ALGORITHMS AND DATA STRUCTURES

Mini Project by Naif Alaqeili, Soud Alaqeili, and Yousef Kharrat

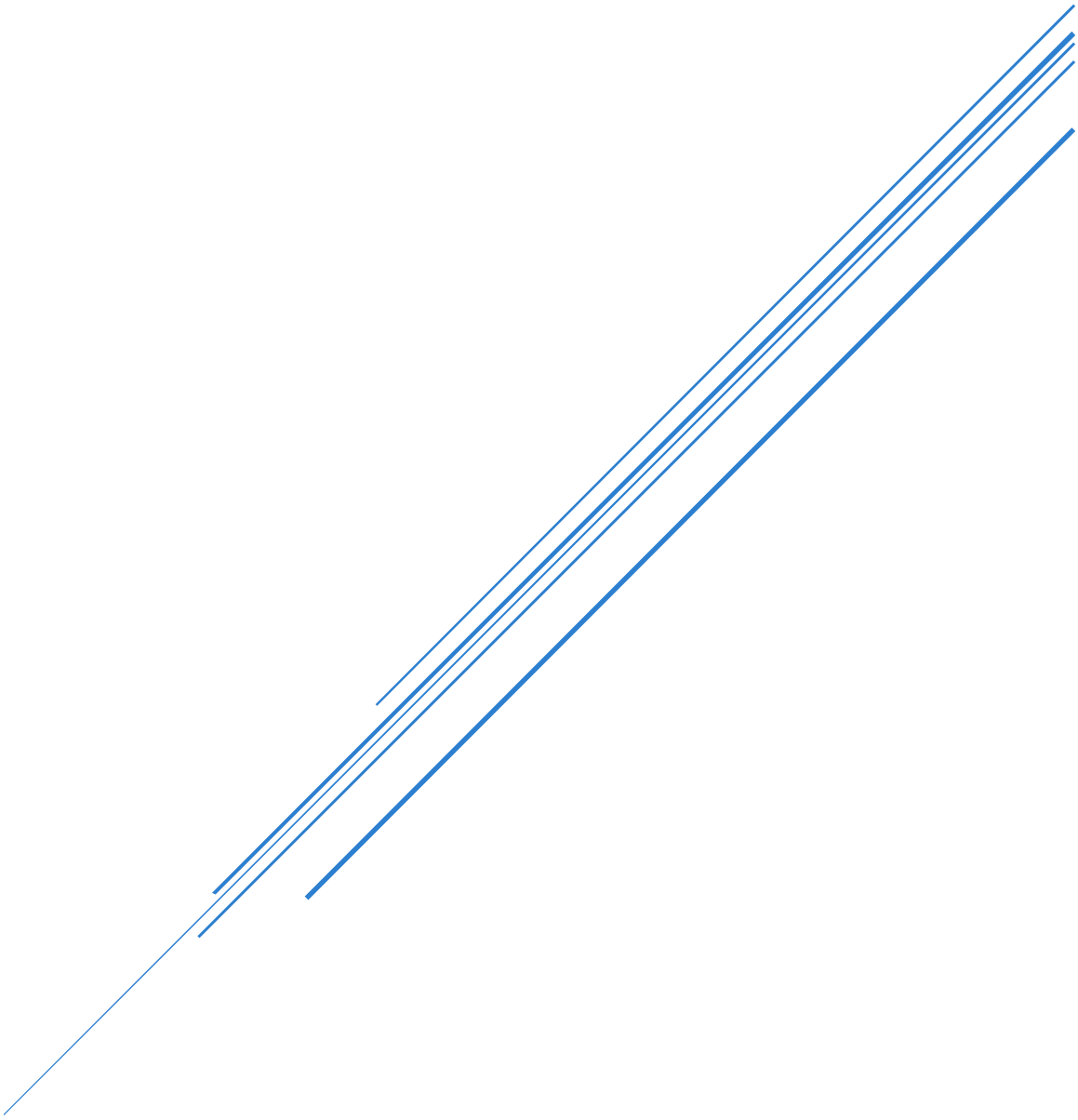

Table of Contents

Group Member Contributions

Youssef Kharat

I completed the first question which was about implementing the ArrayListWithUndo class. For my section, I had to implement 4 methods: append, remove, insert, and undo. This question was crucial since the class I would implement would act as a base for the NetworkWithUndo class, which another group member was going to work on. Implementing the class was quite simple as it only required using the basic methods from the given ArrayList class and adding the functionality of undoing operations. I made sure to complete my part as quickly as possible in order to give time to the rest of my team to build upon what I had written. Since this section was not as large, I chose to also help out with Soud's and Naif's classes as they were more complex and time consuming.

Soud Alaqeili

My contribution was completing the NetworkWithUndo class.To do this, I had to implement 4 methods: add, merge, root, and undo. Regarding difficulty, I found the undo and add methods quite straightforward but I struggled with the merge and root functions. Specifically, it took me some time to figure out how to determine if a node was a root node and also if two root nodes had the same cluster size, which should be merged into the other. Counting the number of visited notes in the root also required excessive testing and debugging but worked well in the end. I was able to complete this class in a reasonable amount of time so that Naif could finish the third and final class.

Naif Alaqeili

The part of the mini project I was working on was the Gadget class. A worrying aspect for me was that I needed to wait for my other group members to finish their respective classes so that I could implement mine which could have meant not having enough time to implement and test effectively. However, the deadline provided sufficient time and my group members and I had a set time line that we agreed upon. For the gadget class, I needed to implement the methods: undo, subnets, clean, connect, and add. My main challenge was figuring out the undo operations and how to sync them correctly with the NetworkWithUndo class. The clean method was also quite tricky but I was able to come up with a innovative solution quite fast. With the help of my other group members, we were able to figure out the issues with the undo operations, and I finished implementing the class fully.

Code Explanation

Below we have described the methods we have implemented. For clarity, we have divided them by their respective questions.

Question 1:

append(self, v):

- Adds an element v to the end of the list.
- Prepares an "undo instruction" that, if needed, can reverse this operation by removing the last element.
- Pushes the undo instruction onto a stack for later retrieval.

insert(self, i, v):

- Inserts an element v at the specified index i.
- Creates an undo instruction to reverse this operation by removing the inserted element from the same index.
- Stores the undo instruction on the stack.

remove(self, i):

- Removes an element at index i.
- First retrieves the value of the element being removed so it can be reinserted during an undo operation.
- Prepares an undo instruction to reverse the removal by reinserting the removed value at the same index.
- Pushes this undo instruction onto the stack.

undo(self):

- Reverses the most recent modification to the list using the latest undo instruction from the stack.
- Checks if there are any undo instructions available; if none, it does nothing.
- Depending on the type of operation (set, rem, or ins), it either:
    - Restores a previous value at an index (set).
    - Removes a value at an index (rem).
    - Inserts a previously removed value back at an index (ins).

Question 2:

add(self)

- Adds a new node to the network, initially forming its own cluster.
- We use a variable, num_operations, to track the number of operations done in the add method, in this case 1.

- Pushes num_operations(1 in this case) onto the undo stack so that we know how many operations we need to reverse

root(self, i)

- Finds the root of a node i while keeping track of all visited nodes by storing them in a list
- Applies path compression: for each visited node, directly points it to the root
- Tracks the number of set operations performed during path compression and pushes this number on the undo stack so that we know how many set operations to undo
- Returns the root node's index.

merge(self, i, j)

- Merges two clusters represented by their root nodes.
- Checks if both i and j are root nodes, and asserts 0 if they aren't. We check if a node is a root node if the value at the node is <=0
- Retrieves the sizes of the clusters for both roots and merges them by pointing the smaller cluster's root to the larger cluster's root
- Updates the size of the new root cluster after merging.
- Tracks and counts the number of set operations performed during the merge and pushes this count onto the undo stack for undoing.

undo(self)

- Reverses the last operation(s) recorded on the undo stack.
- Retrieves the number of operations (num_operations) to undo. It does this by popping the top of the stack which will have the latest number of operations performed by one of the other mthods
- Uses the undo method in the ArrayListWithUndo class implemented earlier to undo the operations

Question 3:

add(self, name)

- Adds a new node (name) to the network.
- If the name already exists, it simply returns. We also push an oth operation with steps 0 to the undos stack
- Maps the name to a unique index using nameMap and increments the network subsize by 1
- We used the helper dictionary in order to implement the clean () functionality
    - When a name is added, the helper stores the name as well as the number of operations on the stack at the time the name was added
    - We can use this to determine how many operations we have to undo to revert back to before the name was added

- We also push a rem operation to the undos stack so that the adding of the name can be undone

connect(self, name1, name2)

- Connects two nodes (name1 and name2) in the network.
- As a precaution, we check if both names are in the network using the provided isIn() method and assert an error if not
- Then we find the root of each node using the root method defined in NetworkWithUndo
- Next we check if the roots are equal to see gf they are already connected
    - If they are, we push an oth operation to the stack with 2 steps to account for the two root calls
- Otherwise we merge the two clusters, and decrement the subsize by 1
    - We also push brk operation with 3 steps (two root calls + merge) on the undo stack.

clean(self, name)

- Removes all operations associated with a specific node (name) from the network.
- To calculates the number of undo steps required to revert back to before a name was added, we can use helper dictionary
    - The helper dictionary stores how many operations were on the stack when the name was added. To get the number of operations that need to be undone, we can just subtract the current number of operations with the number of operations in helper
- Calls undo method to reverse all steps since the node was added.

subnets(self)

- Gathers and returns the connected components (sub-networks) as lists of names.
- Traverses nameMap, finds the root for each name, and groups names by their root index.
- Tracks the number of root operations performed and pushes an oth operation to the undo stack

undo(self, n)

- Undoes the last n recorded operations.
- Pops operations from the undos stack, processing based on their type:
    - **rem**: Deletes the node with the name and decrements subsize by 1
    - **brk**: Reverts a merge operation and decrements subsize by 1
    - **oth**: No action needed since these were internal network changes
- For each operation,we get the number of steps, and undo the same number of steps on the network using the undo method in NetworkWithUndo.

Code Implementation:

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

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

    def append(self, v):
        #undo instruction for append is remove
        undo_instruction = ("rem", self.count, None)
        #push undo instruction to stack
        self.undos.push(undo_instruction)
        #append the element
        super().append(v)

    def insert(self, i, v):
        #undo operation for insert is remove
        undo_instruction = ("rem", i, None)
        #push undo instruction to stack
        self.undos.push(undo_instruction)
        #insert the element
        super().insert(i, v)

    def remove(self, i):
        #get the value of the element to be removed
        removed_value = self.get(i)
        #undo instruction for remove is insert
        undo_instruction = ("ins", i, removed_value)
        #push undo instruction to stack
        self.undos.push(undo_instruction)
        #remove the element
        super().remove(i)

    def undo(self):
        #if the stack is empty, return
        if self.undos.size == 0:
            return
        #get the latest undo instruction
        undo_instruction = self.undos.pop()
```

```
        op, i, v = undo_instruction
        #if the undo operation is set
        if op == "set":
            super().set(i, v)
        #if the undo operation is remove
        elif op == "rem":
            super().remove(i)
        #if the undo operation is insert
        elif op == "ins":
            super().insert(i, v)


    def __str__(self):
        # already implemented
        return str(self.toArray())+"\n-> "+str(self.undos)

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


        def getSize(self):
            # already implemented
            return self.inArray.length()

        def add(self):
            #keep track of number of operations
            num_operations = 0
            # Add a new node to the network as its own cluster
            self.inArray.append(-1)
            #since we did one append operation, increment num_operations by 1
            num_operations += 1
            # Push the number of operations onto the undo stack
            self.undos.push(num_operations)

        def root(self, i):

            #store visited nodes
            visited = []
            #continue iterating until root node met
            while self.inArray.get(i) >= 0:
                visited.append(i)
```

```
            i = self.inArray.get(i)



        #for every node visited, make it point to root
        #keep track of number of set operations well do
        num_operations = 0
        for node in visited:
            #make node point to root and increment num_operations
            if i!=self.inArray.get(node):
                self.inArray.set(node, i)
                num_operations += 1

        #push number of set operations to stack
        self.undos.push(num_operations)

        return i


    def merge(self, i, j):

        #if both nodes are not root nodes, assert 0
        if self.inArray.get(i) >= 0 or self.inArray.get(j) >= 0:
            assert(0)  # One or both are not roots

        # Get sizes of the clusters i and j
        size_i = -self.inArray.get(i)
        size_j = -self.inArray.get(j)

        #keep track of number of operations
        num_operations = 0

        #if i cluster is bigger than j cluster, make root j point to root i
        if size_i > size_j:
            # Make root_j point to root_i
            self.inArray.set(j, i)
            #increment number of operations
            num_operations += 1

            # update size at root i
            self.inArray.set(i, -(size_i + size_j) )
            #increment number of operations
            num_operations += 1
        else:
            # Make root_i point to root_j
            self.inArray.set(i, j)
            #increment number of operations
            num_operations += 1
```

```
                # Update size at root j
                self.inArray.set(j, -(size_i + size_j))
                #increment number of operations
                num_operations += 1

            # Push number of operations performed in merge onto the undo stack
            self.undos.push(num_operations)



        def undo(self):
            #if empty stack, return
            if self.undos.size == 0:
                return

            # Pop the number of operations to undo
            num_operations = self.undos.pop()
            for i in range(num_operations):
                self.inArray.undo()

        def toArray(self):
            # already implemented
            return self.inArray.toArray()

        def __str__(self):
            # already implemented
            return str(self.toArray())+"\n-> "+str(self.undos)

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

        def getSize(self):
            # already implemented
            return self.inNetwork.getSize()

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

        def add(self, name):
            if self.isIn(name):
```

```
                self.undos.push(("oth", 0, None))
                return
            # Add the name to the nameMap and track its index
            index = self.inNetwork.getSize()
            self.nameMap[name] = index
            self.inNetwork.add()
            self.subsize += 1

            # Record number of operations so far in helper and push rem operation
undos stack
            self.helper[name] = self.undos.size
            self.undos.push(("rem", 1, name))

        def connect(self, name1, name2):
            if not self.isIn(name1) or not self.isIn(name2):
                assert("One or both of the names is not in the nameMap")

            index1 = self.nameMap[name1]
            index2 = self.nameMap[name2]

            # Check if already connected
            root1 = self.inNetwork.root(index1)
            root2 = self.inNetwork.root(index2)
            if root1 == root2:
                #if its already connected, push oth with steps=2 to account for the
two root operations we did
                self.undos.push(("oth", 2, None))
                return True

            # Merge clusters and update subsize
            self.inNetwork.merge(root1, root2)
            self.subsize -= 1
            #push brk with steps=3 since we merged and did two root operations
            self.undos.push(("brk", 3, None))
            return False

        def clean(self, name):
            if not self.isIn(name):
                assert("Name is not in the nameMap")

            # Get the number of undo steps required
            undo_steps = self.undos.size - self.helper[name]
            self.undo(undo_steps)

        def subnets(self):
            subnet_map = {}
            operations_performed = 0  # Track the number of operations performed on
the network
```

```
            for name, index in self.nameMap.items():
                root = self.inNetwork.root(index)  # This modifies the internal
network state

                operations_performed += 1  # Increment for each root call
                if root not in subnet_map:
                    subnet_map[root] = []
                subnet_map[root].append(name)


            #push othe with steps number of operations performed to account for number
of root operations we did
            self.undos.push(("oth", operations_performed, None))

            return list(subnet_map.values())


        def undo(self, n):

            while n > 0 and self.undos.size > 0:
                op, steps, name = self.undos.pop()

                if op == "rem":
                    # Remove the name and update subsize
                    del self.nameMap[name]
                    self.subsize -= 1
                elif op == "brk":
                    # Revert a connection and update subsize
                    self.subsize += 1

                #if the op is oth, we don't need to do anything

                #for the number of steps for each operation, do an undo operation in
the network
                for i in range(steps):
                    self.inNetwork.undo()

                n -= 1


        def toArray(self):
            # already implemented
            A = self.inNetwork.toArray()

            for s in self.nameMap:
                i = self.nameMap[s]
                A[i] = (s,A[i])
```

```
            return A

        def __str__(self):
            # already implemented
            return str(self.toArray())+"\n-> "+str(self.nameMap)+"\n->
"+str(self.undos)
```