# R 2.7

root

output: position of the root element of the tree T in vector S

return 1

parent

input: position p of a node in vector S

output: parent node of the given node

return p / 2

leftChild

input: position p of a node in vector S

output: left child node of the given node

return 2 \* p

rightChild

input: position p of a node in vector S

output: right child node of the given node

return 2 \* p + 1

isInternal

input: position p of a node in vector S

output: boolean output of the node if it is an internal node

if 2 \* p <= S.size() or (2 \* p + 1) <= S.size()

return true

return false

isExternal

input: position p of a node in vector S

output: boolean output of the node if it is an internal node

if 2 \* p > S.size() and (2 \* p + 1) > S.size()

return true

return false

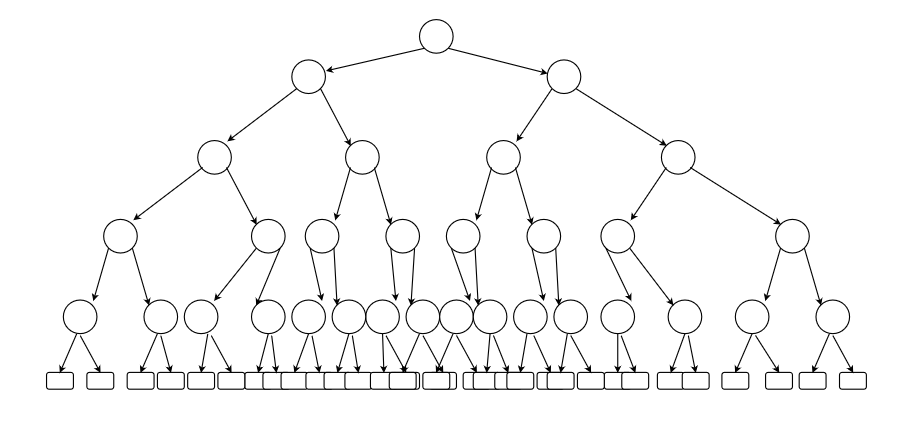
isRoot

input: position p of a node in vector S

output: boolean output of the node if it is the root node

# R 2.8

a.



b. Minimum number of external nodes is h + 1 which is 6 in our case. This case happens if each node has its left child external and right child internal till we get to height 5.

c. Maximum number of external is 2h which in our case is 25. This case happens if all node have two internal child nodes till we get to height 5, as in the figure shown above.

d. Since n >= 2h +1 and n <= 2h+1 - 1

n – 1 >= 2h n + 1 <= 2h+1

h < = (n – 1) / 2 log(n + 1) <= h + 1

h >= log(n + 1) – 1

=> **log(n + 1) – 1 <= h <= (n – 1) / 2**

e. for **h = 0** and **n = 1**

=> log2(1 + 1) – 1 <= h <= (1 – 1) / 2

=> 0 <= h <= 0

# C 2.2

As push, pop and size methods support constant time,

Running time of the dequeue and enqueue = n + n \* k.

T(n) = O(n).

Amortized running time = T(n) / n = 1

# C 2.7

Algorithm putSequenceInRandomOrder(S)

Input:Sequence S with n elements

Output: S in random order

r <- n

while r > 0 do

rand <- randomInt(r)

S.swapElements(S.atRank(r),S.atRank(rand))

r <- r - 1

return S