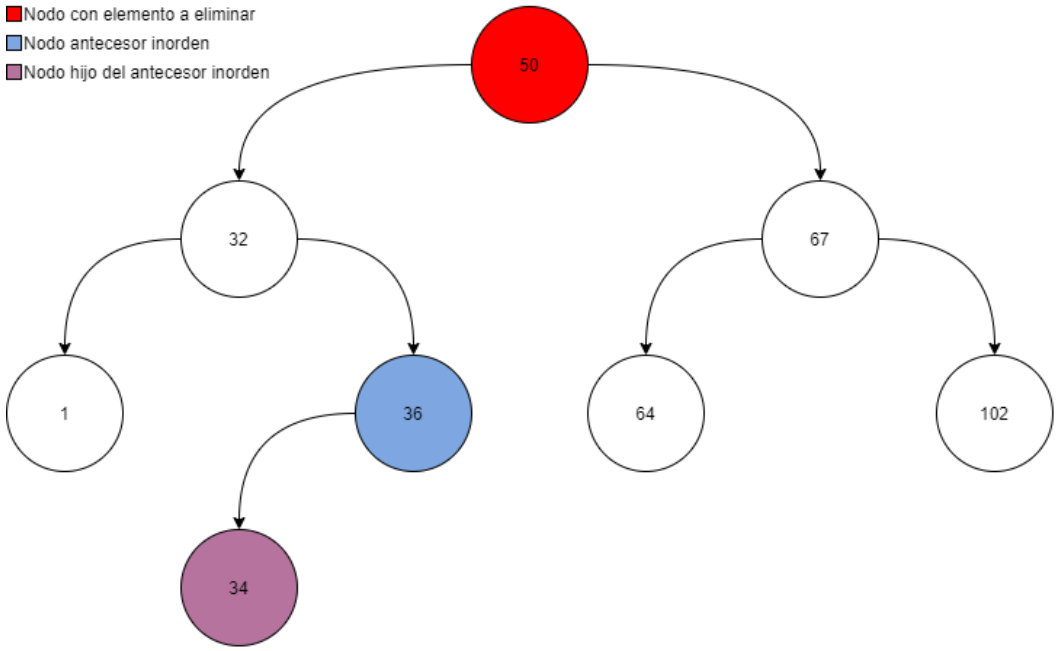
3. `abb_destruir()`, `abb_destruir_todo()` y `abb_destruir_aux()`

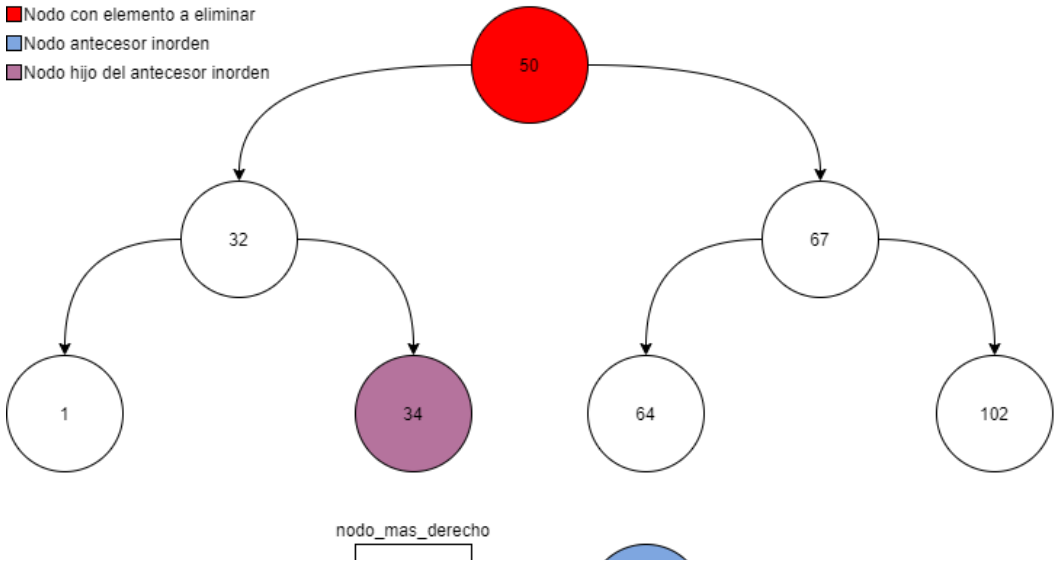
   A la hora de destruir el arbol, desarrolle una funcion auxiliar con el fin de poder hacerlo de forma recursiva. Esta funcion auxiliar es `abb_destruir_aux()` y recibe como parametros al nodo actual y a una funcion de destruccion de elementos. La funcion auxiliar es usada tanto en `abb_destruir()` como en `abb_destruir_todo()`, la unica diferencia es que en el caso de la primera, la funcion de destruccion es `NULL`, mientras que para la segunda, se pasa la funcion de destruccion correspondiente.

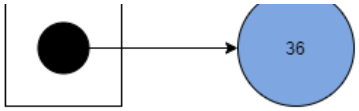
4. `abb_quitar()`, `abb_quitar_aux()` y `extraer_nodo_mas_derecho()` La funcion de quitar elementos llama a una funcion auxiliar llamada `abb_quitar_aux()`, la cual es recursiva y se encarga del verdadero proceso de eliminacion. Como ya fue mencionado previamente, hay que ser realmente cuidadosos a la hora de eliminar nodos en un abb. En especial cuando un nodo tiene 2 hijos ya que, inicialmente, puede parecer un proceso complejo. En mi implementacion, sustituyo el nodo a eliminar por su antecesor inorden y para lograr eso, previamente lo tengo que extraer utilizando la funcion `extraer_nodo_mas_derecho()` a la cual le paso como parametro el nodo izquierdo del nodo que quiero quitar. En la seccion de diagramas hay una explicacion mas visual de esta funcion.

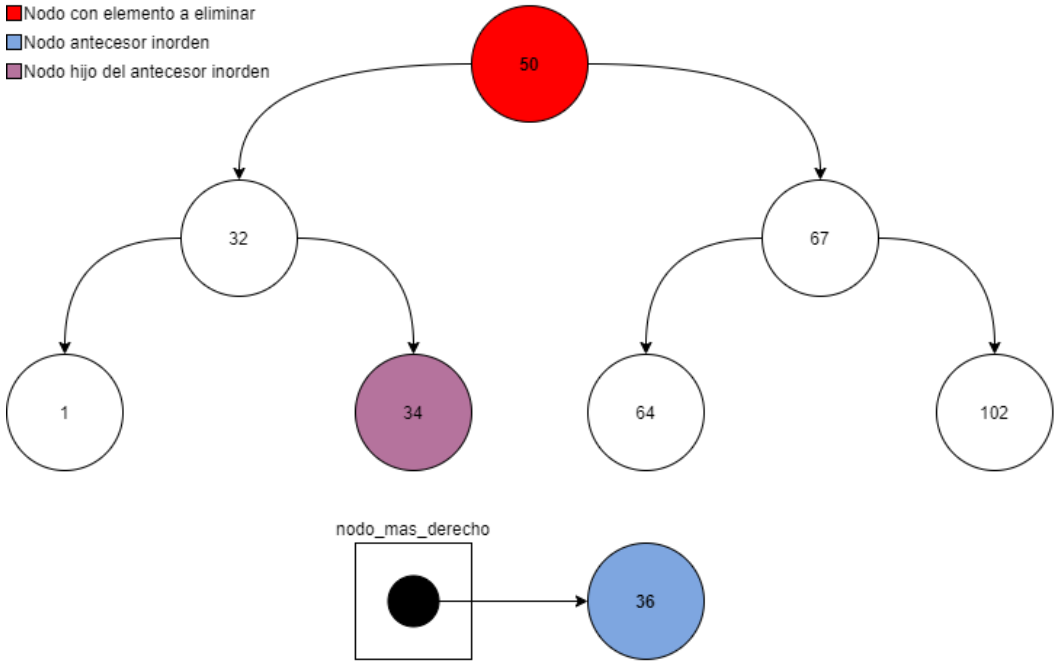
4. Diagramas

1. **Diagrama de `abb_quitar()` de nodo con 2 hijos**

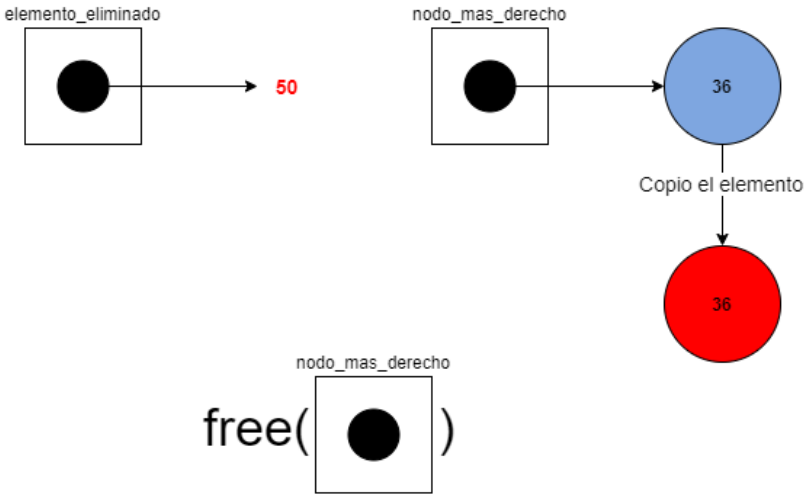


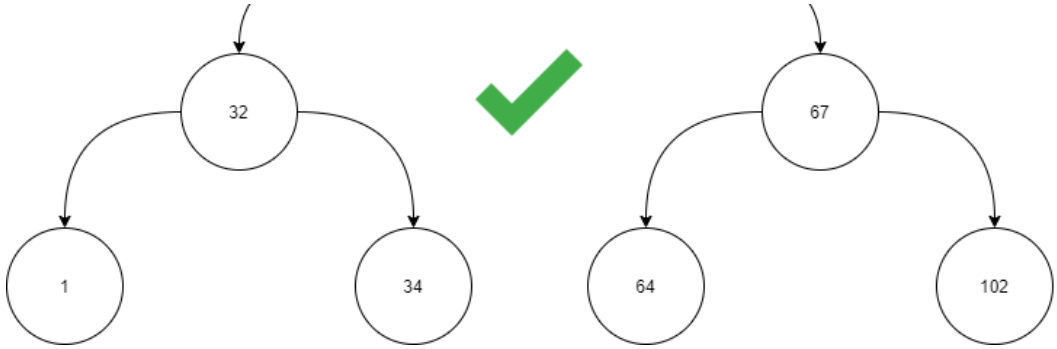






Guardo y sustituyo el elemento a quitar por el elemento del antecesor inorden

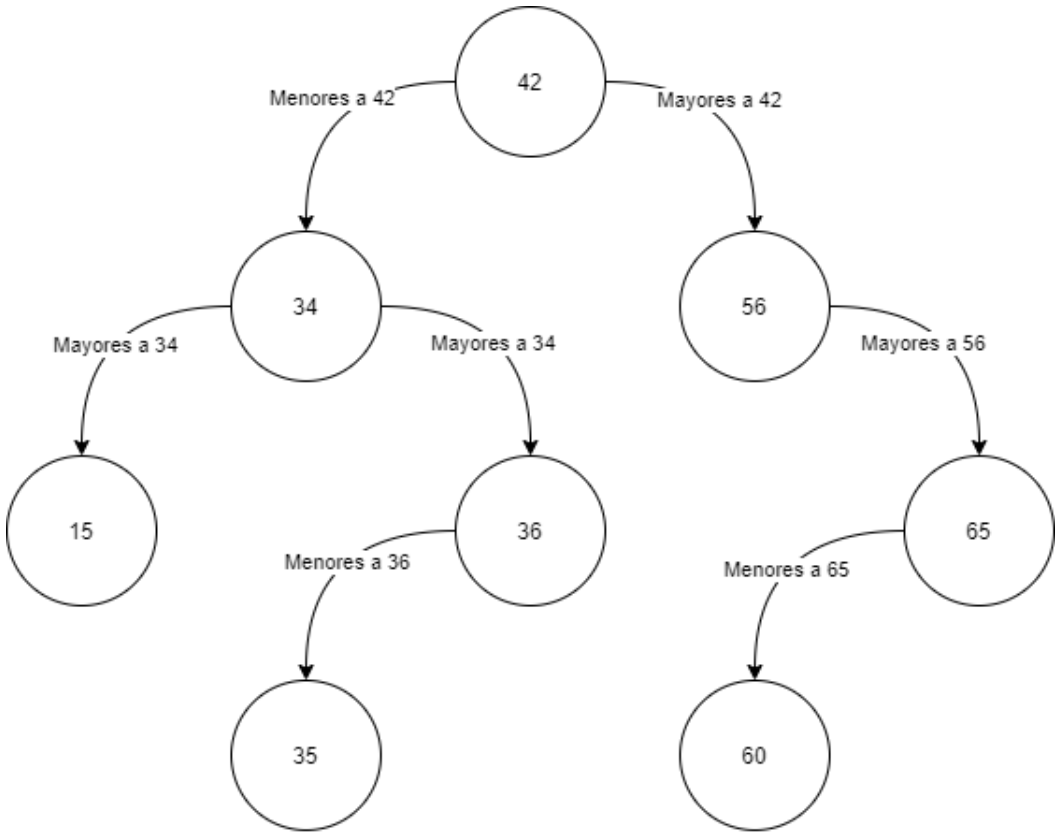




Si se quiere eliminar un elemento de un nodo con 2 hijos, lo que realiza el programa es:

1. Busca el antecesor inorden.
2. Lo extrae del arbol (pone a su hijo, si es que tiene uno, en su lugar).
3. Guarda el elemento a eliminar para poder devolverlo en la funcion luego.
4. Sustituye el elemento a eliminar en el nodo donde se encontraba por el elemento del antecesor inorden.
5. Libera la memoria asignada para el antecesor inorden.

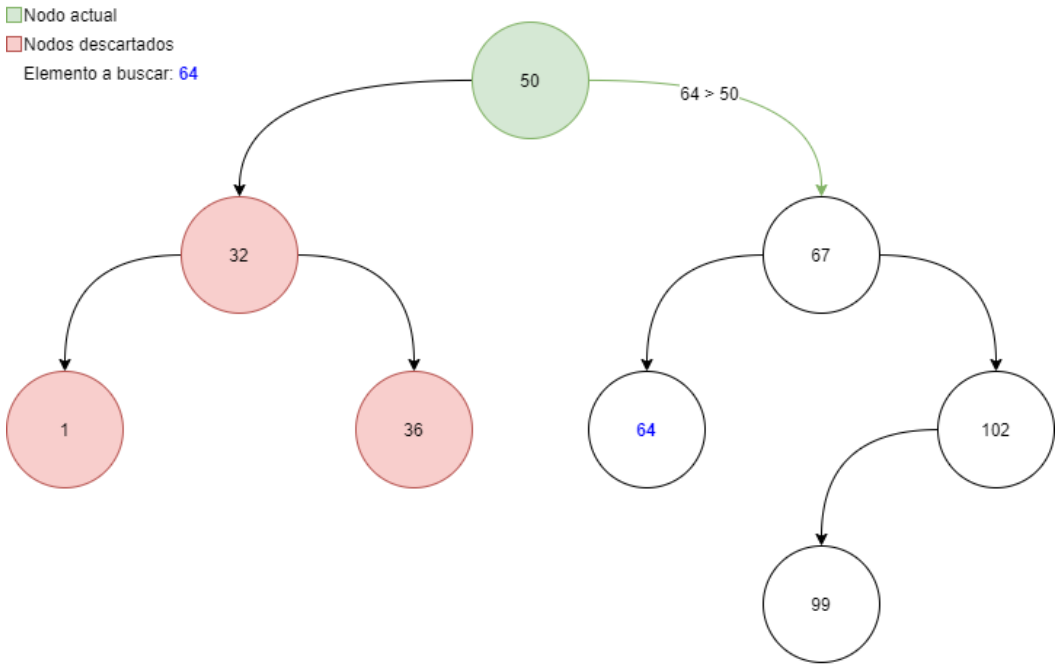
2. **Diagrama de ejemplo de arbol de pruebas**

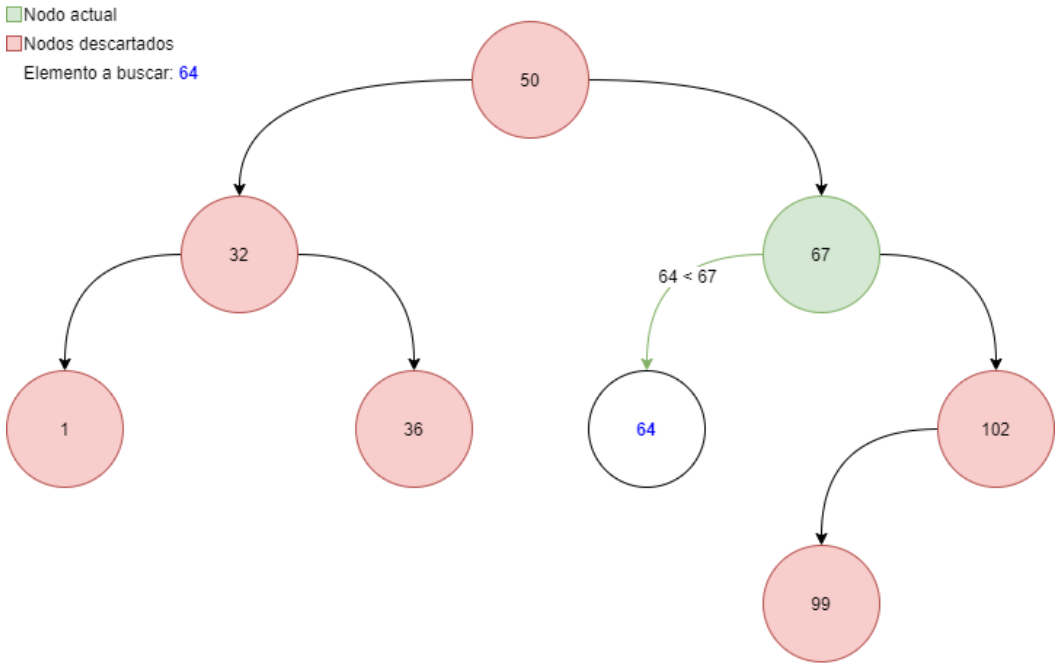


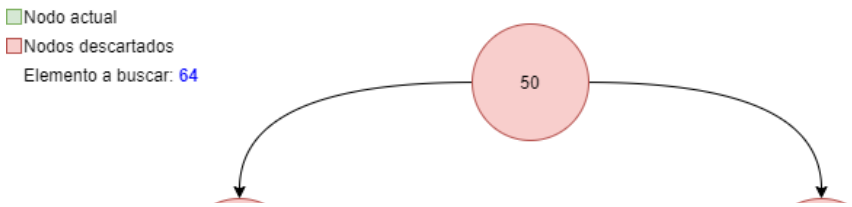
Este es un ejemplo de un arbol utilizado en varias de las pruebas. Para la creacion de las mismas, realice numerosos diagramas en papel con el fin de

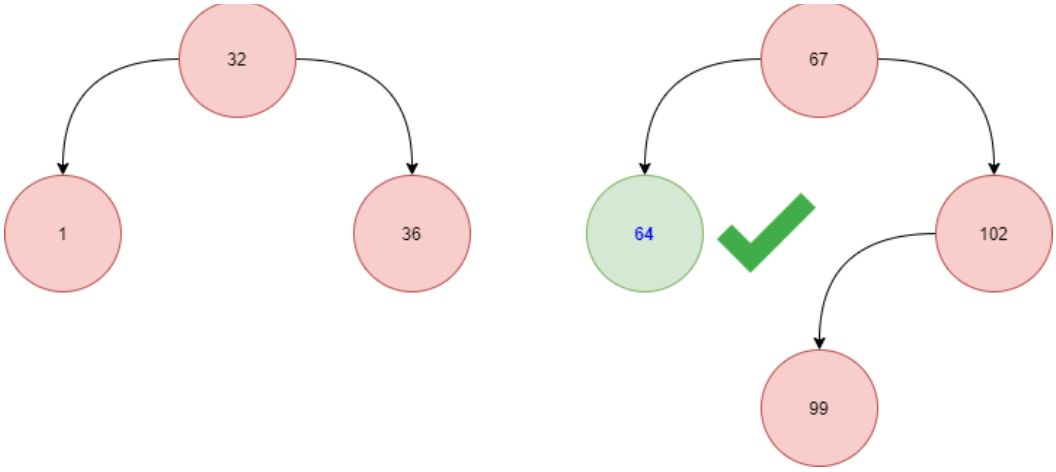
poder proyectar como deberia ser el arbol y cual deberia ser su estado a la hora de realizar diversas operaciones.

3. **Diagrama de busqueda en ABB**









En este ejemplo grafico de la operacion de busqueda se puede ver claramente como se descartan muchisimos nodos sin tener que necesariamente realizar una comparacion para ver si son el elemento buscado. Es por esto que el orden de complejidad se ve bastante reducido en comparacion a una lista no ordenada.