Extensión de un Estructura de Datos

Agustín J. González

ELO-320: Estructura de Datos y Algoritmos

Introducción

- Hay algunas situaciones de ingeniería que no requieren más que las estructuras que hemos visto; sin embargo, hay muchas otras que requieren un poco más de creatividad.
- Sólo en casos muy raros la solución irá por una estructura totalmente nueva. La gran mayoría de los casos es suficiente extender levemente una estructura ya conocida.
- Extender una estructura de datos no es directo. Se requiere:
  - Escoger la estructura de datos base
  - Determinar la información adicional a incorporar
  - Verificar que la información adicional puede ser mantenida por las operaciones de la estructura básica.
  - Desarrollar las nuevas operaciones.
- Veremos la extensión de los árboles de búsqueda binaria para obtener las estadísticas de orden (Cuál es el i-ésimo?) y para determinar el orden que le corresponde a un elemento (Qué orden tiene x?).

Estadísticas de orden en árboles de busqueda binaria

- Ya vimos una solución que toma tiempo O(n). Ahora veremos otra con tiempo O(lg n).
- La idea es agregar un campo a cada nodo. Éste contiene el tamaño del árbol incluyendose.
- La estructura es ahora:
- typedef struct arbol_tag {
  struct arbol_tag p;
  struct arbol_tag left;
  struct arbol_tab right;
  int size;
  elementType element;
  } TREE_NODE;

Obtención de un elemento dada una posición de orden

```
TREE_NODE * OS_Select(TREE_NODE * x, int i) {
        int r;
        if (x == NULL) return NULL;
        if (x->left == NULL)
                r = 1;
        else
                r = x->left->size +1;
/* r contiene el número de nodos del sub-árbol izquierdo más el nodo central*/
        if (i == r)
                return x;
        else if (i < r)
                return OS_Select(x->left, i);
        else
                return OS_Select(x->right, i-r);
}
```

Obtención de la posición de orden dado un elemento

- int OS_Rank(TREE_NODE * T, TREE_NODE * x) {

```
        int r;
        TREE_NODE *y;
        if (x->left == NULL)
                r = 1;
        else
                r = x->left->size + 1;
        y = x;
        while (y != T) {
                if ( y == y -> p->right )
                /*      r = r+y->p->left->size +1;  por tarea de Sebastián Álvarez*/
                        r = r+y->p->size - y->size;
                y = y->p;
        }
        return r;
}
```

- El próximo paso es estudiar cómo debemos modificar las operaciones de inserción y eliminación para mantener la información de tamaño cuando el conjunto dinámico varía.

Modificación del Algoritmo de ***Inserción*** para mantener información sobre el tamaño

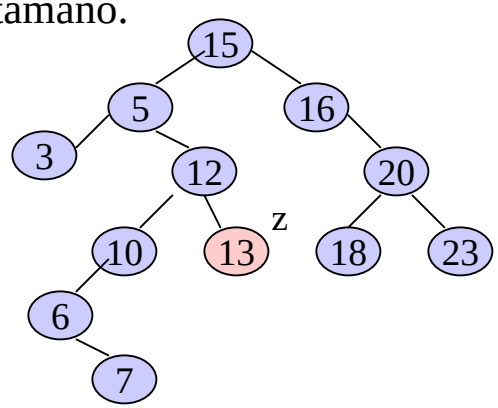
- Suponemos inicialmente que z->left = z->right = NULL.
- Void Tree_Insert(TREE_NODE ** pT, TREE_NODE * z) {

```
        TREE_NODE *y, *x;
        y=NULL;
        x = *pT;
        while (x != NULL) { /* buscamos quien debe ser su padre */
                y = x;
                x->size++;
                if ( z->element < x->element)
                        x = x->left;
                else
                        x= x->right;
        }
        z->size=1;
        z->p = y;
        if (y == NULL)    /* se trata del primer nodo */
                *pT = z;
        else if (z-element < y->element)
                y->left = z;
        else
                y->right = z;
}
```

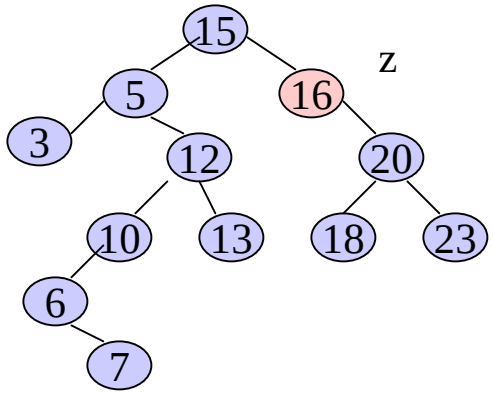
Modificación del Algoritmo de ***Eliminación*** para mantener información sobre el tamaño

- Consideremos cuales son los casos:
- Caso 1: debemos actualizar los nodos desde z hasta la raíz.
- Caso 2: idem caso 1.
- Caso 3: idem caso 1 pero desde nodo ***y*** hasta la raíz.
- Conclusión: en todos los casos recorrer desde ***y*** hasta la raíz disminuyendo en uno el tamaño.

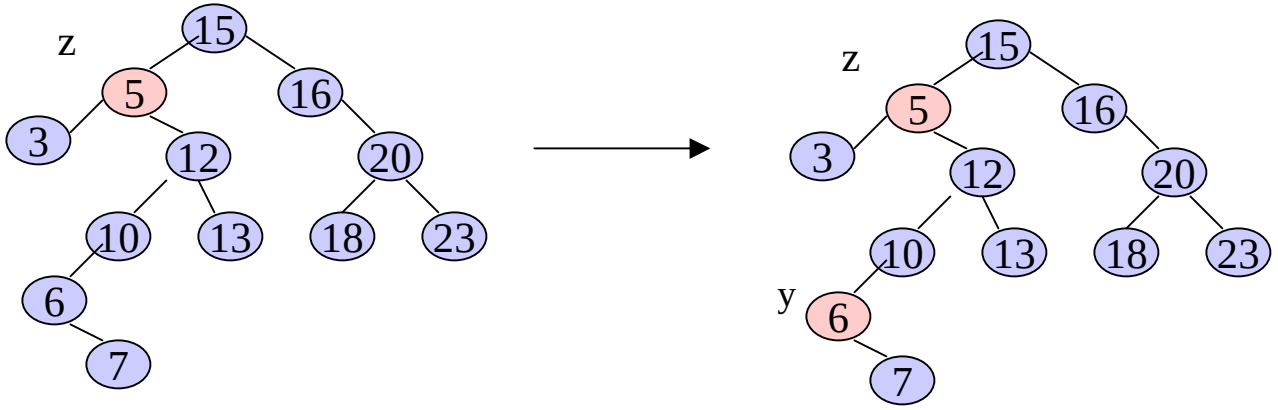
1)



2)



3)



Modificación del Algoritmo de ***Eliminación*** para mantener información sobre el tamaño

- TREE_NODE * Tree-Delete(TREE_NODE **pT, TREE_NODE * z) {

```
        TREE_NODE * x, *y;
        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 =z NULL)   /* estoy eliminado el último nodo */
                *pT = 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;
        x = y->p;
        while (x !=NULL) {
                x->size--;
                x = x->p;.
        }
        return y;
}
```