Práctica 2

Estructura de Datos

J. P. Abarca[1] C. T. Apari[1] C. A. Suca[1] A. Vargas[1]

[1]Universidad Nacionan del San Agustin. Facultad de Producción y Servicios.
Escuela Profesional de Ciencias de la Computación
Maestría en Ciencias de la Computación
Docente: Mg. Vicente Machaca

03 de Julio del 2021

Contenido

1. Arbol AVL
   - Rotación
   - Inserción
   - Eliminación
   - Búsqueda
2. Arbol B
   - Búsqueda
   - Inserción
   - Eliminación
3. Referencias

Contenido

1. Arbol AVL
   - Rotación
   - Inserción
   - Eliminación
   - Búsqueda
2. Arbol B
   - Búsqueda
   - Inserción
   - Eliminación
3. Referencias

Reglas de rotación

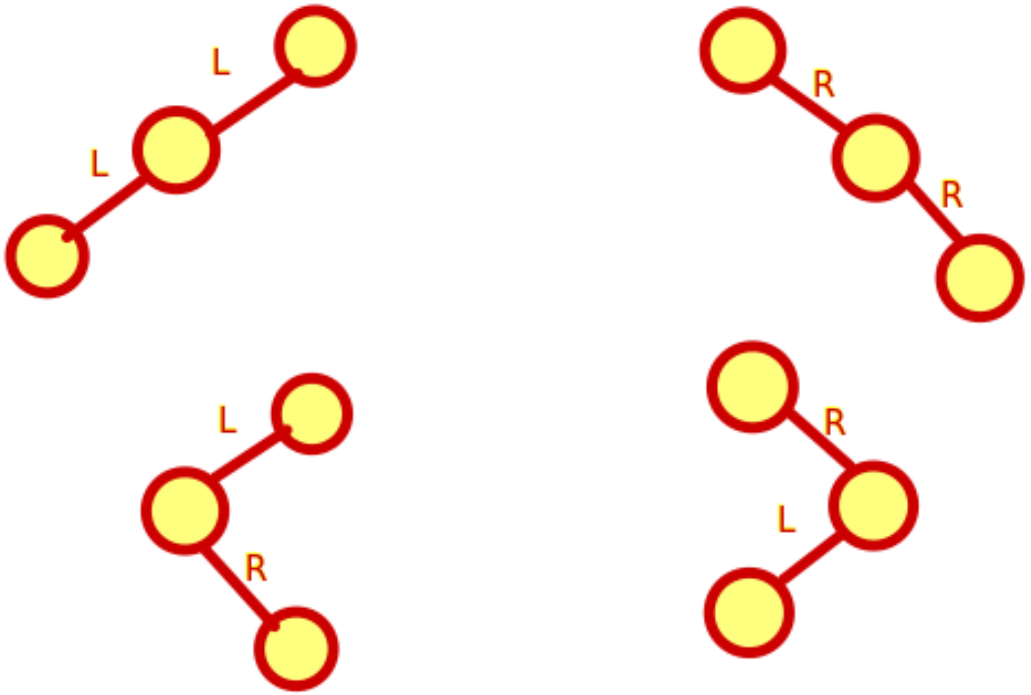


Figura: 4 tipos de rotaciones [1]

Rotación a la izquierda

```
1  rightChild.children[0].parent = node;
2  node.children[1] = rightChild.children[0];
3
4  node.parent = rightChild;
5  rightChild.children[0] = node;
6
7  rightChild.parent = parent;
8  data = rightChild;
```

Listing 1: Rotación a la izquierda

Rotación a la derecha

```
1  leftChild.children[1].parent = node;
2  node.children[0] = leftChild.children[1];
3
4  node.parent = leftChild;
5  leftChild.children[1] = node;
6
7  leftChild.parent = parent;
8  data = leftChild;
```

Listing 2: Rotación a la derecha

Insertar un elemento

```
1  while (!newNode) {
2    if (n <= walker.data) {
3      if (walker.children.length === 0) { // No child
4        walker.children.push({ id: id++, data: n, parent: walker, children: [] }); //
             Left child
5        walker.children.push({ id: id++, data: null, parent: walker, children: [] }); //
             Empty right child
6        newNode = walker.children[0];
7      } else if (walker.children[0].data === null) { // Already have right child, left
           child is empty
8       walker.children[0].data = n;
9       newNode = walker.children[0];
10     } else { // Move left
11      walker = walker.children[0];
12     }
13   } else {
14     if (walker.children.length === 0) { // No child
15       walker.children.push({ id: id++, data: null, parent: walker, children: [] }); //
             Empty left child
16       walker.children.push({ id: id++, data: n, parent: walker, children: [] }); //
             Right child
17       newNode = walker.children[1];
18     } else if (walker.children[1].data === null) { // Already have left child, right
           child is empty
19      walker.children[1].data = n;
20      newNode = walker.children[1];
21     } else { // Move left
```

Eliminación de un elemento

1. Buscar en el subárbol izquierdo el nodo mayor, eliminar el nodo hallado y reemplazar el valor del nodo con el nodo a eliminar.
2. Buscar en el subárbol derecho el nodo menor, eliminar el nodo hallado y reemplazar el valor del nodo con el nodo a eliminar.

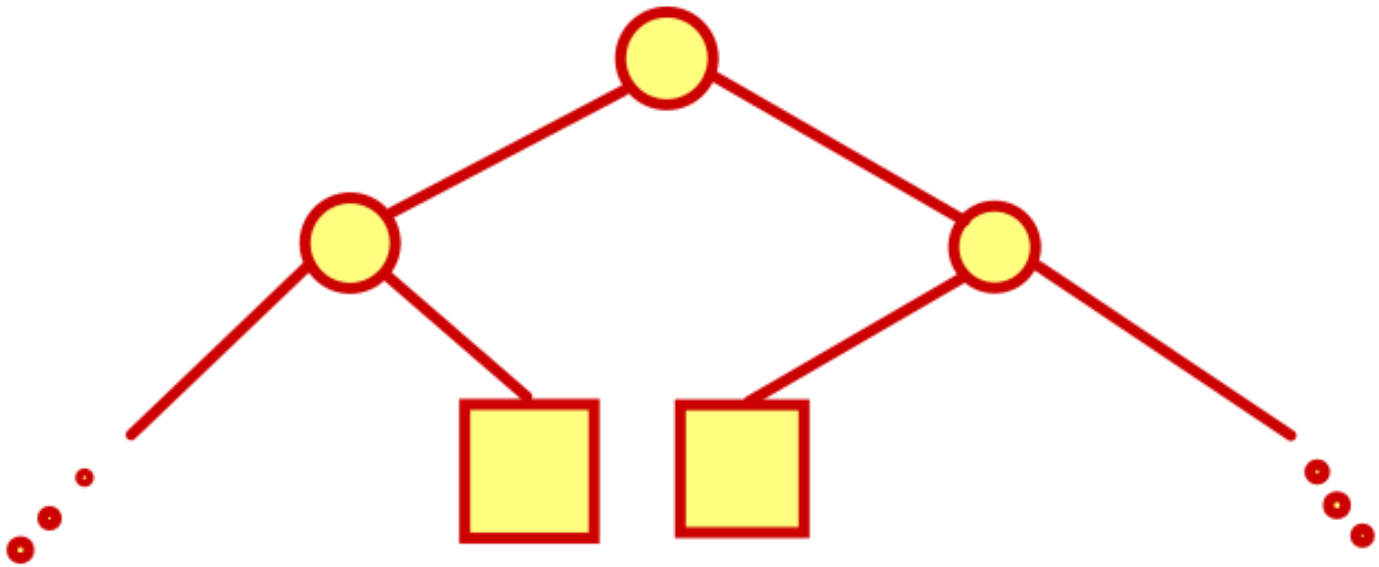


Figura: Eliminación de un nodo

Búsqueda de un elemento

```
1   highlight(nodeFindNodeValue);
2   await timeoutprint(duration*defer*4);
3   removeHighlight(nodeFindNodeValue);
4   var nodo = nodeFindNodeValue.children.length > 0 ? nodeFindNodeValue.children : null;
5
6   // verificar si el recorrido sera por el nodo derecho o izquierdo
7   if(value < nodeFindNodeValue.data){ // nodo izquierdo
8
9    if(nodo){
10    nodeFindNodeValue = nodo[0];
11   } else {
12    break;
13   }
14  } else{ // nodo derecho
15   if(nodo){
16    nodeFindNodeValue = nodo[1];
17   } else {
18    break;
19   }
20  }
21 }
```

Listing 4: Búsqueda de un elemento

Contenido

1. Arbol AVL
   - Rotación
   - Inserción
   - Eliminación
   - Búsqueda
2. Arbol B
   - Búsqueda
   - Inserción
   - Eliminación
3. Referencias

Búsqueda

```
1 BTree.prototype.search = function(value, strict){
2  if (!this.root) return false;
3  else return this.root.traverse(value, strict);
4 }
```

Listing 5: Buscar en árbol

```
1  BTreeNode.prototype.traverse = function(value, strict) {
2   if (this.keys.indexOf(value) > -1) return this;
3   else if (this.isLeaf()) {
4    if (strict) return false;
5    else return this;
6   }
7   else { // find the correct downward path for this value
8    for(var i = 0; i < this.keys.length; i++){
9     if(value < this.keys[i]){
10     return this.children[i].traverse(value, strict);
11    }
12   }
13   return this.children[this.keys.length].traverse(value, strict);
14  }
15 }
```

Listing 6: Desplazamiento

Búsqueda

```
1 BTree.prototype.search = function(value, strict){
2  if (!this.root) return false;
3  else return this.root.traverse(value, strict);
4 }
```

Listing 7: Buscar en árbol

```
1  BTreeNode.prototype.traverse = function(value, strict) {
2   if (this.keys.indexOf(value) > -1) return this;
3   else if (this.isLeaf()) {
4    if (strict) return false;
5    else return this;
6   }
7   else { // find the correct downward path for this value
8    for(var i = 0; i < this.keys.length; i++){
9     if(value < this.keys[i]){
10     return this.children[i].traverse(value, strict);
11    }
12   }
13   return this.children[this.keys.length].traverse(value, strict);
14  }
15 }
```

Listing 8: Desplazamiento

Búsqueda

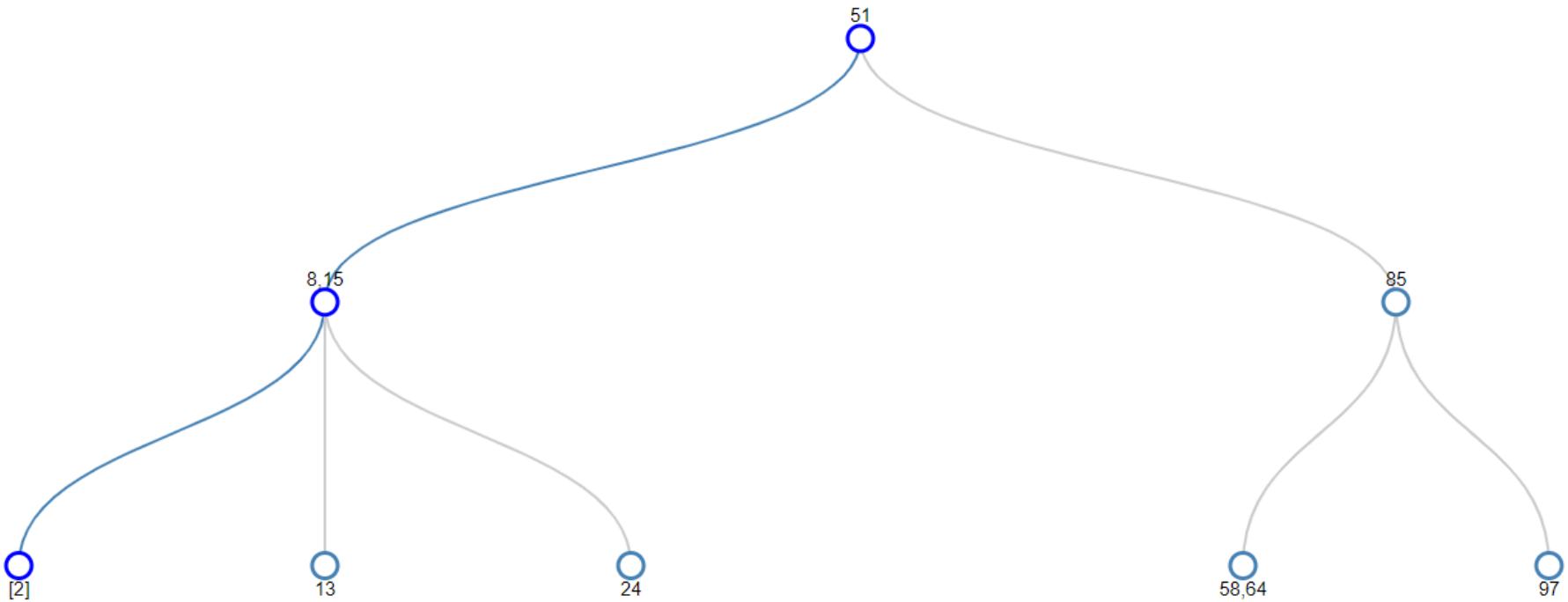


Figura: Buscar en árbol

Máximo

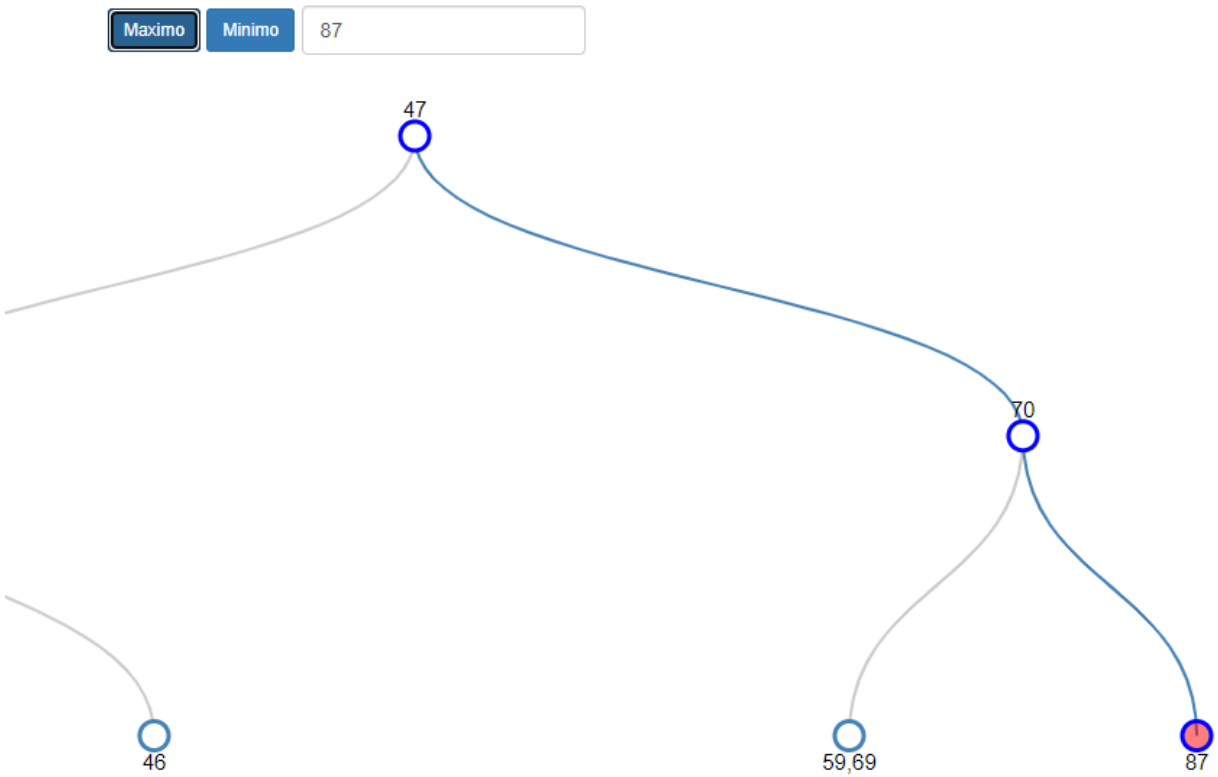


Figura: Buscar Máximo

Mínimo

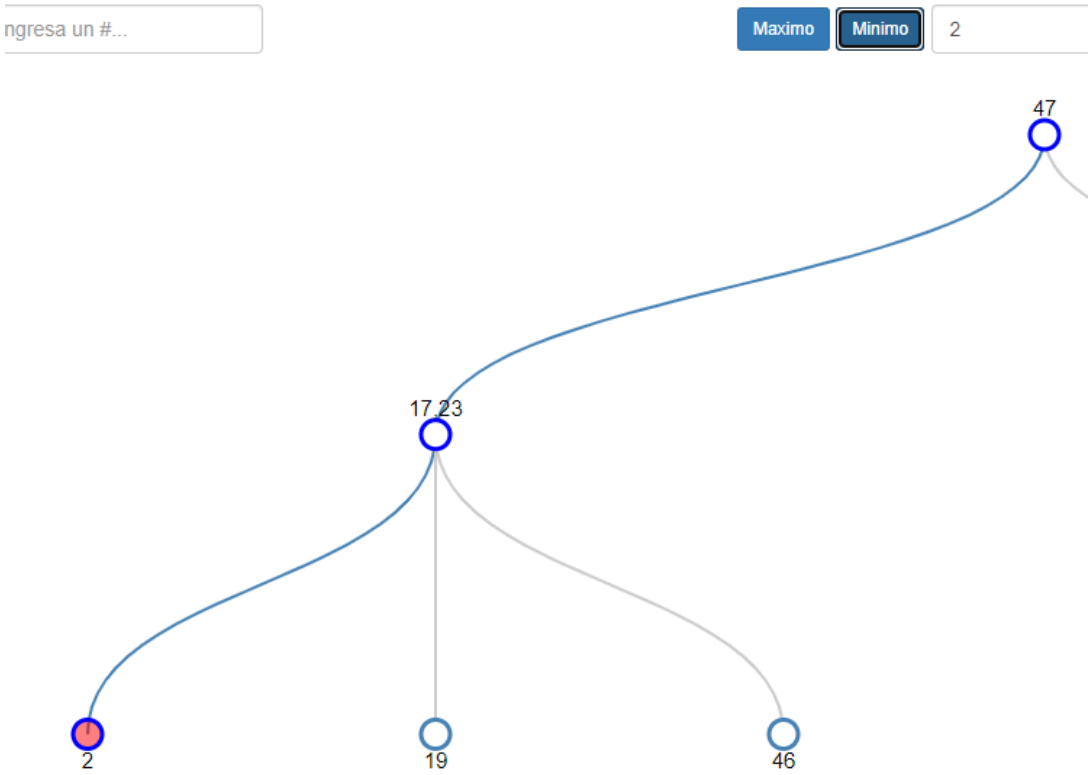


Figura: Buscar Mínimo

Inserción

```
1  BTree.prototype.insert = function(value, silent) {
2
3   if (this.search(value, true)) {
4    if (!silent) alert("The value "+value+" already exists!");
5    return false;
6   }
7
8   //actual hoja desplazada
9   this.current_leaf_offset = 0;
10  //nodos no vinculados
11  this.unattached_nodes = [[]];
12
13  // 1. Find which leaf the inserted value should go
14  var target = this.search(value);
15  if (!target) {
16   // create new root node
17   this.root = this.createNode();
18   target = this.root;
19  }
20
21  // 2. Apply target.insert (recursive)
22  target.insert(value);
23
24 }
```

Listing 9: Inserción

Insertar en nodo (1)

```
1  BTreeNode.prototype.insert = function(value){
2
3   var int = parseInt(value) || 0;
4   if ( int <= 0 || int > 1000000000000 ) {
5    alert('Please enter a valid integer.');
6    return false;
7   }
8
9   // insert element
10  this.keys.push(value);
11  //con el sort el item es insertado en la correcta posiciones
12  //al interior del vector keys
13  this.keys.sort(function(a,b){ // sort numbers ascending
14   if(a > b) return 1;
15   else if(a < b) return -1;
16   else return 0;
17  })
```

Listing 10: Insertar en Nodo 1

Insertar en nodo (2)

```
1
2  //balanceo
3  // if overflow, handle overflow (go up)
4  if(this.keys.length === this.tree.order) {
5   //desvinculo mis hijos y mi padre
6   //e inserto en mi padre el nodo mitad
7   this.handleOverflow();
8  } else { // if not filled, start attaching children
9   //si el padre me permite insertar elmento
10  //desciendo en el arbol vinculando todos mis hijos
11  //con la nueva divisiones ya incorporadas
12  this.attachChildren();
13 }
14 }
```

Listing 11: Insertar en Nodo 2

Dividir(split)

```
1  BTreeNode.prototype.splitMedian = function() {
2   var median_index = parseInt(tree.order/2);
3   var median = this.keys[median_index];
4
5   //separamos las keys izquierdas
6   var leftKeys = this.keys.slice(0,median_index);
7   //creamos un nodo con las key izquierdas
8   var leftNode = tree.createNode(leftKeys); // no children or parent
9   //agregamos a nodos no vinculados en el actual nivel
10  tree.addUnattached(leftNode, tree.current_leaf_offset);
11
12  //separamos keys derechas
13  var rightKeys = this.keys.slice(median_index+1, this.keys.length);
14  //creamos el nodo con keys derechas
15  var rightNode = tree.createNode(rightKeys);
16  //desvinculamos al los nodos derechas
17  tree.addUnattached(rightNode, tree.current_leaf_offset);
18  return median;
19 }
```

Listing 12: Dividir en caso de overflow

Insertar nodo (1)

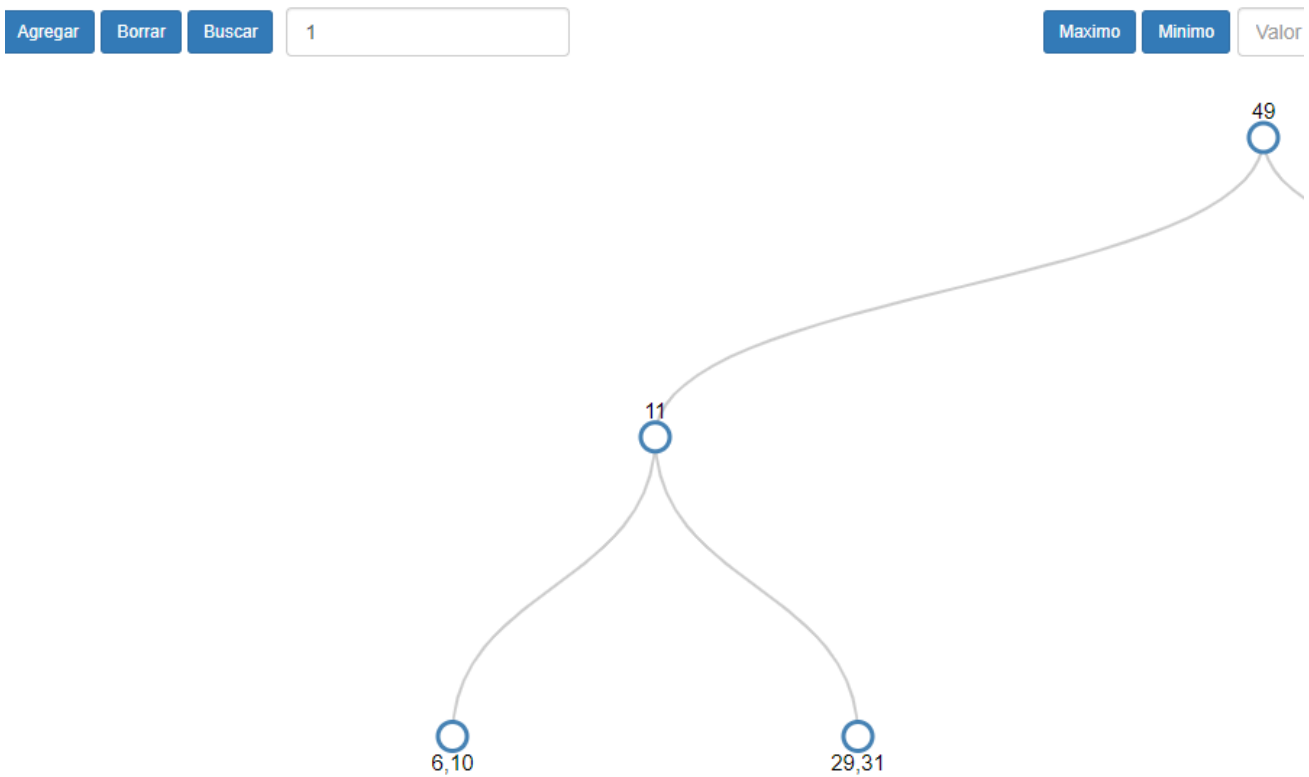


Figura: Insertar nodo $key = 1$ (1)

Insertar nodo (2)

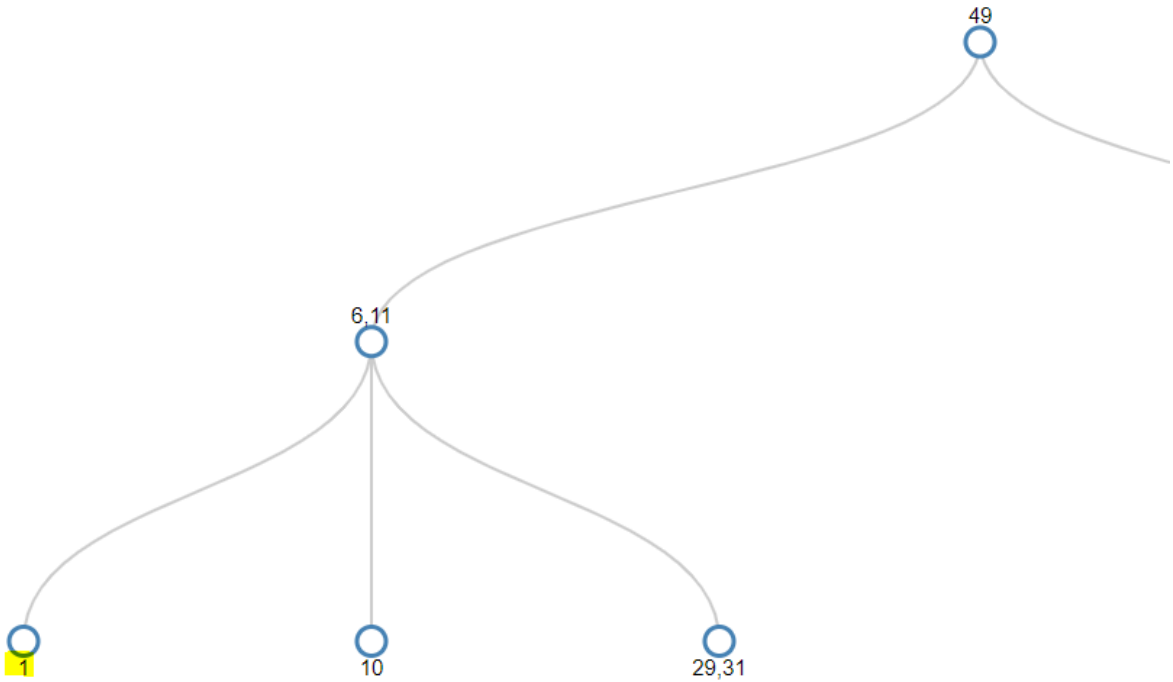


Figura: insertar nodo $key = 1$ (2)

Reglas de eliminación

1. Un nodo puede tener un número m máximo de hijos
2. Un nodo puede tener un máximo de $m - 1$ Claves (Keys)
3. Un nodo debería tener un mínimo de $\lceil m/2 \rceil$ hijos
4. Un nodo (excepto el nodo raíz) debería contener un mínimo de $\lceil m/2 \rceil - 1$ Claves (Keys)

Eliminar - Caso 1

```
1     if (index >= 0) {
2       // Valor presente en el nodo
3       if (node.leaf && node.n > this.order - 1) {
4         // si el nodo es una hoja y tiene mas orden-1 valores, solo borrarlo
5         node.removeValue(node.values.indexOf(value));
6         return true;
7       }
```

Listing 13: Caso 1

Eliminar - Caso 1

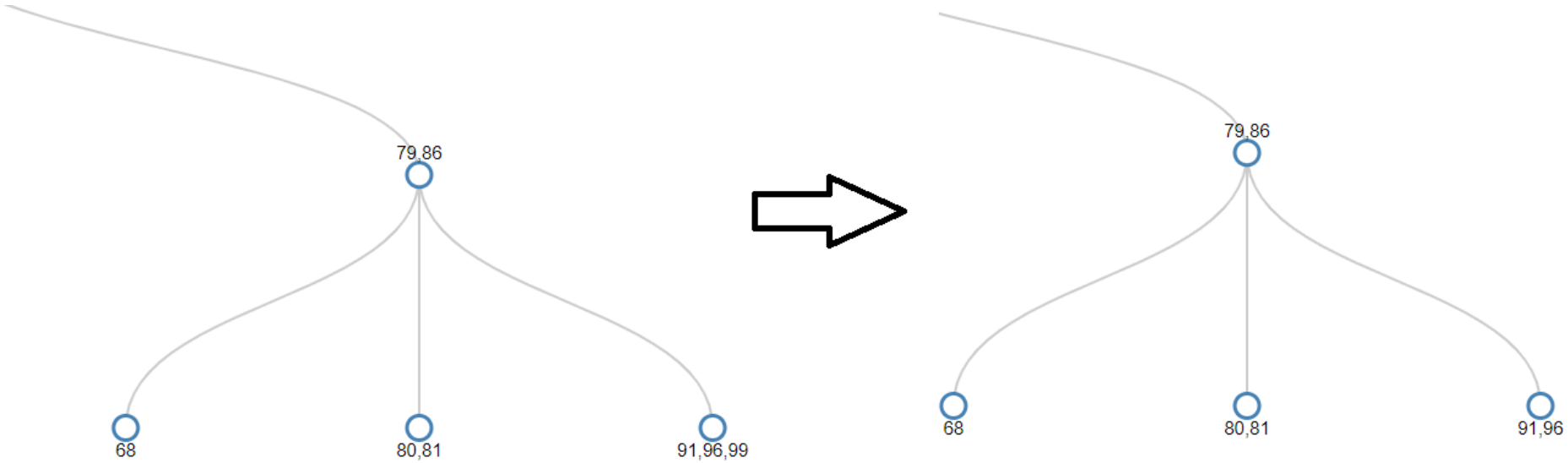


Figura: Caso 1 de Borrado de $Key = 99$ en hoja sin problema [2]

Eliminar - Caso 2

```
1      // Verificar si algun hijo tiene los suficientes valores a transferir
2      if (node.children[index].n > this.order - 1 ||
3        node.children[index + 1].n > this.order - 1) {
4        // Uno de los hijos inmediatos tiene los suficientes valores para transferir
5        if (node.children[index].n > this.order - 1) {
6          // Reemplazar el valor objetivo por el mayor del nodo izquierdo
7          // Luego Borrar el valor del hijo
8          const predecessor = this.getMinMaxFromSubTree(node.children[index], 1);
9          node.values[index] = predecessor;
10         return this.deleteFromNode(node.children[index], predecessor);
11       }
12       const successor = this.getMinMaxFromSubTree(node.children[index+1], 0);
13       node.values[index] = successor;
14       return this.deleteFromNode(node.children[index+1], successor);
15     }
16     // Hijos no tiene los sufientes valores para transferir. Realizar Merge o juntarlos
17     this.mergeNodes(node.children[index + 1], node.children[index]);
18     return this.deleteFromNode(node.children[index], value);
19   }
```

Listing 14: Caso 2

Eliminar - Caso 2

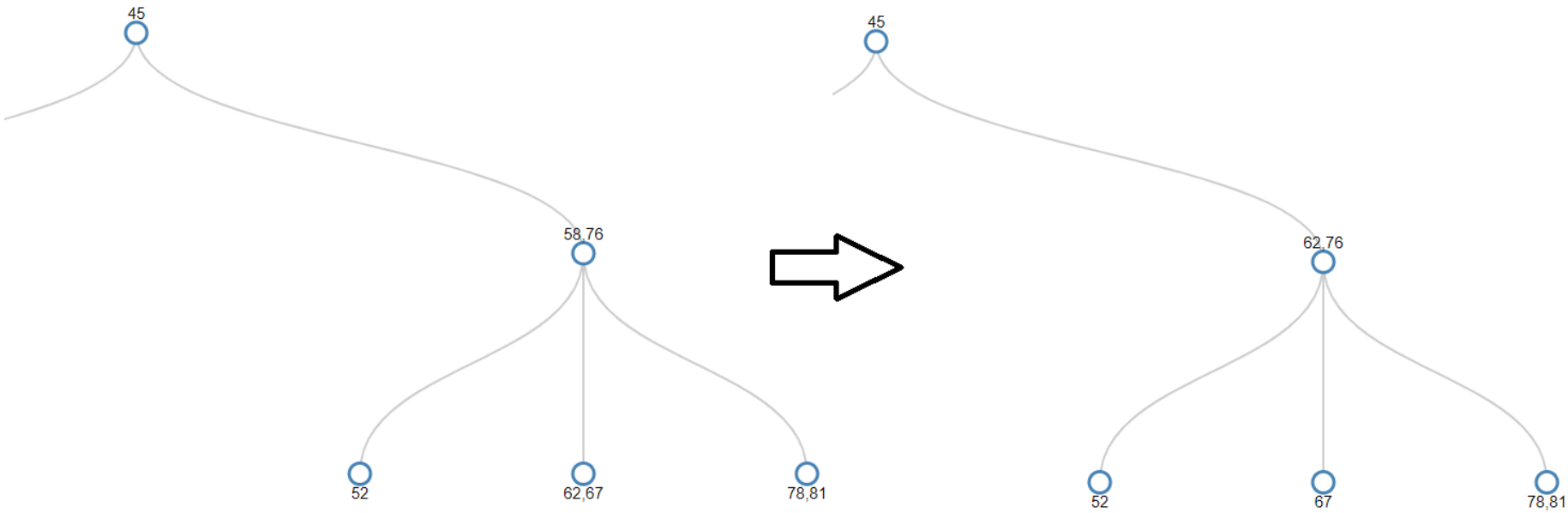


Figura: Caso 1 de Borrado $Key = 58$ esta en el nodo [2]

Eliminar - Caso 3

```
1     //Valor  no esta presente en el nodo
2     if (node.leaf) {
3       // valor no esta en el arbol
4       return false;
5     }
6     // Valor no esta presente en el nodo, buscar en el hijo
7     let nextNode = 0;
8     while (nextNode < node.n && node.values[nextNode] < value) {
9       nextNode++;
10    }
11    if (node.children[nextNode].n > this.order - 1) {
12      // Nodo hijo tiene los suficientes valores para continuar
13      return this.deleteFromNode(node.children[nextNode], value);
14    }
```

Listing 15: Caso 3

Eliminar - Caso 3

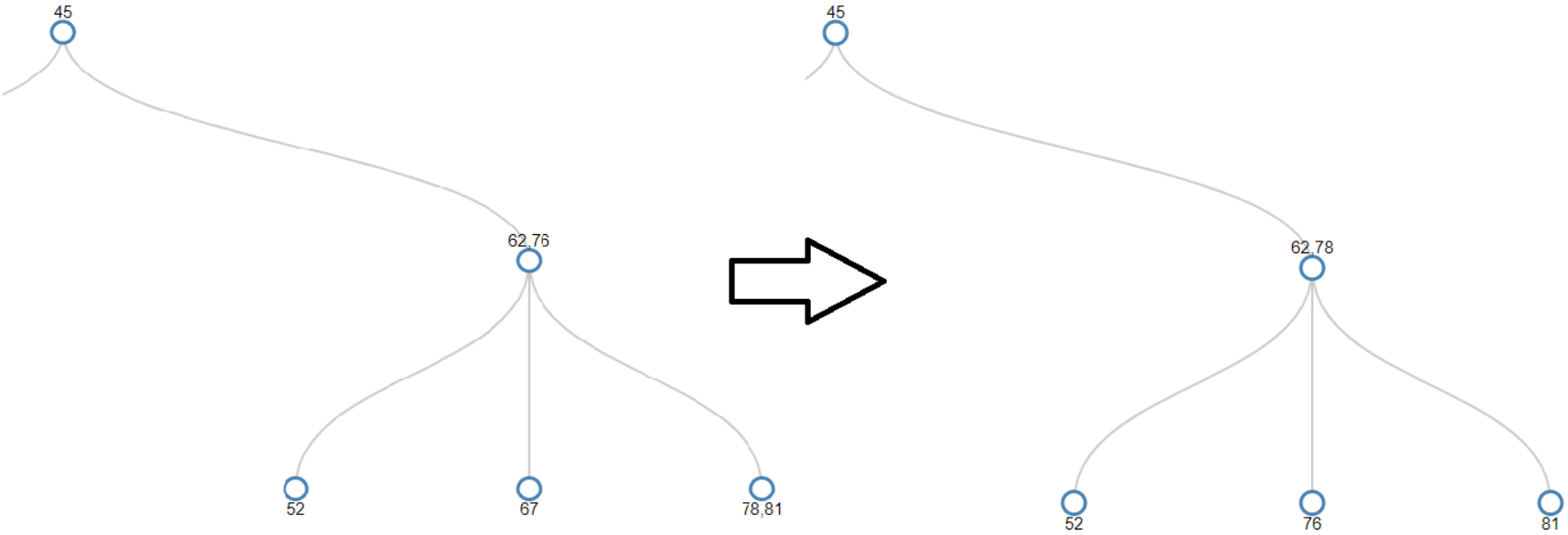


Figura: Caso 1 de Borrado Valor $Key = 67$ no presente en el nodo [2]

Animación Btree

```
248     // NODE POINT
249     nodeEnter.append("circle")
250       .attr("r", 5)
251       .style("fill", "white")
252       .style('opacity',0).transition()
253       .style('opacity',1).duration(250);
254
255     // UPDATE NODE DATA + POSITION
256     node.each(function(d,i){
257         //*** DIBUJAR CIRCULOS EN LAS POSICIONES
            var nodo_act = d3.select('#'+this.id);
259         var tex = d.data.name;
260         var arr = tex.split(',');
261         var array = [];
262         var cx, cy;
263         //*** Recorrer String Interno
264         for(var i=0; i<arr.length; i++){
265             //*** ADD CIRCULO
266             nodo_act.append("circle")
267                 .attr("cx", i*15+i*15-((arr.length-1)*15))
268                 .attr("cy", 20).attr("r", 15)
269                 .style("fill", "yellow");
270             //*** ADD TEXTO
271             nodo_act.append("text")
272                     .text(arr[i])
273                     .attr('transform', 'translate(' +
274                         pos_tex_3d(arr[i],i,arr.length)
275                         + ',' + (25) + ')');
276         }
277     });
```

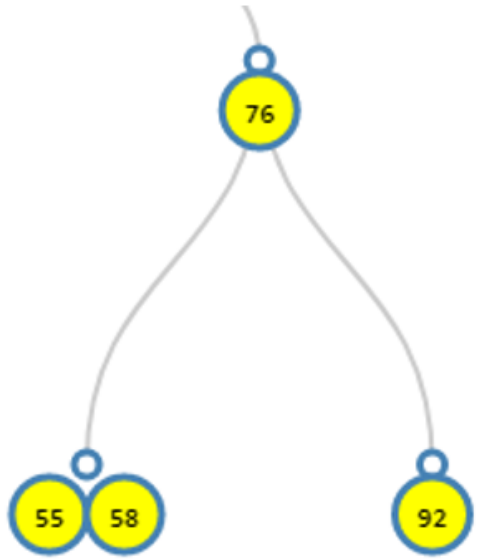


Figura: Distribución de Claves (keys)

Animación Btree

```
160  function colorPath(node){
161
162      //*** Color del Nodo
163      d3.selectAll('g.node').filter(function(d){
164      return d.data.name === node.keys.toString();
165      }).select('circle').style('stroke','red');
166      //*** Recorrer Arbol
167      if(node.isRoot())return;
168      else{
169          //*** Filtrar Contenido de Links
170          d3.selectAll('.link').filter(function(d){
171              return d.data ? d.data.name ===
172                      node.keys.toString() : d.data.name ===
173                      node.keys.toString();
174          }).style('stroke','red');//'steelblue');
175
176          return colorPath(node.parent);
177      }
178  }
```

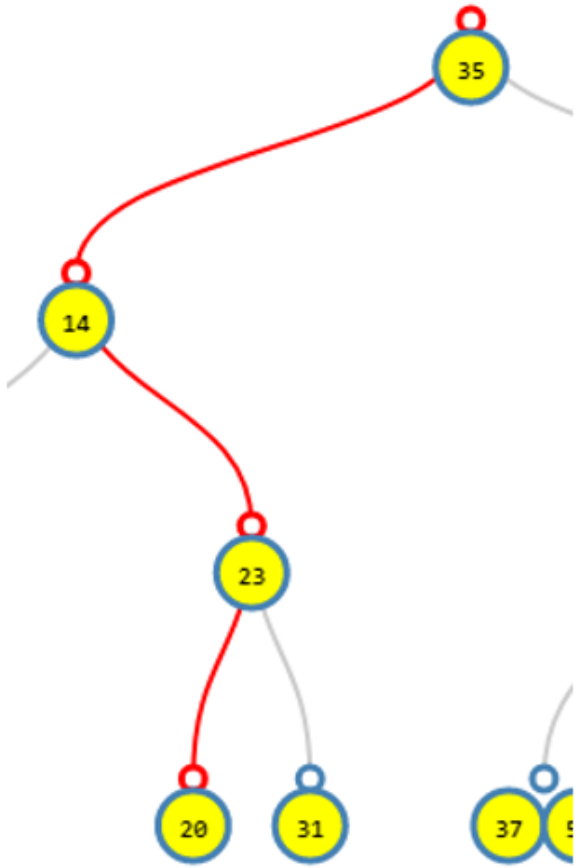


Figura: Animación de Búsqueda

Animación Btree

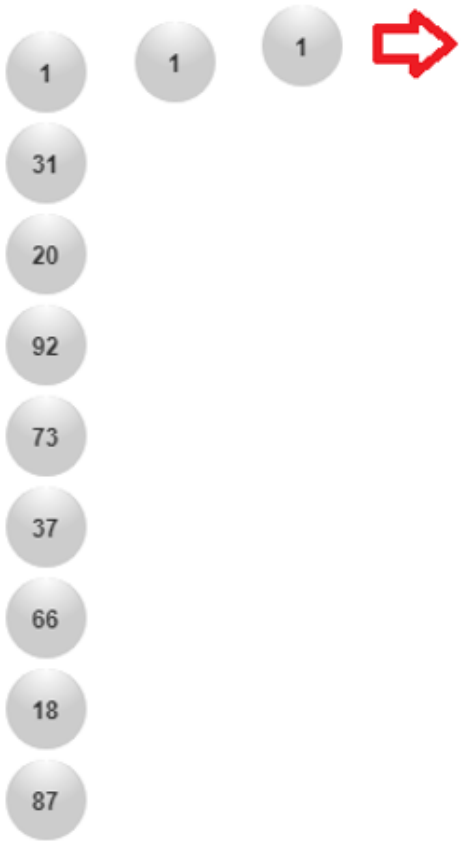
```
function insertar_data(data_value){
    //alert("DATA:"+data_value);
    event.preventDefault();
    var value = parseInt( data_value );
    bTree.insert(value, true); // silently insert

    $("#input-add").val("");

    treeData = bTree.toJSON();
    console.log(treeData);
    update(treeData);

    // Make the current add node highlighted in red
    $("g text").each(function(index) {
        var bTreeNode = bTree.search(value);
        var d3NodeTouched = d3.selectAll('g.node')
                .filter(function(d){
            return d.name === bTreeNode.keys.toString();
        });
       // reset all links and nodes
       d3.selectAll('g.node').select('circle')
                .style({stroke : '#ccc', fill: '#ffffff'});
       d3.selectAll('.link').style('stroke','#ccc');
       // color links and all intermediate nodes
       colorPath(bTreeNode);
       // color bottom node
       d3NodeTouched.select('circle')
                .style({stroke : '#ff0000', fill: '#ffcccc'});
    });

    ga('send', 'event', 'tree', 'inserted value');
}
```



GRACIAS

Repositorio: `https://github.com/jabarcamu/EDA_Practica2`

Contenido

1. Arbol AVL
   - Rotación
   - Inserción
   - Eliminación
   - Búsqueda
2. Arbol B
   - Búsqueda
   - Inserción
   - Eliminación
3. Referencias

References I

[1] S. A.-F. Ellis Horowitz, Sartaj Sahni, *Fundamentals of Data Structures in C: 2nd Edition*.
Silicon Press, 2008.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*.
The MIT Press, 3rd ed., 2009.