Árboles de Búsqueda Binaria

Agustín J. González

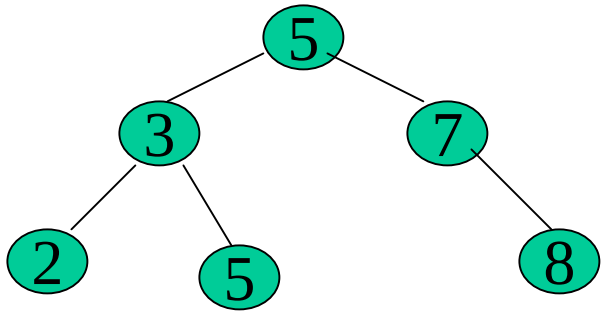
ELO-320: Estructura de Datos y Algoritmos

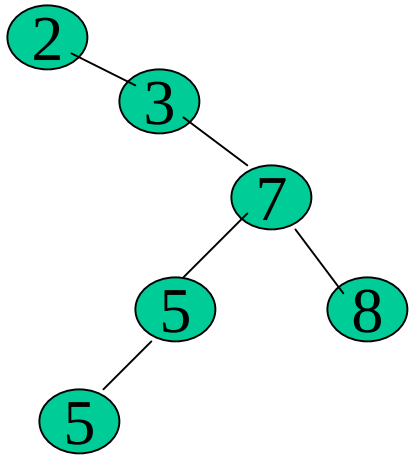
Introducción

- Los árboles de búsqueda son estructuras de datos que soportan las siguientes operaciones de conjuntos dinámicos:
  Search (Búsqueda), Minimum (Mínimo), Maximum (Máximo), Predecessor (Predecesor), Successor (Sicesor), Insert (Inserción), y Delete (eliminación).
- Los árboles de búsqueda se pueden utilizar así como diccionarios y como colas de prioridad.
- Estas operaciones toman tiempo proporcional a la altura del árbol.
- Para un árbol completo binario, esto es $\Theta(\lg n)$ en el pero caso; sin embargo, si el árbol es una cadena lineal de n nodos, las mismas operaciones toman $\Theta(n)$ en el pero caso.
- Para árboles creados aleatoriamente, la altura es $O(\lg n)$, con lo cual los tiempos son $\Theta(\lg n)$.
- Hay varios esquemas para mejorar el peor caso de los árboles de búsqueda. Dos de ellos son los árboles 2-3 y los árboles rojo-negro.

Propiedad de un árbol búsqueda binaria

- Sea x un nodo en un árbol de búsqueda binaria. Si y es un nodo del sub-árbol izquierdo de x, entonces la clave de y ≤ clave de x. Si y es un nodo del sub-árbol dercho de x, entonces la clave de x ≤ clave de y.
- Por ejemplo, dos árboles de búsqueda binaria son:





- La propiedad del árbol de búsqueda nos permite imprimir o recorrer sus nodos en el orden de sus claves haciendo uso de un simple algoritmo recursivo.

Recorrido inorder de un árbol búsqueda binaria

- Suponiendo una estructura como la vista antes para árboles binarios, tenemos:
- typedef struct arbol_tag {
  struct arbol_tag p;
  struct arbol_tag left;
  struct arbol_tab right;
  elementType element;
  } TREE_NODE;
- **void Inorder_Tree_Walk(TREE_NODE * x) {**
  if (x != NULL) {
  Inorder_Tree_Walk(x->left);
  Print(x->element); /* podría ser procesar elemento*/
  Inorder_Tree_Walk(x->right);
  }
  }
- Este algoritmo toma tiempo $\Theta(n)$ porque el procedimiento es llamado exactamente dos veces por cada nodo.
- Análogamente se definen recorridos preorder y postorder del árbol. El único cambio es el lugar de la instrucción de procesamiento del nodo. En preorder, el nodo se procesa primero y en postorder se procesa después.

Recorrido Pre y port-order de un árbol búsqueda binaria

- **void Preorder_Tree_Walk(TREE_NODE * x) {**
  if (x != NULL) {
  Print(x->element); /* podría ser procesar elemento*/
  Preorder_Tree_Walk(x->left);
  Preorder_Tree_Walk(x->right);
  }
  }

- **void Postorder_Tree_Walk(TREE_NODE * x) {**
  if (x != NULL) {
  Postorder_Tree_Walk(x->left);
  Postorder_Tree_Walk(x->right);
  Print(x->element); /* podría ser procesar elemento*/
  }
  }

- **El recorrido de los árboles previos daría:**
- **Preorder: 5, 3, 2, 5, 7, 8**
- **Inorder: 2, 3, 5, 5, 7, 8**
- **Postorder: 2, 5, 3, 8, 7, 5**

Otras operaciones en un árbol de búsqueda binaria

- **Búsqueda de una clave determinada:**

```
TREE_NODE * Tree_Search( TREE_NODE * x, elementType k) {
        if (x == NULL)  return x;
        else if (x->element == k )  /* Ojo esta comparación podría ser una función*/
                return x;
        else  if (k < x->element)
                 return Tree_Search( x->left, k);
        else
                 return Tree_Search( x->right, k);
}
```

- **El tiempo de este algoritmo es O(h) donde h es la altura del árbol.**
- **Este procedimiento se puede "desenrollar" para eliminar el tiempo de múltiples llamados a la misma función.**
-

```
TREE_NODE * Tree_Search( TREE_NODE * x, elementType k) {
        while(x != NULL)
                if (x->element == k )
                        return x;
                else  if (k < x->element)
                        x = x->left;
                else
                        x= x->right;
        return x;
}
```

Máximo y Mínimo en un árbol de búsqueda binaria

- ```
  TREE_NODE * Tree_Maximum( TREE_NODE * x) {
        if (x == NULL)  return x;
        while (x->right != NULL )
                x = x->right;
        return x;
  }
  ```

- ```
  TREE_NODE * Tree_Minimum( TREE_NODE * x) {
        if (x == NULL)  return x;
        while (x->left != NULL )
                x = x->left;
        return x;
  }
  ```

Sucesor y Antecesor en un árbol de búsqueda binaria

- TREE_NODE * Tree_Successor(TREE_NODE * x) {
  TREE_NODE * y;
  ```
  if (x == NULL)  return x;
  if (x->right != NULL)
          return Tree_Minimum(x->right);
  y = x->p;
  while ( y != NULL )
          if (x == y->right) {
                  x = y;
                  y = y->p;
          }
          else break;
  return y;
  ```
  }
- TREE_NODE * Tree_Predecessor(TREE_NODE * x) {
  TREE_NODE * y;
  ```
  if (x == NULL)  return x;
  if (x->left != NULL)
          return Tree_Maximum(x->left);
  y = x->p;
  while ( y != NULL )
          if (x == y->left) {
                  x = y;
                  y = y->p;
          }
          else break;
  return y;
  ```
  }

Inserción en un árbol de búsqueda binaria

- Suponemos inicialmente que z->left = z->right = NULL.
- Void Tree_Insert(TREE_NODE ** T, TREE_NODE * z) {

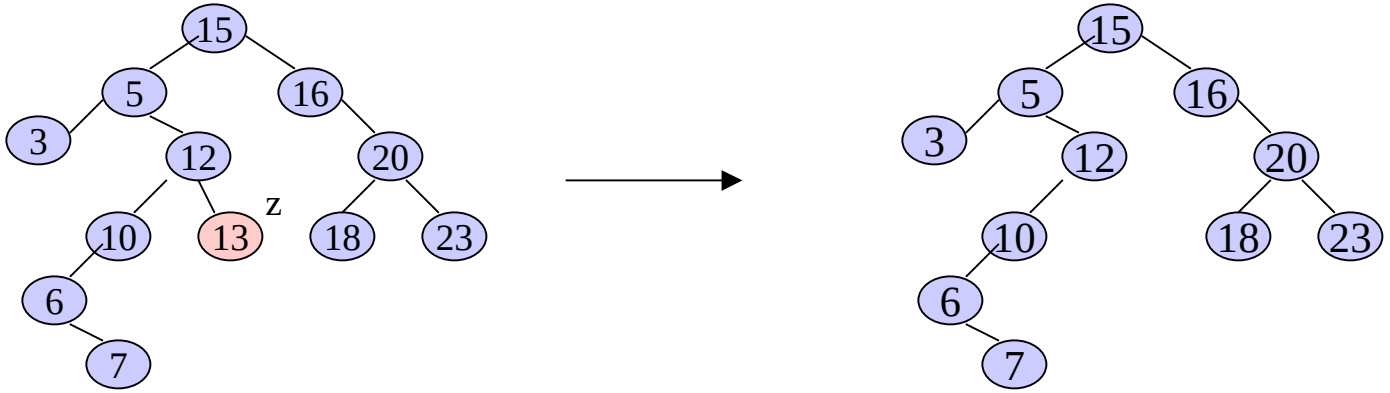
```
        TREE_NODE *y, *x;
        y=NULL;
        *x = *T;
        while (x != NULL) { /* buscamos quien debe ser su padre */
                y = x;
                if ( z->element < x->element)
                        x = x->left;
                else
                        x= x->right;
        }
        z->p = y;
        if (y == NULL)    /* se trata del primer nodo */
                *T = z;
        else if (z-element < y->element)
                y->left = z;
        else
                y->right = z;
}
```

- Como el procedimiento de búsqueda este algoritmo tiene toma un tiempo O(h), h es la altura del árbol.

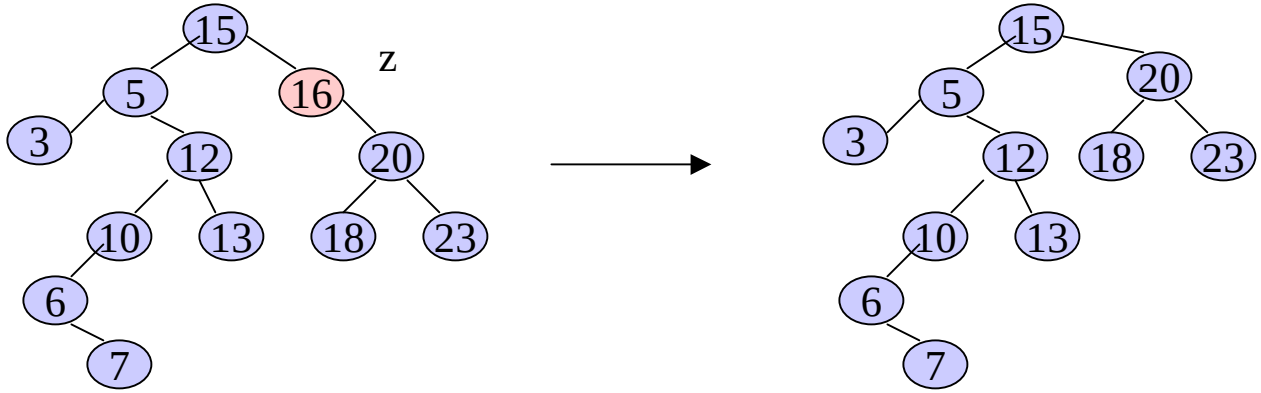
Eliminación en un árbol de búsqueda binaria

- Como entrada disponemos de z, un puntero al nodo a remover.
- Hay tres casos a considerar:
  1.- Que *z sea un nodo hoja. En este caso se elimina fácilmente.
  2.- Que *z sea un nodo sin hijo izquierdo o derecho. En este caso su único sub-árbol sube para toma el lugar de *z.
  3.-*z posee dos sub-árboles. En este caso, su sucesor no posee hijo izquierdo, luego éste puede ser movido desde su posición a la de *z.

1)

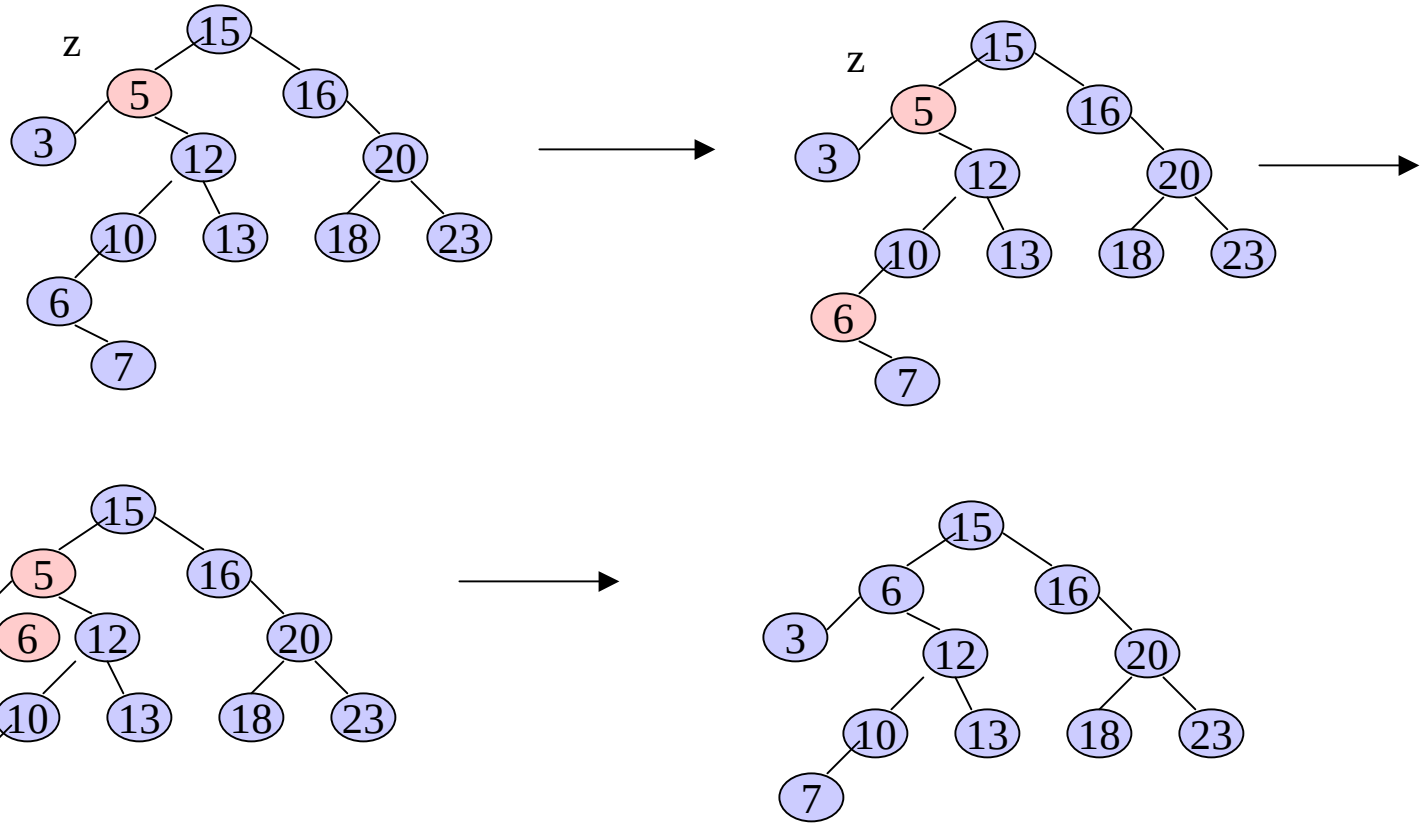


2)



Eliminación tercer caso

- Caso 3)



Eliminación: Algoritmo

```
• TREE_NODE * Tree-Delete(TREE_NODE **T, TREE_NODE * z) {
        TREE:_NODE * x;
        if (z->left == NULL || z->right == NULL)
                y = z;     /* caso 1 y 2 */
        else
                y = Tree_Successor(z);  /* caso 3 */
  /* hasta aquí y es un nodo con menos de dos hijos y debe ser extraído del árbol*/
        if (y->left != NULL)
                x = y->left;
        else
                x = y->right;
        if (x != NULL)
                x->p = y->p;
        if (y->p == NULL)   /* estoy eliminado el último nodo */
                *T = x;
        else if (y == y->p->left) /* y es  hijo izquierdo */
                y->p->left = x;
        else
                y->p->right = x;
        if (y !=z)  /* caso 3 */
                z->element = y->element;
        return y;
  }
```