ADS MiniProject

The Planet is Under Attack!

Pranav Ramanathan (230840106)

Enes Furkan Serin (220566933)

Hamza Mayou (230086575)

Table of Contents

ArrayListWithUndo

In all the methods mentioned below, we inherited the existing methods of `append(self, v)`, remove(self, i), `insert(self, i, v)` from the `ArrayList` class.

Insert Method

Here we call the insert method we inherited and then push the undo instruction triplet to the undo stack. The undo instruction is the remove method in this case.

Append Method

Here we call the append method and push the undo instruction triplet which is remove in this case and since append is adding to the end of the ArrayList we needed to find the index for the last element of the ArrayList to include in the triplet. We used the length method of ArrayList to achieve this.

Remove Method

Contrary to the two methods mentioned above, in the `remove` method we pushed the undo instruction to the stack before we called the inherited remove method. This way we preserved the value at the index we are removing from the ArrayList in order to be able to push the undo instruction triplet with the value.

Undo Method

Here, we unpack the popped triplet (the last triplet pushed) from the `undos` stack. Using the undo instruction we either set, remove, or insert using the values and indexes in the triplet we popped. Since there are only three cases, the default case would be insert. If the stack of `undos` is empty, it does nothing.

NetworkWithUndo

Add Method

To add we create a new cluster which is done by adding the value -1 in the next position, showing that is the root node of the cluster with length 1.

Root Method

First, we go through the nodes of a cluster by checking what the value at the given index is then going to the index that the value represents until we reach a negative value (i.e. a root node). We save the nodes we visit in a stack and then go through the stack to change the

corresponding value of each index in the stack to the index of the root node. By doing so we make every non root node point directly to the root of the cluster (i.e. apply path flattening to the cluster). We push the size of the stack we created to the undos stack since that is the number of operations we performed on the inArray of the network.

Merge Method

First, we calculate the new length of the merged cluster which will be the sum of the values stored in the indexes of the root nodes of the clusters. To make the program more efficient we merge the clusters by making the root node of the shorter cluster point to the root node of the other. We push the integer 2 onto the undos stack since we are using the set method twice on the inArray of the network.

Undo Method

We implemented the undo method by leveraging the fact that the inArray of the network is an ArrayListWithUndo. This design allows us to track the number of operations performed in each function and push that count onto the undo stack. To perform an undo on the network, we pop and integer from the stack and then call the undo method of ArrayListWithUndo on the inArray of the network that many times. This effectively reverses the last operation performed on the network.

Gadget

Add Method

We first check to see if the name is already in Gadget and if it is we do not alter the gadget. Otherwise, after mapping an integer that's the size of the gadget to the name we add the new node to the inner Network in the Gadget.

Connect Method

First, we find the nodes corresponding to the keys. Then, we checked if the roots of the two nodes were the same using the root of the inner network and returned true if they are. Otherwise, we merge the two nodes by merging the nodes in the inner network.

Clean function

We first check if the name exists using `isIn()` and exits if it doesn't. It calculates the required undo steps as the difference between the current undo stack size and the recorded size when the node was added. Finally, it calls `self.undo()` with this value to fully revert the Gadget to before the name was added.

Subnets function

The `subnets()` method simply asks us to return an array of string arrays (which represent clusters). To implement this, we grouped the nodes by cluster by creating a dictionary, which has roots of the clusters as keys and all the nodes in the cluster as values. We simply iterated through the keys in `self.nameMap` and assigned each corresponding node to their key-value pair in the dictionary, by checking the root node of the cluster that they are in.

Undo function

To implement the undo method, with the parameter representing the number of undo operations, by using the unpacked triplet popped from the undos stack. This triplet tells us the operation we need to perform, and we adjust the gadget accordingly. Depending on the number of steps given to us by the triplet, we perform undo on the inner network. We do this repeatedly with a while loop, until we either do the amount of undo operations or until we get an empty gadget.

Code Implementation

```
class ArrayListWithUndo(ArrayList):
  def __init__(self):
    super().__init__()
    self.undos = Stack()

  def set(self, i, v):
    self.undos.push(("set", i, self.inArray[i]))
    self.inArray[i] = v

  def append(self, v):
    super().append(v)
    self.undos.push(("rem", self.length()-1, None))

  def insert(self, i, v):
    super().insert(i, v)
    self.undos.push(("rem", i, None))

  def remove(self, i):
    self.undos.push(("ins", i, self.inArray[i]))
    super().remove(i)

  def undo(self):
    if self.undos.size == 0:
      return
    instruction, idx, val = self.undos.pop()
    if instruction == "set":
      self.inArray[idx] = val
    elif instruction == "rem":
      super().remove(idx)
    else:
      super().insert(idx, val)
  def __str__(self):
    return str(self.toArray())+"\n-> "+str(self.undos)

class NetworkWithUndo:
  def __init__(self, N):
    self.inArray = ArrayListWithUndo()
    for _ in range(N): self.inArray.append(-1)
    self.undos = Stack()
    self.undos.push(N)

  def getSize(self):
```

```
  return self.inArray.length()

def add(self):
  self.inArray.append(-1)
  self.undos.push(1)
  Return

def root(self, i):
  indexToCheck = i
  visitedNodes = Stack()
  while self.inArray.get(indexToCheck) >= 0:
    visitedNodes.push(indexToCheck)
    indexToCheck = self.inArray.get(indexToCheck)
  rootIdx = indexToCheck
  self.undos.push(visitedNodes.size)
  while visitedNodes.size > 0:
    currentIndex = visitedNodes.pop()
    self.inArray.set(currentIndex, rootIdx)
  return rootIdx

def merge(self, i, j):
  iLength = self.inArray.get(i)
  jLength = self.inArray.get(j)
  newLength = iLength + jLength
  if iLength < jLength:
    self.inArray.set(i, newLength)
    self.inArray.set(j, i)
  else:
    self.inArray.set(j, newLength)
    self.inArray.set(i, j)
  self.undos.push(2)
  Return

def undo(self):
  if self.undos.size > 0:
    numberOfOperations = self.undos.pop()
    for _ in range(numberOfOperations):
      self.inArray.undo()
    Return

def toArray(self):
  return self.inArray.toArray()

def __str__(self):
```

```
    return str(self.toArray())+"\n-> "+str(self.undos)

class Gadget:
  def __init__(self):
    self.inNetwork = NetworkWithUndo(0)
    self.subsize = 0
    self.nameMap = {}
    self.undos = Stack()
    self.helper = {}

  def getSize(self):
    return self.inNetwork.getSize()

  def isIn(self, name):
    return name in self.nameMap

  def add(self, name):
    if self.isIn(name):
      self.undos.push(("oth", 0, None))
      return
    else:
      self.nameMap[name] = self.getSize()
      self.helper[name] = self.undos.size
      self.subsize += 1
      self.inNetwork.add()
      self.undos.push(("rem", 1, name))

  def connect(self, name1, name2):
    node1 = self.nameMap[name1]
    node2 = self.nameMap[name2]
    rootNode1 = self.inNetwork.root(node1)
    rootNode2 = self.inNetwork.root(node2)
    if rootNode1 == rootNode2:
      self.undos.push(("oth", 2, None))
      return True
    else:
      self.subsize -= 1
      self.inNetwork.merge(rootNode1, rootNode2)
      self.undos.push(("brk", 3, None))
      return False

  def clean(self, name):
    if not self.isIn(name):
      return
```

```
    numUndos = self.undos.size - self.helper[name]
    self.undo(numUndos)

  def subnets(self):
    arr = []
    clusters = {}
    for string in self.nameMap.keys():
      index = self.nameMap[string]
      rootIdx = self.inNetwork.root(index)
      if rootIdx not in clusters:
        clusters[rootIdx] = []
      clusters[rootIdx].append(string)
      for arrayOfStrings in clusters.values():
        arr.append(arrayOfStrings)
      self.undos.push(("oth", len(self.nameMap), None))
      return arr

  def undo(self, n):
    while n != 0 and self.getSize() != 0:
      op, numberOfSteps, s = self.undos.pop()
      if op == "rem":
        self.nameMap.pop(s)
        self.helper.pop(s)
        self.subsize -= 1
      elif op == "brk":
        self.subsize += 1
      for _ in range(numberOfSteps):
        self.inNetwork.undo()
      n -= 1
    return
  def toArray(self):
    A = self.inNetwork.toArray()
    for s in self.nameMap:
       i = self.nameMap[s]
       A[i] = (s,A[i])
    return A
  def __str__(self):
    return str(self.toArray())+"\n-> "+str(self.nameMap)+"\n->
"+str(self.undos)
```

Contributions Section

Pranav

I implemented the clean function in the Gadget class. Additionally, I fixed the undo function and corrected the add method to push the right triple onto the stack when both root nodes were the same. I also corrected the root method in NetworkWithUndo by fixing the checking logic. Furthermore, I fixed the subnets method to use an array instead of an Array List.

Enes

I implemented the subnets, add, and undo functions in the Gadget class. I also developed the undo method to revert operations when necessary. Additionally, I fixed the subnets method to use arrays, implemented the undo functionality, corrected comments in the add method, and handled root and merge operations within the NetworkWithUndo class.

Hamza

I implemented the connect function in the Gadget class. I also managed the undo stack to allow rollback of connections. Furthermore, I fixed issues in ArrayListWithUndo and NetworkWithUndo by using the correct syntax for getting and setting elements from ArrayLists.

Collaborative Efforts

Together, we implemented the NetworkWithUndo and ArrayListWithUndo classes through live coding sessions. We all contributed to writing the report, accurately describing each functionality, and refactoring each other's sections to ensure clarity and consistency.