Bueno, supongo que Cris ya les dio la teoría necesaria de árboles y ahora nos vamos a meter un poco en lo que sería una implementación en código de un Arbol Binario de Busqueda (los ABB).

A alguno le quedo alguna duda sobre el tema? Vamos a repasar un poco lo que es un árbol binario. Es una estructura de datos que como su palabra lo indica se comporta parecido un árbol, por tener una raíz y hojas.

La raiz la podemos pensar el nodo superior del arbol, y las hojas son todos aquellos nodos que no tienen hijos. Lo que lo hace binario al arbol es que tiene como máximo 2 hijos, que vamos a referenciar como el hijo izquierdo y el hijo derecho. Un nodo que tiene al menos un hijo se convierte en padre de ese hijo.

El que se usa generalmente de los que vieron hoy es el Arbol de Binario de Busqueda (ABB), que no es mas es un árbol binario ordenado y el hecho de que este ordenado nos permite buscar facilmente el contenido de cualqueir nodo usando el algoritmo de búsqueda binaria que vieron esta semana, que se acuerdan que usa la estrategia “divide y venceras” para recorrer de a mitades.

EJEMPLO: El arbol que vamos a usar de ejemplo en nuestro código es el que surge de insertar estos numeros en este orden

Para crear un ABB en C++ vamos a necesitar tener entonces una clase que sea el arbol binario de busqueda y otra para el nodo especial que vamos a usar para arboles. Que a simple vista que atributos les parece que tiene que tener?

(Mostrar clase BSTNode)

Bien, tiene entonces un puntero a los nodos hijos izquierdo y derecho y le agregamos tambien un puntero al padre para poder rastrear al padre de cada nodo. Esto nos va a facilitar las cosas para cuando atravesemos el arbol.

Hay varias operaciones básicas que generalmente tiene un ABB, entre ellas vamos a ver:

* Insert (): Para agregar un nuevo nodo al Arbol actual. Si es la primera vez que agregamos un nodo, el nodo que insertamos será un nodo raíz.
* Print\_in\_order (): para imprimir todos los datos del arbol, ordenados de menor a mayor.
* Search(): para encontrar un dato determinado en . Si la clave existe, devuelve VERDADERO; de lo contrario, devuelve FALSO.
* FindMin () y FindMax () para encontrar el valor mínimo y el valor máximo que existen en el arbol.
* Successor () y Predecessor () se usan para encontrar el sucesor y predecesor de una clave determinada. Si no vieron lo que es con cris no se preocupen que vamos hablar sobre esto cuando lleguemos a esa porcion de codigo
* Y finalmente Eliminar () se utiliza para eliminar un dato determinado del arbol

**Insertar** una nueva clave en un ABB

Para insertar una clave en el ABB tenemos que agregar un nuevo nodo basandonos en el comportamiento del ABB. Cada vez que queremos insertar un dato, vamos a tener que compararlo con el nodo raíz y fijarnos si es más chico o más grande que la clave de la raíz. Obviamente que si no tenemos una raíz todavia, la clave insertada se convierte en una raíz.

Como se habran dado cuenta aca tengo metodos privados y metodos publicos. Lo hice de esta forma porque no me interesa que de afuera accedan a los metodos que puse como publicos. Todos estos métodos privados lo que hacen es pasarle como argumento la raiz actual del ABB. Lo que me permite esto es implementar los métodos de forma recursiva. Porque en primer lugar se van a llamar pasandole la raiz como en este de insertar, pero despues dentro de esa funcion lo puedo volver a llamar pasandole otro nodo distinto. Esto lo vamos a hacer para ir dividiendo al problema en dos como la busqueda binaria.

Por ejemplo, este metodo insertar, el que es publico que vamos a usar desde afuera de la clase lo que hace es ir actualizando la raiz del arbol. Fijense que llama a insert y le pasa como nodo de parametro la misma raiz. Entonces se llama a insert y lo primero que se fija es si el nodo esta apuntando a NULL. Como le pase la raiz entonces se esta fijando si tengo una raiz o no. Como al principio no tengo me crea un nuevo nodo y me lo devuelve.

Si vamos al constructor del nodo vemos que se crea con el dato que le pasamos y con su padre e hijos apuntando a NULL. Esta seria la situacion en la que tenemos el 23 solo.

Ahora si agregamos el 12 se vuelve a pasar de argumento el root pero ahora el root no esta vacio entonces vamos a ir a alguno de los otros condiconales. Aca se fija si el dato es mayor al dato de mi raiz, cosa que no. Entonces termino en el ultimo condicional que dice que el nodo que pase como parametro, le setea como hijo izquierdo lo que devuelve la llamada a insert… Y fijense que este node->get\_left esta vacio todavia porque no se lo inserte, entonces vuelvo como siempre pasa en una funcion recursiva al caso mas simple que es este que tenemos que el nodo es nulo. Entonces ahí me vuelve a crear el nodo y ese nodo se setea como left de la raiz. Y asi sucesivamente con todos los nodos que se van agregando de forma recursiva.

Basicamente la logica es que si el valor que agrego es menor a la raiz o al nodo que le pase como parametro me voy yendo a los subarboles izquierdos hasta encontrar el nodo hoja que no tiene hijo y que me va a devolver que su get\_left o get\_right es nulo. Lo mismo con los subarboles derechos.

Y que complejidad les parece que tiene este algoritmo?

es O (h), donde h es la altura del BST. Si tenemos un arbol muy degenerado o no balanceado que se parece a una lista vamos a tener una complejidad de O(n), con n la cantidad de nodos. Sino, si el arbol esta balanceado vamos a estar en aproximadamente O(log(n))

Mostrar couts para ver que se insertó bien todo

**print\_in\_order**

Ahora que ya tenemos el abb creado con todos los datos insertados podemos querer imprimir los datos en orden y para eso hacemos un recorrido en profundidad inorden. Esto va a atravesar el arbol desde el valor mas chico al valor mas grande. Para esto, vamos a posicionarnos en el nodo de mas a la izquierda que tiene el valor mas chico y terminar en el nodo de mas a la derecha que tiene el valor mas grande.

De la misma forma que antes tenemos dos metodos, uno que empieza llamando al otro desde la raiz y despues se vuelve a llamar recursivamente.

Basandonos en el arbol de ejemplo lo primero que se hace es imprimir todos los datos del subarbol izquierdo a la raiz. Fijense que esta linea se va llamando sucesivamente hasta llegar a la hoja de mas a la izquierda que no tiene hijos. Entonces cuando se llame a print\_in\_order una vez llegado al nodo 3 vemos que este no tiene un hijo a la izquierda entonces devuelve NULL. Y como dice aca, si es NULL no se hace nada. Entonces recien ahí pasa a la siguiente linea que dice imprimir lo que esta en ese nodo. Y despues llama al print\_in\_order para lo que este a la derecha de ese nodo. Que si no es hoja como en este caso va a pasarle el 7 y lo va a imprimir. Si era hoja va a seguir avanzando en todas las llamadas que se habian quedado trabadas en esta linea de get left que ya fueron resueltas. Es decir va a ir yendo hacia arriba en los padres hasta llegar a la raiz y pasar al subarbol derecho. Recuerden que el primer nodo que llamamos es el root y que le quedaron un monton de lineas pendientes por ejecutar que se habian ido por la rama de esta linea de get left.

La complejidad de PrintInOrder () va a ser O (N), con N es el número total de nodos tanto para el mejor como para el peor de los casos, porque siempre va a atravesar todos los nodos.

(mostrar cout)

**Search()**

Bueno ahora supongamos que queremos ver si un dato esta o no en el arbol. Esto va a ser bastante facil porque solo necesitamos comparar el dato que tenemos con el dato del nodo actual. Si el dato que tenemos es más chico que el dato del nodo actual, vamos al subárbol izquierdo, de lo contrario, vamos al subárbol derecho. Vamos a hacer esto hasta que encontremos la clave o no haya más nodos para buscar.

Igual que antes tenemos dos metodos, uno que invoca al otro pasandole la raiz y el dato y despues se fija si el nodo que devuelve no es NULL entonces se encontro el dato y devuelvo true.

Ahora suponganse que queremos ver si el numero 53 esta en el arbol. Como dijimos la primera vez se llama a esta funcion con la raiz y como la raiz no es nula ni contiene el dato que buscamos esto no pasa. Va a ir al segundo condicional porque 53 es mayor a 23 que es lo que contiene la raiz. Entonces pasamos al buscar en el subarbol derecho y le pasamos ahora a la funcion el nodo que contiene al 31. Como 53 es mayor a 31, vamos a volver a este condicional y se le va a pasar ahora el 88. Y como 53 es menor a 88 se va a llamar al ultimo condicional pasandole a search el nodo a la izquierda de 88, que es 53. Ahora hay una coincidencia en el primer condicional asi que me devuelve el dato, y como es un nodo no nulo el otro metodo me va a dar true.

Si hubiesemos buscado un numero que no esta en el arbol, dependiendo del valor del mismo vamos a ir a subarboles derechos o izquierdos pero finalmente vamos a llegar a un nodo hoja y vamos a querer buscar en los hijos de esa hoja, que son nulos. Entonces se va a pasar un nodo nulo al search y va a devolver NULL. Y como dice el otro metodo, si devuelve NULL el dato no esta en el arbol.

La complejidad es O (h), donde h es la altura del ABB. Si encontramos una clave que se encuentra en el nodo raíz, la complejidad va a ser O (1), que es el mejor de los casos. Si buscamos una clave en un árbol degenerado, la complejidad del tiempo va a ser O (N), donde N es el número total de nodos, que es el peor de los casos.

(mostrar cout)

**Find Min**

Despues podemos querer encontrar el valor minimo almacenado en el arbol, que tambien es bastante sencillo. Como mencione antes el valor mas chico va a estar a la izquierda de todo en el arbol.

Partimos de nuevo llamando a la raiz y vamos bajando a la izquierda. En nuestro ejemplo va a saltear al ultimo condicional y va a llamar al find\_min del nodo a la izquierda de la raiz, que es el 12. Despues va a llamar de nuevo a find\_min pasandole el nodo a la izquierda del 12, que es el 3. Y ahora como el 3 no tiene un hijo a la iquierda, voy a entrar en el segundo condicioanal en el que el hijo es NULL y voy a devolver el dato, es decir, 3, que es el valor minimo de mi arbol.

**Find Max**

Con el max la logica es la misma solo que el valor mas grande va a estar a la derecha de todo del arbol. Entonces se va a ir llamando esta linea que va llamando los nodos hacia la derecha hasta encontrar el nodo mas a la derecha que no tenga hijos y lo devuelve.

Similar al Search (), la complejidad de FindMin () y FindMax () es O (h), donde h es la altura del arbol. Sin embargo, si encontramos el valor de clave máximo en un Arbol degenerado hacia la izquierda, la complejidad va a ser erá O (1), que es el mejor de los casos, porque no tiene ningún subárbol derecho. Lo mismo pasa si buscamos el valor mínimo en un arbol degenerado hacia la derecha. El peor de los casos se va a dar si intentamos encontrar el valor mínimo en un arbol degenerado a la izquierda o el valor máximo en un arbol degenerado a la derecha, la complejidad temporal será O (N) con N la cantidad de nodos.

(mostrar cout)

**Succesor**

Otra de las propiedades que pueden encontrarse en un arbol binario con lo que se conoce como el sucesor y el predecesor. Para eso agegre los metodos Successor () y Predecessor () que ahora vamos a ver. Pero antes de ver el codigo vamos a ver que son estos sucesores y predecesores

Hay tres reglas para averiguar el sucesor de un valor de un ABB.

Supongamos que tenemos el valor de algun nodo que lo encontramos haciendo un search y queremos saber cual es el sucesor.

(ver ppt)

Basados en esta explicacion podemos crear el metodo para hallar el sucesor. Tenemos de nuevo dos funciones. En la funcion que es publica primero buscamos el nodo que contiene el dato del que queremos saber el sucesor. Despues nos fijamos que si ese dato no esta en el arbol o no se encontro un sucesor, se devuelve -1. Y sino se llama recursivamente al otro metodo privado de successor.

La primera vez que la llamemos le vamos a pasar el nodo del que queremos saber cual es el sucesor. Supongamos el caso 1 primero que queriamos saber el sucesor de 31. Vamos a llamar a succesor pasandole el nodo que contiene a 31.

Lo primero que se fija es si tiene subarbol derecho, que es nuestro caso. Y si lo tiene entonces devuelve el valor minimo del subarbol derecho, que es el 53. Entonces 53 es el sucesor.

Despues podemos considerar el segundo caso que teniamos en el que no tiene subarbol derecho. Queremos por ejemplo hallar el sucesor de 15. Lo que hace primero es crear dos nuevos nodos auxiliares. Uno donde vamos a guardar el sucesor que apunta a null y otro llamado ancestro apuntando inicialmente a la raiz que siempre es mayor a 15.

Despues iteramos mientras que el nodo sea distinto al ancestro que en un inicio es la raiz. Y nos fijamos, si lo que esta en el nodo, que es 15 es menor a lo que esta en la raiz que es 23 encontres mi sucesor va a ser ese ancestro por ahora. Y al ancestro ahora lo hago avanzar hacia la izquierda y pasa a ser el 12. Se sigue cumpliendo que 15 es distinto de 12 asi que sigo en el ciclo. Pero ahora mi dato, 15 no es menor a lo que apunta el ancentro (12) asi que me voy al else y cambio al ancestro por lo que esta a la derecha del mismo. Lo que esta a la derecha de 12 es 15, asi que ahora 15 es su propio ancestro y se sale del ciclo. Finalmente se devuelve el valor que esta dentro del sucesor que era 23.

Successor (), complejidad promedio de O (h), donde h es la altura del arbol. Sin embargo, si queremos encontrar el sucesor del valor máximo en un arbol degenerado hacia la derecha la complejidad va a pasar a ser O (N), que es el peor de los casos.

**Predecesor**

La implementacion es analoga a la del sucesor solo que en vez de fijarnos si tiene subarbol derecho nos fijamos que tenga subarbol izquierdo y en ese caso encontramos el maximo. Y sino tiene subarbol izquierdo iteramos para encontrar el primer ancestro que sea mas chico al valor actual.

La complejidad es la misma que el de sucessor peroa hora el peor caso va a estar para cuando queramos encontrar el sucesor del valor minimo en un arbol degenerado a la izquierda.

**Remove**

La última operación que vamos a ver es eliminar un nodo basado en un dato dado. Hay tres casos posibles para eliminar un nodo de un arbol:

1. Eliminar una hoja, es decir un nodo que no tiene ningún hijo. En este caso, solo necesitamos eliminar el nodo. En nuestro ejemplo, podemos eliminar los valores 7, 15, 29 y 53.

2. Eliminar un nodo que solo tiene un hijo. En este caso, tenemos que conectar al hijo con el padre del nodo a eliminar. Después de eso, podemos eliminar el nodo que queriamos de forma segura. Por ejemplo, si queremos eliminar el nodo 3, tenemos que apuntar el puntero padre del nodo 7 al nodo 12 y hacer que el nodo izquierdo de 12 apunte a 7. Despues eliminar facilmente el nodo 3.

3. Eliminar un nodo que tiene dos hijos. En este caso, tenemos que averiguar el sucesor (o predecesor) del nodo. Después de eso, podemos reemplazar el nodo de destino con el nodo sucesor (o predecesor). Supongamos que queremos eliminar el nodo 31, y que el sucesor tome su lugar. Entonces podemos eliminar el nodo 31 y reemplazarlo con el nodo 53 que es su sucesor. Ahora, el nodo 53 va a tener dos hijos, el nodo 29 a la izquierda y el nodo 88 a la derecha.

Las primeras lineas son para cuando el nodo no fue encontrado. El primer if se va a llegar recien cuando encontremos el nodo que estamos queriendo eliminar, sino lo encontramos vamos a llamar recursivamente a la funcion para que lo busque en el subarbol izquierdo o derecho según el valor que tenga. Cuando lo encuentre entramos en este if.

Veamos los 3 casos que mencionamos recien, en el codigo. Supongamos que queremos eliminar una hoja.