<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

\{0\} \{1\} \{2\} \{3\} \{4\} \{5\} \{6\} \{7\} \{8\} \{9\} \{10\} \{11\} \{12\} \{13\} \{14\} \{15\} \{16\} \{17\} \{18\} \{19\} \{20\} \{21\} \{22\} \{23\} \{24\}
\{0, 1\} \{2\} \{3\} \{4, 6, 7, 8, 9, 13, 14\} \{5\} \{10, 11, 15\} \{12\} \\
\{16, 17, 18, 22\} \{19\} \{20\} \{21\} \{23\} \{24\}
{0, 1} {2} {3} {5} {10, 11, 15} {12}
{4, 6, 7, 8, 9, 13, 14, 16, 17, 18, 22} {19} {20} {21} {23} {24}
\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24\}
A forest of 8 trees
The forest after the union of trees with 4 and 5
The forest after the union of trees with roots 6 and 7
The forest after the union of trees with roots 4 and 6
The forest formed by union-by-size, with the size encoded as negative numbers
Worst-case tree for N=16
The forest formed by union-by-height, with the height encoded as negative numbers
Path compression resulting from a find(14) on the tree
package weiss.nonstandard;

// DisjointSets class

// CONSTRUCTION: with int representing initial number of sets

// ***********************************PUBLIC OPERATIONS***********************************
// void union( root1, root2 ) --> Merge two sets
// int find( x ) --> Return set containing x

// ***********************************ERRORS***********************************
// Error checking or parameters is performed

public class DisjointSets
{
    public DisjointSets( int numElements )
    { /* Figure 24.21 */ }

    public void union( int root1, int root2 )
    { /* Figure 24.21 */ }

    public int find( int x )
    { /* Figure 24.21 */ }

    private int [ ] s;

    private void assertIsRoot( int root )
    {
        assertIsItem( root );
        if( s[ root ] >= 0 )
            throw new IllegalArgumentException( );
    }

    private void assertIsItem( int x )
    {  
        if( x < 0 || x >= s.length )
            throw new IllegalArgumentException( );
    }
}
/**
 * Construct the disjoint sets object.
 * @param numElements the initial number of disjoint sets.
 */

public DisjointSets( int numElements )
{
    s = new int[ numElements ];
    for( int i = 0; i < s.length; i++ )
        s[ i ] = -1;
}

/**
 * Union two disjoint sets using the height heuristic.
 * root1 and root2 are distinct and represent set names.
 * @param root1 the root of set 1.
 * @param root2 the root of set 2.
 * @throws IllegalArgumentException if root1 or root2
 * are not distinct roots.
 */

public void union( int root1, int root2 )
{
    assertIsRoot( root1 );
    assertIsRoot( root2 );
    if( root1 == root2 )
        throw new IllegalArgumentException( );

    if( s[ root2 ] < s[ root1 ] ) // root2 is deeper
        s[ root1 ] = root2;       // Make root2 new root
    else
    {
        if( s[ root1 ] == s[ root2 ] )
            s[ root1 ]--;          // Update height if same
        s[ root2 ] = root1;       // Make root1 new root
    }
}

/**
 * Perform a find with path compression.
 * @param x the element being searched for.
 * @return the set containing x.
 * @throws IllegalArgumentException if x is not valid.
 */

public int find( int x )
{
    assertIsItem( x );
    if( s[ x ] < 0 )
        return x;
    else
    {
        return s[ x ] = find( s[ x ] );
    }
}
Minimum spanning tree
Kruskal’s algorithm
The nearest common ancestor (NCA) for each request in the pair sequence

- \( \text{NCA}(x, y) \) is A
- \( \text{NCA}(x(u, z)) \) is C
- \( \text{NCA}(x(w, x)) \) is A
- \( \text{NCA}(x(z, w)) \) is B
- \( \text{NCA}(x(w, y)) \) is y
Before we return from D in post-order traversal, this is how the disjoint sets look like. Anchors (A, B, C, D) are nodes in stack.
After we return from D in post-order traversal, we union(C, D) and we can answer NCA(D, x) for all x that has been visited, such as NCA(D, p) and NCA(D, q) but not NCA(D, r)