Week2

2. Let's imagine we add support to our dynamic array for a new operation PopBack (which removes the last element). PopBack will reallocate the dynamically-allocated array if the size is ≤ the capacity / 2 to a new array of half the capacity. So, for example, if, before a PopBack the size were 5 and the capacity were 8, then after the PopBack, the size would be 4 and the capacity would be 4.

Give an example of *n* operations starting from an empty array that require *O*(*n*2) copies.

Let *n* be a power of 2. Add *n*/2 elements, then alternate *n*/4 times between doing a PushBack of an element and a PopBack.

**Correct**

Once we have added *n*/2 elements, the dynamically-allocated array is full (size=*n*/2, capacity=*n*/2). When we add one element, we resize, and copy *n*/2 elements (now: size = *n*/2+1, capacity=*n*) When we then remove an element (with PopBack), we reallocate the dynamically allocated array and copy *n*/2 elements. So, each of the final *n*/2 operations costs *n*/2 copies, for a total of *n*2/4 moves, or *O*(*n*2)

3.

Let's imagine we add support to our dynamic array for a new operation PopBack (which removes the last element). Calling PopBack on an empty dynamic array is an error.

PopBack reallocates the dynamically-allocated array to a new array of half the capacity if the size is ≤ the capacity / 4 . So, for example, if, before a PopBack the size were 5 and the capacity were 8, then after the PopBack, the size would be 4 and the capacity would be 8. Only after two more PopBack when the size went down to 2 would the capacity go down to 4.

We want to consider the worst-case sequence of any *n* PushBack and PopBack operations, starting with an empty dynamic array.

What potential function would work best to show an amortized *O*(1) cost per operation?

Φ(*h*)=*max*(2×*size*−*capacity*,*capacity*/2−*size*)

**Correct**

This is a valid potential function since:

* When we start, size=capacity=0, so Φ(*h*0)=0
* Φ(*ht*)≥0 since, if size > capacity/2, the first term of the max is non-negative, and if size ≤ capacity/2, the second term of the max is non-negative.

The analysis of PushBack remains just as in lecture. The question is what happens when we do a PopBack. Amortized cost = true cost + Φ(*ht*)−Φ(*ht*−1)

* No resize needed: true cost = 1. If size > capacity/2, the change in Φ=Φ(*ht*)−Φ(*ht*−1)=2. If size ≤ capacity/2, the change in Φ=1. Max total amortized cost is 3.
* Resize needed: true cost = capacity/4 + 1. Φ(*ht*)=0,Φ(*ht*−1) = capacity/2 - (capacity/4 + 1) = capacity/4 - 1. Total amortized cost = capacity/4 + 1 - (capacity/4 - 1) = 2.

4. Imagine a stack with a new operation: PopMany which takes a parameter, *i*, that specifies how many elements to pop from the stack. The cost of this operation is *i*, the number of elements that need to be popped.

Without this new operation, the amortized cost of any operation in a sequence of stack operations (Push, Pop, Top) is *O*(1) since the true cost of each operation is *O*(1).

What is the amortized cost of any operation in a sequence of *n* stack operations (starting with an empty stack) that includes PopMany (choose the best answers)?

*O*(1) because we can place one token on each item in the stack when it is pushed. That token will pay for popping it off with a PopMany.

**Correct**

Correct. Add a token to each element on the stack as it is pushed. Then, on a PopMany, use those tokens to pay for the popping cost of each. Thus, the amortized cost is 2 which is *O*(1).

*O*(1) because the sum of the costs of all PopMany operations in a total of n operations is *O*(*n*).

**Correct**

Correct. Since over *n* operations starting with an empty stack there can be at most *n*items in the stack, the sum of the costs of the PopMany operations can be at most *n*. Thus, the total actual costs of *n* operations is at most *O*(*n*), so the amortized cost is *O*(*n*)/*n*=*O*(1).

Week3

Quiz 1

Assume that we represent a complete *d*-ary tree in an array *A*[1…*n*] (this is a 1-based array of size *n*). What is the right formula for the indices of children of a node number *i*?



{(*i*−1)*d*+2,…,(*i*−1)*d*+*d*+1}



{(*i*−1)*d*+2,…,min{*n*,(*i*−1)*d*+*d*+1}}

**Correct**



{*id*+2,…,min{*n*,*id*+*d*+1}}



{(*i*−1)*d*+1,…,min{*n*,(*i*−1)*d*+*d*}}

Week3

Quiz2

4. Consider the following program:

for i from 1 to 60:

MakeSet(i)

for i from 1 to 30:

Union(i, 2\*i)

for i from 1 to 20:

Union(i, 3\*i)

for i from 1 to 12:

Union(i, 5\*i)

for i from 1 to 60:

Find(i)

Assume that the disjoint sets data structure is implemented as disjoint trees with union by rank heuristic and with path compression heuristic.

Compute the maximum height of a tree in the resulting forest. (Recall that the height of a tree is the number of edges on a longest path from the root to a leaf. In particular, the height of a tree consisting of just one node is equal to 0.)



**Correct Response**

There is at least one tree of height 1 in the forest. Also, all trees have height at most 1, since the last for-loop calls Find() for all 60 elements. Since path compression is used, each non-root node will be attached directly to the corresponding root in this loop, and hence all the trees will have height at most 1.

2.

You've organized a party, and your new robot is going to meet and greet the guests. However, you need to program your robot to specify in which order to greet the guests. Of course, guests who came earlier should be greeted before those who came later. If several guests came at the same time or together, however, you want to greet first the older guests to show them your respect. You want to use a min-heap in the program to determine which guest to greet next. What should be the comparison operator of the min-heap in this case?

def GreetBefore(A, B):

if A.arrival\_time != B.arrival\_time:

return A.arrival\_time < B.arrival\_time

return A.age > B.age

3.

You want to implement a Disjoint Set Union data structure using both path compression and rank heuristics. You also want to store the size of each set to retrieve it in *O*(1). To do this, you've already created a class to store the nodes of DSU and implemented the *Find* method using the path compression heuristic. You now need to implement the *Union* method which will both use rank heuristics and update the size of the set. Which one is the correct implementation?

def Union(a, b):

pa = Find(a)

pb = Find(b)

if pa.rank <= pb.rank:

pa.parent = pb

pb.size += pa.size

if pa.rank == pb.rank:

pb.rank += 1

else:

pb.parent = pa

pa.size += pb.size

week5

1. Your colleague proposed a different definition of a binary search tree: it is such binary tree with keys in the nodes that for each node the key of its left child (if exists) is less than its key, and the key of its right child (if exists) is bigger than its key. Is this a good definition for a binary search tree?



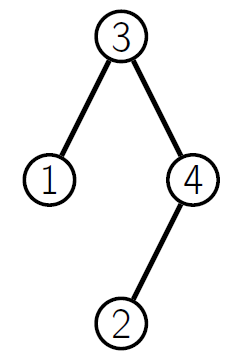
Yes



No

**Correct**

Correct! Binary search for a key in such tree might not work: try finding the key 2in the tree below. Your first move would be to go left from the root, because 2<3. However, the key 2 is in the right subtree of the root. The condition that the key of the left child of each node is less than the key of the parent, and the key of the right child is greater than the key of the parent is satisfied for all nodes in the tree. What's missing is that not just the key of the left child, but the keys of the whole left subtree of each node should be less than the key of the subtree root, and similarly for the right subtree.



3. Can the Insert operation be implemented given only Split and Merge operations?



Yes. First create a new tree with single key - the key to be inserted. Then merge current tree with the new tree.



No



Yes. First create a new tree with single key - the key to be inserted. Then split the current tree by this key. Then merge the left splitted part with the new tree. Then merge the result with the right splitted part.

4. Can the Delete operation be implemented given only Split and Merge operations?



Yes. Suppose we are deleting key *x*. Split by the key twice: one split such that all the keys <*x* go to the left left and all the keys ≥*x* go to the right. Then split the right part of the first split such that all the keys ≤*x* go to the left and all the keys >*x* go to the right. Then merge the left part of the first split and the right part of the second split - thus leaving out the node with key *x* if there was such a node.