```
//R-2.7

Algorithm Root (S)

   return S[1]



Algorithm Parent(S,p)

   if p==1 return null

   return S[p//2]



Algorithm leftChild(S,p)

   if (2*p)> size(s) then return null

   return S[2*p]


Algorithm rightChild(S,p)

   if (2*p+1)> size(s) then return null

   return S[2*p +1]


Algorithm isInternal(S,p)

   return 2*p < size(S)


Algorithm isExternal(S,p)

   return 2*p > size(S)


Algorithm isRoot(S,p)

   return p==1
```

//R-2.8

a. will be 16 children at the level 5

b.For height h, the minimum number of external nodes is 1,

since a degenerate tree has just one leaf at the deepest level.

c.For height h, the maximum number of external nodes is 2^h,

because a complete binary tree doubles the number of external nodes at each level.

exple : level 5 -> 2^5=32

d.

In a binary tree, for n internal nodes, the number of external nodes e is n + 1

(because a binary tree has one more external node than internal nodes).

The height of a complete binary tree is the minimum height, which is log(n + 1).

The height of a degenerate (linear) tree is the maximum height, which is n.

Hence, the height h satisfies:

$\log(n+1)<h<n$

The logarithmic term represents the minimum height (complete tree),

and the linear term represents the maximum height (degenerate tree).

e.

Lower Bound (log(n+1) = h): Achieved when the binary tree is complete,

meaning all levels except possibly the last are fully filled, and the last level has nodes as far left as possible.

Upper Bound (h = n): Achieved when the binary tree is degenerate, i.e., each internal node has only one child, making the tree resemble a linked list.

C-2.2

When using two stacks to implement a queue:

Enqueue (push): This operation always takes constant time O(1) because we simply push elements onto the first stack (stack1).

Dequeue (pop):

When stack2 is empty, all elements from stack1 are transferred to stack2, which takes O(n) in the worst case, where n is the number of elements.

However, this transfer happens infrequently. After transferring, each of these n elements can be dequeued in O(1) time.

Therefore, the amortized cost of a dequeue operation is O(1) because the expensive O(n) transfer is spread out over the n dequeues.

Conclusion: The amortized running time for both enqueue and dequeue is O(1).

C-2.7

We need to shuffle a sequence of n elements (representing cards) so that each possible order is equally likely.

The approach involves using a variation of the Fisher-Yates shuffle (also known as the Knuth shuffle).

```
Algorithm shuffle(S, n):
  for i = 0 to n-2:
    j = randomInt(n-i) + i
    swap(S[i], S[j])
```

//randomInt(n) generates a random integer between 0 and n-1.

For each position i, you randomly select an index j from the unselected portion of the sequence and swap the elements at i and j.

This ensures that every element is selected exactly once, guaranteeing that all possible orderings are equally likely.