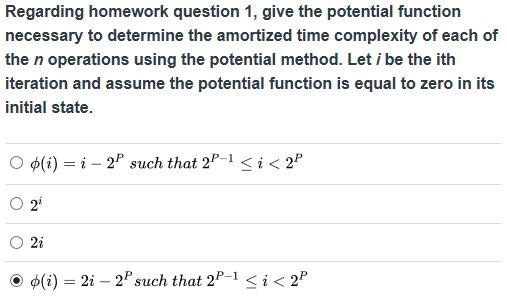
**[Q1] Prompt**: **A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. That is, operation i costs f(i), where f(i) = i, if i = 2k for some k ≥ 0, and f(i) = 1 otherwise. Determine the amortized cost per operation using the following methods of analysis:**

**(a) Aggregate method**

**(b) Accounting method**

**(c) Potential method**



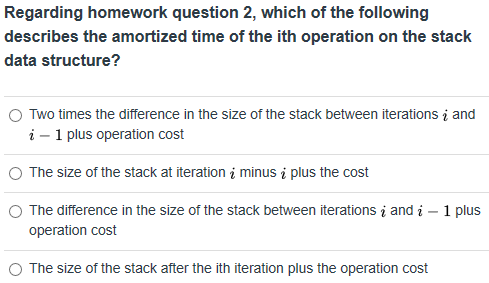
φ(i) = “bank account balance” if you think like accounting method

2p = next power of 2 so i is defined as being between subsequent powers of 2, *e.g*. between 2 and 4

if i = 3 then (i) – 2p = 2(7) – 2p which = 3 – 4 or –1 which is not allowed as it is a negative “balance” so first wrong

if i = 3 then 2(i) – 2p = 2(7) – 2p which = 6 – 4 or +2 which is a positive “balance” **so fourth answer is correct**

**[Q2] Prompt**: **Suppose that we have a stack S that supports the following operations: Push (S,x), which pushes an element x on top of the stack S, Pop (S), which removes the element at the top of stack S, and MultiPop (S,k), which removes the top k elements from the stack S (if we try to remove more elements than the current number of elements on the stack in a Pop or MultiPop operation, only the elements currently on the stack will be returned). Assume that the stack is initially empty. Give a potential function argument showing that the amortized running time of each of the three stack operations is Θ(1).**



<https://www.youtube.com/watch?v=CGEcK7ULnY8>

Aggregate Method = multipop = O(n) which dominates O(1) so ultimately O(n)/n = O(1)

Accounting Method = cannot pop w/o a push, so charge 2 for each push you cover the costs of pops and multipops

Potential Method = ai = ci + φ(Di) – φ(Di-1) and we know ci = 1 and s = size of Stack, so for push φ(Di) – φ(Di-1) = (s + 1) – s = 1 thus:

Push => ai = 1 + (s + 1) – s = 2 because s cancels and 1 + 1 = 2

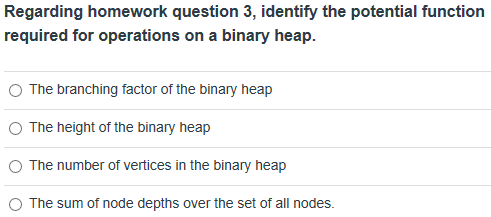
Pop => ai = 1 + (s – 1) – s = 0 because s cancels and 1 – 1 = 2

Multip-pop => ai = k’ + (s – k’) – s = 0 because both k’ and s cancels

k' = cost of multipop (s – k’) – s = Δφ which is the difference in the stack from before to after which = cost of multipop

**So the answer is the third selection**

**[Q3] Prompt**: **Consider an ordinary binary heap data structure with n elements that supports the instructions Insert and Extract-Min in O(log n) worst- case time. Give a potential function Φ such that the amortized cost of Insert is O(log n) and the amortized cost of Extract-Min is O(1), and show that it works.**



**A heap is a binary tree so like a splay tree but left and right children can be > or < parent as long as < root for max heap**

**At first it looks like “The height of binary tree” answer works, but**

On first page of splay tree amortized bounds pdf r(v) = rank of node v and s(v) = size of subtree rooted at node

φt(T) = Σ v є V(T): r(v) which equals φt(T) = Σ v є V(T): [floor]log(s(v))

so if we use depth instead of rank since heap not splay tree, where d(v) = depth of node v, we get

φ(t) = Σ v є V(T): d(v) ≤ n log n which is a more granular definition for an unbalanced tree then pure height, which means that **last “The sum of node depths over the set of all nodes” is the correct answer**

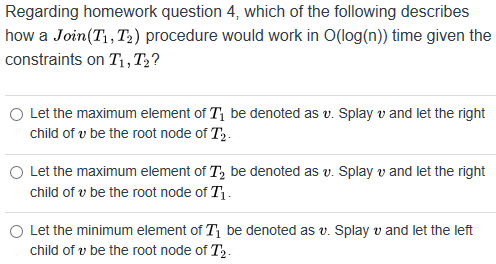
**[Q4] Prompt**: **The following binary search tree (BST) operations can be implemented efficiently on splay trees. For simplicity, assume that all elements are distinct.**

**Join (T1, T2): Assume every element in BST T1 is less than or equal to every element in BST T2, and returns the BST formed by combining the trees T1 and T2.**

**Split (T,a): Split BST T, containing element a, into two BSTs: T1, containing all elements in T with key less or equal to a; and T2, containing all elements in T with key greater than a.**

**Show how to correctly implement those operations on splay trees so that the amortized running time of your operations is O(log n). Justify your procedures and their corresponding running times.**

**Would you be able to also implement the Join and Split operations on a Red-Black tree in O(log n) worst-case running time? Briefly justify your answer.**

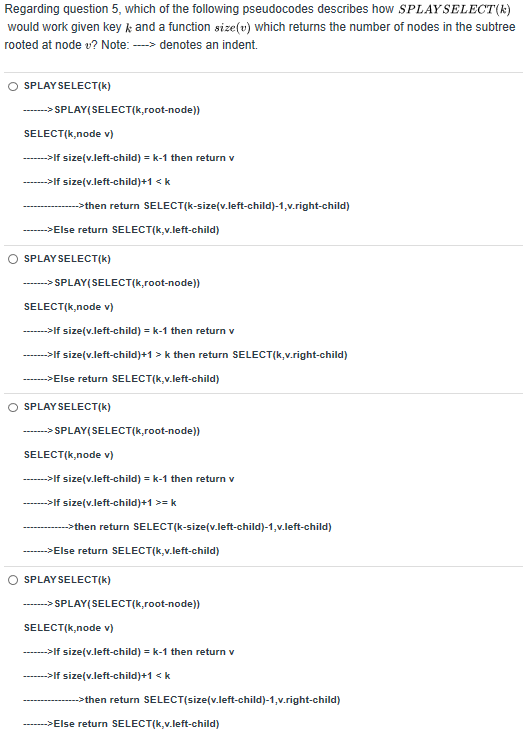
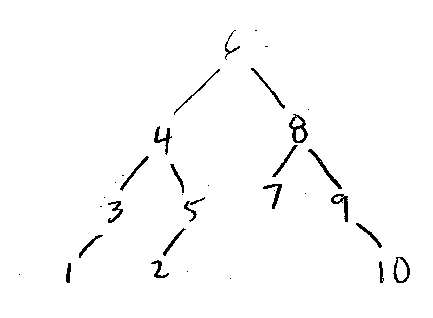


It is the **first answer** because v cannot have a right child so works for root when you add T2 as right child

**[Q5] Prompt**: **We define an augmented binary search tree to be a binary search tree where each node v in the tree also stores the value size[v], which is equal to the number of nodes in the subtree rooted at v.**

**(a)  Show that a rotation in an augmented binary tree can be performed in constant time.**

**(b)  Describe an algorithm SplaySelect (k), 1 ≤ k ≤ n, that selects the kth smallest item in an augmented splay tree with n nodes in O(log n) amortized time. Justify your algorithm and its running time.**



**I made a BST that had nodes 1-10 with 6 as the root then worked through algorithms:**

**1) SELECT(8, 6)**

**Is 5 = 7 NO**

**Is 5 < 8 YES**

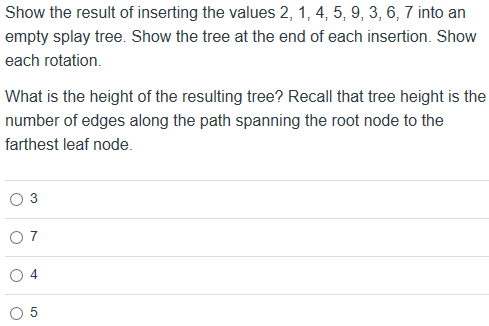
**SELECT(2, 8)**

**Is 1 = 1 YES**

**Return 8 The rest of work-throughs failed so answer is first option**

**[Q6]**

**Show the result of inserting the values 2, 1, 4, 5, 9, 3, 6, 7 into an empty splay tree. Show the tree at the end of each insertion. Show each rotation.**



Worked through twice and answer = 4