R-2.7

Algorithm root()
    Return S.elemAtRank(1)

Algorithm parent(v)
    If p(v) mod 2 > 0
        Return S.elemAtRank((p(v)-1) /2)
    Else
        Return S.elemAtRank(P(v)/2)

Algorithm leftChild(v)
    Return S.elemAtRank(2p(v))

Algorithm rightChild(v)
    Return S.elemAtRank(2p(v) + 1)

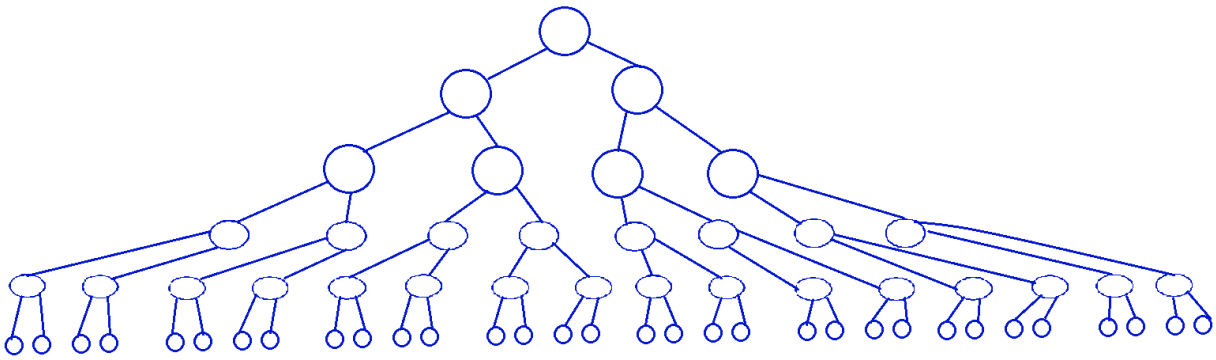
Algorithm isInternal(v)
    Return ((2 p(v)+ 1)< (S.size()-1) ∧ (leftChild(v) ¬= null ∨ rightChild(v) ¬= null))

Algorithm isExternal(v)
    Return ¬ isInternal(v)

Algorithm isRoot(v)
    Return v= root()

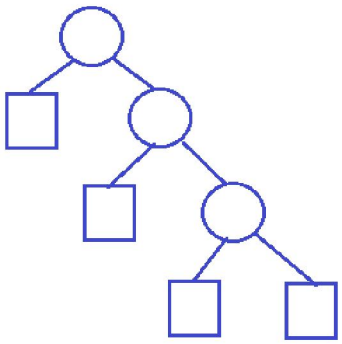
R-2.8

a)

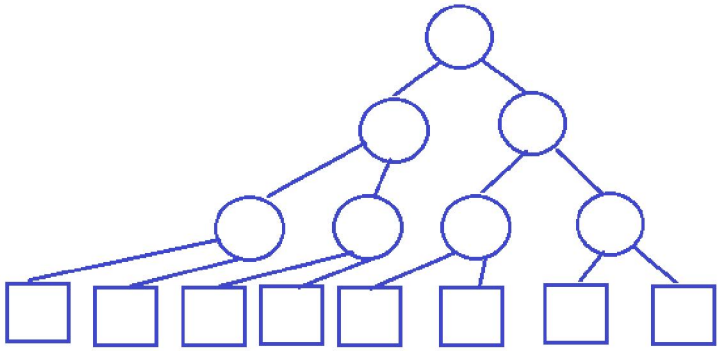


b) e= i+1
i >= h
e >= h+1

The minimum number of external nodes is h +1



c)
$I <= 2^h - 1$
I = e – 1
$e <= 2^h$
The maximum number of external nodes for a binary tree is 2^h



d) $h <= I <= 2^h - 1$
$h+1 <= e <= 2^h$
$2h+1 <= n <= 2^{h+1} - 1$

$n >= 2h+1 \quad \wedge \quad n <= 2^{h+1} - 1$

$h <= (n-1)/2 \quad \wedge \quad h >= \log(n+1) - 1$

$\log(n+1) - 1 <= h <= (n-1)/2$

e) $\log(n+1) - 1 = (n-1)/2$

$\log(n+1) = (n+1)/2$

The above relation is true for:

$n+1=2$ or $n+1 = 4$

$n=1$ and $h=0$

$n=3$ and $h=1$

C-2.2

Suppose we have two stacks S1 and S2. Enqueue just push the element to S1. Dequeue is implemented by doing pop on S2 if S2 is not empty. If S2 is empty then pop all the elements from S1 to S2 then pop the first element from S2.

If we set the weight for Enqueue to be 2, then we will have one extra credit when moving each element from S1 to S2. Thus the for n operations the running time is O(n). As a result, each operation runs in O(1) amortized time.

C-2.7

Algorithm shuffleDeck(s)

{Input s: sequence of cards to be shuffles}

	Array	Linked List
shuffledSquence ← Create new sequence		
while ¬ s.isEmpty() do	O(n)	O(n)
randomElement ← s.removeAtRank(randomInt(s.size()))	$O(n^2)$	O(n)
shuffledSquence.addLast(randomElement)	O(n)	O(n)
return shuffledSquence		

The running time Array based sequence is $O(n^2)$

The running time Linked List based sequence is O(n)