Assignment 3

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.

|  |
| --- |
| Algorithm root()  if n = 0 then  return None  else  return S[1] |
| Algorithm parent(p)  if p = 1 then  return None  else  return S[floor(p/2)] |
| Algorithm leftChild(p)  if 2\*p <=n then  return S[2\*p]  else  return None |
| Algorithm rightChild()  if 2\*p+1 <= n then  return S[2\*p+1]  else  return None |
| Algorithm isInternal()  return leftChild != None or rightChild(p) != None |
| Algorithm isExternal()  return leftChild == None and rightChild(p) == None |
| Algorithm isRoot(p)  return p=0 |

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

1. **Draw a binary tree with height 5 and with the maximum number of external nodes.**

**Answer**

Print middle level of perfect binary tree without finding height -  GeeksforGeeks

1. **What is the minimum number of external nodes for a binary tree with height h? Justify your answer.**

Answer

2h-1/2 = 8

1. What is the maximum number of external nodes for a binary tree with height h? Justify your answer.

Answer

2h-1=16

1. Let T be a binary tree with height h and n nodes. Show that

log(n+1) -1 < h < (n-1)/2

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

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?

**Answer:**

|  |  |
| --- | --- |
| **Stack stack1, stack2**  **Algorithm *enqueue***(***o***)  **if stack1.*size***() = ***N*** **then**  **throw *FullQueueException***  **else**  ***stack1.push(o)*** | **O(1)**  **O(1)**  **O(1)**  **Amortized running time: O(1)** |
| **Algorithm *dequeue***()  **if *isEmpty***() **then**  **throw *EmptyQueueException***  **else**  **if stack2.isEmpty() then   while !stack1.isEmpty() do  stack2.push(stack1.pop())**  **return stack2.pop()** | **O(1)**  **O(1)**  **O(n)**  **O(n)**  **Total running time: O(n/2)** |

**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?**

Answer

|  |  |
| --- | --- |
| **Algorithm cardDeck**(L)  if L.isEmpty() then throw None  e := L.first()  **if stack1.*size***() = ***N*** **then**  **throw *FullQueueException***  **else**  ***stack1.push(o)*** | **O(1)**  **O(1)**  **O(1)**  **Amortized running time: O(1)** |
| **Algorithm *dequeue***()  **if *isEmpty***() **then**  **throw *EmptyQueueException***  **else**  **if stack2.isEmpty() then   while !stack1.isEmpty() do  stack2.push(stack1.pop())**  **return stack2.pop()** | **O(1)**  **O(1)**  **O(n)**  **O(n)**  **Total running time: O(n/2)** |