Árboles de búsqueda binaria

Carlos E. Alvarez[1].

[1]Dep. de Matemáticas aplicadas y Ciencias de la Computación, Universidad del Rosario

2019-II

Árboles de búsqueda binaria

¿Por qué usar árboles?

- La búsqueda lineal de un elemento en un arreglo desordenado o una lista enlazada es en el peor caso $\Theta(N)$
- En un arreglo ordenado puede usarse búsqueda binaria, que es en el peor caso $O(\lg N)$ -en donde $\lg N$ significa $\log_2 N$-.
- Dada una ubicación en la estructura, la inserción o remoción de un elemento en un arreglo es en el peor caso $\Theta(N)$, mientras que en una lista enlazada es $\Theta(1)$
- Por lo tanto, la ubicación + inserción/remoción de un elemento es en el peor caso del orden $O(\lg N) + \Theta(N) = \Theta(N)$ en un arreglo ordenado y $\Theta(N) + \Theta(1) = \Theta(N)$ en una lista enlazada

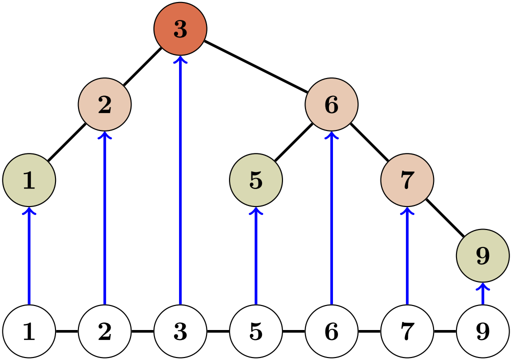
Árboles de búsqueda binaria

¿Por qué usar árboles?

Un árbol de búsqueda binaria es una estructura en donde se puede lograr que el proceso de búsqueda + inserción/remoción, que en este caso van unidos, sea en el peor caso del orden $O(\lg N)$.

Árboles de búsqueda binaria

¿Por qué usar árboles?

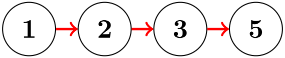


De lista enlazada a árbol binario

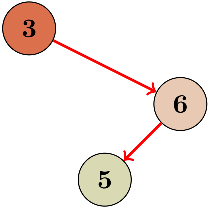
Árboles de búsqueda binaria

¿Por qué usar árboles?

Buscando el 5 en la lista enlazada (búsqueda lineal):



Buscando el 5 en el árbol (de arriba a abajo):



Árboles de búsqueda binaria

ADT:

Estructura de nodos en donde todo nodo X, excepto la raíz, tiene un nodo padre y puede tener hasta 2 nodos hijos denominados *izquierdo* y *derecho*. Los nodos están ordenados de acuerdo a su clave $X.key$, de manera que para cada X los valores de $.key$ en su sub-árbol izquierdo son $< X.key$ y los de su sub-árbol derecho son $> X.key$.

Árboles de búsqueda binaria

ADT (operaciones):

- Contains(S, x): Retorna true si el elemento x se encuentra en el S o `false` en caso contrario
- FindMin/FindMax(S): Retorna el elemento mínimo/máximo de S
- Insert(S, x): Inserta el elemento x en S, preservando las propiedades de S
- Remove(S, x): Remueve el elemento x de S, preservando las propiedades de S

Árboles de búsqueda binaria

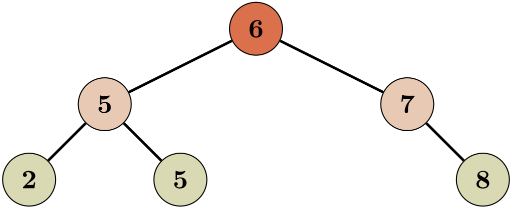
- Los árboles son apuntadores a nodos
- Los nodos son estructuras que contienen árboles

Nodo de árbol de búsqueda binaria

```
template <typename T>
struct BSTNode {
  T key;
  BSTNode *left;
  BSTNode *right;
};
```

Árboles de búsqueda binaria

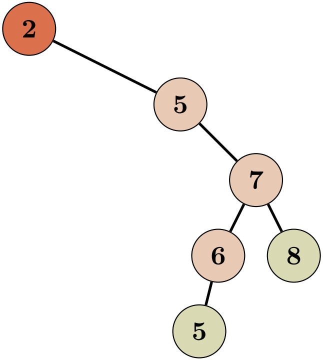
Propiedad de árbol de búsqueda binaria



Ejemplo

Árboles de búsqueda binaria

Propiedad de árbol de búsqueda binaria



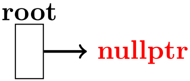
Ejemplo

Creando el árbol

Estructura recursiva:

- El árbol(`BSTNode*`) as un apuntador a un nodo
- Un nodo(`BSTNode`) es una estructura que contiene árboles (`left` y `right`)

Al crear un árbol, este se encuentra vacío.



Creando el árbol

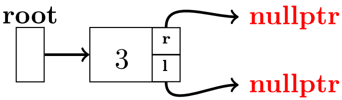
```
template <typename T>
class BST {
private:
  BSTNode<T> *root;

public:
  BST() { root = nullptr; }
};
```

Creando el árbol

Insertando un nodo (3):

```
insert(root=null,3)
```

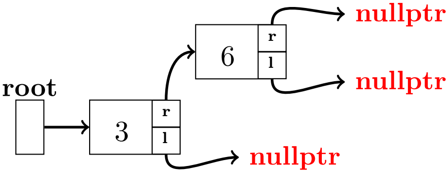


Primer nodo.

Creando el árbol

Insertando un nodo (6>3):

```
insert(root->3,6) => insert(p3r=null,6)
```

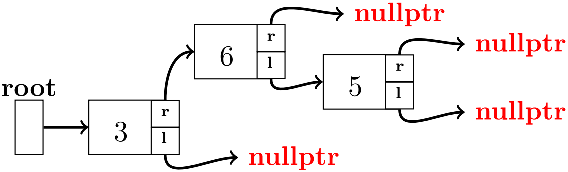


Segundo nodo.

Creando el árbol

Insertando un nodo (6>5>3):

```
insert(root->3,5) => insert(p3r->6,5) => insert(p6l=null,5)
```

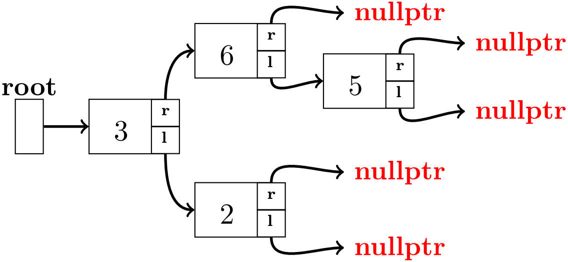


Tercer nodo.

Creando el árbol

Insertando un nodo ($2<3$):

```
insert(root->3,2) => insert(p3l=null,2)
```



Cuarto nodo.

Creando el árbol

```
template <typename T>
void BST<T>::insertNode(BSTNode<T>* &t, T k) {
  if(t == nullptr){
    t = new BSTNode<T>;
    t->key = k;
    t->left = t->right = nullptr;
  }else{
    if(k != t->key){
      if(k < t->key){
        insertNode(t->left, k);
      }else{
        insertNode(t->right, k);
      }
    }
  }
}
```

Destruyendo el árbol

```
template <typename T>
void BST<T>::destroyRecursive(BSTNode<T> *t) {
  if(t != nullptr){
    destroyRecursive(t->left);
    destroyRecursive(t->right);
    delete t;
  }
}
```

Recorriendo el árbol

Quisiéramos recorrer todos los nodos en orden, realizando una operación sobre cada uno de ellos.

Algoritmo Inorder-tree-walk: Algoritmo recursivo que recorre todos los nodos.

```
Inorder-tree-walk(x)
  if x ≠ NULL
    Inorder-tree-walk(x.left)
    perform operations
    Inorder-tree-walk(x.right)
```

Recorriendo el árbol

Imprimiendo todos los nodos:

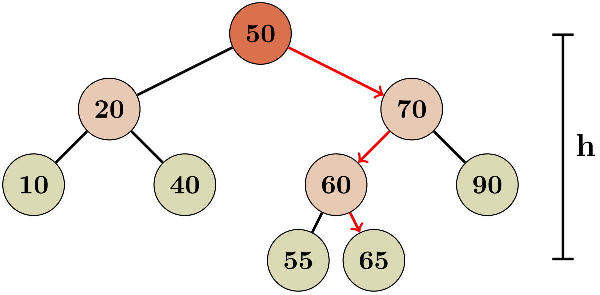
```
template <typename T>
void BST<T>::displayNode(BSTNode<T> *t, int count) {
  if(t != nullptr){
    count++;
    displayNode(t->left, count);
    cout << "(" << count-1 << ")" << t->key << " ";
    displayNode(t->right, count);
  }
}
```

Ejercicios:

1. Implemente la clase `BST`, con los métodos constructor, destructor, `insert` (recibe la clave e inserta un nodo) y `display` (imprime el árbol a manera de lista ordenada)
2. Escriba un programa que cree un árbol de tipo `string` e inserte los nombres de los siete enanos: Grumpy, Doc, Sleepy, Bashful, Dopey, Happy y Sneezy. Imprima el árbol
3. Repita el programa insertando los enanos en orden distinto ¿Que cambia?

Búsqueda

Busca el 65



Búsqueda

```
template <typename T>
BSTNode<T>* BST<T>::findNode(BSTNode<T> *t, T k) {
  if(t == nullptr) return nullptr;
  if(k == t->key) return t;
  if(k < t->key){
    return findNode(t->left, k);
  }else{
    return findNode(t->right, k);
  }
}
```

Mínimo/máximo

```
template <typename T>
BSTNode<T>* BST<T>::minimum(BSTNode<T> *t) {
  while(t->left != nullptr)
    t = t->left;
  return t;
}

template <typename T>
BSTNode<T>* BST<T>::maximum(BSTNode<T> *t) {
  while(t->right != nullptr)
    t = t->right;
  return t;
}
```

Predecesor/sucesor

- Predecesor de x: Es el nodo con la mayor clave menor que x.key
- Sucesor de x: Es el nodo con la menor clave mayor que x.key

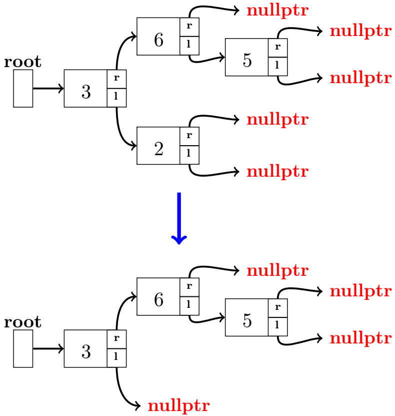
Taller:

- Añada los métodos `findNode`, `minimum` y `maximum` a su clase
- Añada el atributo `parent` a la estructura `BSTNode`. Modifique el método `insertNode` para que actualice correctamente este atributo
- Válgase de los métodos `maximum`, `minimum` y el nuevo atributo `parent`, para crear los métodos `predecessor` y `successor`

Pruebe que todo funcione correctamente.

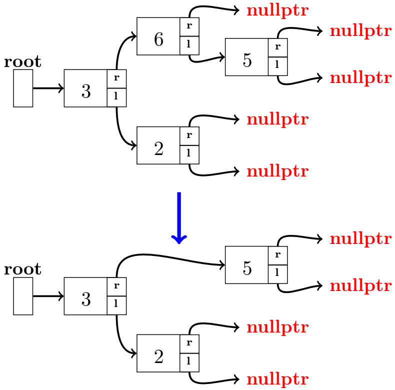
Removiendo un nodo

Nodo no tiene hijos (remueve 2):



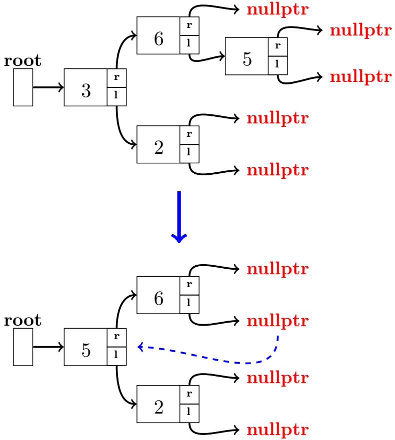
Removiendo un nodo

Nodo tiene un hijo (remueve 6):



Removiendo un nodo

Nodo tiene dos hijos (remueve 3):



Removiendo un nodo

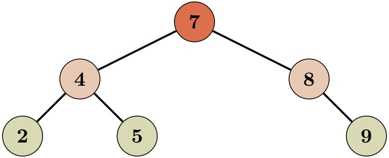
Taller:

- Teniendo en cuenta los tres casos posibles, implemente la remoción de un nodo creando el método `removeNode(BSTNode<T>* t, T k)`, en donde `k` es la clave y `t` es el árbol (apuntador a nodo)

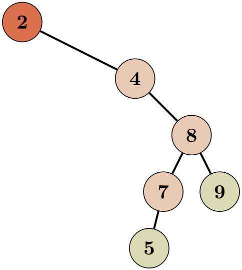
Árboles balanceados

las alturas h_r y h_l de los sub-árboles derecho e izquierdo de cualquier sub-árbol no deben diferir por más de 1:

$$|h_r - h_l| \leq 1$$



Balanceado



Desbalanceado

Altura de un árbol balanceado $\rightarrow O(\log_2 N)$.

Árboles balanceados

Árboles AVL: A cada nodo se le asigna un factor de balanceo bf

$$bf = h_l - h_r,$$

en donde h_l es la altura del sub-árbol izquierdo del nodo y h_r la del sub-árbol derecho.

En un árbol AVL tenemos que $-1 \leq bf \leq 1$ para todos los nodos.

- Añada el atributo bf a la estructura `BSTNode`

Árboles balanceados

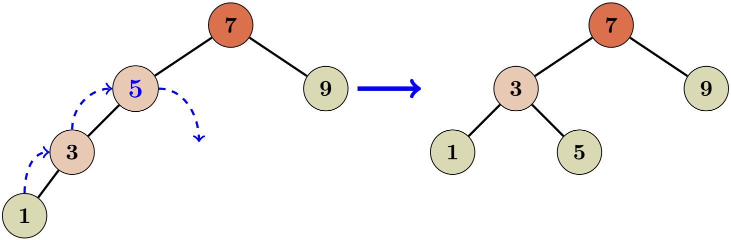
Método que asigna los parámetros `bf` en un sub-árbol con raíz en el nodo `t`:

```
template <typename T>
int BST<T>::balFactOfNode(BSTNode<T>* &t) {
  if(t->left == nullptr && t->right == nullptr){ //no childre
    t->bf = 0; //balancing factor
    return 0; //h - depth of the node
  }else{
    int lf=-1, rf=-1;
    if(t->left != nullptr)
      lf = balFactOfNode(t->left);
    if(t->right != nullptr)
      rf = balFactOfNode(t->right);

    t->bf = rf-lf; //balancing factor
    return max(lf,rf)+1; //h - depth of the node
  }
}
```

Árboles balanceados

Para el nodo 5 tenemos $b = -2$.

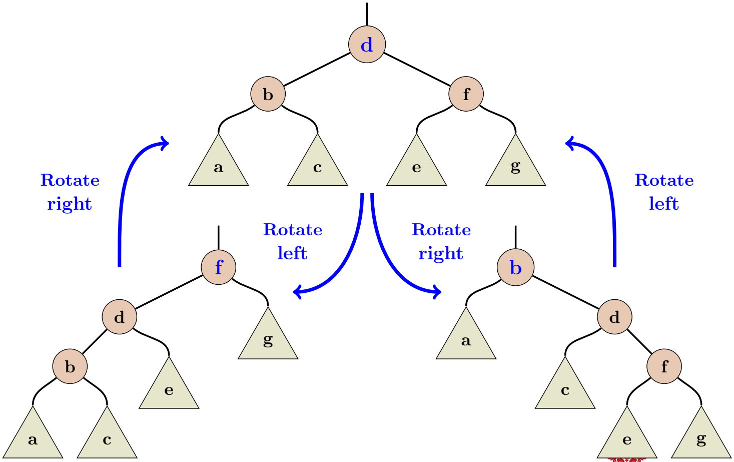


Rotación simple a la derecha sobre el nodo 5

Si $b < -1$ rotamos a la derecha. Si $b > 1$ rotamos a la izquierda.

Árboles balanceados

Rotaciones simples: Caso general



Universidad del Rosario | MACC Matemáticas Aplicadas y Ciencias de la Computación

Árboles balanceados

Taller:

- Teniendo en cuenta que nuestros nodos están implementados como estructuras `BSTNode`, escriba paso a paso lo que debe hacerse para implementar una rotación a la izquierda

Árboles balanceados

```
template <typename T>
void BST<T>::rotateLeft(BSTNode<T>* &t) {
  BSTNode<T>* child = t->right;
  t->right = child->left;
  if(t->right != nullptr)
    t->right->parent = t;
  child->left = t;
  child->parent = t->parent;
  t->parent = child;

  BSTNode<T>* p = child->parent;
  if(p != nullptr){
    if(p->key > child->key)
      p->left = child;
    else
      p->right = child;
  }else
    root = child;
}
```

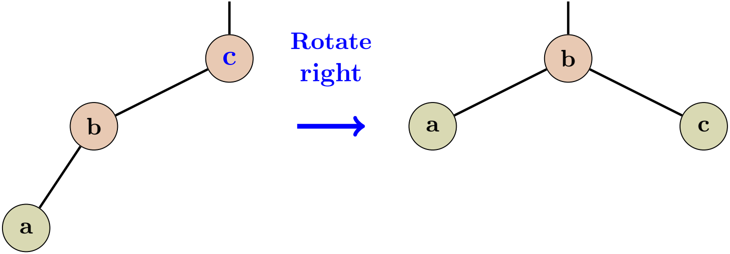
Árboles balanceados

```
template <typename T>
void BST<T>::rotateRight(BSTNode<T>* &t) {
  BSTNode<T>* child = t->left;
  t->left = child->right;
  if(t->left != nullptr)
    t->left->parent = t;
  child->right = t;
  child->parent = t->parent;
  t->parent = child;

  BSTNode<T>* p = child->parent;
  if(p != nullptr){
    if(p->key > child->key)
      p->left = child;
    else
      p->right = child;
  }else
    root = child;
}
```

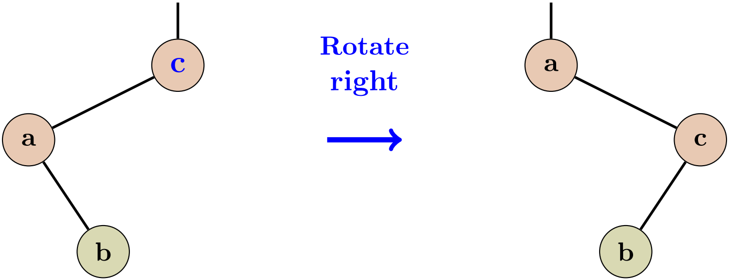
Árboles balanceados

Caso 1: Rotación simple reduce la altura del árbol

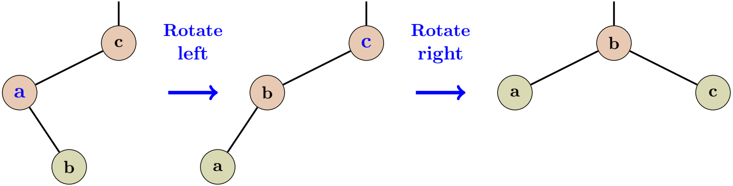


Árboles balanceados

Caso 2: Rotación simple NO reduce la altura del árbol



Solución caso 2:



Para eliminar el desbalance en c, cuyo hijo izquierdo/derecho tiene a su vez un único hijo del lado derecho/izquierdo, hay que hacer una rotación doble, primero sobre el hijo de c a la izquierda/derecha y luego sobre c a la derecha/izquierda.

Árboles balanceados

Taller:

- Implemente la rotación doble a la derecha (como la que se muestra en la diapositiva anterior) como `dRotateRight(BSTNode<T>* t)`
- Implemente la rotación doble a la izquierda (el opuesto de la anterior) como `dRotateLeft(BSTNode<T>* t)`
- Pruebe que los métodos funcionan correctamente

Árboles balanceados

Para mantener un árbol AVL al insertar/remover elementos, debemos en ocasiones rebalancear los nodos.

Si al insertar/remover un nodo se produce un desbalance, debemos detectar el nodo `t` en el que se presenta éste y rotar el sub-árbol con `t` como pivote, de manera a seguir cumpliendo con la propiedad AVL.

Árboles balanceados

```
template <typename T>
void BST<T>::fixImbalance(BSTNode<T>* &t) {
  bool btest = false;
  while(t != nullptr){ //search backwards for the first inbal
    if(fabs(t->bf) > 1){
      cout << t->key << " is imbalanced ["
           << t->bf << "], and my ";
      if(t->bf < 0){
        cout << "left child's bf is ["
             << t->left->bf;
        if(t->left->bf < 0){
          cout << "<0]\n Performing a "
               << "single rotation to the right.\n";
          rotateRight(t);
        }else{
          cout << ">=0]\n Performing a "
               << "double rotation to the right.\n";
          dRotateRight(t);
        }
```

Árboles balanceados

```
    }else{
      cout << "right child's bf is ["
           << t->right->bf;
      if(t->right->bf < 0){
        cout << "<0]\n Performing a "
             << "double rotation to the left.\n";
        dRotateLeft(t);
      }else{
        cout << ">=0]\n Performing a "
             << " single rotation to the left.\n";
        rotateLeft(t);
      }
    }
    btest = true;
  }
```

Árboles balanceados

```
    t = t->parent;
    if(t != nullptr){
      //set balancing factors for the sub-tree
      balFactOfNode(t);
    }

    if(btest) break;
  }
}
```

Árboles balanceados

Taller:

- Implemente el método `AVLinsert(T k)` que inserte un nodo y luego busque y arregle el desbalance que la inserción pudo generar
- Implemente el método `AVLremove(T k)` que remueva un nodo, actualice los `bf` en el árbol, y luego busque y arregle el desbalance que la remoción pudo generar

Cuide que ninguna de las operaciones que realiza sea de orden mayor a $O(\lg N)$ en el peor caso.