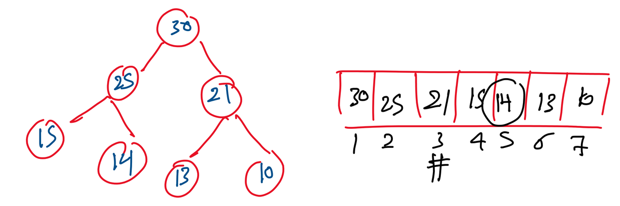
assignment-3

Friday, 1 November 2024 10:37 PM

R-2.7 Let T be a binary tree with n nodes that is implemented with a vector, S, and let p be the level numbering of the nodes in T, as given in Section 2.3.4. Give pseudo-code descriptions of each of the methods root, parent, leftChild, rightChild, isInternal, isExternal, and isRoot.



① Root

to return the root node of a binary tree

```
Algorithm root()
    return S[1]    // root at index 1
```

② parent

return the parent of a node @ index i

```
Algorithm parent(i)
    if i == 0
        return null
    return S[ i / 2 ]
```

③ leftChild

return the left child of the node @ index i

```
Algorithm leftChild(i)
    leftIndex = 2 * i
    if leftIndex >= len(S)
        return null
    return S[leftIndex]
```

④ rightChild

return the right child of a node @ index i

```
Algorithm RightChild(i)
    rightIndex = 2 * i + 1
    if (rightIndex >= len(S))
        return null
    return S[rightIndex]
```

⑤ IsInternal (i)

checks if the node @ index i is an internal node

```
Algorithm isInternal(i)
    if leftChild(i) == null or rightChild(i) == null
        return false
    return true
```

⑥ IsExternal

```
Algorithm IsExternal(i)
    if leftChild(i) == null ∧ right child(i) == null
        return true
    return false
```

⑦ isRoot

```
Algorithm IsRoot(i)
    if i == 1 then
        return true
    return false
```

R-2.8 Answer the following questions so as to justify Theorem 2.8.

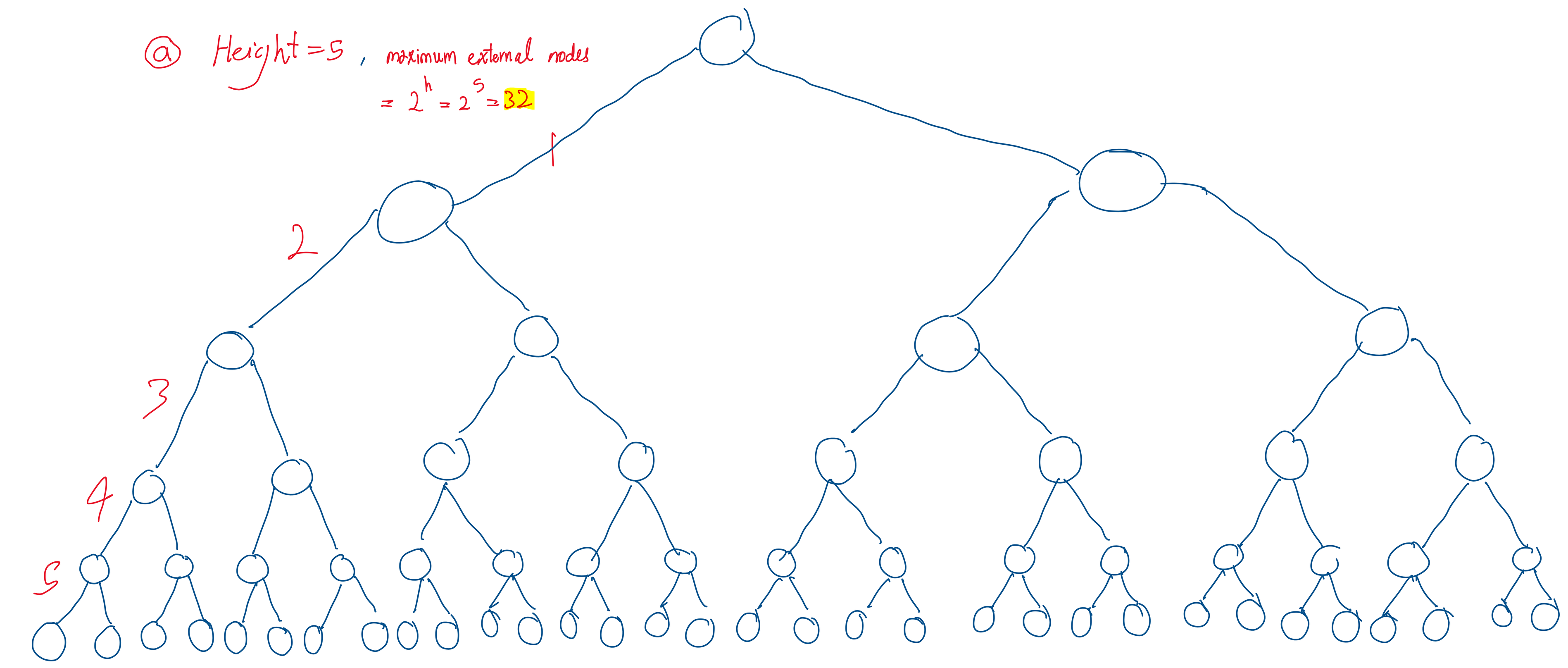
a. Draw a binary tree with height 5 and with the maximum number of external nodes.
b. What is the minimum number of external nodes for a binary tree with height h? Justify your answer.
c. What is the maximum number of external nodes for a binary tree with height h? Justify your answer.
d. Let *T* be a binary tree with height *h* and *n* internal nodes. Show that

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

e. For which values of *n* and *h* can the above lower and upper bounds on *h* be attained with equality?

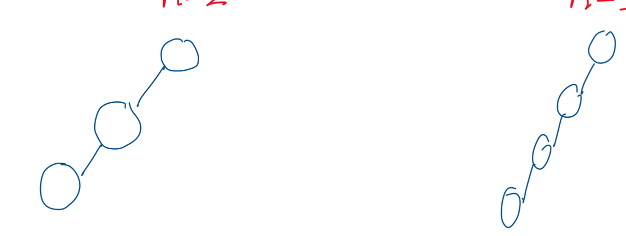
ⓐ Height = 5, maximum external nodes

$= 2^h = 2^5 = 32$



ⓑ

h=2 h=3



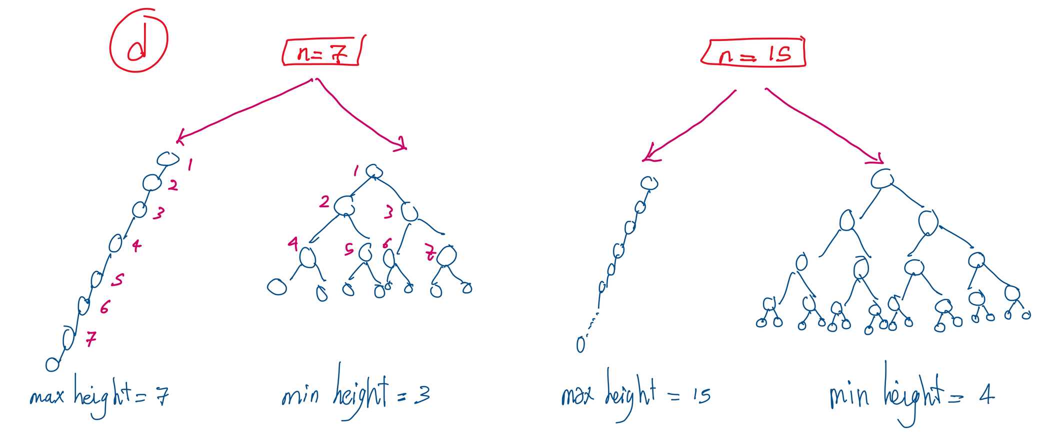
So the minimum number of external nodes is 1 as the tree will be a linear chain

ⓒ The maximum number of external nodes

$e = 2^h$

e: maximum number of external nodes
h: height of the tree.

ⓓ



We can see that minimum height ≥ log(n+1)
〃 〃 〃 〃 maximum height ≤ number of internal nodes n

So

$\boxed{\log(n+1) \le h \le n}$

ⓔ Lower Bound Equality (h = log(n+1))
↳ this equality holds for a Complete binary tree.

upper Bound Equality (h = n)
↳ this equality holds for a degenerate (linear) tree

C-2.2 Analyze your implementation of the queue ADT that used two stacks (from assignment 2). What is the amortized running time for dequeue and enqueue, assuming that the stacks support constant time push, pop, and size methods?

the amortized running time for both "enqueue" and "dequeue" operations in a a queue implemented with two stacks is O(1). This means that although an individual dequeue operation may take O(n) time (when transferring all elements from stack1 to stack2). the average time per operation across a sequence of operation remains O(1)

C-2.7 Using the Sequence ADT, describe an efficient way of putting a sequence representing a deck of *n* cards into random order. Use the function randomInt(n), which returns a random number between 0 and n-1, inclusive. Your method should guarantee that every possible ordering is equally likely. Equally likely means that each element/card must be selected exactly one time. **Hint**: randomly select an element from the unselected elements, then swap it from the unselected segment of the Sequence and into the selected segment; keep doing this until there is only one element left in the unselected segment. After each swap, there will be one fewer in the unselected segment and one more in the selected segment. What is the running time of your method, if the sequence is implemented with an array? What if it is implemented with a linked list?

```
Algorithm RandomCards( deckSeq )
    i ← 0
    n = deckSeq.size()
    while i < n - 2 :
        r ← randomInt(n - i)
        swap(deckSeq[i], deckSeq[i+r])
```

○ For Array-based sequence, both accessing & swapping are O(1) so when we perform n-1 swaps, the total running time is O(n)

○ For linkedlist-based sequence, accessing an element at an arbitrary index i is O(i) because we must traverse the list to reach it so the total complexity is

$O(1+2+3+\dots+n) = O\left(\frac{n*(n+1)}{2}\right) = O(n^2)$