Version 1, 2006/11/06 by Hyo-Sil Kim

The Rank Tree

We introduce a data structure called the *rank tree* to solve the *Josephus problem* efficiently.

The Josephus Problem

n players, numbered 1 to n, are sitting in a circle; starting at player 1, a hot potato is passed; after m passes, the player holding the hop potato is eliminated, the circle closes ranks, and the game continued with the player who was sitting after the eliminated player picking up the hot potato; the last remaining player wins.

The Josephus problem arose in the first century A.D in a cave on a mountain in Israel where Jewish zealots were being besieged Roman soldiers. The historian Josephus was among them. To Josephus's consternation, the zealots voted to enter into a suicide pact rather than surrender to the Romans. He suggested the game that now bears his name. The hot potato was sentence of death to the person next to the one who got the potato. Josephus rigged to get the last lot and convinced the remaining intended victim that the two of them should surrender. That is how we know about this game; in effect, Josephus cheated.

Fig. 1 is an example with 5 players and the number of passes being 1.

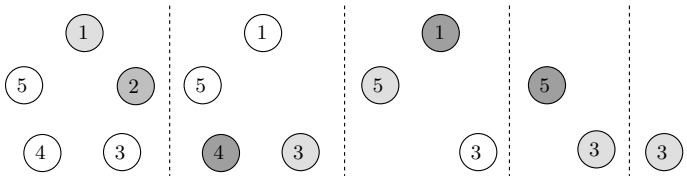


Figure 1: The Josephus problem: At each step, the darkest circle represents the initial holder and the lightly shaded circle represents the player who received the hot potato (and is eliminated). Passes are made clockwise.

What data structure to use? We need a data structure to store the circle of player with following methods:

- Pass the potato to the next person.
- Delete the person holding the potato.

The Simple Solutions

Using a Linked List. The required methods suggest that we can represent the players in a linked list, actually in a circular linked list which is even better considering that players are sitting in a circle. Even though Java does not provide one, we can simulate it easily. If you see the following code,

```
// Return the winner in the Josephus problem.
// Linked list implementation.
public static int josephus(int people, int passes)
{
    List<Integer> list = new LinkedList<Integer>();
    for (int i = 1; i <= people; i++)
        list.add(i);

    // Play the game;
    Iterator<Integer> itr = list.iterator();
    while (people > 1) {
        for (int i = 0; i <= passes; i++) {
1.          if (!itr.hasNext())
2.              itr = list.iterator();
```

```
            itr.next();
        }
        itr.remove();
        --people;
    }
    // find out who the winner is
    itr = list.iterator();
    return itr.next( );
}
```

you simply notice that adding the line number 1 and 2 only makes the circular linked list simulation possible.

How much time takes to solve the Josephus problem with a linked list? It takes $O(m+1)$ for the hot potato to reach to the right player who will be eliminated. Since we continue to eliminate players until only one player is left, $O(n-1)$ rounds are needed. Multiplying them, we get $O(mn)$, which means quadratic time. Can we do better? What about using an array list?

Using an Array List. To use an array list, first observe that if the potato is currently at position p, then after m passes, it will be at position $(p+m)\%n$. (Here, positions are numbered from 0 to $n-1$.) Then we can easily find the player who will be eliminated in $O(1)$. But how much does it take to remove the player at the position? Remember that after removing the player, we need to compact the array, which means we have to shift $O(n-p-1)$ elements behind the removed one. Multiplying this with $O(n-1)$ rounds, we get $O(n^2)$. Still quadratic time takes to solve our problem.

A linked list is good at deleting an object, but not very good at finding it; An array list is good at finding an object, but not very good at removing it; somehow we need a data structure which good at both finding and deleing an object.

A More Efficient Algorithm: the Rank Tree

Here, we introduce the *rank tree* which supports the following operations.

1. Construct a rank tree from an array with n elements;
2. find(int k) returns the item at rank (index) k;
3. remove(int k) removes the item at rank k;
4. size() returns the current size;

Idea.

- Store the elements in a binary tree in in-order sequence. Store in each node t the size of the subtree whose root is t.
- To find the node with rank k, we just have to follow the path from the root.
- To remove the node with rank k, there are three cases.
  - if t has zero or one subtree, just remove t and change a reference of its parent.
  - if t has two subtrees, delete the leftmost node in its right subtree instead, and move the element stored there to t. (rename it)

  Of course we have to recompute the size of nodes after deleting a node t.

Let us look at implementations of above method with examples.

A Node. An element of the binary tree.

```
private static class Node<AnyType> {
    AnyType iElement;
    Node<AnyType> iLeft;
    Node<AnyType> iRight;
    int iSize;

    public Node(AnyType element) {
        iElement = element;
        iLeft = iRight = null;
        iSize = 1;
    }
    public void recomputeSize() {
        int leftSize = (iLeft != null) ? iLeft.iSize : 0;
        int rightSize = (iRight != null) ? iRight.iSize : 0;
        iSize =  leftSize + 1 + rightSize;
    }
}
```

A Construction Method. We construct a binary tree in in-order sequence with a size.

```
 // Construct the tree.
public RankTree(AnyType [] elements) {
    iRoot = buildTree(elements, 0, elements.length - 1);
}
 // Internal method to build tree for elements from i to j.
private Node<AnyType> buildTree(AnyType [] elements, int i, int j)
{
    if (j < i)  return null;
    if (i == j) return new Node<AnyType>(elements[i]);
    int mid = (i + j) / 2;
    Node<AnyType> t = new Node<AnyType>(elements[mid]);
    t.iLeft = buildTree(elements, i, mid - 1);
    t.iRight = buildTree(elements, mid + 1, j);
    t.recomputeSize();
    return t;
}
```

A Find Method: Let us assume that we want to find a node t with rank k and also let us denote the number of elements in the left subtree of the current node as s. Then, there are the following three cases:

- if $k < s$, go to the left subtree and apply it again.
- if $k = s$, return the current node.
- if $k > s$, go to the right subtree, change the rank into $k - (s + 1)$ instead of k, and apply it again.

As you can see, in all the cases, you only need to follow a path from the root.

```
 // Find item at rank k in the tree.
 // returns null if not found.
public AnyType find(int k) {
    return elementAt(find(k, iRoot));
}
```

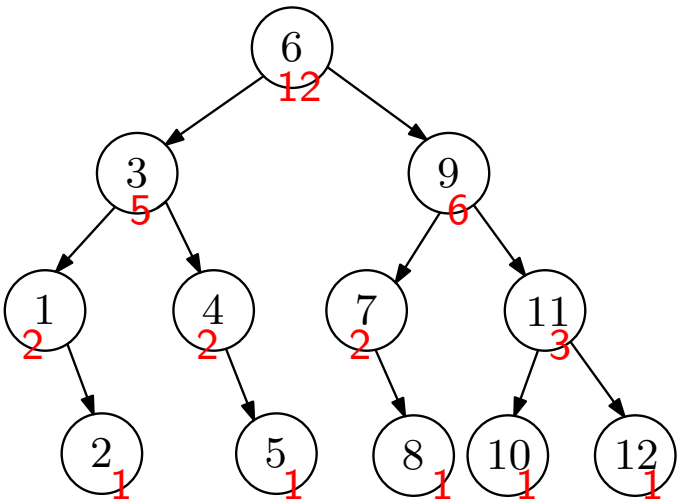


Figure 2: Construction of the rank tree with 12 elements and with the size of each node written in red.

```
 // Internal method to find node at rank k in a subtree rooted at t.
 // returns node containing the matched item.
private Node<AnyType> find(int k, Node<AnyType> t) {
    if (t == null)  throw new IllegalArgumentException();
    int leftSize = (t.iLeft != null) ? t.iLeft.iSize : 0;
    if (k < leftSize)   return find(k, t.iLeft);
    if (k == leftSize)  return t;
    return find(k - leftSize - 1, t.iRight);
}
```

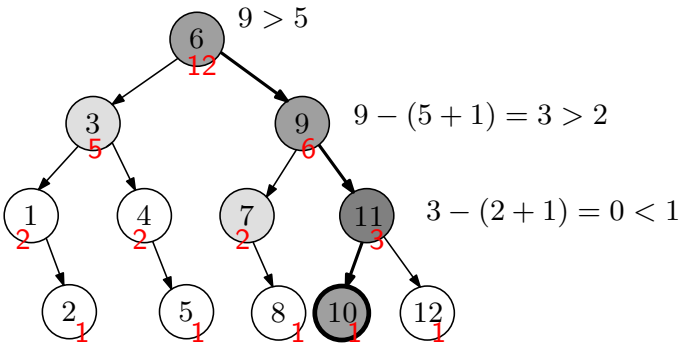
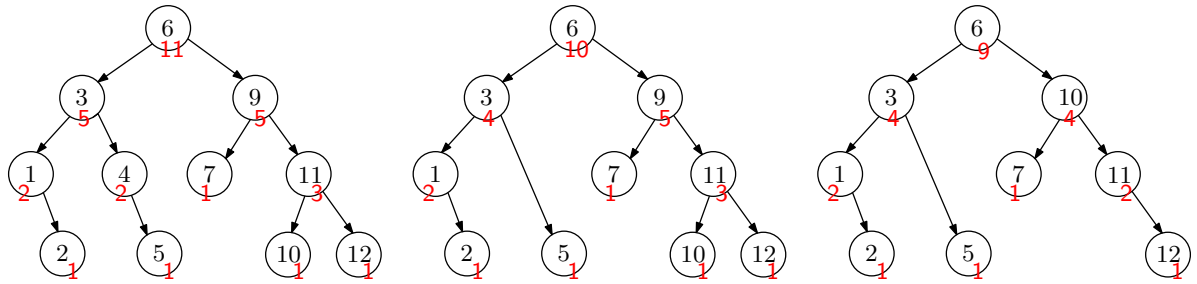


Figure 3: Find(rank 9): The darkest nodes are visited to find it and lightly shaded nodes are referenced nodes to know the number of elements in the left subtrees.

A Delete Method.

```
 // Remove element at rank k from the tree..
public void remove(int k) {
    iRoot = remove(k, iRoot);
}
  // Internal method to remove node at rank k from a subtree rooted at t.
  // returns the new root.
private Node<AnyType> remove(int k, Node<AnyType> t) {
```

```
        if (t == null)  throw new IllegalArgumentException();
        int leftSize = (t.iLeft != null) ? t.iLeft.iSize : 0;
        if (k < leftSize)       t.iLeft = remove(k, t.iLeft);
        else if (k > leftSize)  t.iRight = remove(k - leftSize - 1, t.iRight);
        else {
            // need to remove node t
            if (t.iLeft != null && t.iRight != null) { // Two children
                t.iElement = findFirst(t.iRight).iElement;
                t.iRight = removeFirst(t.iRight);
            }
            else    t = (t.iLeft != null) ? t.iLeft : t.iRight;
        }
        if (t != null)  t.recomputeSize();
        return t;
    }
```



(a) After deleting the node with rank 7 (b) After deleting the node with rank 3 (c) After deleting the node with rank 6

Figure 4: Delete a node with given rank k

As you can see, there are a lot of recursion going on. Do you know why this is so? It is because using recursion, we can remember the path from the root. If you notice that a node of a tree has only outgoing edges to its children, not its parent, you may wonder how we can possibly go upward directly to answer questions like recomputing the `size` of each root. Here, we do not need to go upward explicitly. Instead, recursion tells you where you stayed and computes the value where you stayed. That is the beauty of recursion. :-)

Analysis.

1. `find(int k)`: $O(h)$
2. `remove(int k)`: $O(h)$
3. `size()`: $O(1)$

where h is the height of the tree. Recall from the last class, that the *height* is the length of the longest path from the root. Since we can make a perfectly balanced tree in the construction step and by not having an insertion operator, the balance is never broken, we know that the height of the rank tree is $\lceil \log(n+1) \rceil - 1$.

Therefore `find` and `remove` take time $O(\log n)$, and the total running time for the Josephus problem is $O(n \log n)$.

The running time of the `RankTree` constructor, by the way, is $O(n)$. This follows from solving the recursive formula $T(n) = 2T(n/2) + O(1)$.