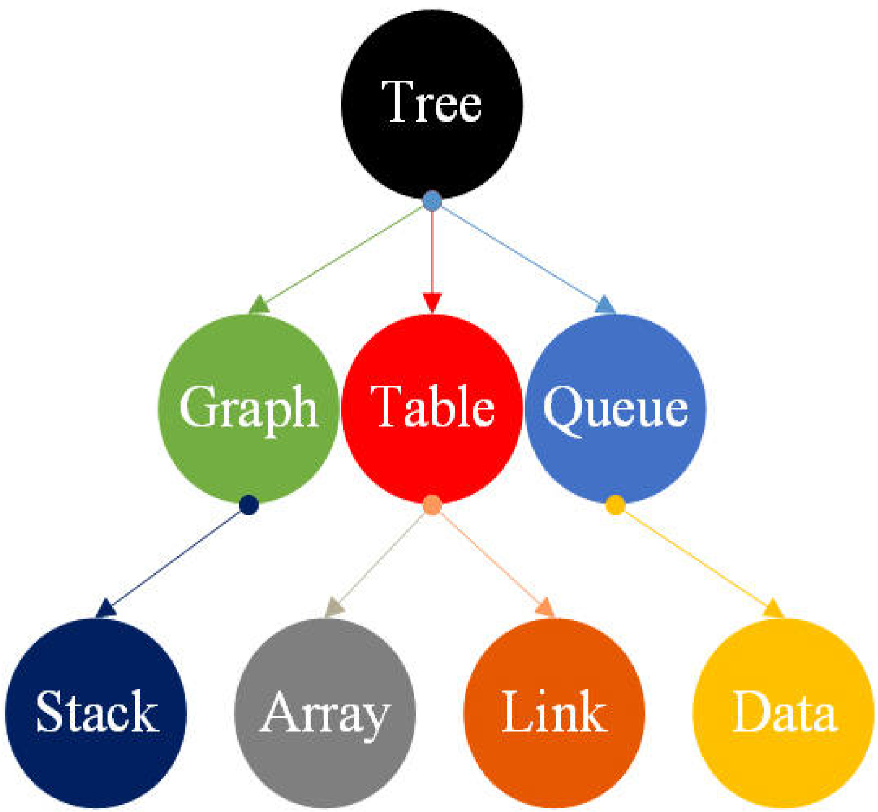
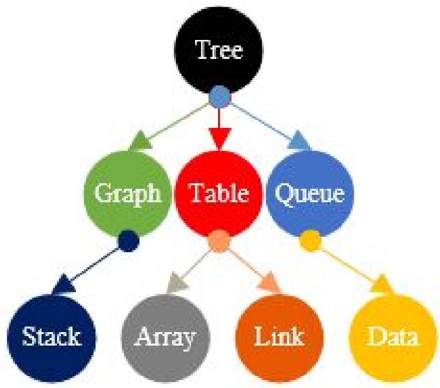
Easy Learning Data Structures Algorithms Python 3



Python 3 Data Structures Algorithms

Easy Learning
Data Structures Algorithms
Python 3



YANG HU

Simple is the beginning of wisdom. From the essence of practice, this book to briefly explain the concept and vividly cultivate programming interest,you will learn it easy, fast and well.

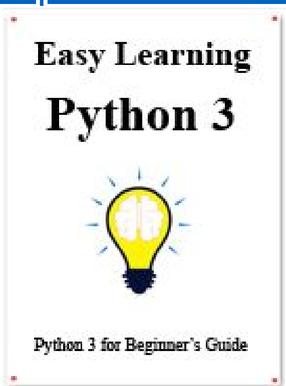
http://en.verejava.com

ISBN: 9781098735487

CONTENTS

If you want to learn this book, you must have basic knowledge of Python, you can learn book: << Easy Learning Python 3>>

https://www.amazon.com/dp/1092328122



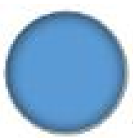
If you already have basic knowledge of Python, skip it, start an exciting journey

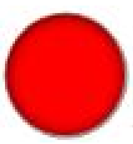
Bubble Sorting Algorithm

Bubble Sorting Algorithm:
Compare arrays[j] with arrays[j + 1], if arrays[j] > arrays[j + 1] are exchanged.
Remaining elements repeat this process, until sorting is completed.

Sort the following numbers from small to large

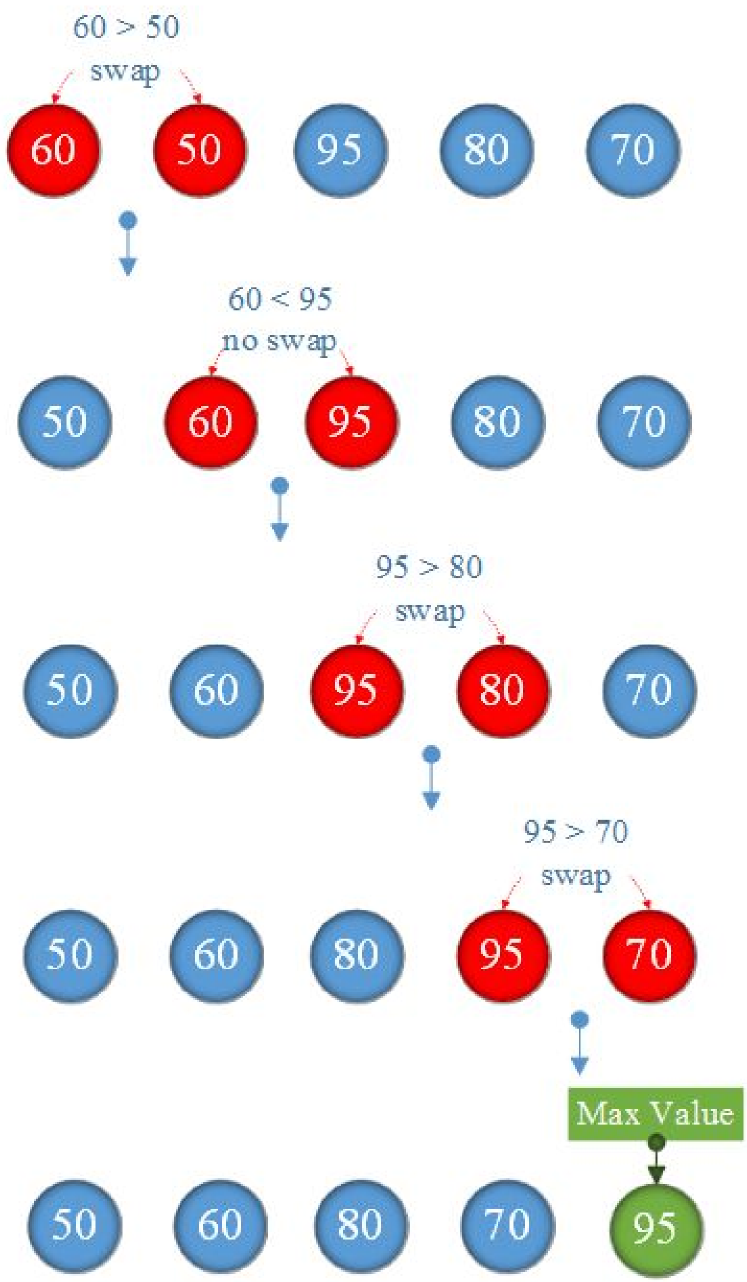
Explanation:


No sorting,

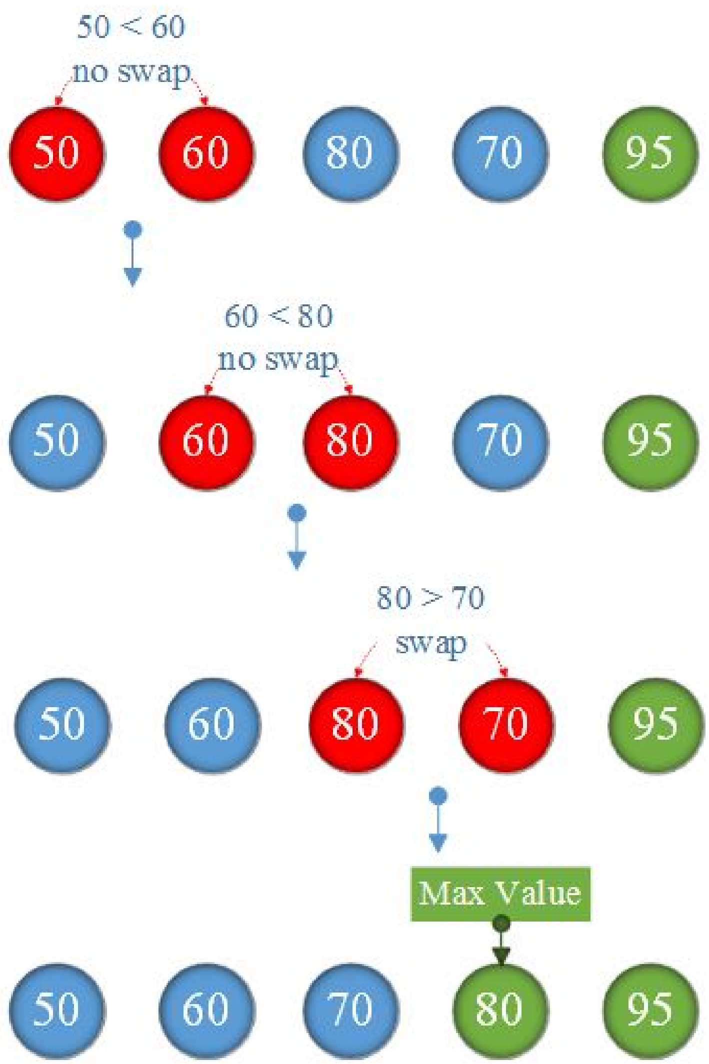

Comparing,


Already sorted

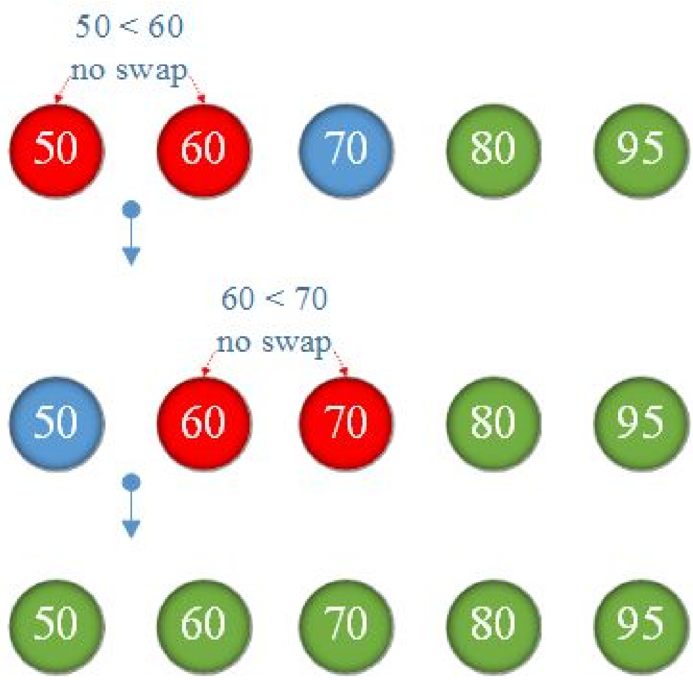
1. **First sorting:**



2. Second sorting:



3. **Third sorting:**



No swap so terminate sorting : we can get the sorting numbers from small to large

Create Test File : test_bubble_sort.py

```
from array import array

def sort(arrays):
    length = len(arrays)
    for i in range(0, length-1):
        for j in range(0, length-i-1):
            if arrays[j] > arrays[j+1]:
                flag = arrays[j]
                arrays[j] = arrays[j+1]
                arrays[j+1] = flag

def main():
    scores = array('i') # Create an int type array
    scores = array( 'i', [60, 50, 95, 80, 70 ])

    sort(scores)

    for score in scores:
        print(score, ",", end="")

if __name__ == "__main__":
    main()
```

Result:

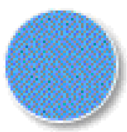
50 ,60 ,70 ,80 ,95 ,

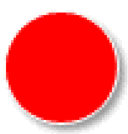
Select Sorting Algorithm

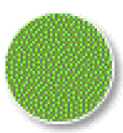
Select Sorting Algorithm:
Sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning.

Sort the following numbers from small to large

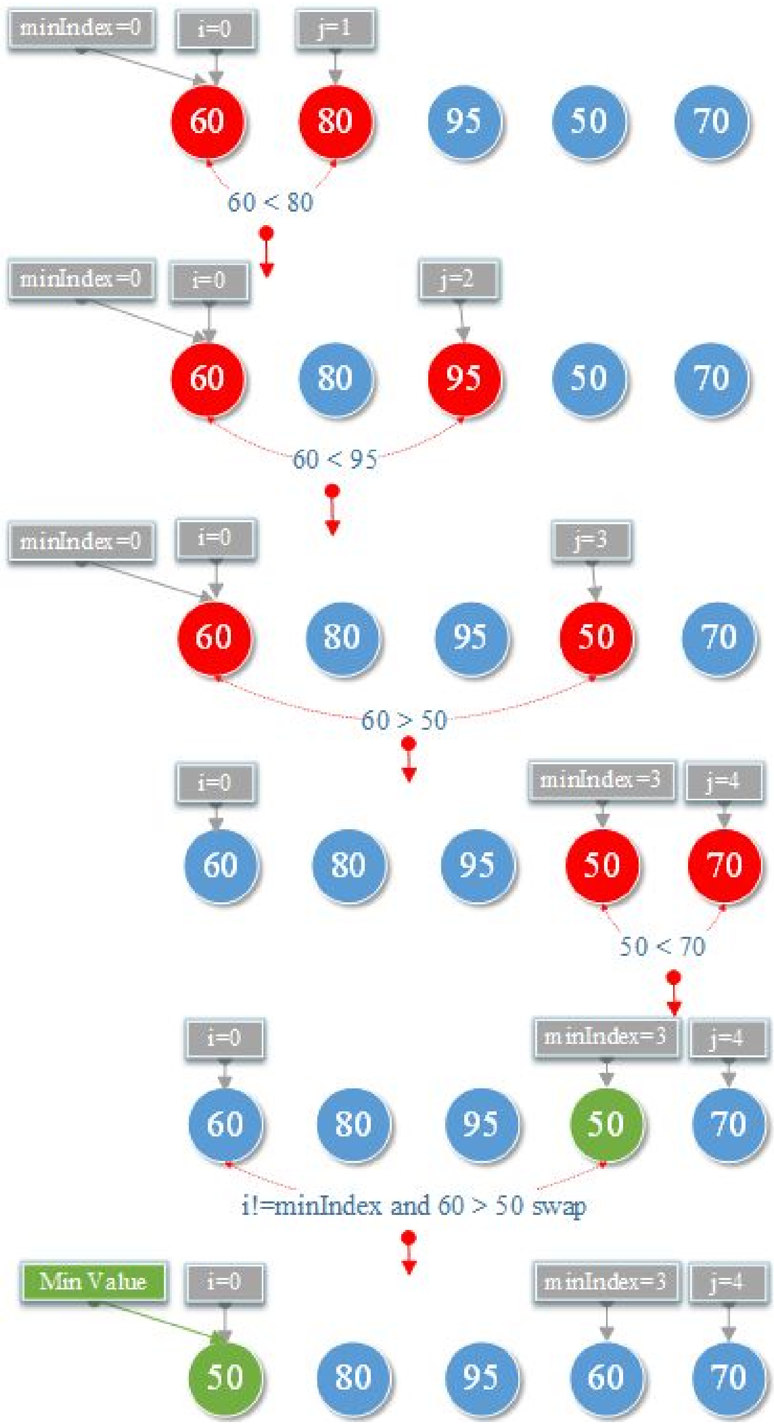
Explanation:

 No sorting,

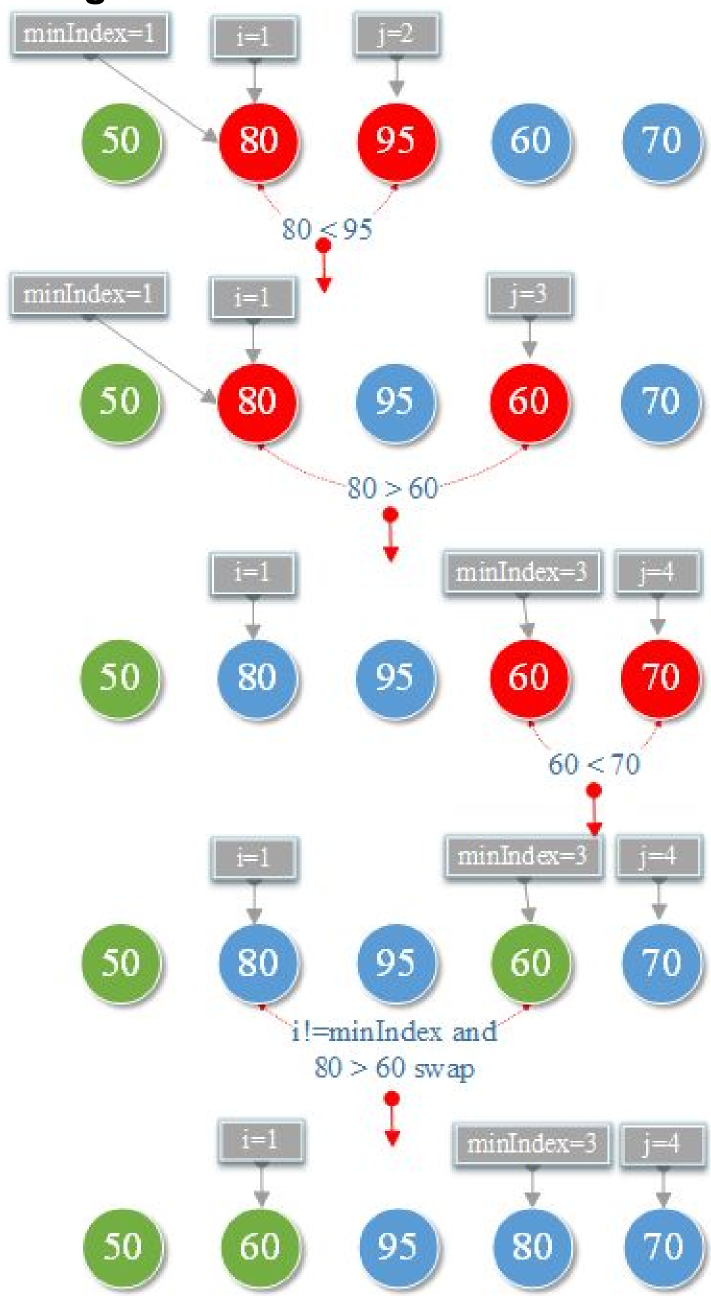
 Comparing,

 Already sorted.

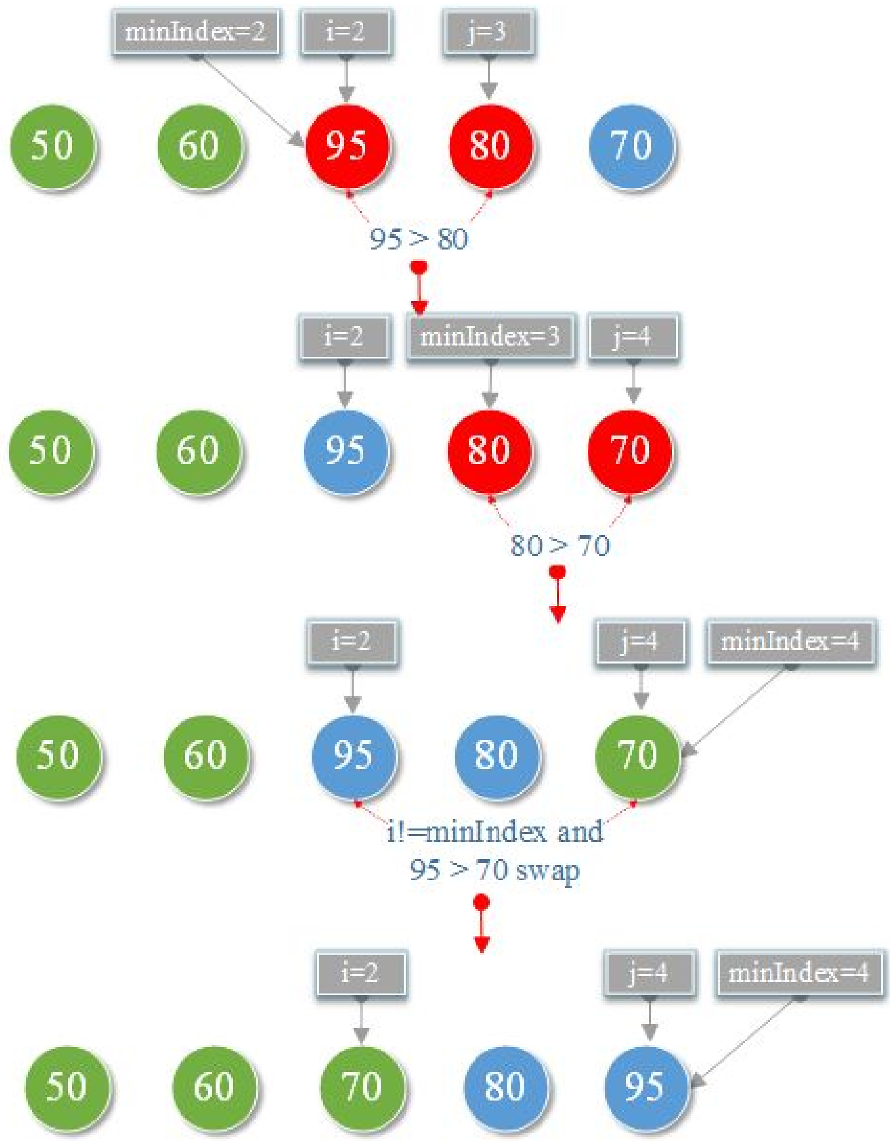
1. First sorting:



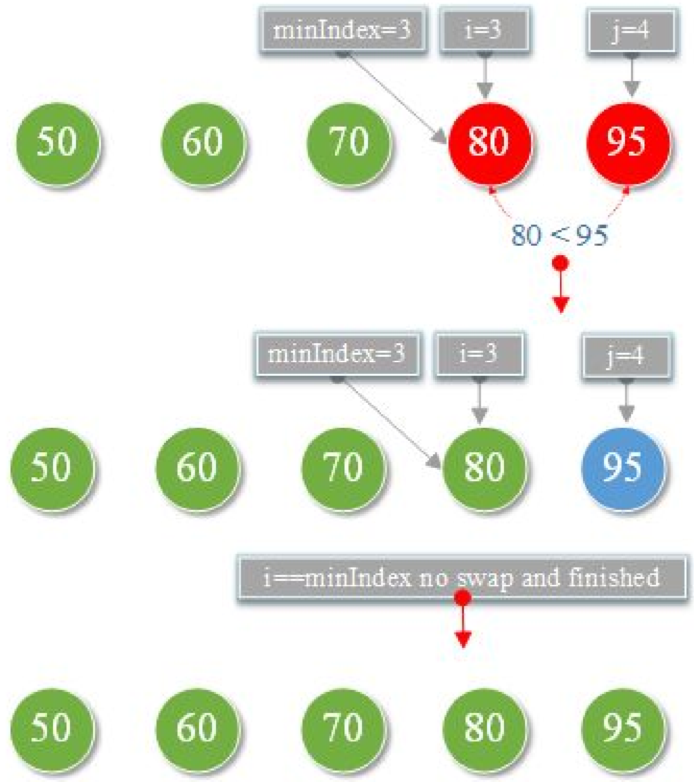
2. **Second sorting:**



3. Third sorting:



4. Forth sorting:



we can get the sorting numbers from small to large

test_select_sort.py

```
from array import array

def sort(arrays):
    length = len(arrays) - 1
    min_index = 0 # the index of the selected minimum

    for i in range(0, length):
        min_index = i
        # the minimum value of each loop as the first element
        min_value = arrays[min_index]
        for j in range(i, length):
            # Compare with each element if it is less than the minimum
value, exchange the min_index
            if min_value > arrays[j + 1]:
                min_value = arrays[j + 1]
                min_index = j + 1

        # if the minimum index changes, the current minimum is
exchanged with the min_index
        if i != min_index:
            temp = arrays[i]
            arrays[i] = arrays[min_index]
            arrays[min_index] = temp

def main():
    scores = array('i') # Create an int type array
    scores = array( 'i',[60, 50, 95, 80, 70 ])
    sort(scores)
    for score in scores:
        print(score, ",", end="")

if __name__ == "__main__":
```

```
    main()
```

Result:

50 ,60 ,70 ,80 ,95 ,

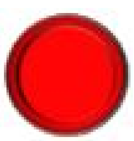
Insert Sorting Algorithm

Insert Sorting Algorithm:
Take an unsorted new element in the array, compare it with the already sorted element before, if the element is smaller than the sorted element, insert new element to the right position.

Sort the following numbers from small to large

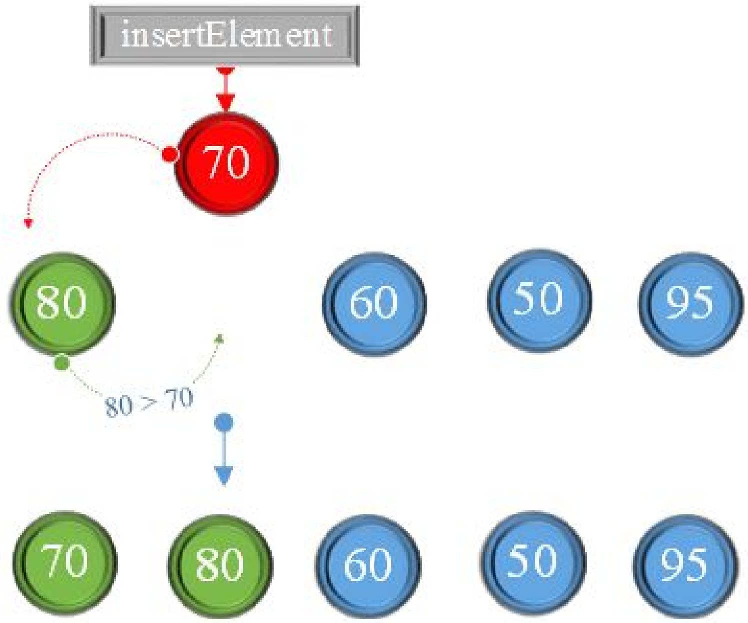
Explanation:

 No sorting,

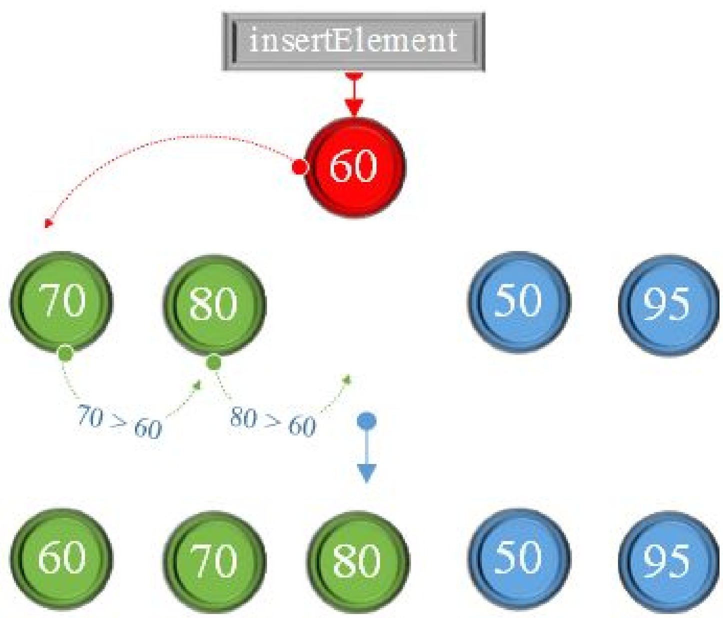
 Inserting,

 Already sorted

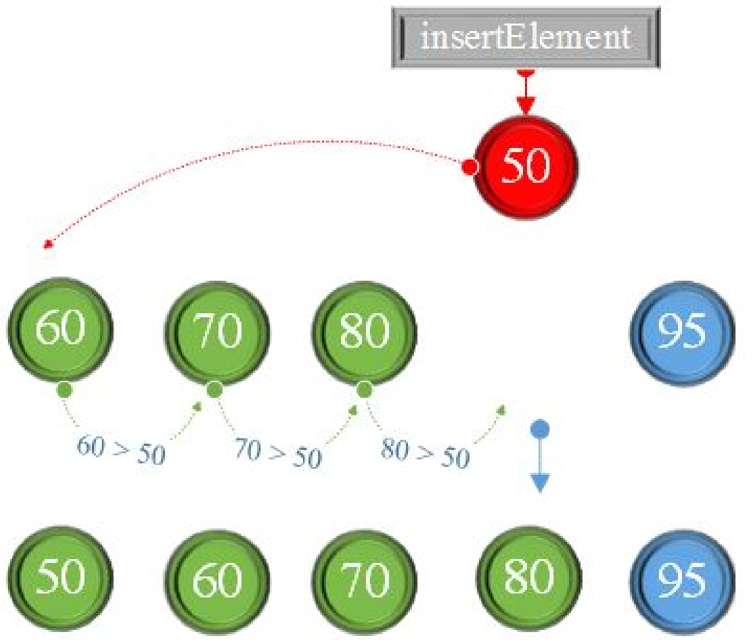
1. **First sorting:**



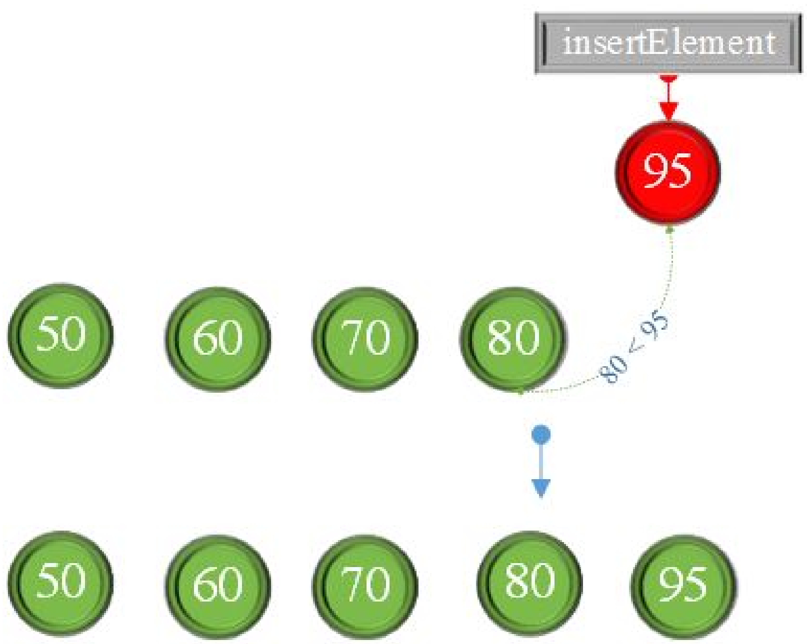
2. **Second sorting:**



3. **Third sorting:**



4 **Third sorting:**



test_insert_sort.py

```
from array import array

def sort(arrays):
    length = len(arrays)
    for i in range(0, length-1):
        insert_element = arrays[i] #Take unsorted new elements
        insert_position = i # Inserted position
        for j in range(insert_position - 1, -1, -1):
            # If the new element is smaller than the sorted element, it is
shifted to the right
            if insert_element < arrays[j]:
                arrays[j + 1] = arrays[j]
                insert_position-=1
        arrays[insert_position] = insert_element # Insert the new
element


def main():
    scores = array('i') # Create an int type array
    scores = array( 'i',[80, 70, 60, 50, 95])
    sort(scores)
    for score in scores:
        print(score, ",", end="")

if __name__ == "__main__":
    main()
```

Result:

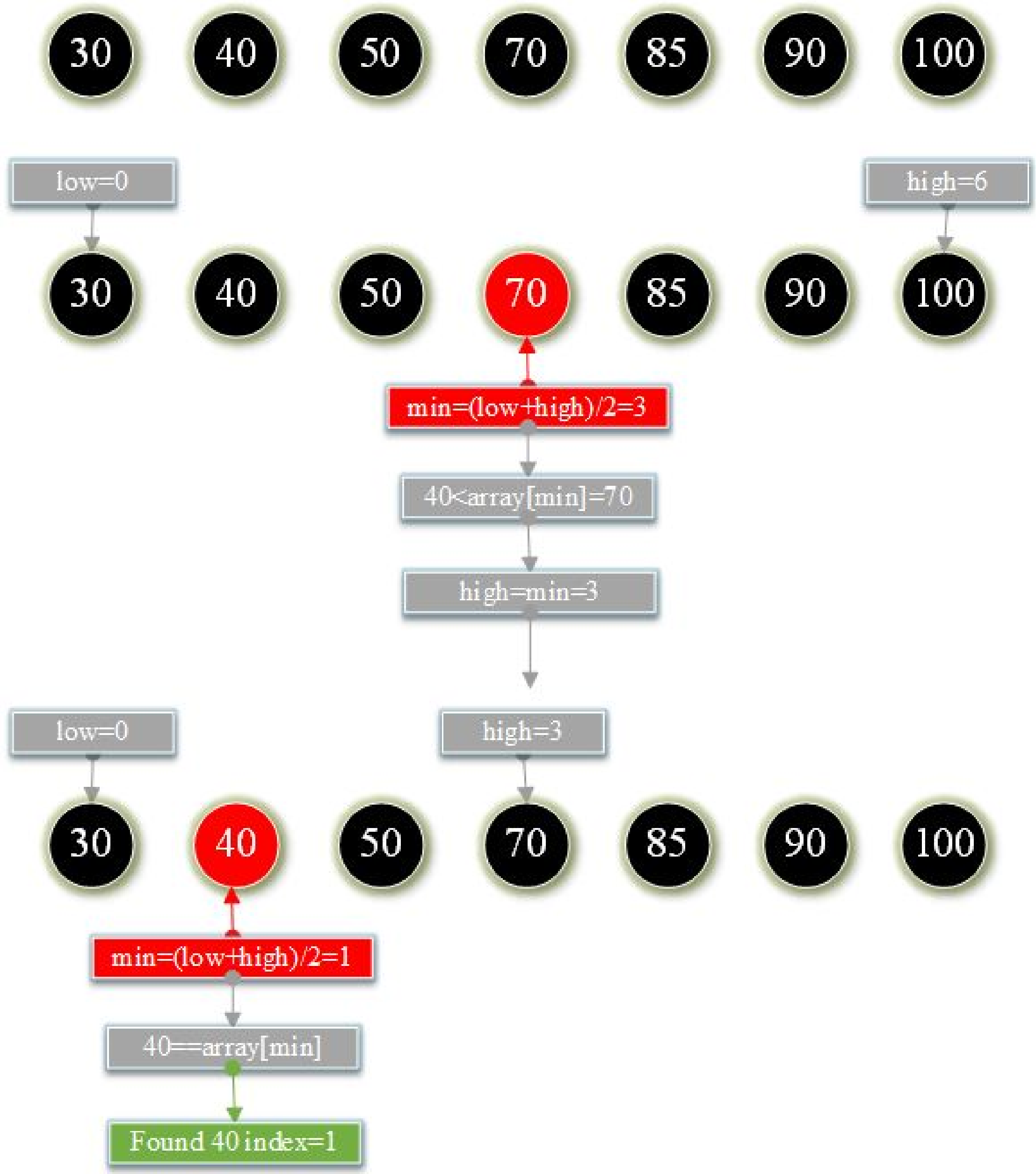
50 ,60 ,70 ,80 ,95 ,

Dichotomy Binary Search

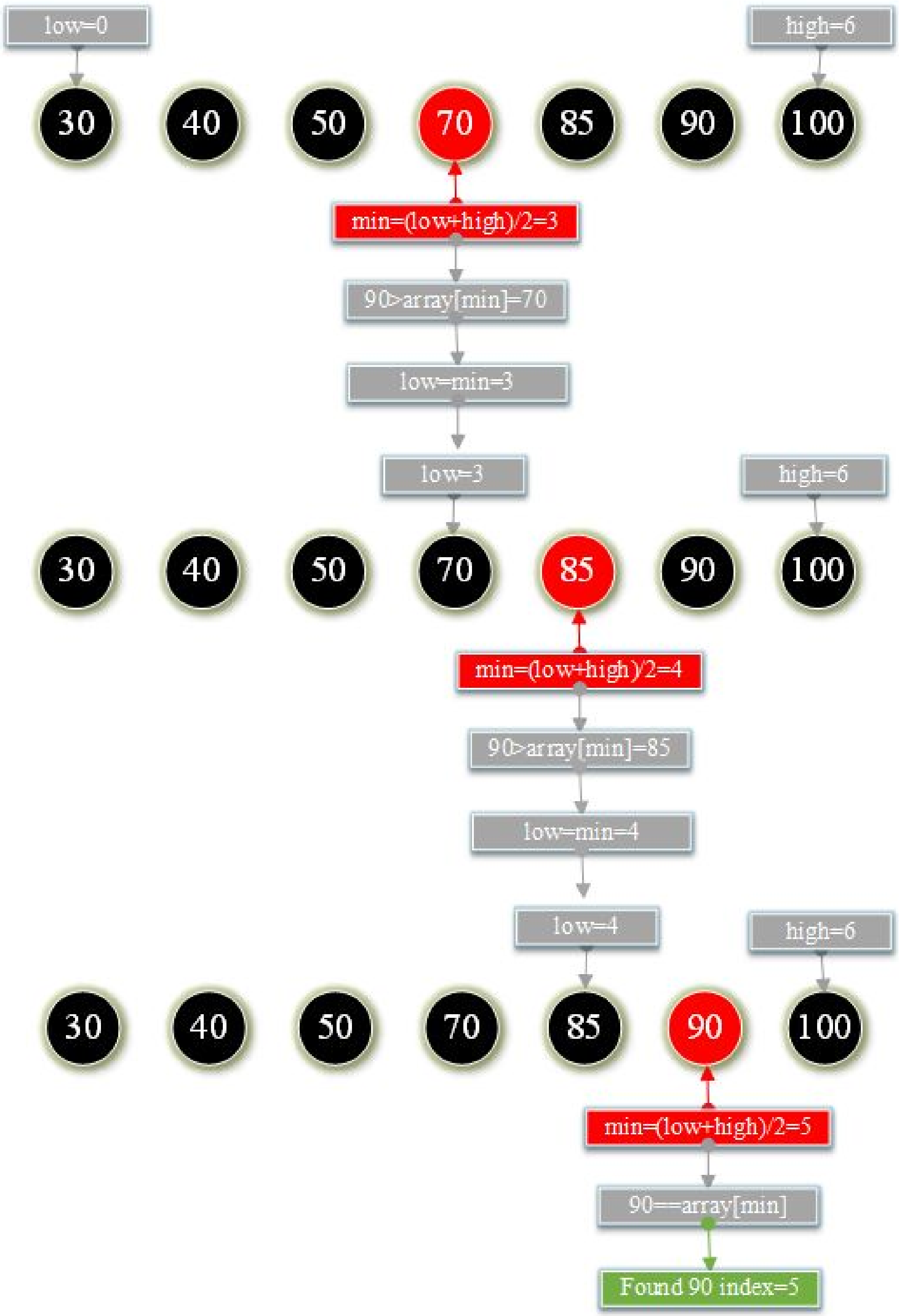
Dichotomy Binary Search:
Find the index position of a given value from an already ordered array.

1. Initialize the lowest index low=0, the highest index high=scores.length-1
2. Find the searchValue of the middle index mid=(low+high)/2 scores[mid]
3. Compare the scores[mid] with searchValue
   If the scores[mid]==searchValue print current mid index,
   If scores[mid]>searchValue that the searchValue will be found between low and mid-1
4. And so on. Repeat step 3 until you find searchValue or low>=high to terminate the loop.

Example 1 : Find the index of searchValue=40 in the array that has been sorted below.



Example 2 : Find the index of searchValue=90 in the array that has been sorted below.



test_binary_search.py

```
from array import array

def binary_search(arrays, search_value):
    length = len(arrays)
    low = 0
    high = length - 1
    mid = 0
    while low <= high:
        mid = (low + high) // 2
        if arrays[mid] == search_value:
            return mid
        elif arrays[mid] < search_value:
            low = mid + 1
        elif arrays[mid] > search_value:
            high = mid - 1
    return -1

def main():
    scores = array('i') # Create an int type array
    scores = array( 'i',[30, 40, 50, 70, 85, 90, 100])
    search_value = 40;
    position = binary_search(scores, search_value)
    print(search_value, " position:", position)

    print("-----------------------------")

    search_value = 90
    position = binary_search(scores, search_value)
    print(search_value, " position:", position)

if __name__ == "__main__":
    main()
```

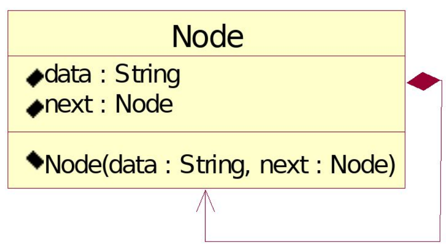
Result:

40 position: 1
----
90 position: 5

Unidirectional Linked List

Unidirectional Linked List Single Link:
Is a chained storage structure of a linear table, which is connected by a node. Each node consists of data and next pointer to the next node.

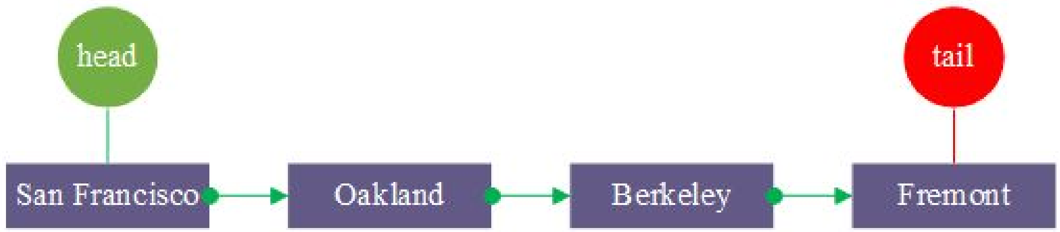
UML Diagram



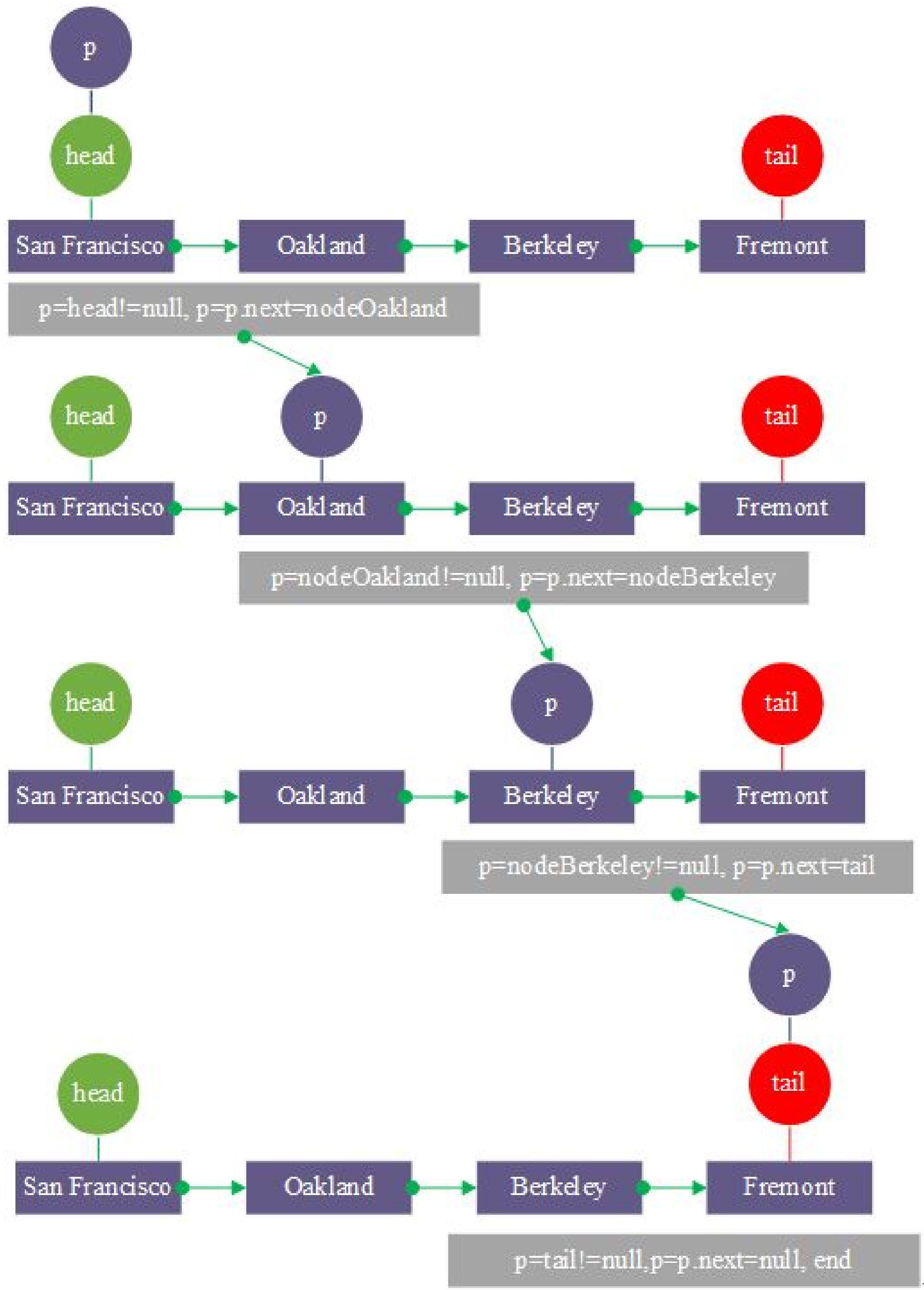

```python
class Node:

    data = ''
    next = None

    def __init__(self, data, next):
        self.data = data
        self.next = next
```

1. Unidirectional Linked List initialization.

Example : Construct a San Francisco subway Unidirectional linked list



2. traversal output.



test_unidirectional_linkedlist.py

```
class Node:
    data = ''
    next = None

    def __init__(self, data, next):
        self.data = data
        self.next = next

class LinkedList:
    __head = None
    __tail = None

    def init(self):
        # the first node called head node
        self.__head = Node("San Francisco", None)

        node_oakland = Node("Oakland", None)
        self.__head.next = node_oakland

        node_berkeley = Node("Berkeley", None)
        node_oakland.next = node_berkeley

        # the last node called tail node
        self.__tail = Node("Fremont", None)
        node_berkeley.next = self.__tail

        return self.__head

    def output(self, node):
        p = node
        while (p != None): # From the beginning to the end
            data = p.data
```

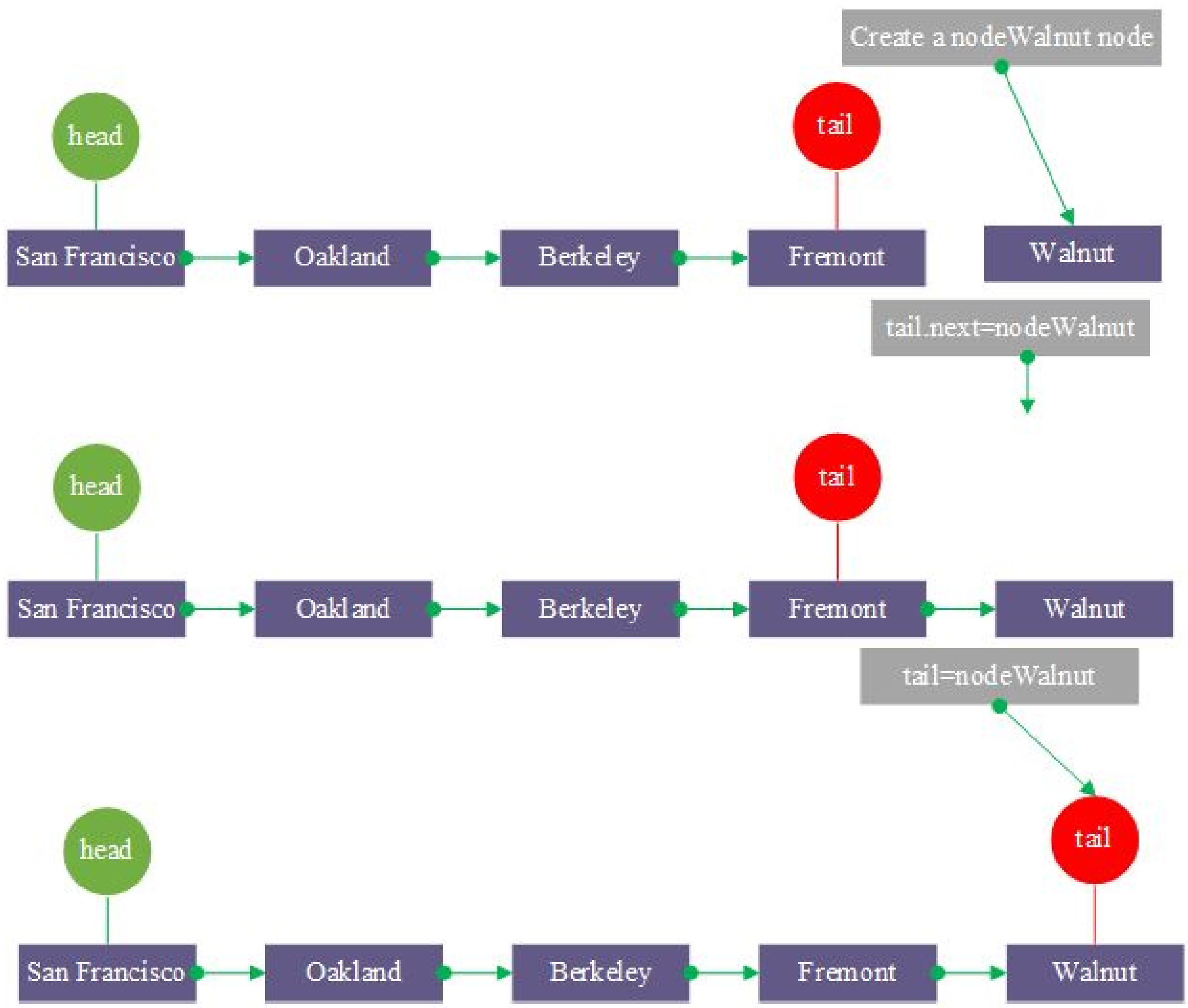
```
        print(data, " -> ", end="")
        p = p.next
    print("End\n\n")
```

```python
def main():
    linkedlist = LinkedList()
    head = linkedlist.init()
    linkedlist.output(head)

if __name__ == "__main__":
    main()
```

Result:

San Francisco -> Oakland -> Berkeley -> Fremont -> End

3. Append a new node name: Walnut to the end.



test_unidirectional_linkedlist.py

```
class Node:
    data = ''
    next = None

    def __init__(self, data, next):
        self.data = data
        self.next = next

class LinkedList:
    __head = None
    __tail = None

    def init(self):
        # the first node called head node
        self.__head = Node("San Francisco", None)

        node_oakland = Node("Oakland", None)
        self.__head.next = node_oakland

        node_berkeley = Node("Berkeley", None)
        node_oakland.next = node_berkeley

        # the last node called tail node
        self.__tail = Node("Fremont", None)
        node_berkeley.next = self.__tail

        return self.__head

    def add(self, new_node):
        self.__tail.next = new_node
        self.__tail = new_node
```

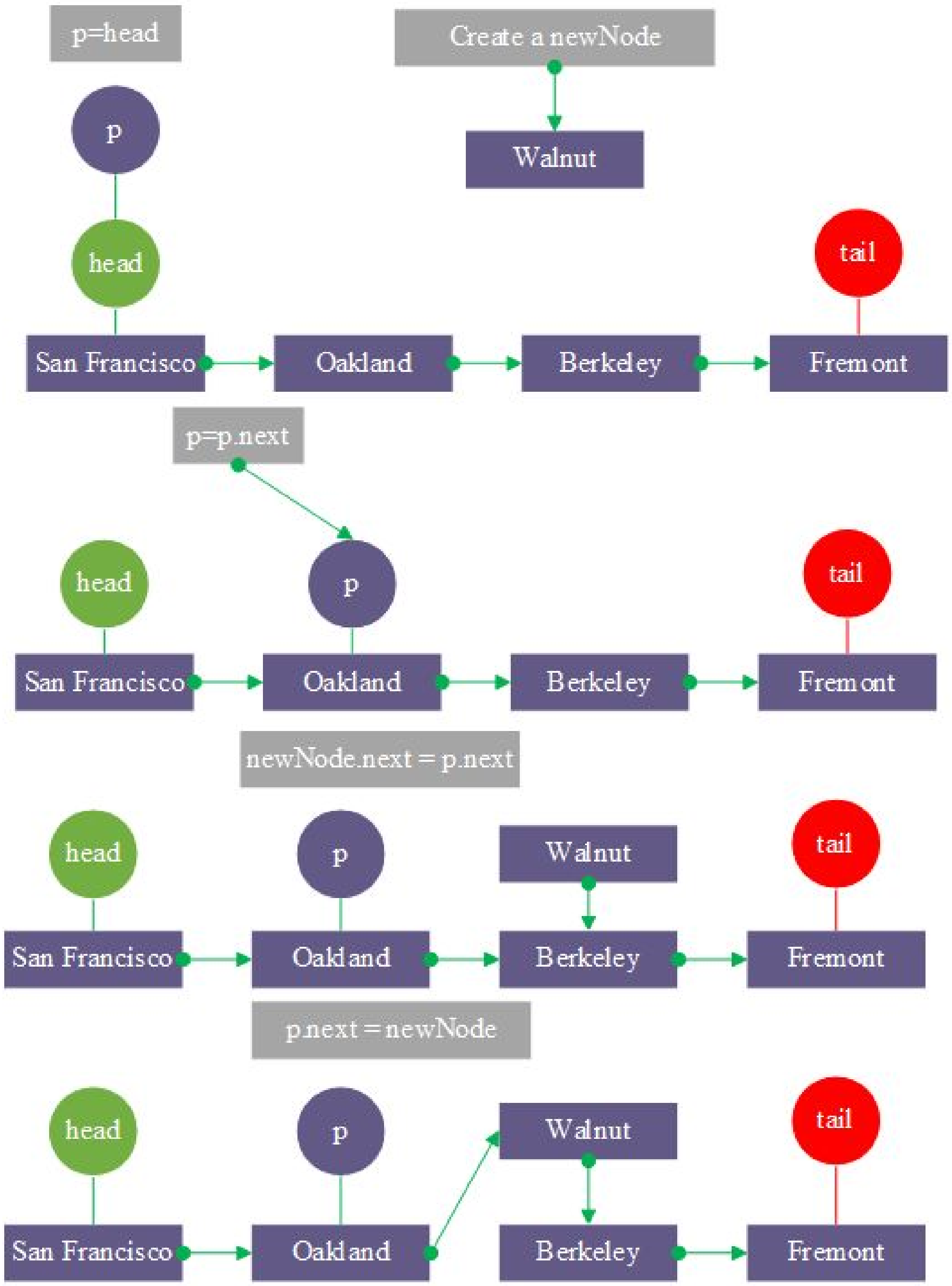
```python
    def output(self, node):
        p = node
        while (p != None): # From the beginning to the end
            data = p.data
            print(data, " -> ", end="")
            p = p.next
        print("End\n\n")
```

```python
def main():
    linkedlist = LinkedList()
    head = linkedlist.init()

    print("Append a new node name: Walnut  to the end: ")
    linkedlist.add(Node("Walnut", None))

    linkedlist.output(head)

if __name__ == "__main__":
    main()
```

Result:

Append a new node name: Walnut to the end:
San Francisco -> Oakland -> Berkeley -> Fremont -> Walnut -> End

3. Insert a node Walnut in position 2.



test_unidirectional_linkedlist.py

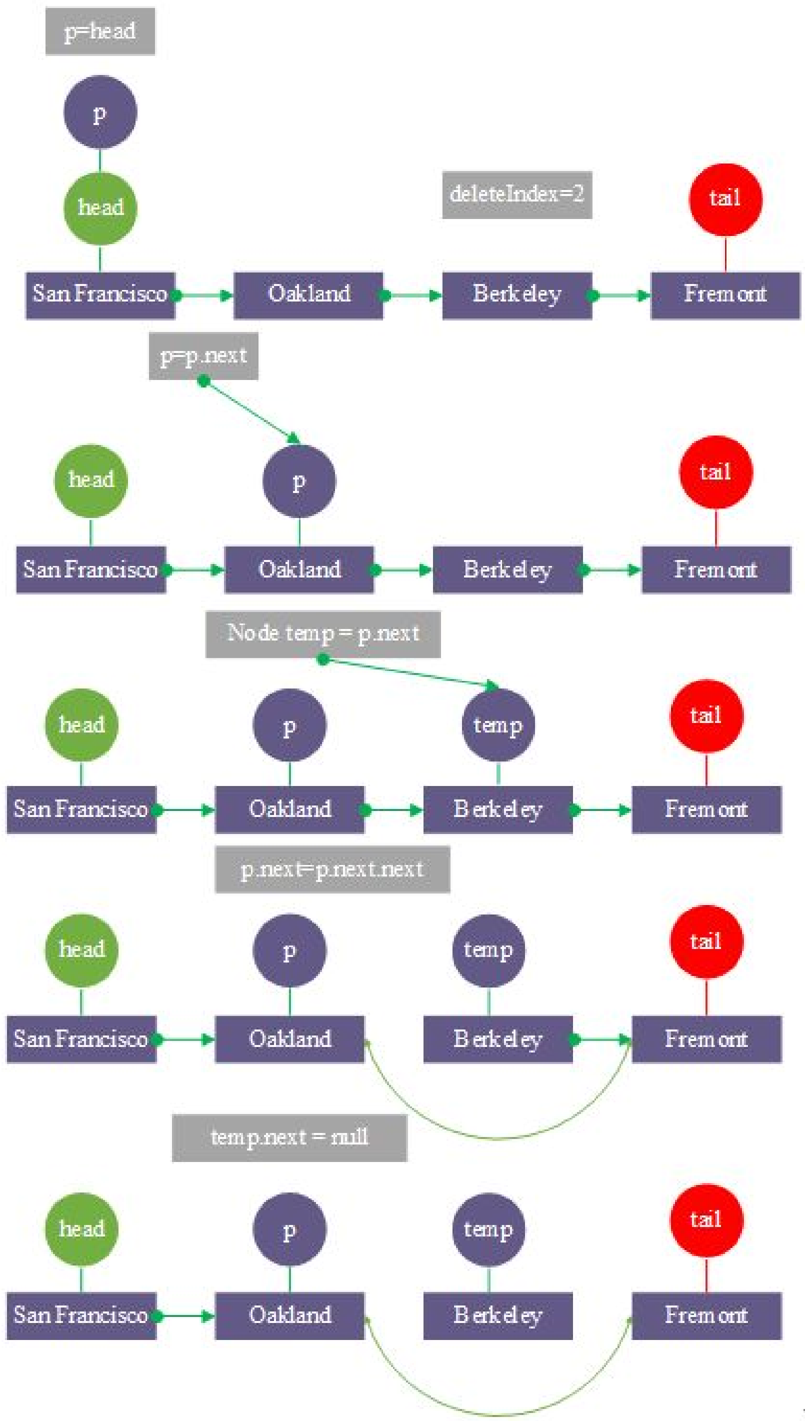
```
class Node:
    data = ''
    next = None

    def __init__(self, data, next):
        self.data = data
        self.next = next

class LinkedList:
    __head = None
    __tail = None

    def init(self):
        # the first node called head node
        self.__head = Node("San Francisco", None)

        node_oakland = Node("Oakland", None)
        self.__head.next = node_oakland

        node_berkeley = Node("Berkeley", None)
        node_oakland.next = node_berkeley

        # the last node called tail node
        self.__tail = Node("Fremont", None)
        node_berkeley.next = self.__tail
        return self.__head

    def insert(self, insert_position, new_node):
        p = self.__head
        i = 0
        # Move the node to the insertion position
        while p.next != None and i < insert_position - 1:
```

```python
            p = p.next
            i+=1

        new_node.next = p.next # new_node next point to next node
        p.next = new_node # current next point to new_node




    def output(self, node):
        p = node
        while (p != None): # From the beginning to the end
            data = p.data
            print(data, " -> ", end="")
            p = p.next
        print("End\n\n")

def main():
    linkedlist = LinkedList()
    head = linkedlist.init()

    print("Insert a new node Walnut at index = 2 : ")
    linkedlist.insert(2, Node("Walnut", None))

    linkedlist.output(head)

if __name__ == "__main__":
    main()
```

Result:

Insert a new node Walnut at index = 2 :
San Francisco -> Oakland -> Walnut -> Berkeley -> Fremont -> End

4. Delete the index=2 node.



test_unidirectional_linkedlist.py

```
class Node:
    data = ''
    next = None

    def __init__(self, data, next):
        self.data = data
        self.next = next

class LinkedList:
    __head = None
    __tail = None

    def init(self):
        # the first node called head node
        self.__head = Node("San Francisco", None)

        node_oakland = Node("Oakland", None)
        self.__head.next = node_oakland

        node_berkeley = Node("Berkeley", None)
        node_oakland.next = node_berkeley

        # the last node called tail node
        self.__tail = Node("Fremont", None)
        node_berkeley.next = self.__tail
        return self.__head

    def remove(self, remove_position):
        p = self.__head
        i = 0
        # Move the node to the previous node position that was
deleted
```

```
        while p.next != None and i < remove_position - 1:
            p = p.next
            i+=1
        temp = p.next # Save the node you want to delete
        p.next = p.next.next # Previous node next points to next of delete the node
        temp.next = None




    def output(self, node):
        p = node
        while (p != None): # From the beginning to the end
            data = p.data
            print(data, " -> ", end="")
            p = p.next
        print("End\n\n")

def main():
    linkedlist = LinkedList()
    head = linkedlist.init()

    print("Delete a new node Berkeley at index = 2 : ")
    linkedlist.remove(2)

    linkedlist.output(head)

if __name__ == "__main__":
    main()
```

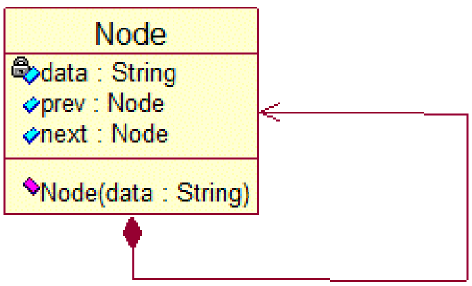
Result:

Delete a new node Berkeley at index = 2 :
San Francisco -> Oakland -> Fremont -> End

Doubly Linked List

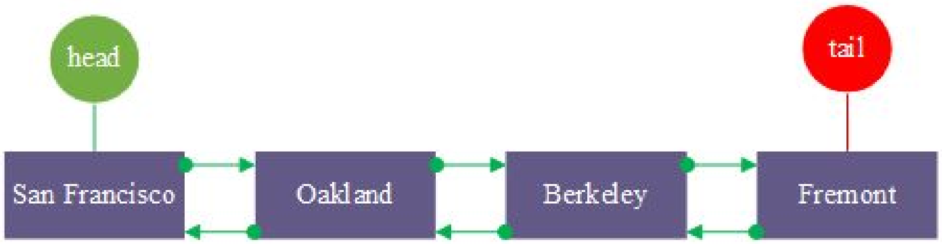
Doubly Linked List:
It is a chained storage structure of a linear table. It is connected by nodes in two directions. Each node consists of data, pointing to the previous node and pointing to the next node.

UML Diagram




```python
class Node:
    data = ''
    prev = None
    next = None

    def __init__(self, data):
        self.data = data
```

1. Doubly Linked List initialization.
Example : Construct a San Francisco subway Doubly linked list



2. traversal output.

test_doubly_linkedlist.py

```
class Node:
    data = ''
    prev = None
    next = None

    def __init__(self, data):
        self.data = data

class LinkedList:
    __head = None
    __tail = None

    def init(self):
        self.__head = Node("San Francisco")
        self.__head.prev = None
        self.__head.next = None

        node_oakland = Node("Oakland")
        node_oakland.prev = self.__head
        node_oakland.next = None
        self.__head.next = node_oakland

        node_berkeley = Node("Berkeley")
        node_berkeley.prev = node_oakland
        node_berkeley.next = None
        node_oakland.next = node_berkeley

        self.__tail = Node("Fremont")
        self.__tail.prev = node_berkeley
        self.__tail.next = None
```

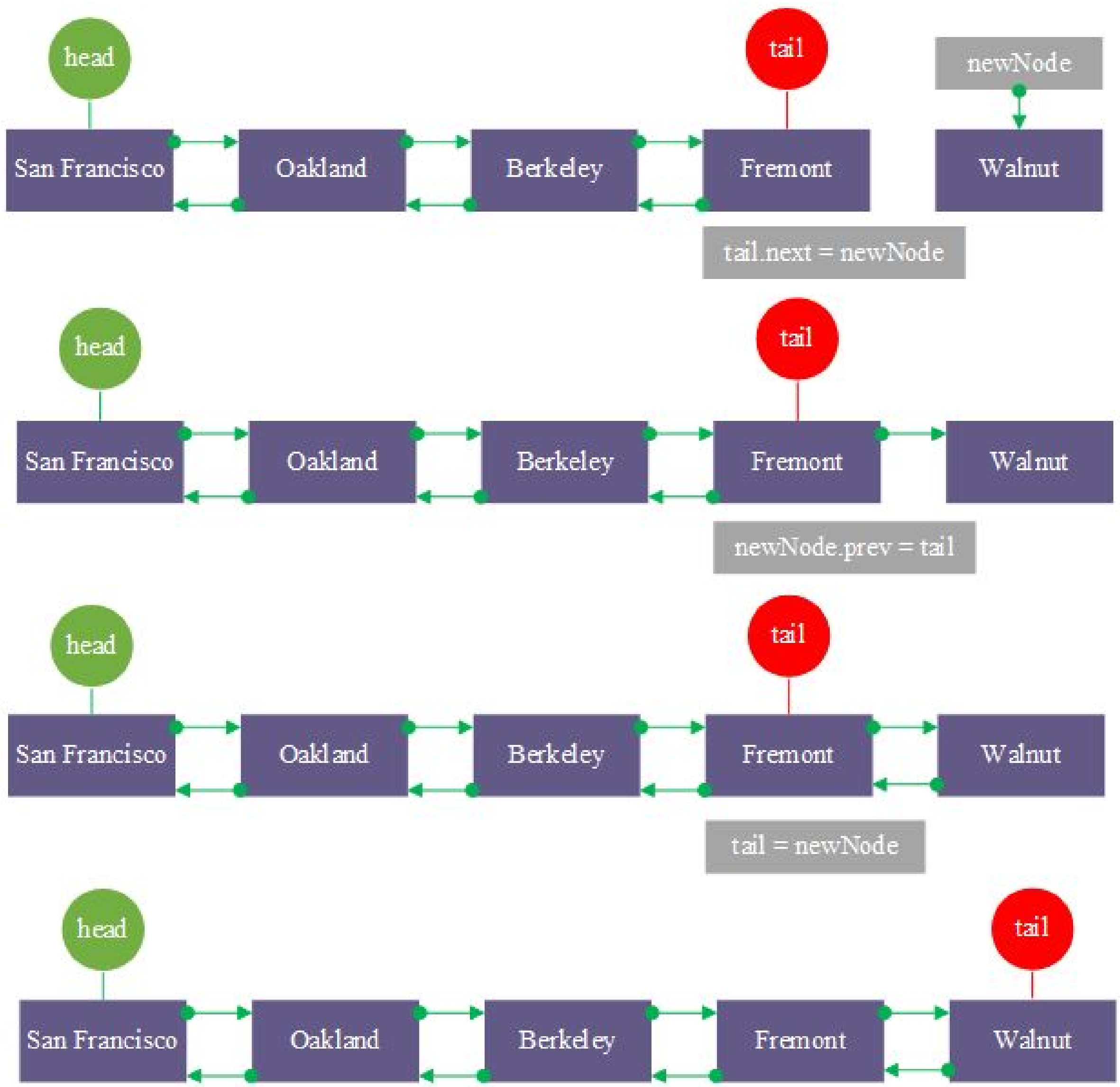
```
        node_berkeley.next = self.__tail
        return self.__head









    def output(self, node):
        p = node
        end = None
        while p != None:
            print(p.data, " -> ", end="")
            end = p
            p = p.next
        print("End\n")

        p = end
        while p != None:
            print(p.data, " -> ", end="")
            p = p.prev
        print("Start\n\n")

def main():
    linkedlist = LinkedList()
    head = linkedlist.init()
    linkedlist.output(head)

if __name__ == "__main__":
    main()
```

Result:

San Francisco -> Oakland -> Berkeley -> Fremont -> End

Fremont -> Berkeley -> Oakland -> San Francisco -> Start

3. add a node Walnut at the end of Fremont.



test_doubly_linkedlist.py

```
class Node:
    data = ''
    prev = None
    next = None

    def __init__(self, data):
        self.data = data

class LinkedList:

    __head = None
    __tail = None

    def init(self):
        self.__head = Node("San Francisco")
        self.__head.prev = None
        self.__head.next = None

        node_oakland = Node("Oakland")
        node_oakland.prev = self.__head
        node_oakland.next = None
        self.__head.next = node_oakland

        node_berkeley = Node("Berkeley")
        node_berkeley.prev = node_oakland
        node_berkeley.next = None
        node_oakland.next = node_berkeley

        self.__tail = Node("Fremont")
        self.__tail.prev = node_berkeley
        self.__tail.next = None
        node_berkeley.next = self.__tail
```

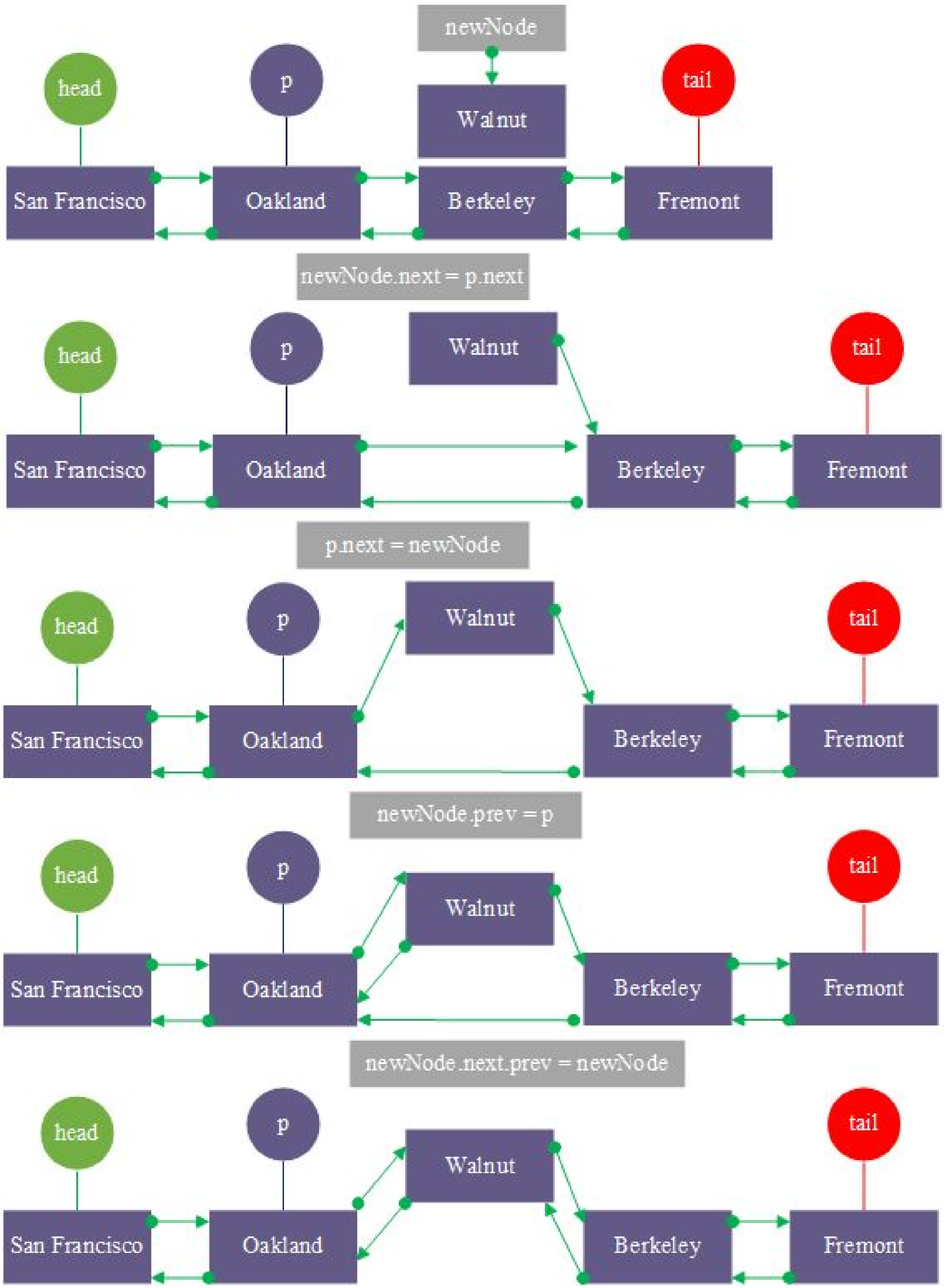
```
        return self.__head

    def add(self, new_node):
        self.__tail.next = new_node;
        new_node.prev = self.__tail
        self.__tail = new_node




    def output(self, node):
        p = node
        end = None
        while p != None:
            print(p.data, " -> ", end="")
            end = p
            p = p.next
        print("End\n")

        p = end
        while p != None:
            print(p.data, " -> ", end="")
            p = p.prev
        print("Start\n\n")

def main():
    linkedlist = LinkedList()
    head = linkedlist.init()

    print("Append a new node name: Walnut  to the end: ")
    linkedlist.add(Node("Walnut"))

    linkedlist.output(head)

if __name__ == "__main__":
    main()
```

Result:

Append a new node name: Walnut to the end:
San Francisco -> Oakland -> Berkeley -> Fremont -> Walnut -> End

Walnut -> Fremont -> Berkeley -> Oakland -> San Francisco -> Start

3. Insert a node Walnut in position 2.



test_doubly_linkedlist.py

```
class Node:
    data = ''
    prev = None
    next = None

    def __init__(self, data):
        self.data = data

class LinkedList:

    __head = None
    __tail = None

    def init(self):
        self.__head = Node("San Francisco")
        self.__head.prev = None
        self.__head.next = None

        node_oakland = Node("Oakland")
        node_oakland.prev = self.__head
        node_oakland.next = None
        self.__head.next = node_oakland

        node_berkeley = Node("Berkeley")
        node_berkeley.prev = node_oakland
        node_berkeley.next = None
        node_oakland.next = node_berkeley

        self.__tail = Node("Fremont")
        self.__tail.prev = node_berkeley
        self.__tail.next = None
        node_berkeley.next = self.__tail
```

```python
        return self.__head

    def insert(self, insert_position, new_node):
        p = self.__head
        i = 0
        while p.next != None and i < insert_position-1:
            p = p.next
            i+=1
        new_node.next = p.next
        p.next = new_node
        new_node.prev = p
        new_node.next.prev = new_node

    def output(self, node):
        p = node
        end = None
        while p != None:
            print(p.data, " -> ", end="")
            end = p
            p = p.next
        print("End\n")

        p = end
        while p != None:
            print(p.data, " -> ", end="")
            p = p.prev
        print("Start\n\n")

def main():
```

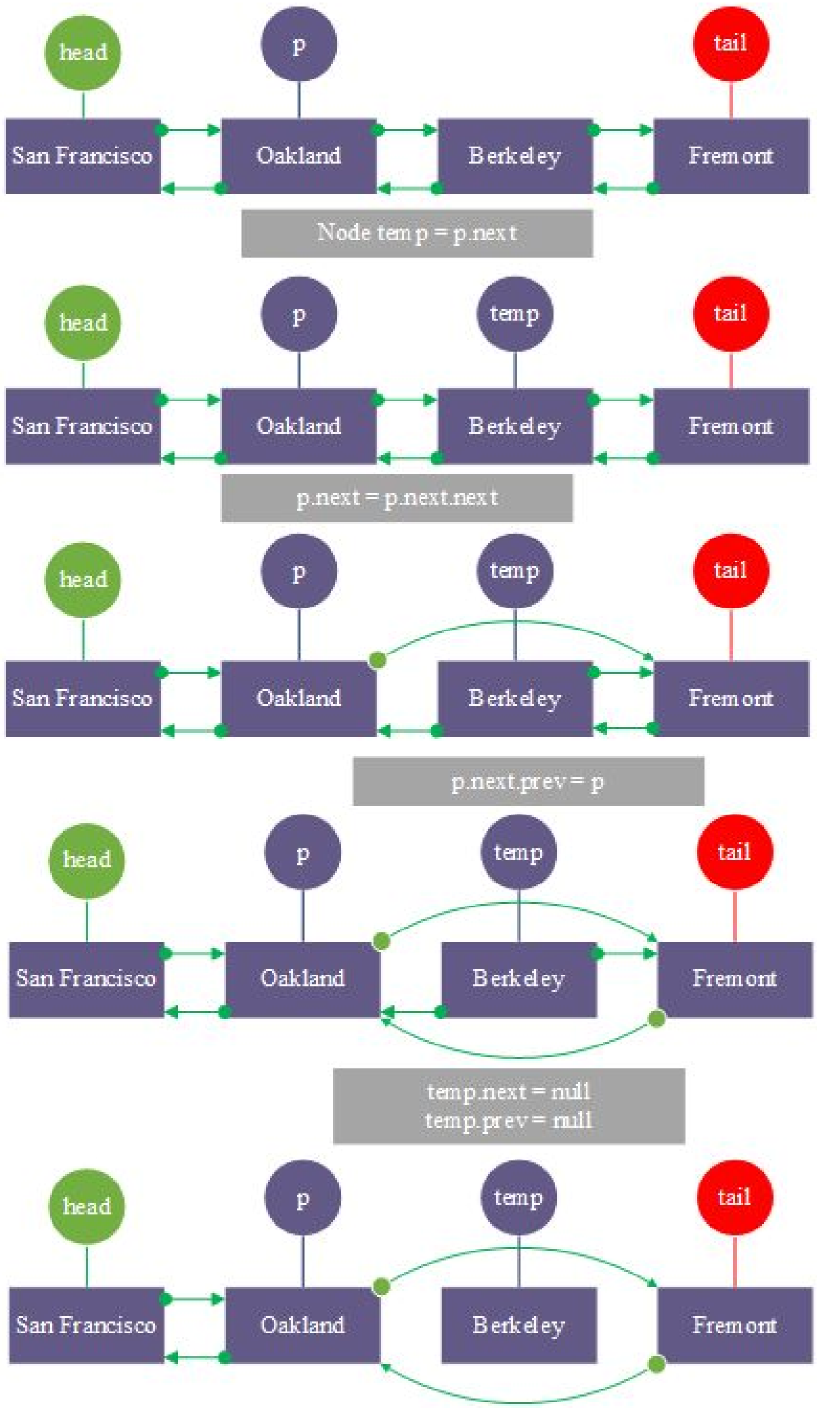
```
    linkedlist = LinkedList()
    head = linkedlist.init()

    print("Insert a new node Walnut at index = 2 : ")
    linkedlist.insert(2,Node("Walnut"))
    linkedlist.output(head)

if __name__ == "__main__":
    main()
```

Result:

Insert a new node Walnut at index = 2 :
San Francisco -> Oakland -> Walnut -> Berkeley -> Fremont -> End

Fremont -> Berkeley -> Walnut -> Oakland -> San Francisco -> Start

4. Delete the index=2 node.



TestDoubleLink.py

```
class Node:
    data = ''
    prev = None
    next = None

    def __init__(self, data):
        self.data = data

class LinkedList:

    __head = None
    __tail = None

    def init(self):
        self.__head = Node("San Francisco")
        self.__head.prev = None
        self.__head.next = None

        node_oakland = Node("Oakland")
        node_oakland.prev = self.__head
        node_oakland.next = None
        self.__head.next = node_oakland

        node_berkeley = Node("Berkeley")
        node_berkeley.prev = node_oakland
        node_berkeley.next = None
        node_oakland.next = node_berkeley

        self.__tail = Node("Fremont")
        self.__tail.prev = node_berkeley
        self.__tail.next = None
        node_berkeley.next = self.__tail
```

```python
        return self.__head

    def remove(self, remove_position):
        p = self.__head
        i = 0
        # Move the node to the previous node  that was deleted
        while p.next != None and i < remove_position - 1:
            p = p.next
            i+=1

        temp = p.next # Save the node you want to delete
        p.next = p.next.next
        p.next.prev = p
        temp.next = None # Set the delete node next to null
        temp.prev = None # Set the delete node prev to null

    def output(self, node):
        p = node
        end = None
        while p != None:
            print(p.data, " -> ", end="")
            end = p
            p = p.next
        print("End\n")

        p = end
        while p != None:
            print(p.data, " -> ", end="")
            p = p.prev
        print("Start\n\n")
```

```python
def main():
    linkedlist = LinkedList()
    head = linkedlist.init()
    print("Delete node Berkeley at index = 2 : ")
    linkedlist.remove(2)
    linkedlist.output(head)

if __name__ == "__main__":
    main()
```

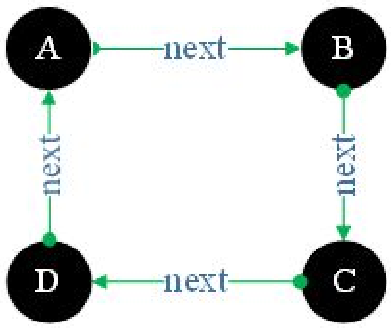
Result:

Delete node Berkeley at index = 2 :
San Francisco -> Oakland -> Fremont -> End

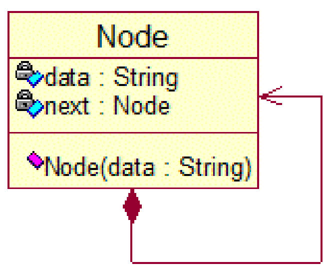
Fremont -> Oakland -> San Francisco -> Start

One-way Circular LinkedList

One-way Circular List:
It is a chain storage structure of a linear table, which is connected to form a ring, and each node is composed of data and a pointer to next.

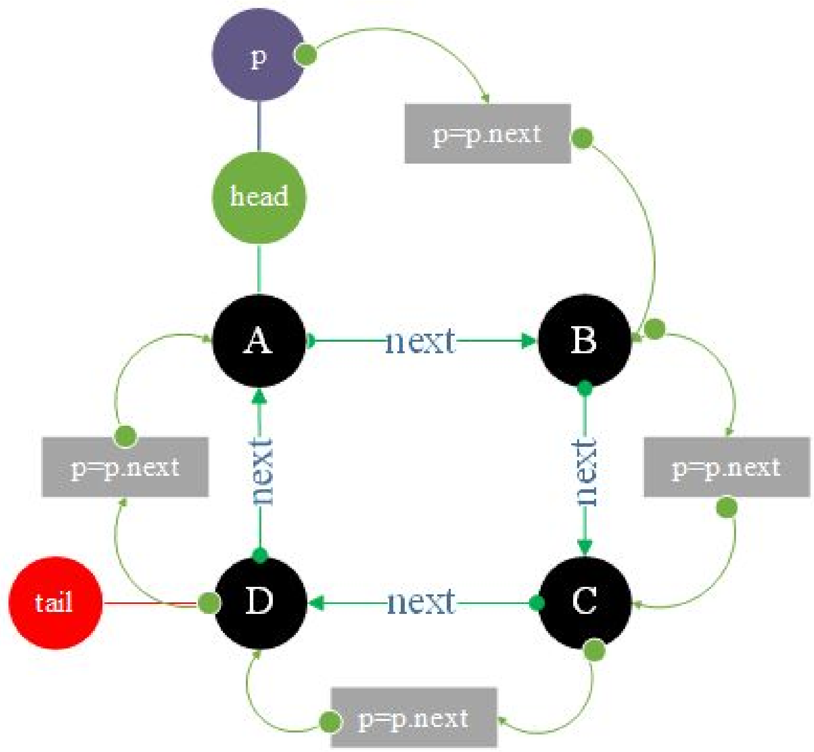


UML Diagram




```
class Node:

    data = ''
    next = None

    def __init__(self, data):
        self.data = data
```

1. One-way Circular Linked List initialization and traversal output.



test_oneway_circular_linkedlist.py

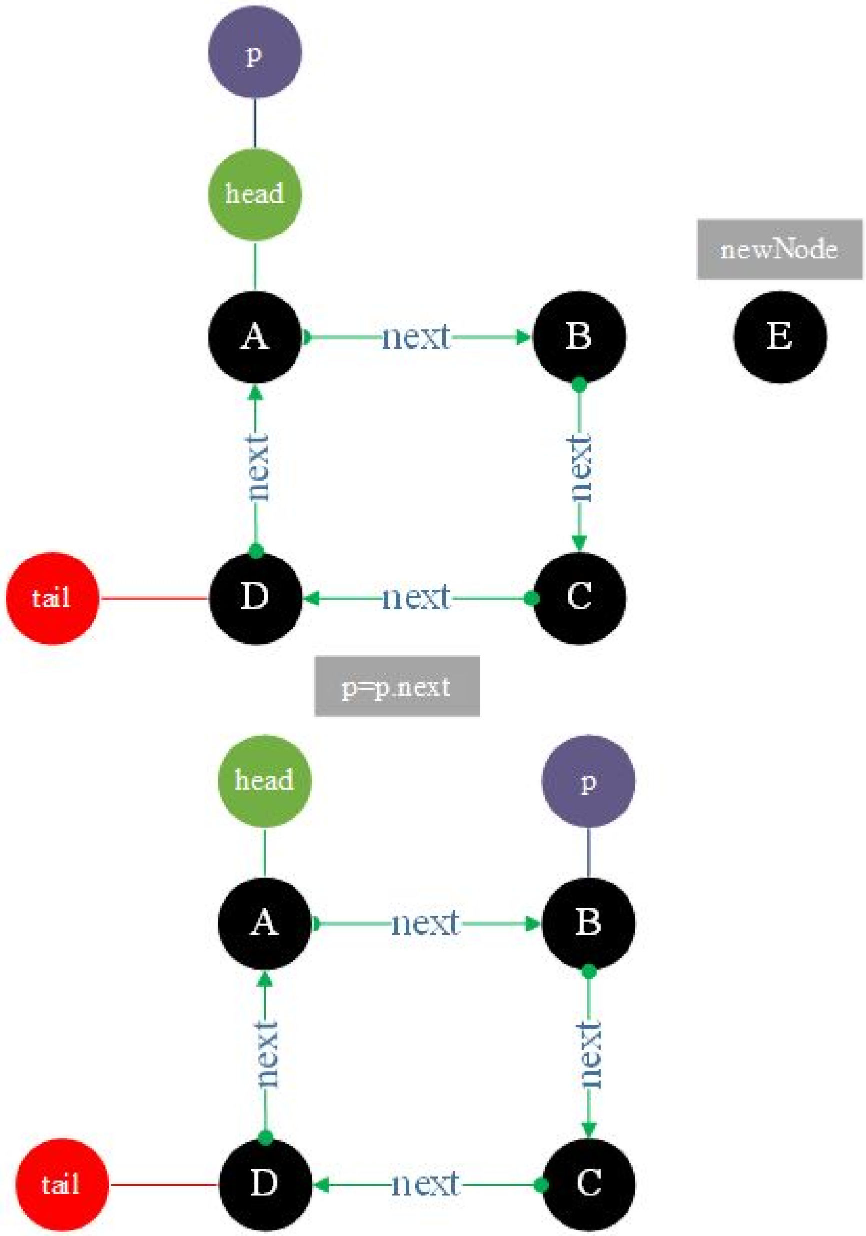
```python
class Node:

    data = ''
    next = None

    def __init__(self, data):
        self.data = data

class LinkedList:
    __head = None
    __tail = None

    def init(self):
        #the first node called head node
        self.__head = Node("A")
        self.__head.next = None

        nodeB = Node("B")
        nodeB.next = None
        self.__head.next = nodeB

        nodeC = Node("C")
        nodeC.next = None
        nodeB.next = nodeC

        # the last node called tail node
        self.__tail = Node("D")
        self.__tail.next = self.__head
        nodeC.next = self.__tail

    def output(self):
        p = self.__head
```

```
        print(p.data, " -> ", end="")
        p = p.next
        while p != self.__head:
            print(p.data, " -> ", end="")
            p = p.next

        print(p.data, "\n\n")



def main():
    linkedlist = LinkedList()
    linkedlist.init()
    linkedlist.output()

if __name__ == "__main__":
    main()
```

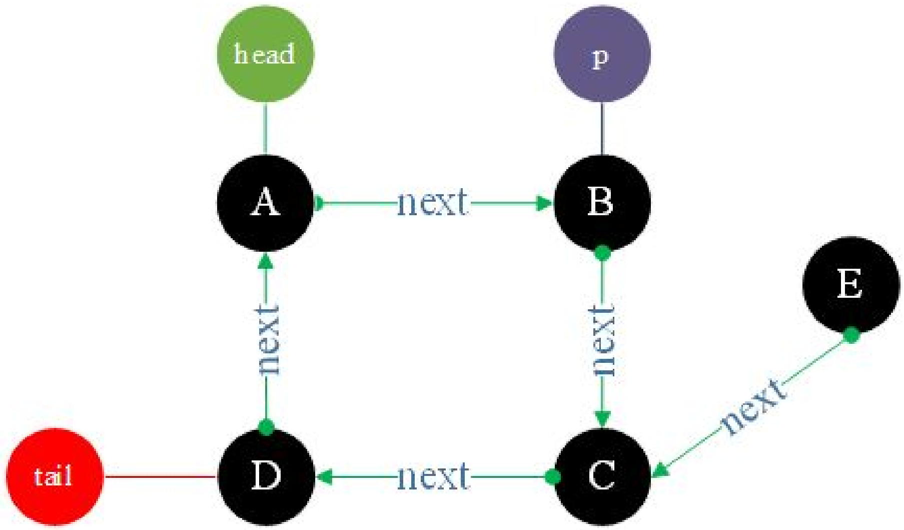
Result:

A -> B -> C -> D -> A

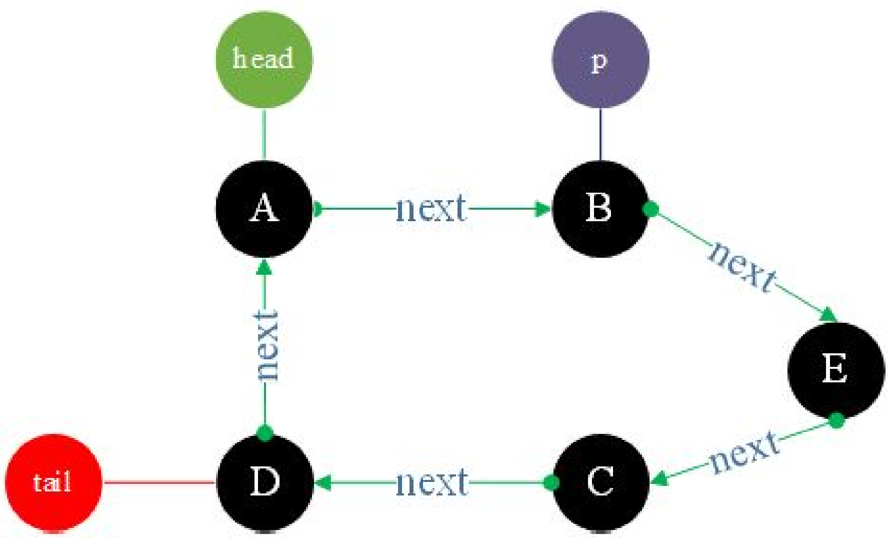
3. Insert a node E in position 2.



newNode.next = p.next



p.next = newNode



test_oneway_circular_linkedlist.py

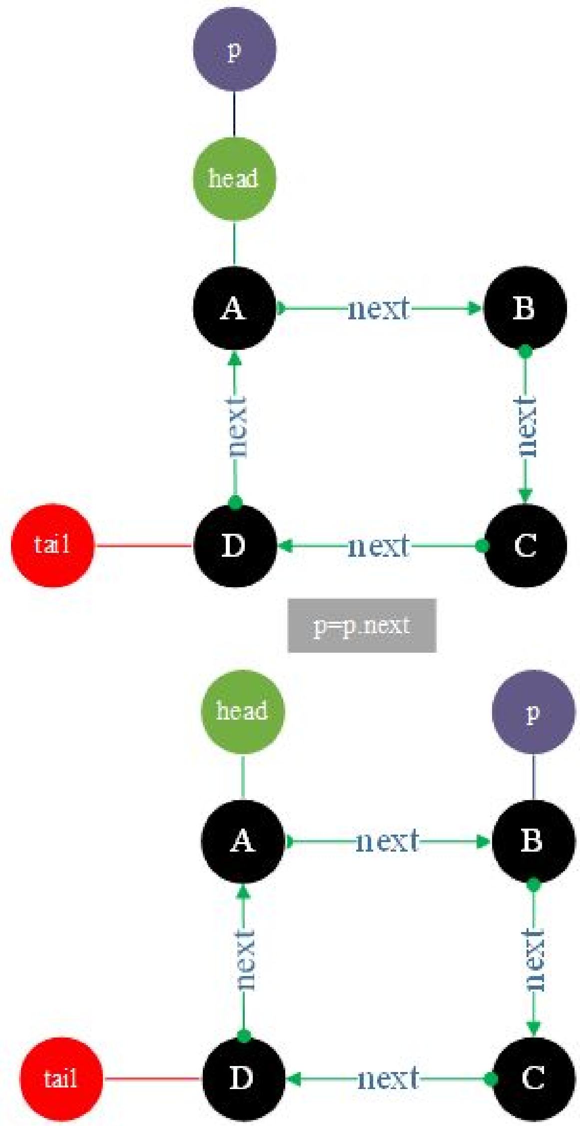
```
class Node:

    data = ''
    next = None

    def __init__(self, data):
        self.data = data

class LinkedList:
    __head = None
    __tail = None

    def init(self):
        self.__head = Node("A")
        self.__head.next = None

        nodeB = Node("B")
        nodeB.next = None
        self.__head.next = nodeB

        nodeC = Node("C")
        nodeC.next = None
        nodeB.next = nodeC

        self.__tail = Node("D")
        self.__tail.next = self.__head
        nodeC.next = self.__tail

    def insert(self, insert_position, new_node):
        p = self.__head
        i = 0
        # Move the node to the insertion position
```

```
        while p.next != None and i < insert_position - 1:
            p = p.next
            i+=1
        new_node.next = p.next
        p.next = new_node





    def output(self):
        p = self.__head
        print(p.data, " -> ", end="")
        p = p.next
        while p != self.__head:
            print(p.data, " -> ", end="")
            p = p.next
        print(p.data, "\n\n")

def main():
    linkedlist = LinkedList()
    linkedlist.init()

    print("Insert a new node E at index = 2 : ")
    linkedlist.insert(2,Node("E"))

    linkedlist.output()

if __name__ == "__main__":
    main()
```

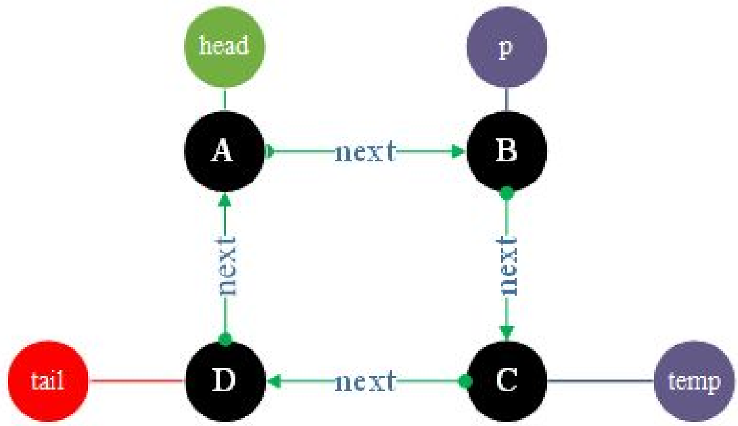
Result:

Insert a new node E at index = 2 :
A -> B -> E -> C -> D -> A

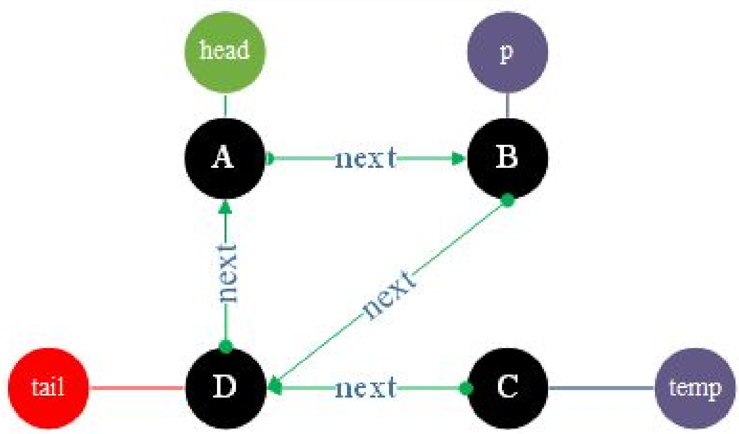
4. Delete the index=2 node.



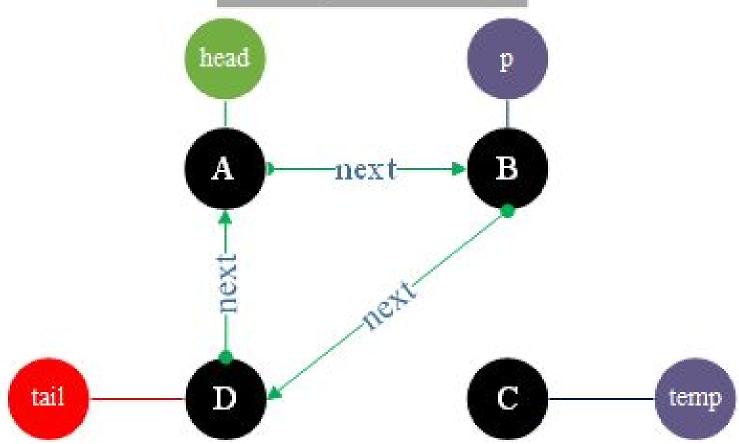
Node temp = p.next



p.next = p.next.next



temp.next = null



test_oneway_circular_linkedlist.py

```
class Node:

    data = ''
    next = None

    def __init__(self, data):
        self.data = data

class LinkedList:
    __head = None
    __tail = None

    def init(self):
        self.__head = Node("A")
        self.__head.next = None

        nodeB = Node("B")
        nodeB.next = None
        self.__head.next = nodeB

        nodeC = Node("C")
        nodeC.next = None
        nodeB.next = nodeC

        self.__tail = Node("D")
        self.__tail.next = self.__head
        nodeC.next = self.__tail

    def remove(self, remove_position):
        p = self.__head
        i = 0
        while p.next != None and i < remove_position - 1:
```

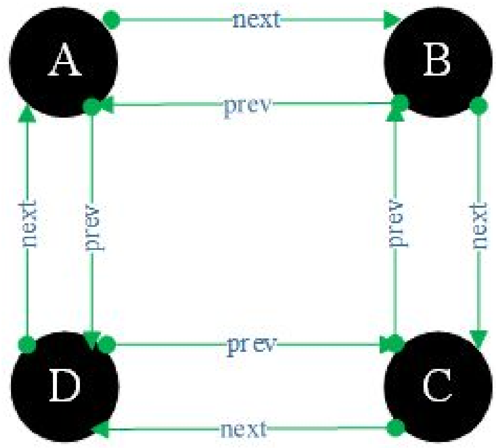
```
            p = p.next
            i+=1
        temp = p.next
        p.next = p.next.next
        temp.next = None





    def output(self):
        p = self.__head
        print(p.data, " -> ", end="")
        p = p.next
        while p != self.__head:
            print(p.data, " -> ", end="")
            p = p.next
        print(p.data, "\n\n")

def main():
    linkedlist = LinkedList()
    linkedlist.init()

    print("Delete a node at index = 2 : ")
    linkedlist.remove(2)

    linkedlist.output()

if __name__ == "__main__":
    main()
```

Result:

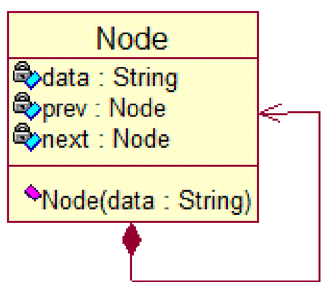
Delete a node at index = 2 :
A -> B -> D -> A

Two-way Circular LinkedList

Two-way Circular List:
It is a chain storage structure of a linear table. The nodes are connected in series by two directions, and is connected to form a ring. Each node is composed of data, pointing to the previous node prev and pointing to the next node next.

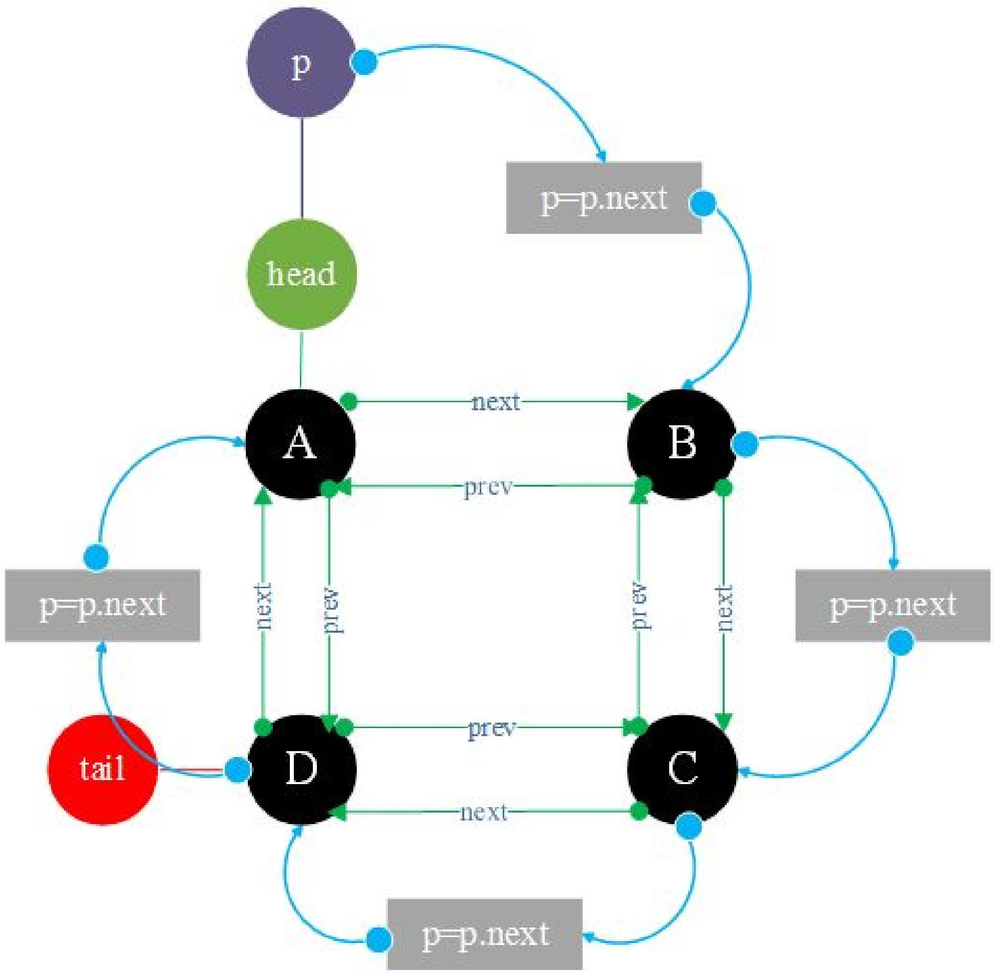


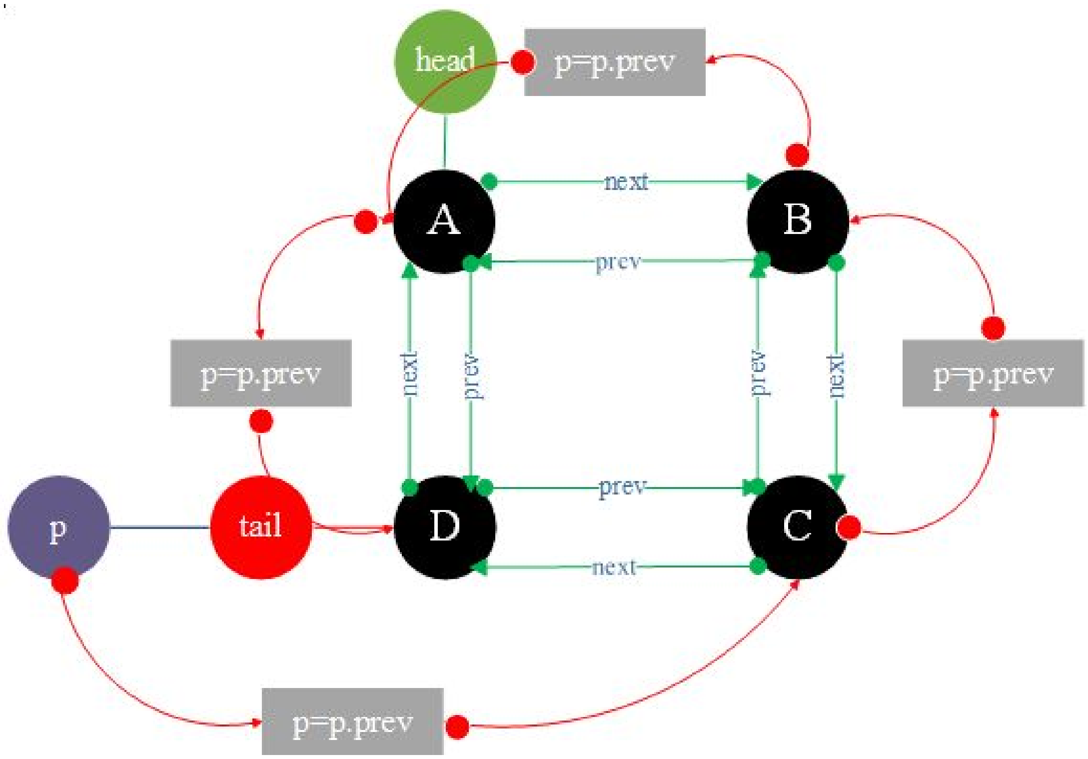
UML Diagram




```
class Node:

    data = ''
    next = None
    prev = None

    def __init__(self, data):
        self.data = data
```

1. Two-way Circular Linked List **initialization and traversal output**.




head
p=p.prev
A
B
next
prev
p=p.prev
next
prev
prev
next
p=p.prev
p
tail
D
C
prev
next
p=p.prev

test_twoway_circular_linkedlist.py

```
class Node:
    data = ''
    next = None
    prev = None

    def __init__(self, data):
        self.data = data

class LinkedList:
    __head = None
    __tail = None

    def init(self):
        self.__head = Node("A")
        self.__head.next = None
        self.__head.prev = None

        nodeB = Node("B")
        nodeB.next = None
        nodeB.prev = self.__head
        self.__head.next = nodeB

        nodeC = Node("C")
        nodeC.next = None
        nodeC.prev = nodeB
        nodeB.next = nodeC

        self.__tail = Node("D")
        self.__tail.next = self.__head
        self.__tail.prev = nodeC
        nodeC.next = self.__tail
        self.__head.prev = self.__tail
```

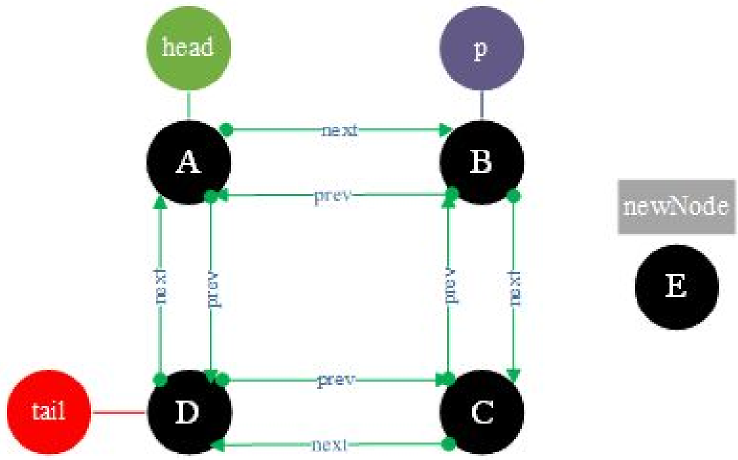
```python
    def output(self):
        p = self.__head
        print(p.data, " -> ", end="")
        p = p.next;
        while p != self.__head:
            print(p.data, " -> ", end="")
            p = p.next
        print(p.data, "\n\n", end="")

        p = self.__tail
        print(p.data, " -> ", end="")
        p = p.prev
        while p != self.__tail:
            print(p.data, " -> ", end="")
            p = p.prev

        print(p.data, "\n\n", end="")

def main():
    linkedlist = LinkedList()
    linkedlist.init()
    linkedlist.output()

if __name__ == "__main__":
    main()
```

Result:

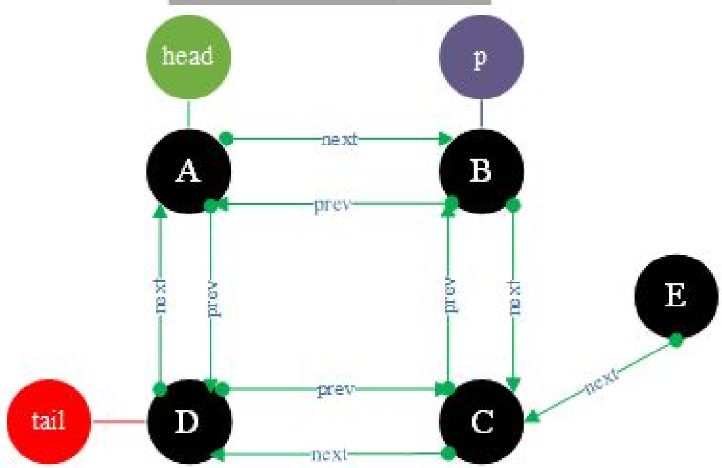
A -> B -> C -> D -> A

D -> C -> B -> A -> D

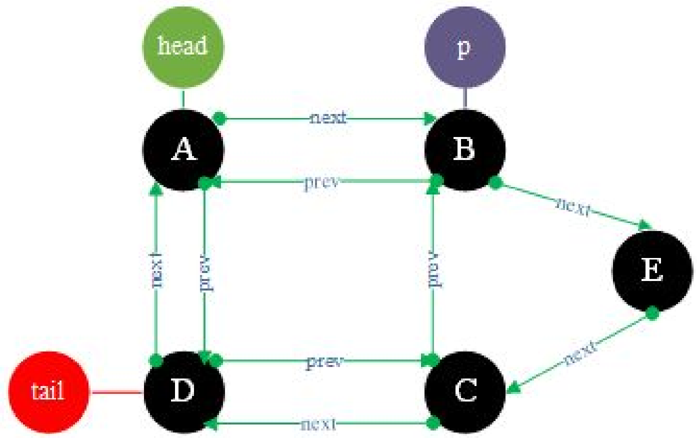
3. Insert a node E in position 2.



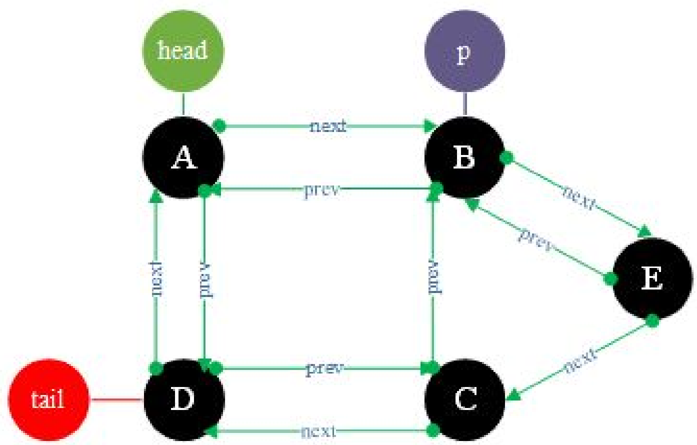
newNode.next = p.next



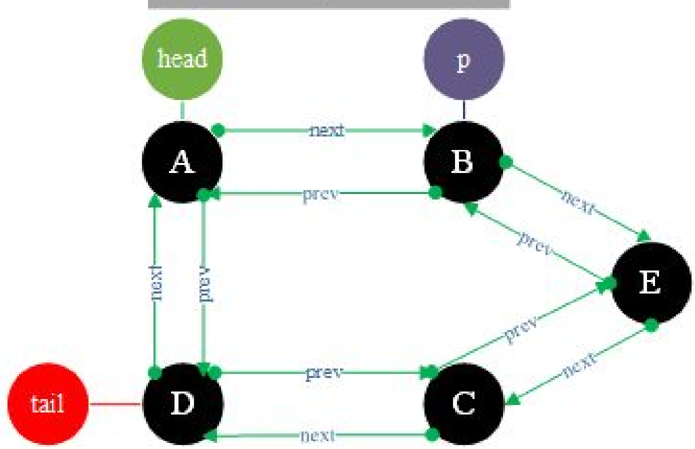
p.next = newNode



newNode.prev = p



newNode.next.prev = newNode



test_twoway_circular_linkedlist.py

```
class Node:
    data = ''
    next = None
    prev = None

    def __init__(self, data):
        self.data = data

class LinkedList:
    __head = None
    __tail = None

    def init(self):
        self.__head = Node("A")
        self.__head.next = None
        self.__head.prev = None

        nodeB = Node("B")
        nodeB.next = None
        nodeB.prev = self.__head
        self.__head.next = nodeB

        nodeC = Node("C")
        nodeC.next = None
        nodeC.prev = nodeB
        nodeB.next = nodeC

        self.__tail = Node("D")
        self.__tail.next = self.__head
        self.__tail.prev = nodeC
        nodeC.next = self.__tail
        self.__head.prev = self.__tail
```

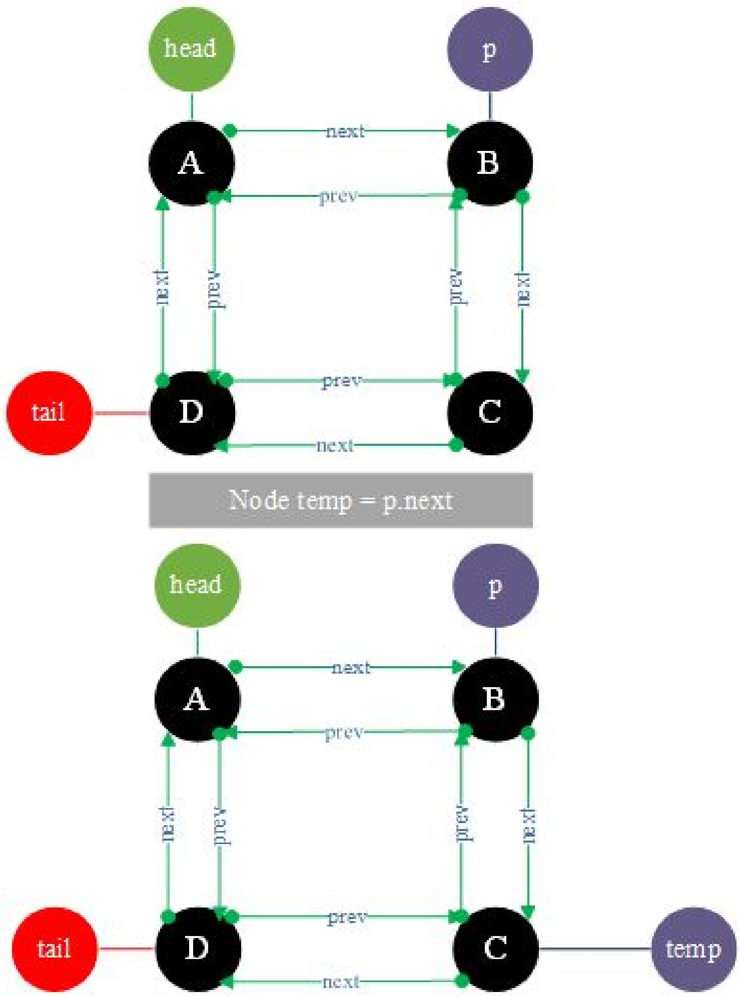
```python
def insert(self, insert_position, new_node):
    p = self.__head
    i = 0
    #Move the node to the insertion position
    while p.next != None and i < insert_position - 1:
        p = p.next
        i+=1
    new_node.next = p.next
    p.next = new_node
    new_node.prev = p
    new_node.next.prev = new_node

def output(self):
    p = self.__head
    print(p.data, " -> ", end="")
    p = p.next;
    while p != self.__head:
        print(p.data, " -> ", end="")
        p = p.next
    print(p.data, "\n\n", end="")

    p = self.__tail
    print(p.data, " -> ", end="")
    p = p.prev
    while p != self.__tail:
        print(p.data, " -> ", end="")
        p = p.prev
```

```
        print(p.data, "\n\n", end="")

def main():
    linkedlist = LinkedList()
    linkedlist.init()
    print("Insert a new node E at index = 2 : ")
    linkedlist.insert(2,Node("E"))
    linkedlist.output()

if __name__ == "__main__":
    main()
```

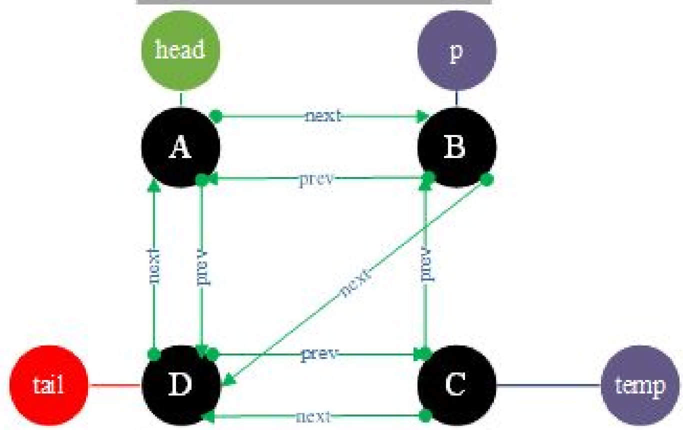
Result:

Insert a new node E at index = 2 :
A -> B -> E -> C -> D -> A
D -> C -> E -> B -> A -> D

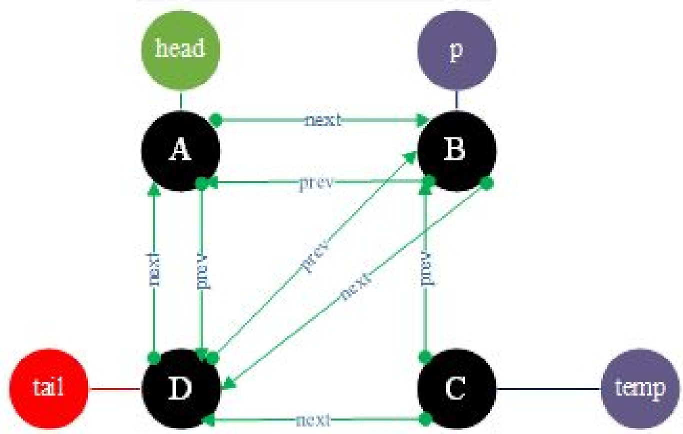
4. Delete the index=2 node.



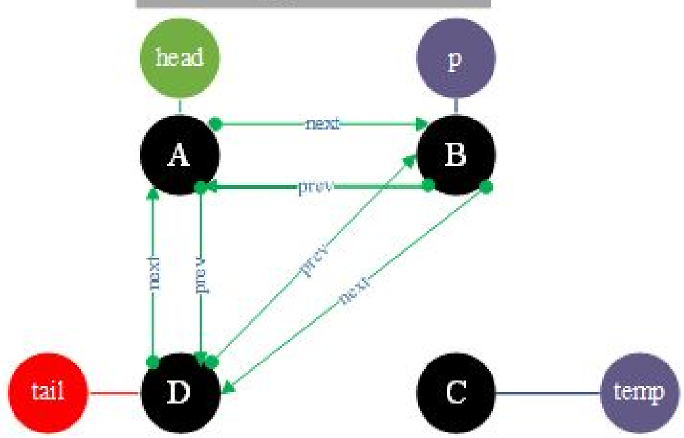
p.next = p.next.next



p.next.prev = p



temp.next = null
temp.prev = null



test_twoway_circular_linkedlist.py

```
class Node:
    data = ''
    next = None
    prev = None

    def __init__(self, data):
        self.data = data

class LinkedList:
    __head = None
    __tail = None

    def init(self):
        self.__head = Node("A")
        self.__head.next = None
        self.__head.prev = None

        nodeB = Node("B")
        nodeB.next = None
        nodeB.prev = self.__head
        self.__head.next = nodeB

        nodeC = Node("C")
        nodeC.next = None
        nodeC.prev = nodeB
        nodeB.next = nodeC

        self.__tail = Node("D")
        self.__tail.next = self.__head
        self.__tail.prev = nodeC
        nodeC.next = self.__tail
        self.__head.prev = self.__tail
```

```python
def remove(self, remove_position):
    p = self.__head
    i = 0
    while p.next != None and i < remove_position - 1:
        p = p.next
        i+=1
    temp = p.next
    p.next = p.next.next
    p.next.prev = p
    temp.next = None # he delete node next to null
    temp.prev = None # Set the delete node prev to null

def output(self):
    p = self.__head
    print(p.data, " -> ", end="")
    p = p.next;
    while p != self.__head:
        print(p.data, " -> ", end="")
        p = p.next
    print(p.data, "\n\n", end="")

    p = self.__tail
    print(p.data, " -> ", end="")
    p = p.prev
    while p != self.__tail:
        print(p.data, " -> ", end="")
        p = p.prev
```

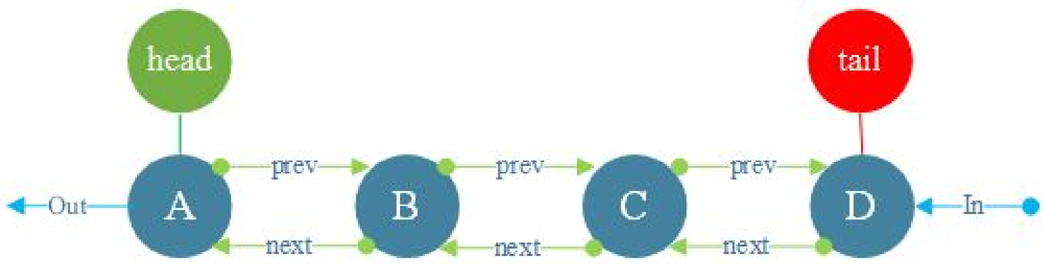
```python
        print(p.data, "\n\n", end="")

def main():
    linkedlist = LinkedList()
    linkedlist.init()
    print("Delet a node C at index = 2 : ")
    linkedlist.remove(2)
    linkedlist.output()

if __name__ == "__main__":
    main()
```

Result:

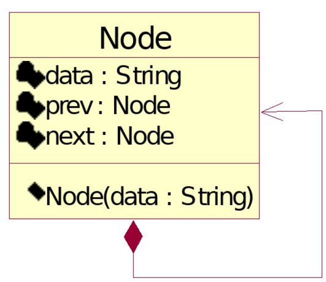
```
Delet a node C at index = 2 :
A  -> B  -> D  -> A
D  -> B  -> A  -> D
```

Queue

Queue:
FIFO (First In First Out) sequence.



UML Diagram

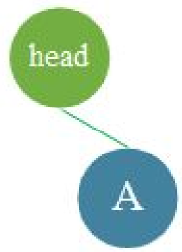



```
class Node:

    data = ''
    next = None
    prev = None

    def __init__(self, data):
        self.data = data
```

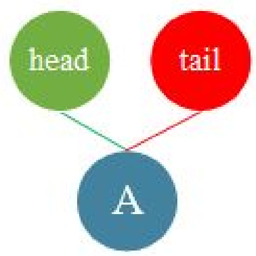
1. Queue initialization and traversal output.

Initialization Insert A

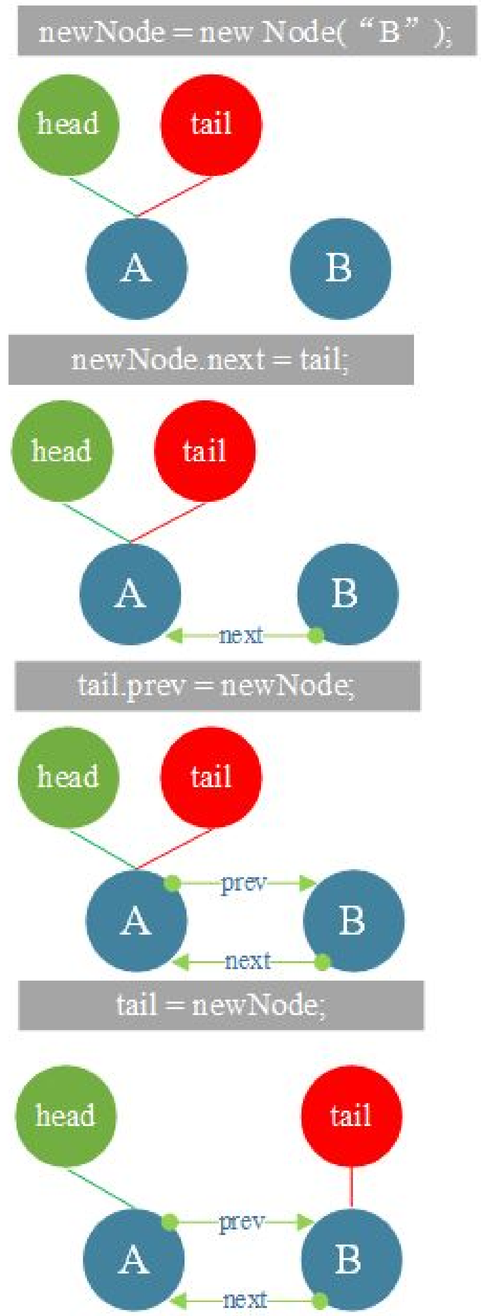
```
head = new Node( “A” );
```



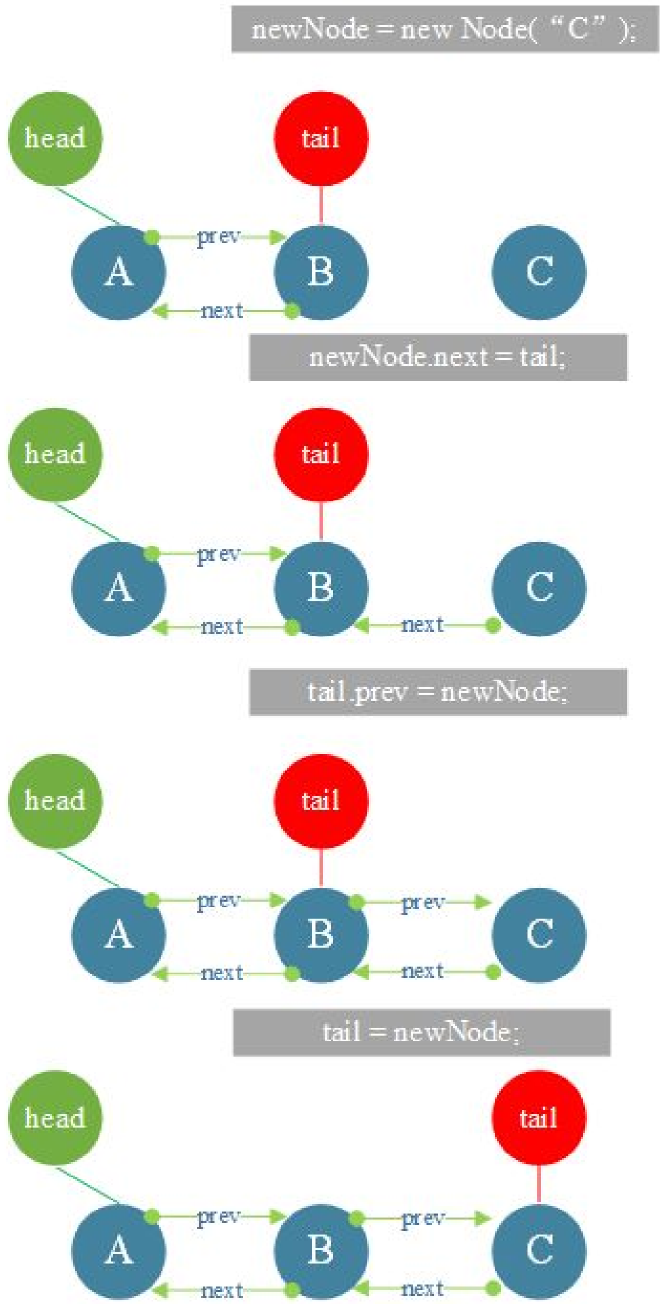

```
tail = head;
```



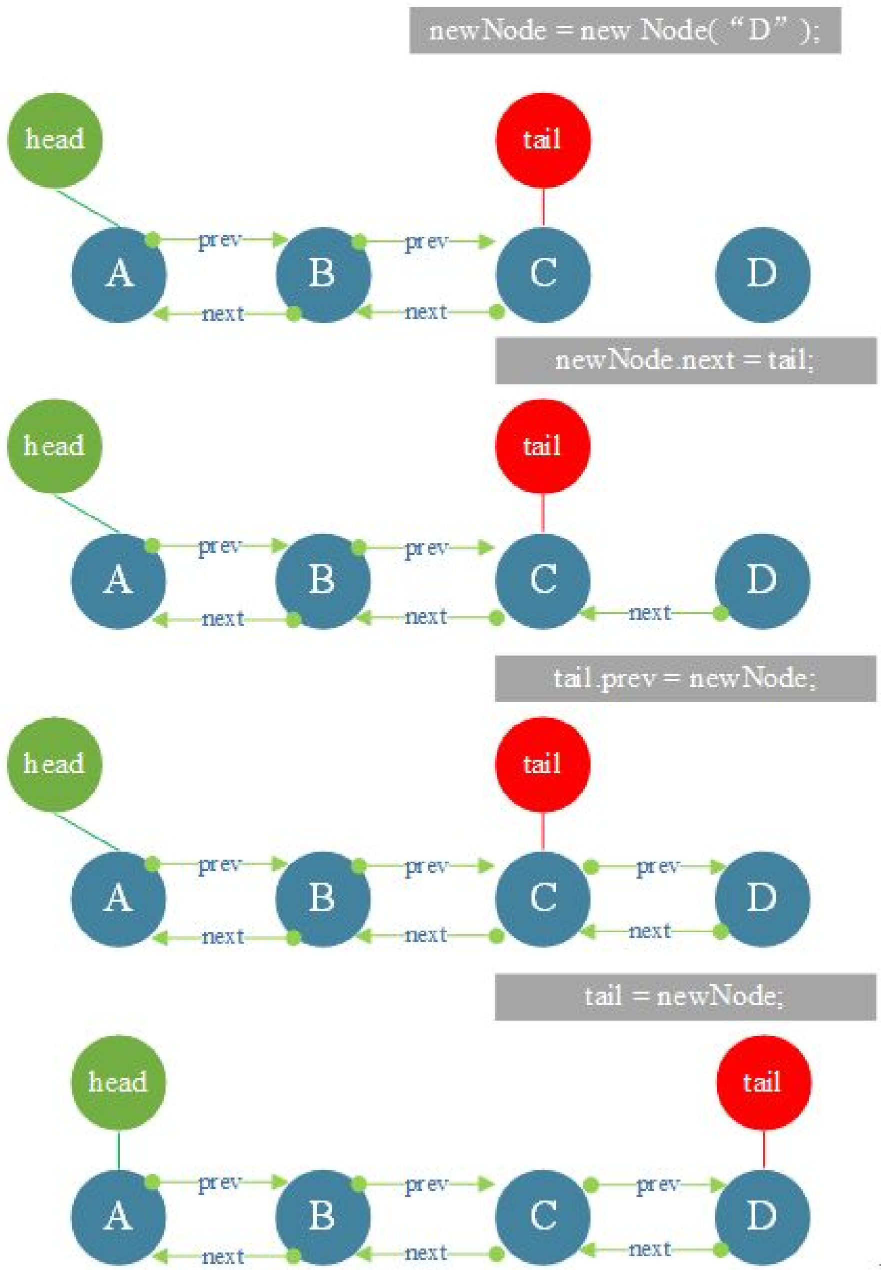
Initialization Insert B



Initialization Insert C



Initialization Insert D



TestQueue.py

```
class Node:
    data = ''
    next = None
    prev = None

    def __init__(self, data):
        self.data = data

class Queue:
    __head = None
    __tail = None
    __size = 0

    def offer(self, element):
        if self.__head == None:
            self.__head = Node(element)
            self.__tail = self.__head
        else:
            new_node = Node(element)
            new_node.next = self.__tail
            self.__tail.prev = new_node
            self.__tail = new_node
        self.__size+=1

    def poll(self):
        p = self.__head
        if p == None:
            return None
        self.__head = self.__head.prev
        p.next = None
        p.prev = None
        self.__size-=1
```

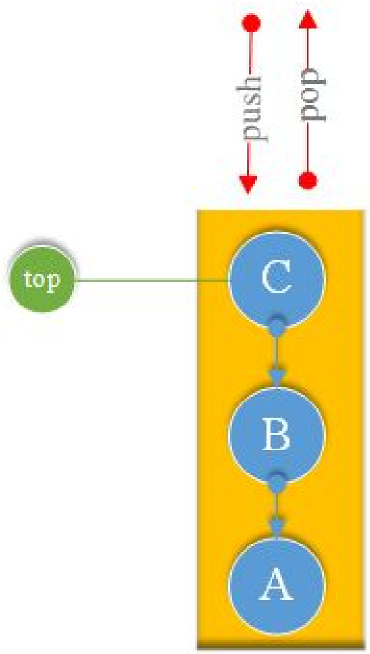
```
        return p

    @property
    def size(self):
        return self.__size




    def output(self, queue):
        print("Head ", end="")
        node = queue.poll()
        while node != None:
            print(node.data, " <- ", end="")
            node = queue.poll()
        print("Tail\n")

def main():
    queue = Queue()
    queue.offer("A")
    queue.offer("B")
    queue.offer("C")
    queue.offer("D")
    queue.output(queue)

if __name__ == "__main__":
    main()
```

Result:

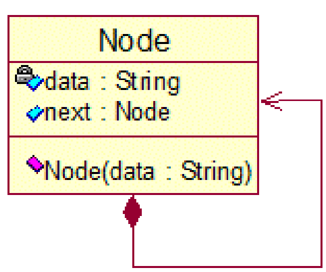
Head A <- B <- C <- D <- Tail

Stack

Stack:
FILO (First In Last Out) sequence.



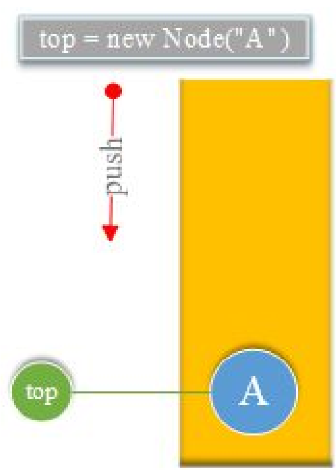
UML Diagram



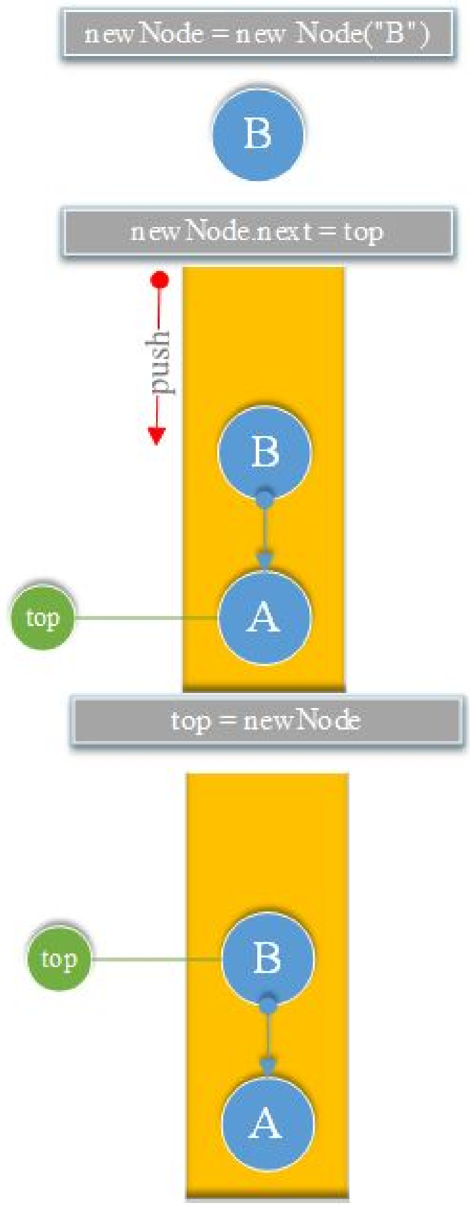
```python
class Node:

    data = ''
    next = None

    def __init__(self, data):
        self.data = data
```

1. Stack initialization and traversal output.

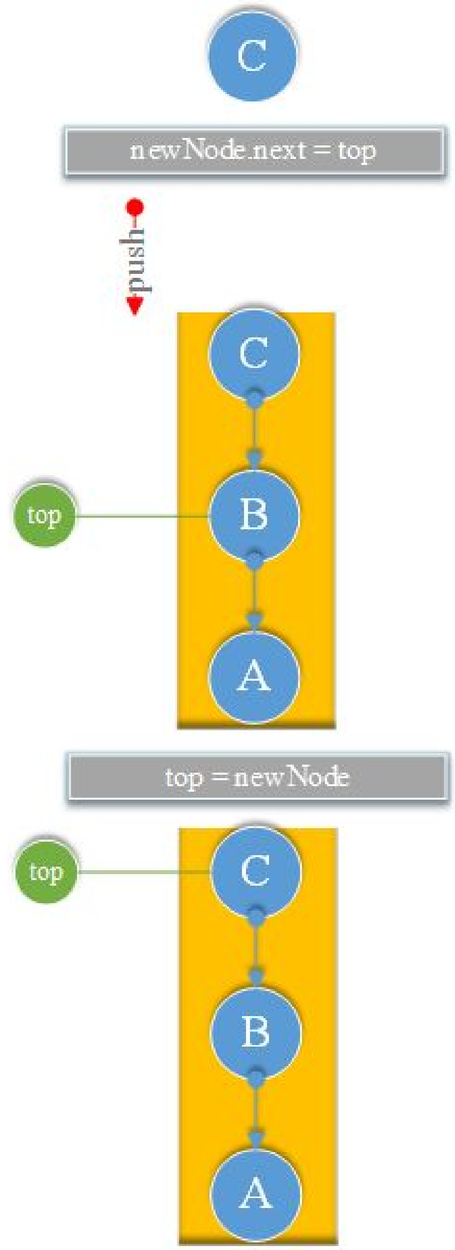
Push A into Stack



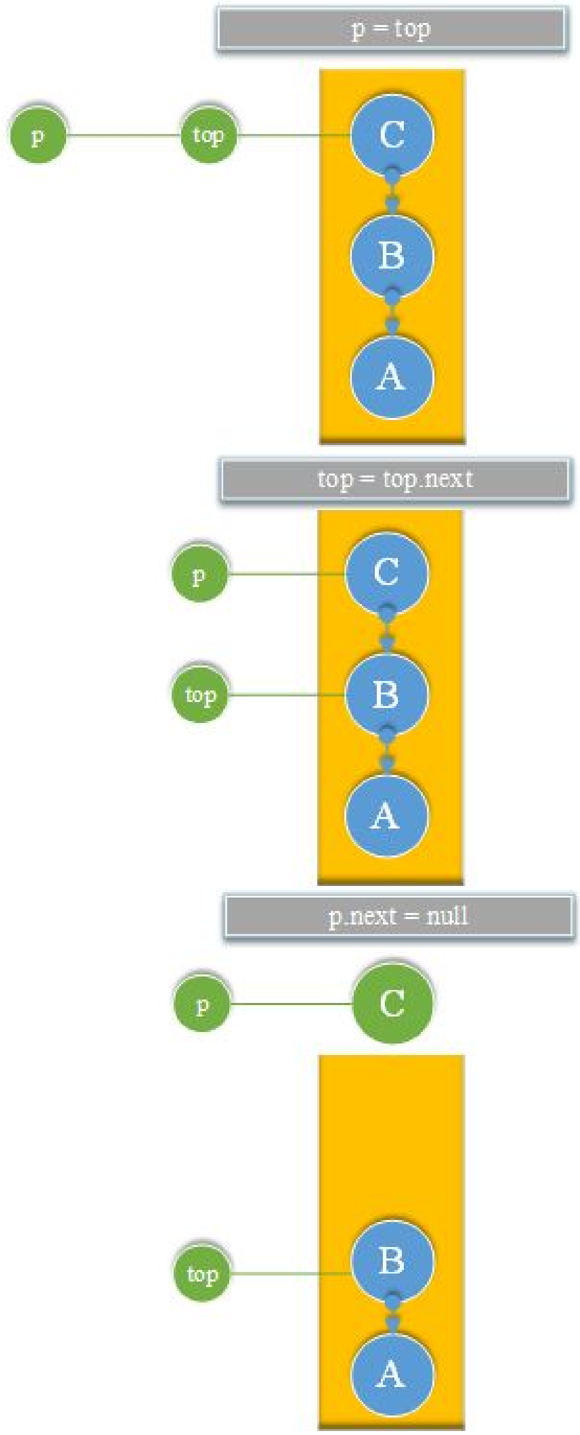
Push **B** into Stack



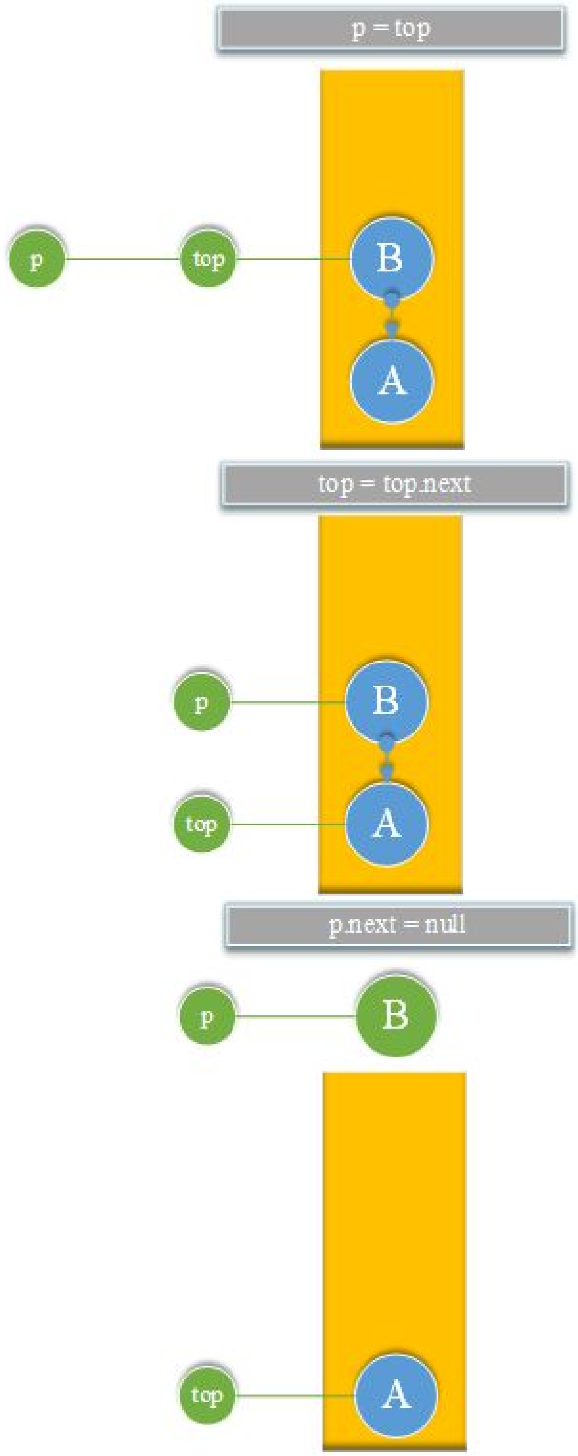
Push C into Stack



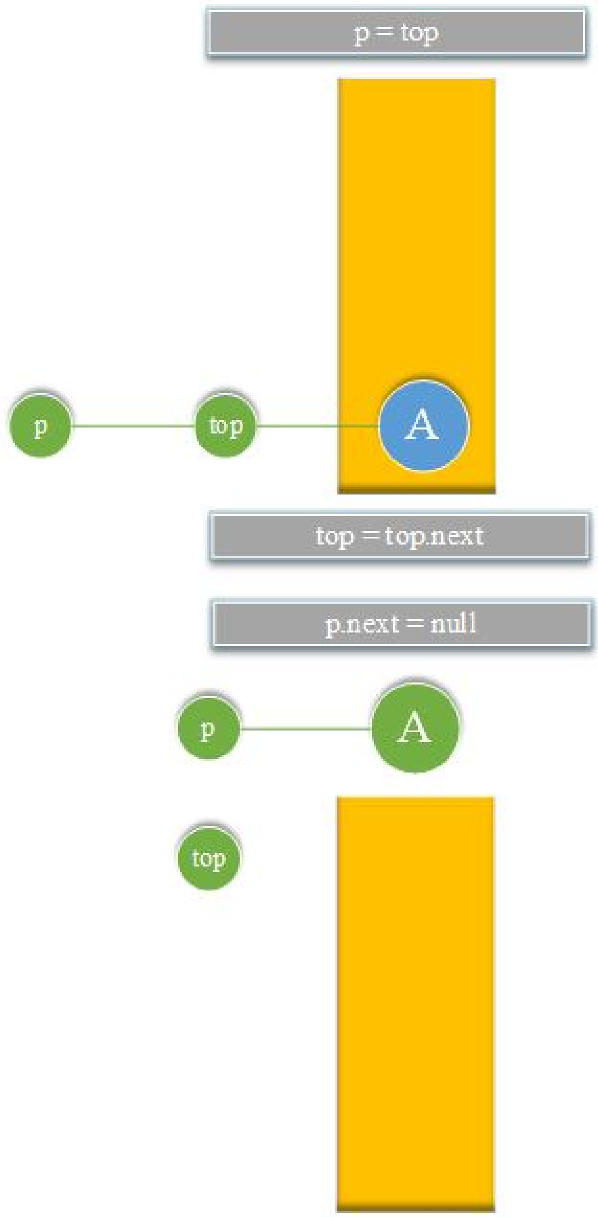
If pop C from Stack:



If pop B from Stack:



If pop A from Stack:



test_stack.py

```
class Node:
    data = ''
    next = None

    def __init__(self, data):
        self.data = data

class Stack:
    __top = None
    __size = 0

    def push(self, element):
        if self.__top == None:
            self.__top = Node(element)
        else:
            new_node = Node(element)
            new_node.next = self.__top
            self.__top = new_node
        self.__size+=1

    def pop(self):
        if self.__top == None:
            return None
        p = self.__top
        self.__top = self.__top.next # top move down
        p.next = None
        self.__size-=1
        return p

    @property
    def size(self):
        return self.__size
```

```
    def output(self, stack):
        print("Top ", end="")
        node = stack.pop()
        while node != None:
            print(node.data, " -> ", end="")
            node = stack.pop()
        print("End\n")

def main():
    stack = Stack()
    stack.push("A")
    stack.push("B")
    stack.push("C")
    stack.push("D")
    stack.output(stack)

if __name__ == "__main__":
    main()
```

Result:

Top D -> C -> B -> A -> End

Recursive Algorithm

Recursive Algorithm:
The program function itself calls its own layer to progress until it reaches a certain condition and step by step returns to the end..

1. Factorial of n : n*(n-1)*(n-2) *2*1

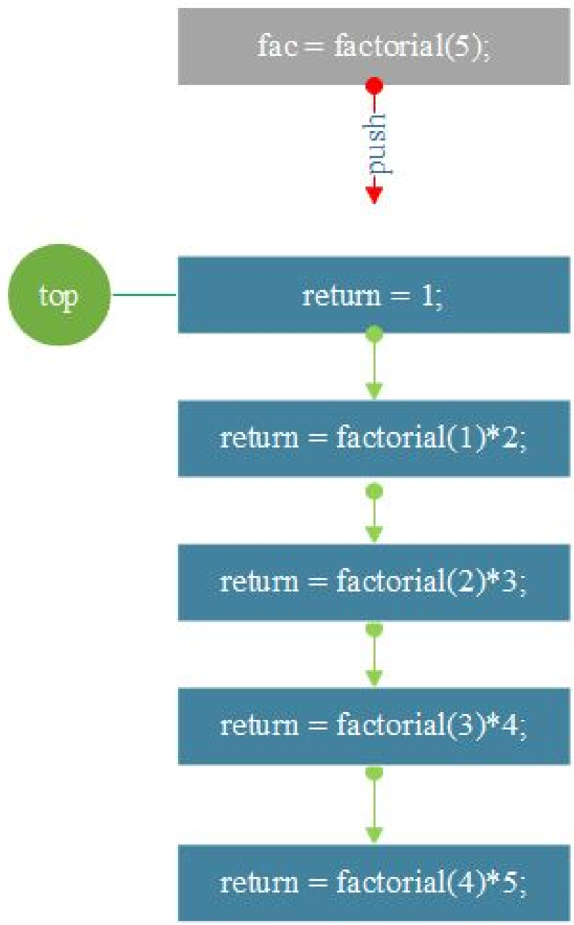
TestFactorial.py

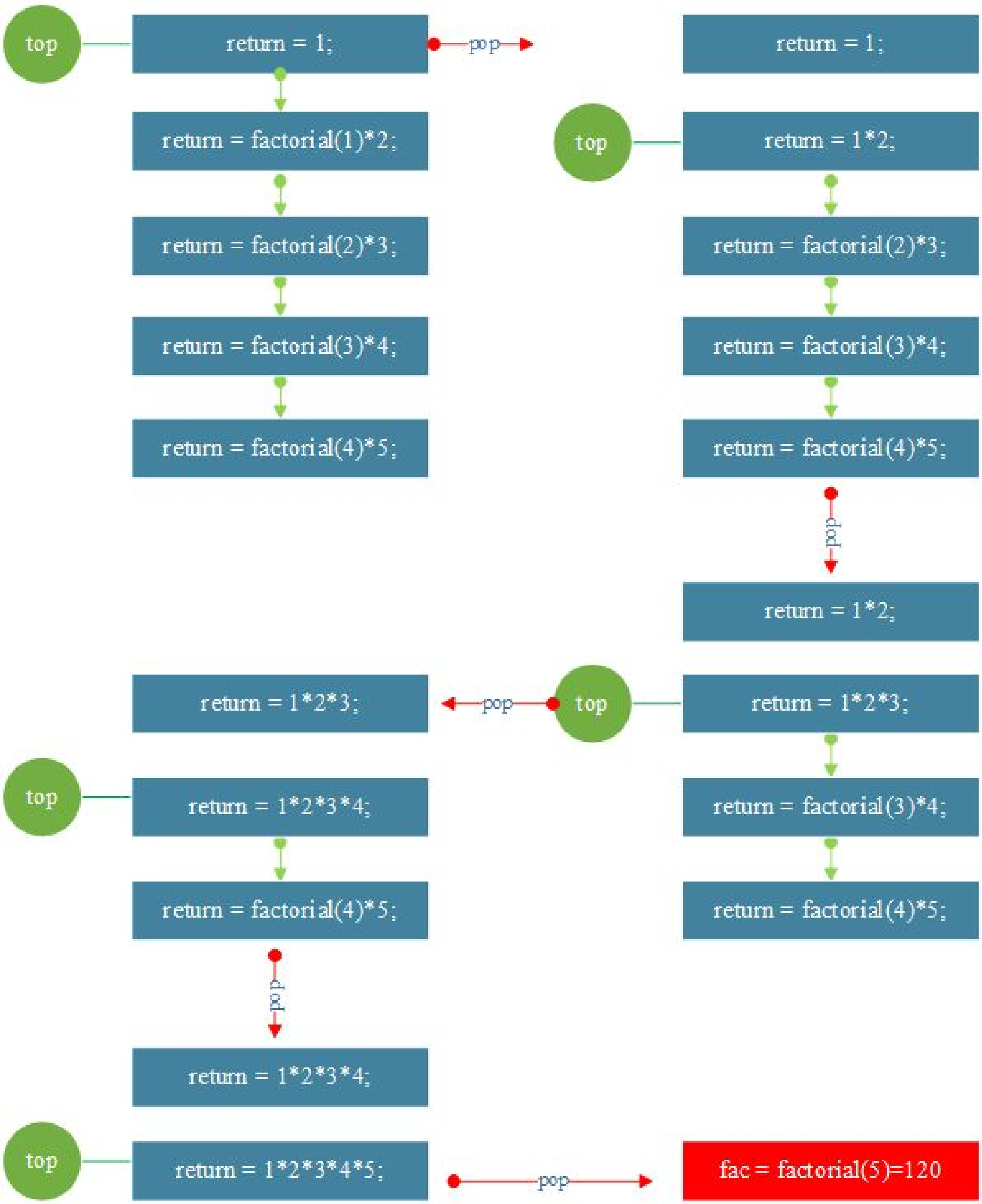
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return factorial(n - 1) * n # Recursively call yourself until the end of the return

def main():
    n = 5
    fac = factorial(n)
    print("The factorial of 5 is :", fac)

if __name__ == "__main__":
    main()
```

Result:

The factorial of 5 is : 120

Graphical analysis:



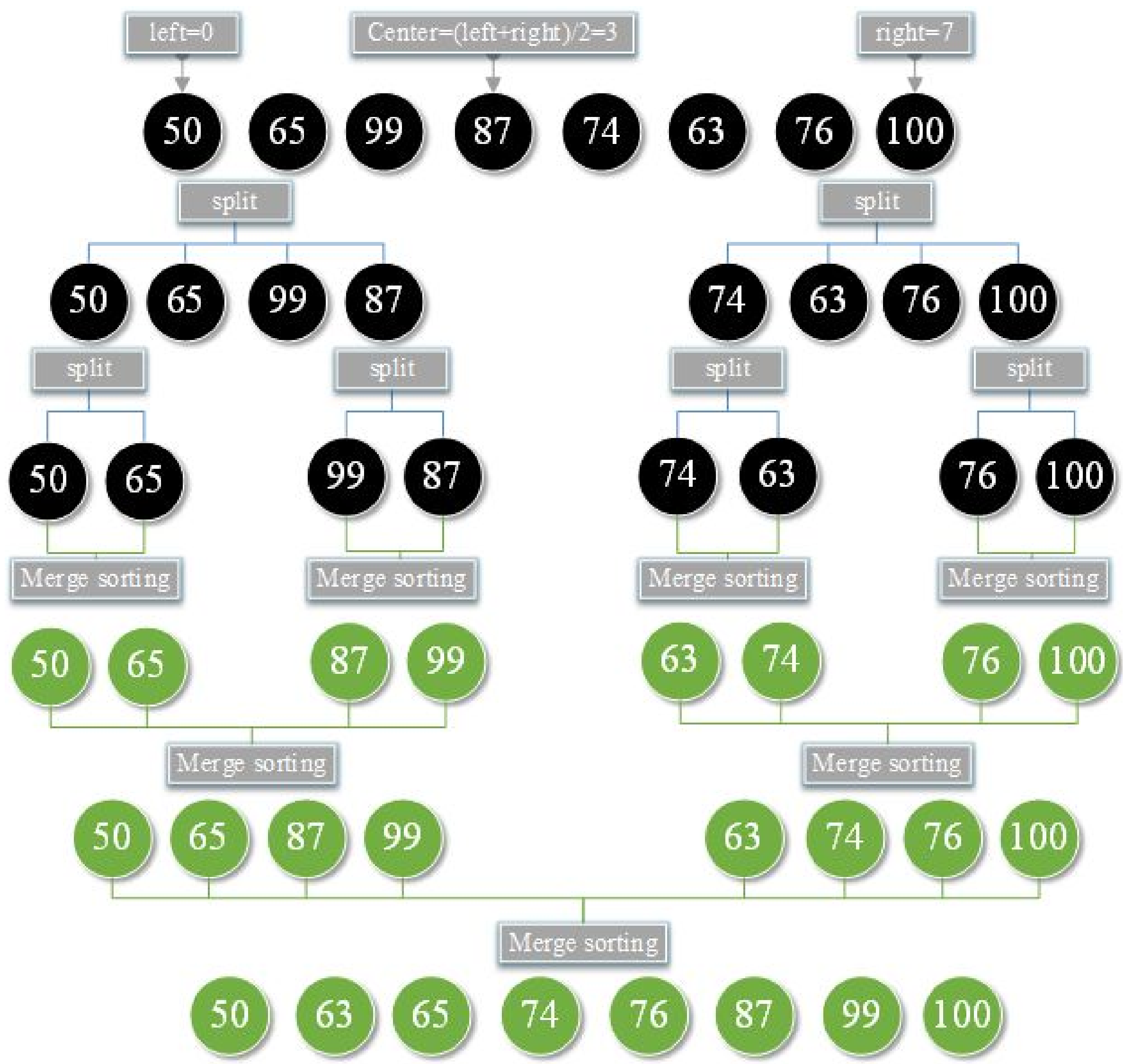


top
return = 1;
pop
return = 1;
return = factorial(1)*2;
top
return = 1*2;
return = factorial(2)*3;
return = factorial(2)*3;
return = factorial(3)*4;
return = factorial(3)*4;
return = factorial(4)*5;
return = factorial(4)*5;
pop
return = 1*2;
return = 1*2*3;
pop
top
return = 1*2*3;
top
return = 1*2*3*4;
return = factorial(3)*4;
return = factorial(4)*5;
return = factorial(4)*5;
pop
return = 1*2*3*4;
top
return = 1*2*3*4*5;
pop
fac = factorial(5)=120

Two-way Merge Algorithm

Two-way Merge Algorithm:
The data of the first half and the second half are sorted, and the two ordered sub-list are merged into one ordered list, which continue to recursive to the end.

1. The scores {50, 65, 99, 87, 74, 63, 76, 100} by merge sort



test_merge_sort.py

```python
from array import array

def sort(array):
    length = len(array)
    temp = [0]*length
    merge_sort(array, temp, 0, length - 1)

def  merge_sort(array, temp, left, right):
    if left < right:
        center = (left + right) // 2
        merge_sort(array, temp, left, center) # Left merge sort
        merge_sort(array, temp, center + 1, right) # Right merge sort
        merge(array, temp, left, center + 1, right) # Merge two ordered
arrays

#Combine two ordered list into an ordered list
def merge(array, temp, left, right, rightEndIndex):
        leftEndIndex = right - 1 #/ End subscript on the left
        tempIndex = left #Starting from the left count
        elementNumber = rightEndIndex - left + 1
        while left <= leftEndIndex and right <= rightEndIndex:
            if array[left] <= array[right]:
                temp[tempIndex] = array[left]
                tempIndex+=1
                left+=1
            else:
                temp[tempIndex] = array[right]
                tempIndex+=1
                right+=1

        while left <= leftEndIndex: # If there is element on the left
            temp[tempIndex] = array[left]
            tempIndex+=1
```

```
        left+=1

    while right <= rightEndIndex: #If there is element on the right
        temp[tempIndex] = array[right]
        tempIndex+=1
        right+=1

    for i in range(0, elementNumber):
        array[rightEndIndex] = temp[rightEndIndex]
        rightEndIndex-=1

def main():
    scores = array('i') # Create an int type array
    scores = array( 'i',[50, 65, 99, 87, 74, 63, 76, 100, 92])
    sort(scores)
    for score in scores:
        print(score, ",", end="")

if __name__ == "__main__":
    main()
```

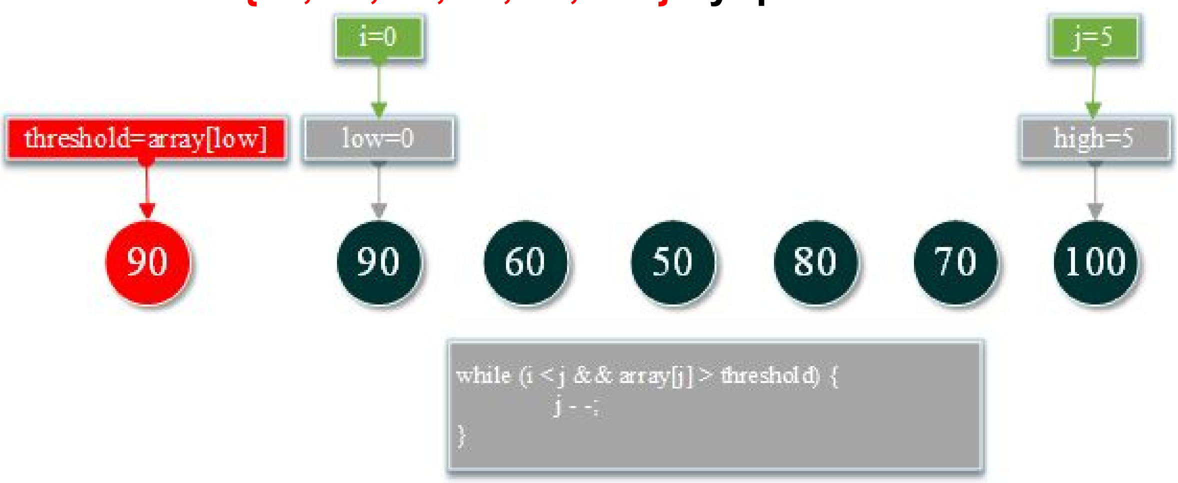
Result:

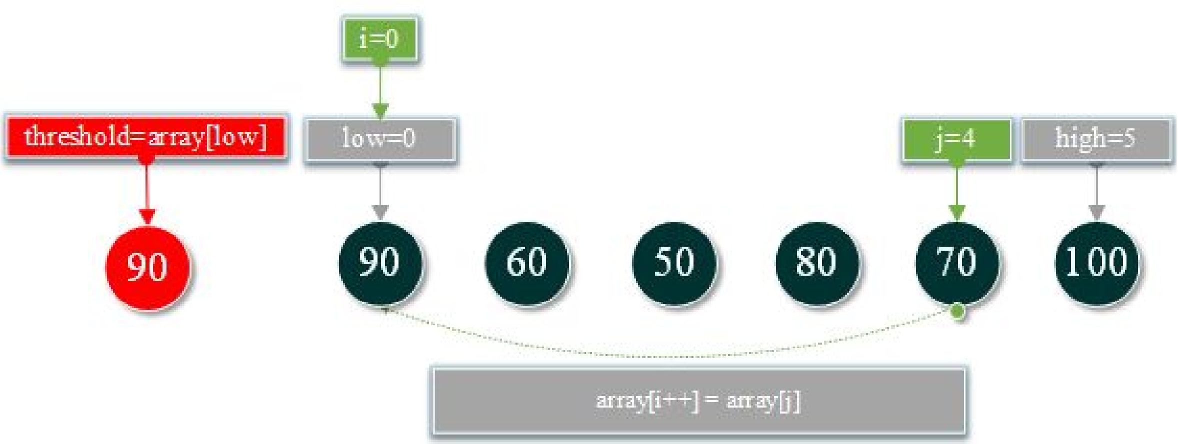
50 ,63 ,65 ,74 ,76 ,87 ,92 ,99 ,100 ,

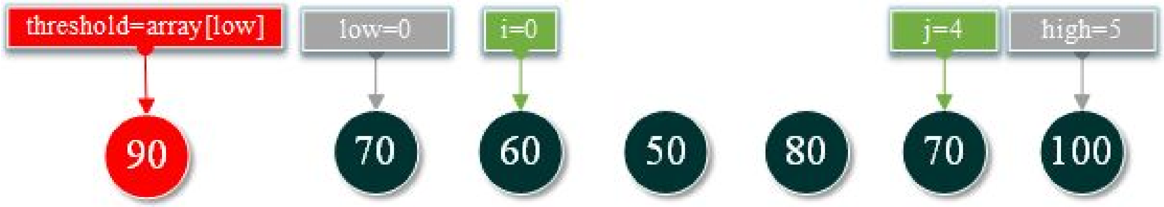
Quick Sort Algorithm

Quick Sort Algorithm:
Quicksort is a popular sorting algorithm that is often faster in practice compared to other sorting algorithms. It utilizes a divide-and-conquer strategy to quickly sort data items by dividing a large array into two smaller arrays.

1. The scores {90, 60, 50, 80, 70, 100} by quick sort

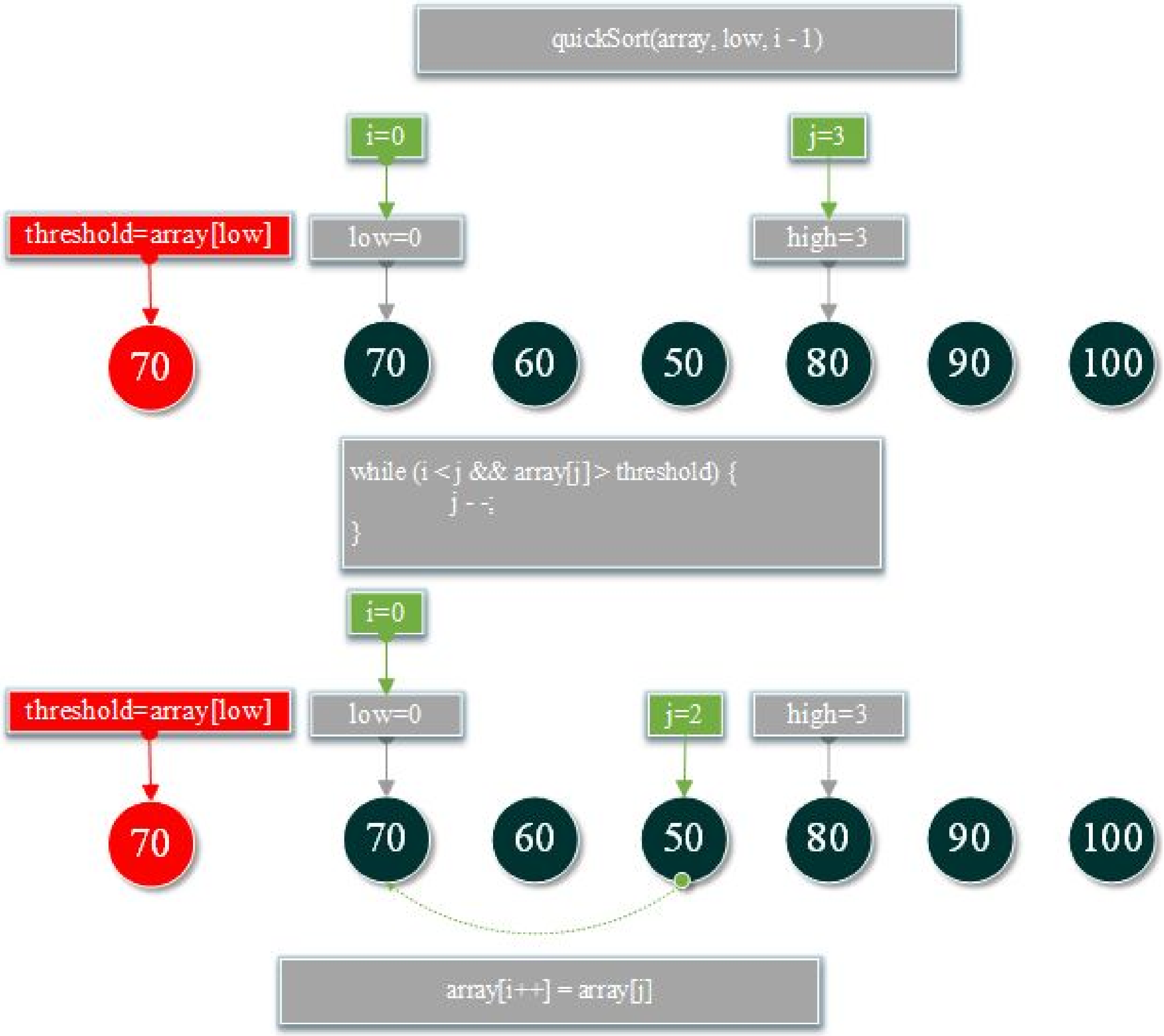




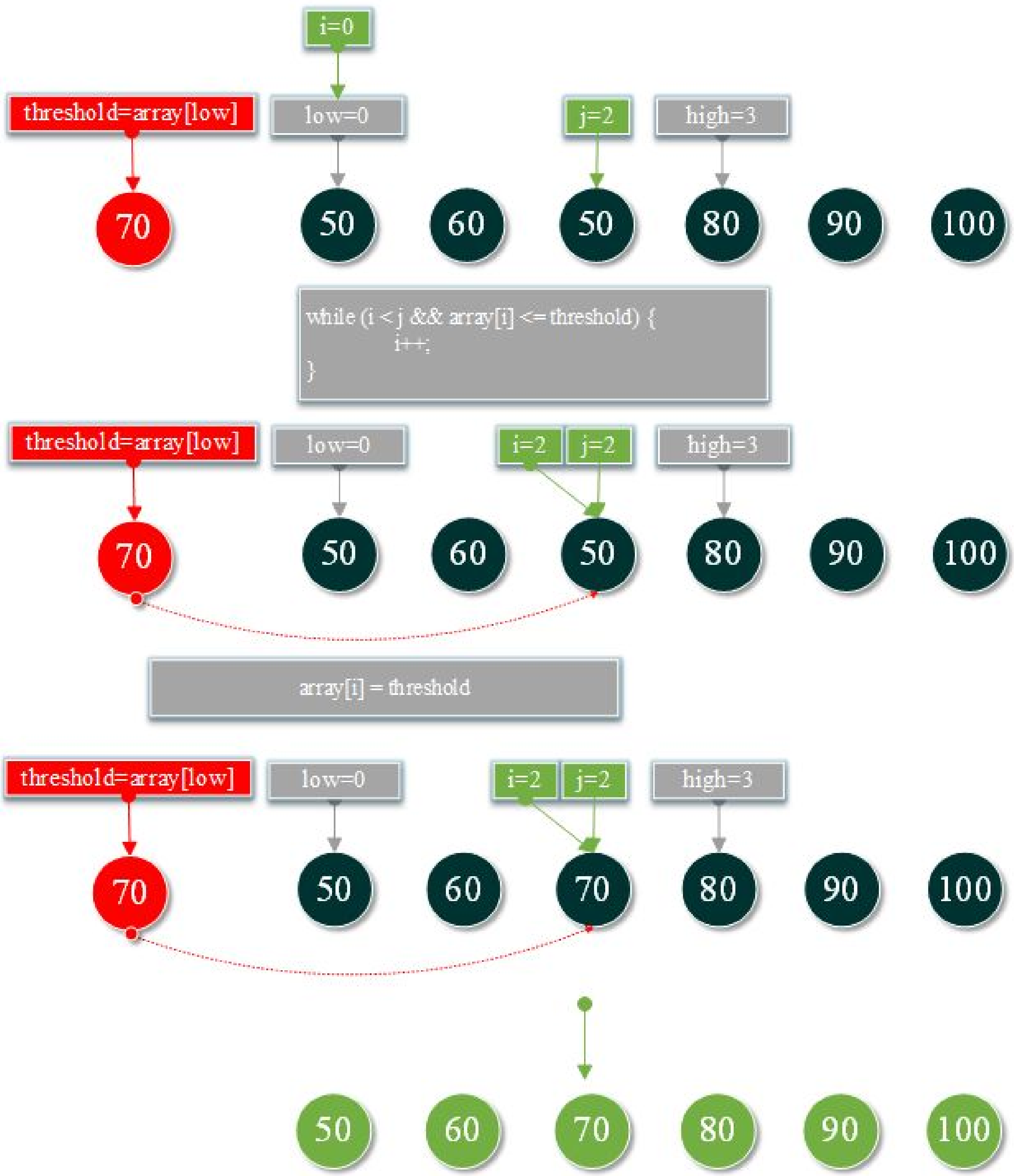



```
while (i < j && array[i] <= threshold) {
        i++;
}
```


```
array[i] = threshold
```



quickSort(array, low, i - 1)
i=0
j=3
threshold=array[low]
low=0
high=3
70
70
60
50
80
90
100
while (i < j && array[j] > threshold) {
j - -;
}
i=0
threshold=array[low]
low=0
j=2
high=3
70
70
60
50
80
90
100
array[i++] = array[j]



i=0
threshold=array[low]
low=0
j=2
high=3
70
50
60
50
80
90
100
while (i < j && array[i] <= threshold) {
i++;
}
threshold=array[low]
low=0
i=2
j=2
high=3
70
50
60
50
80
90
100
array[i] = threshold
threshold=array[low]
low=0
i=2
j=2
high=3
70
50
60
70
80
90
100
50
60
70
80
90
100

test_quick_sort.py

```
from array import array

def sort(array):
   length = len(array)
   if length > 0:
      quick_sort(array, 0, length - 1)

def quick_sort(array, low, high):
      if low > high :
         return

      i = low
      j = high
      threshold = array[low]

      # Alternately scanned from both ends of the list
      while i < j:
         # Find the first position less than threshold from right to left
         while i < j and array[j] > threshold:
            j-=1

         #Replace the low with a smaller number than the threshold
         if i < j:
            array[i] = array[j]
            i+=1

         # Find the first position greater than threshold from left to
right
         while i < j and array[i] <= threshold:
            i+=1

         # Replace the high with a number larger than the threshold
         if i < j:
```

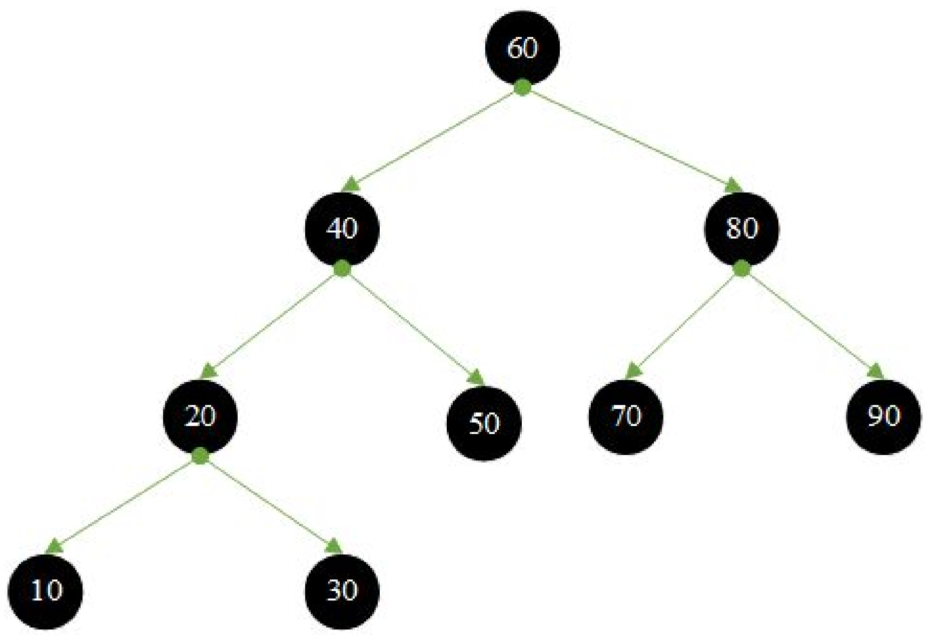
```
            array[j] = array[i]
            j-=1

        array[i] = threshold

        # left quick_sort
        quick_sort(array, low, i - 1)
        # right quick_sort
        quick_sort(array, i + 1, high)


def main():
    scores = array('i') # Create an int type array
    scores = array( 'i',[90, 60, 50, 80, 70, 100])
    sort(scores)
    for score in scores:
        print(score, ",", end="")

if __name__ == "__main__":
    main()
```

Result:

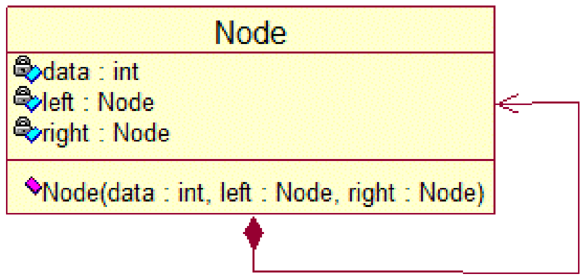
50 ,60 ,70 ,80 ,90 ,100 ,

Binary Search Tree

Binary Search Tree:
1. If the left subtree of any node is not empty, the value of all nodes on the left subtree is less than the value of its root node;
2. If the right subtree of any node is not empty, the value of all nodes on the right subtree is greater than the value of its root node;
3. The left subtree and the right subtree of any node are also binary search trees.



Node UML Diagram

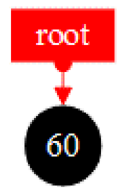



```
class Node:
    data = ''
    left = None
    right = None

    def __init__(self, data, left, right):
        self.data = data;
        self.left = left;
        self.right = right;
```

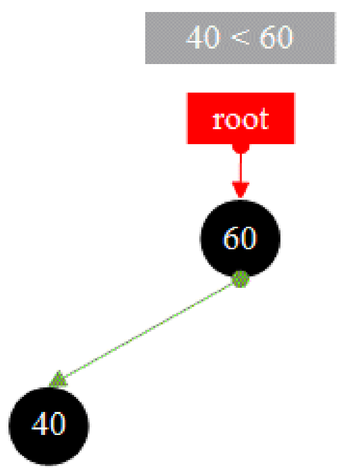
1.Construct a binary search tree, insert node

The inserted nodes are compared from the root node, and the smaller than the root node is compared with the left subtree of the root node, otherwise, compared with the right subtree until the left subtree is empty or the right subtree is empty, then is inserted.

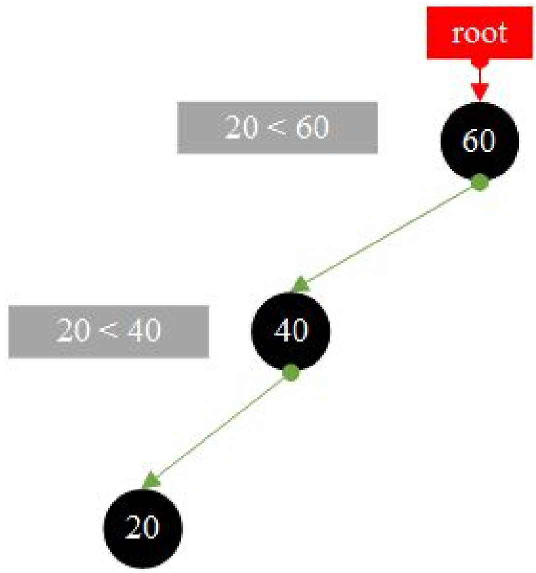
Insert 60



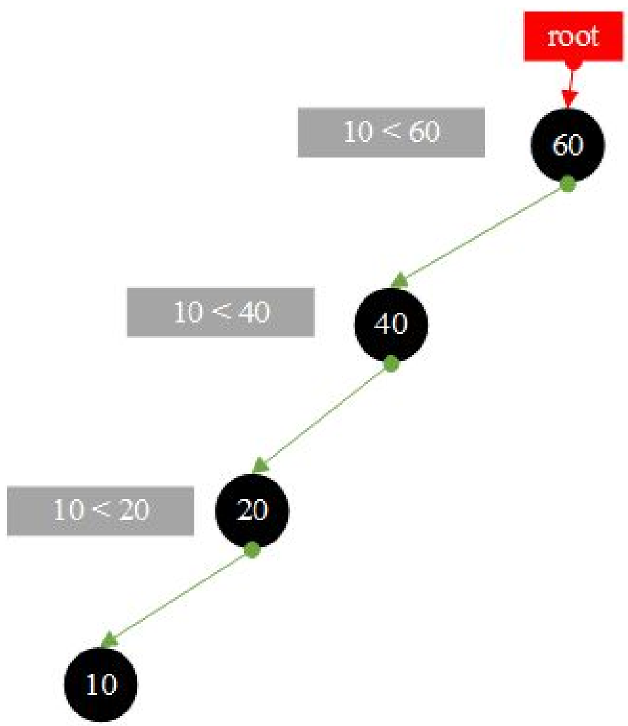
Insert 40



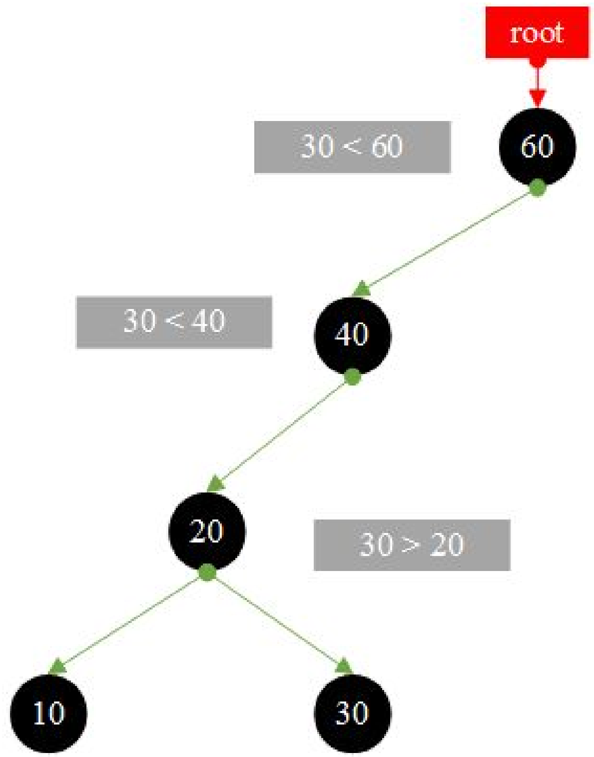
Insert 20



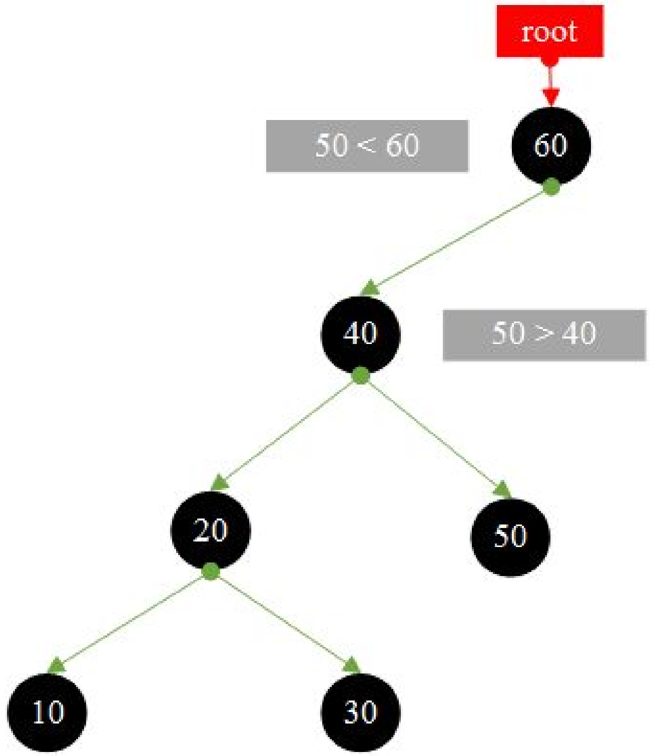
Insert 10



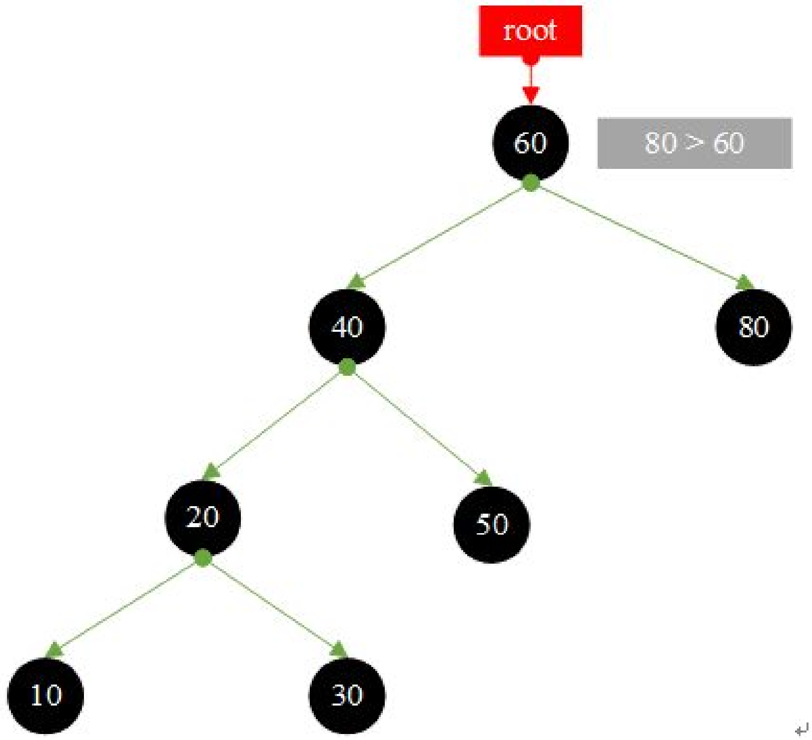
Insert 30



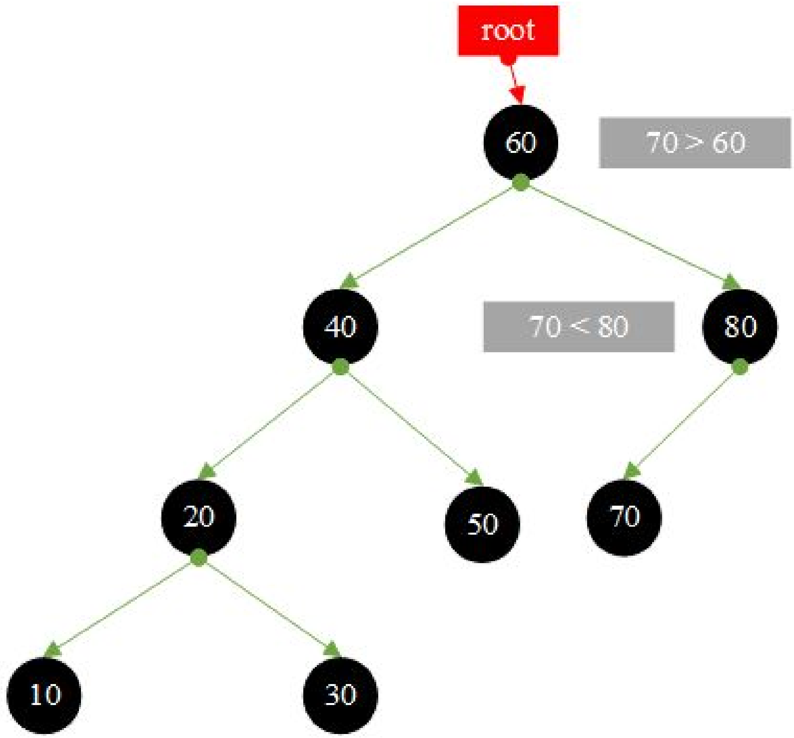
Insert 50



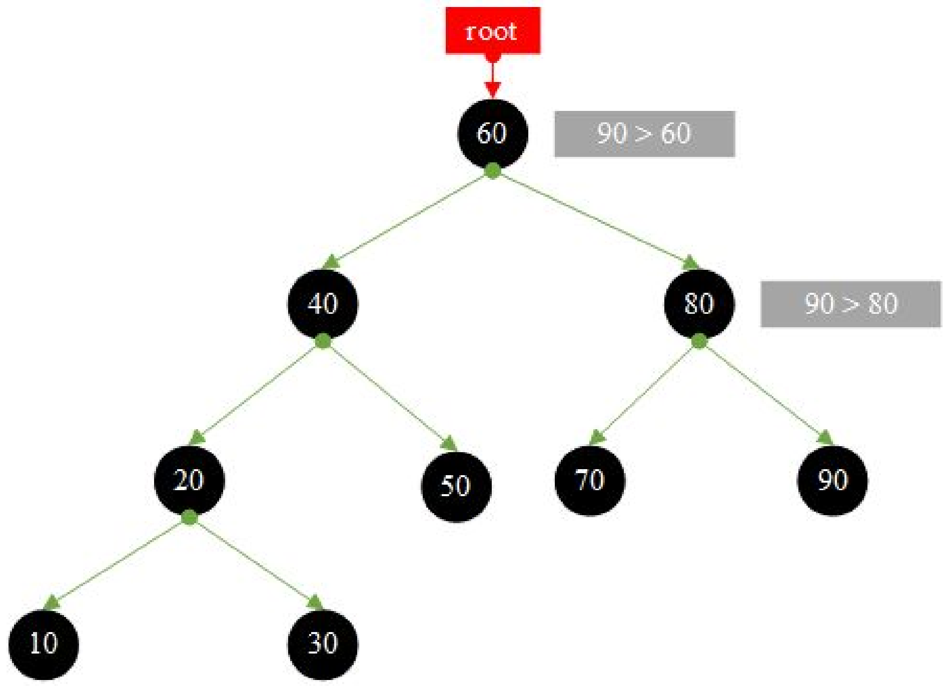
Insert 80



Insert 70

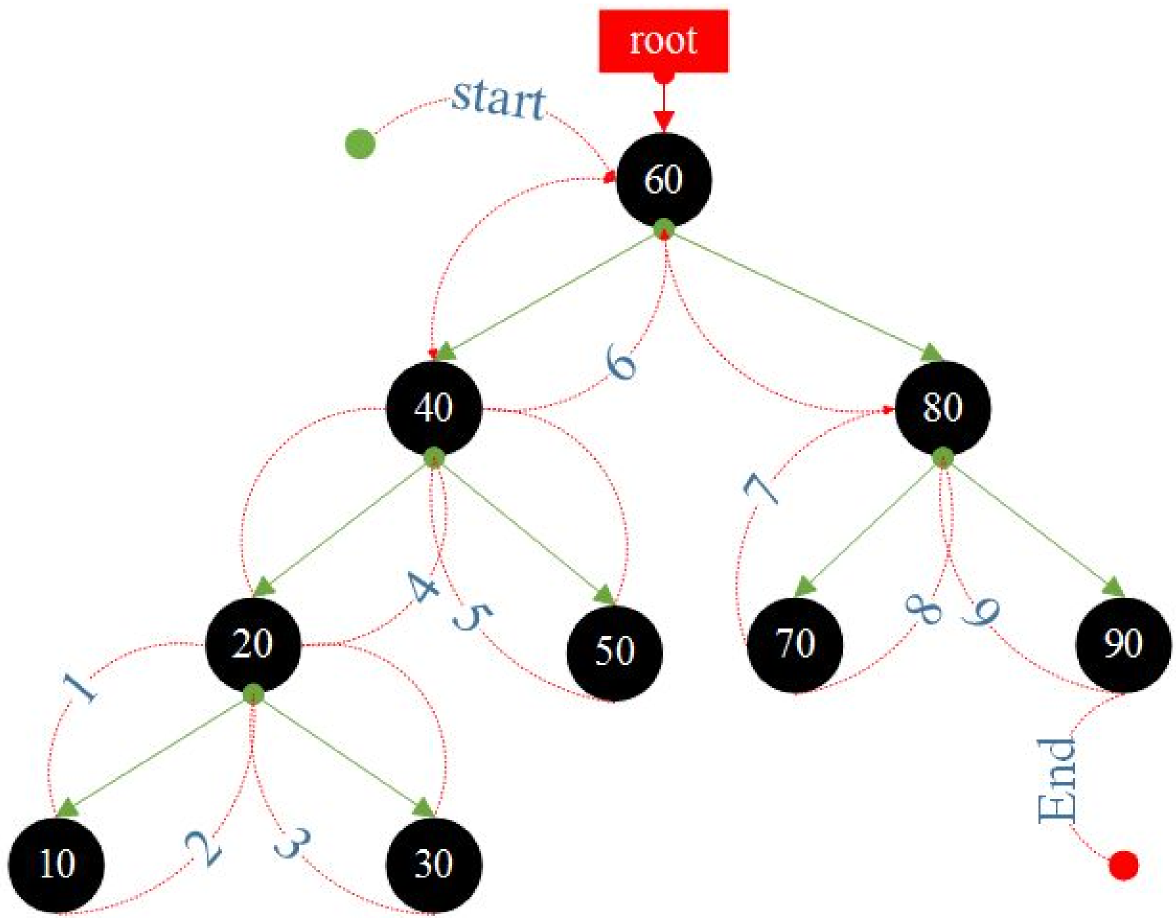


Insert 90

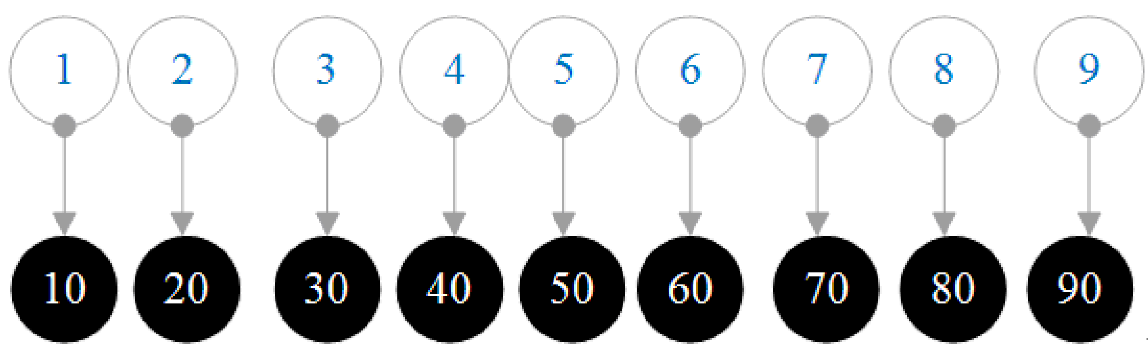


2. binary search tree In-order traversal

In-order traversal : left subtree -> root node -> right subtree



Result:



test_binary_tree.py

```
class Node:
    data = ''
    left = None
    right = None

    def __init__(self, data, left, right):
        self.data = data;
        self.left = left;
        self.right = right;

class BinaryTree:
    __root = None

    @property
    def root(self):
        return self.__root

    def inOrder(self, root):
        if root == None:
            return
        self.inOrder(root.left) # Traversing the left subtree
        print(root.data, ", ", end="")
        self.inOrder(root.right) # Traversing the right subtree

    def insert(self, node, new_data):
        if self.__root == None:
            self.__root = Node(new_data, None, None)
            return
        compareValue = new_data - node.data
        #Recursive left subtree, continue to find the insertion position
        if compareValue < 0:
```

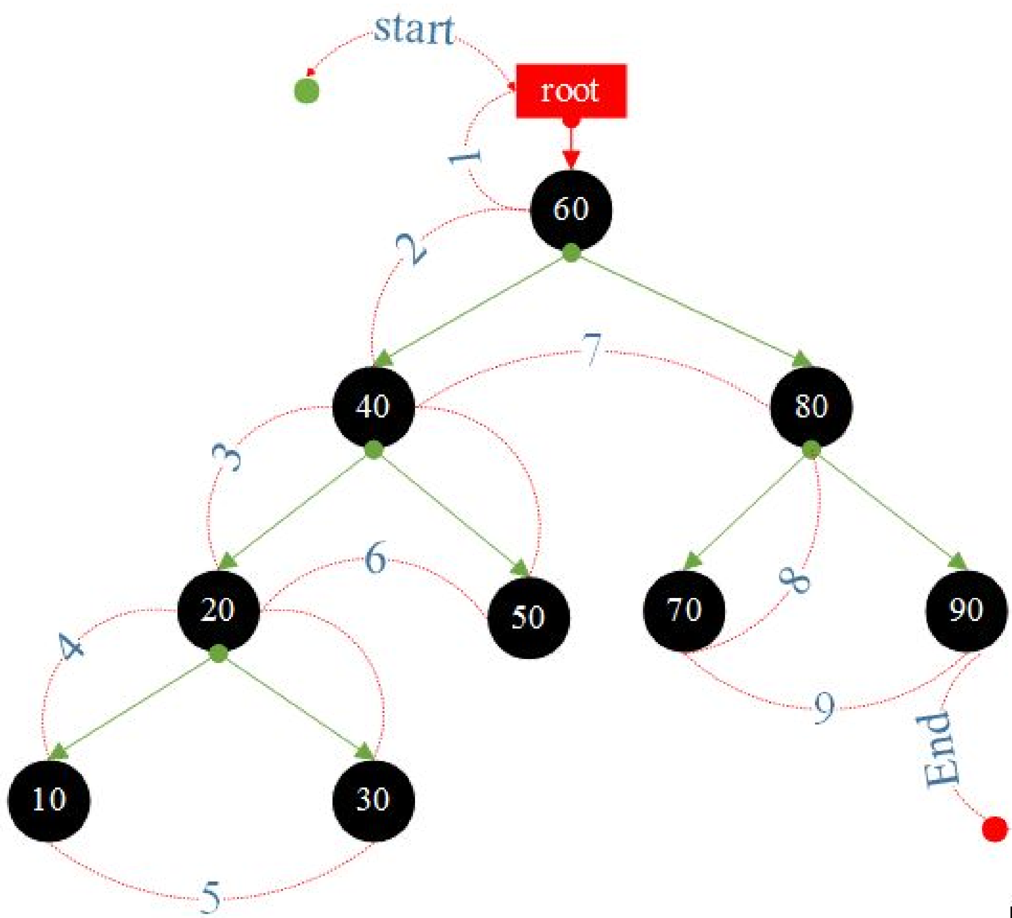
```
            if node.left == None:
                node.left = Node(new_data, None, None)
            else:
                self.insert(node.left, new_data)
        elif compareValue > 0: #Recursive right subtree, continue to find the insertion position
            if node.right == None:
                node.right = Node(new_data, None, None)
            else:
                self.insert(node.right, new_data)

def main():
    binary_tree = BinaryTree()
    #Constructing a binary search tree
    binary_tree.insert(binary_tree.root, 60)
    binary_tree.insert(binary_tree.root, 40)
    binary_tree.insert(binary_tree.root, 20)
    binary_tree.insert(binary_tree.root, 10)
    binary_tree.insert(binary_tree.root, 30)
    binary_tree.insert(binary_tree.root, 50)
    binary_tree.insert(binary_tree.root, 80)
    binary_tree.insert(binary_tree.root, 70)
    binary_tree.insert(binary_tree.root, 90)

    print("\nIn-order traversal binary search tree")
    binary_tree.inOrder(binary_tree.root)

if __name__ == "__main__":
    main()
```

Result:

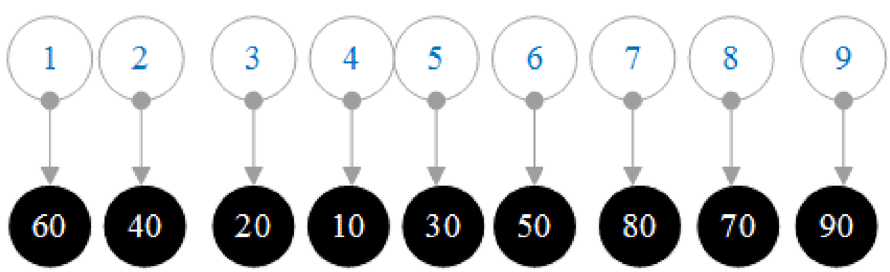
In-order traversal binary search tree
10 , 20 , 30 , 40 , 50 , 60 , 70 , 80 , 90 ,

3. binary search tree Pre-order traversal

Pre-order traversal : root node -> left subtree -> right subtree



Result:



test_binary_tree.py

```
class Node:
    data = ''
    left = None
    right = None

    def __init__(self, data, left, right):
        self.data = data;
        self.left = left;
        self.right = right;

class BinaryTree:
    __root = None

    @property
    def root(self):
        return self.__root

    def preOrder(self,  root):
        if root == None:
            return
        print(root.data, ", ", end="")
        self.preOrder(root.left) # Recursive Traversing the left subtree
        self.preOrder(root.right) # Recursive Traversing the right
subtree

    def insert(self, node, new_data):
        if self.__root == None:
            self.__root = Node(new_data, None, None)
            return
        compareValue = new_data - node.data
        #Recursive left subtree, continue to find the insertion position
        if compareValue < 0:
```

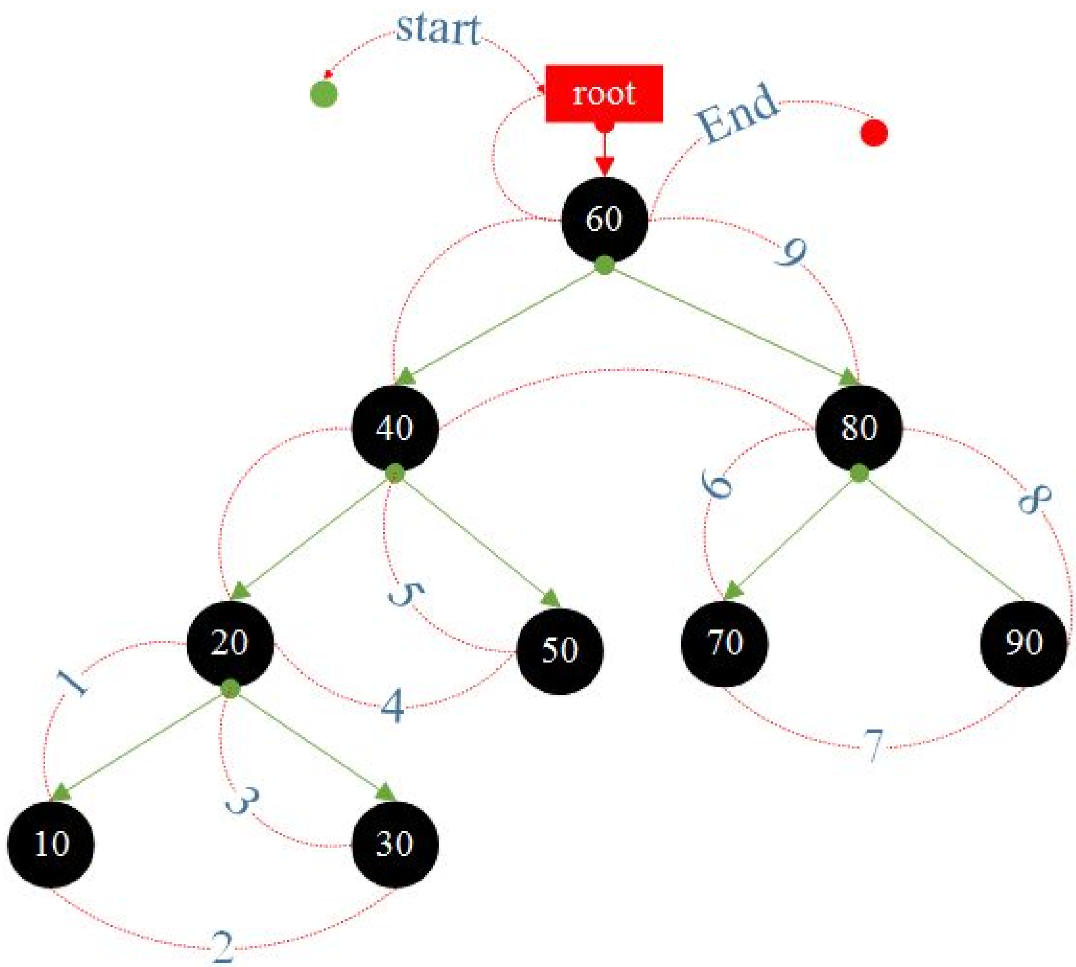
```
            if node.left == None:
                node.left = Node(new_data, None, None)
            else:
                self.insert(node.left, new_data)
        elif compareValue > 0: #Recursive right subtree, continue to find the insertion position
            if node.right == None:
                node.right = Node(new_data, None, None)
            else:
                self.insert(node.right, new_data)


def main():
    binary_tree = BinaryTree()
    #Constructing a binary search tree
    binary_tree.insert(binary_tree.root, 60)
    binary_tree.insert(binary_tree.root, 40)
    binary_tree.insert(binary_tree.root, 20)
    binary_tree.insert(binary_tree.root, 10)
    binary_tree.insert(binary_tree.root, 30)
    binary_tree.insert(binary_tree.root, 50)
    binary_tree.insert(binary_tree.root, 80)
    binary_tree.insert(binary_tree.root, 70)
    binary_tree.insert(binary_tree.root, 90)

    print("Pre-order traversal binary search tree")
    binary_tree.preOrder(binary_tree.root)

if __name__ == "__main__":
    main()
```

Result:

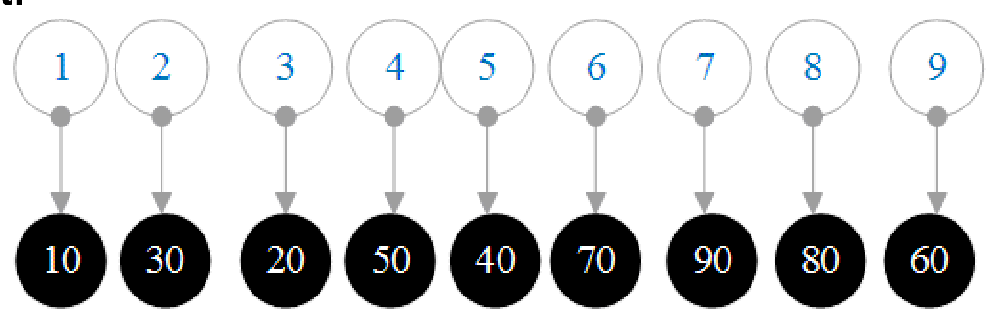
Pre-order traversal binary search tree
60 , 40 , 20 , 10 , 30 , 50 , 80 , 70 , 90 ,

4. binary search tree Post-order traversal

Post-order traversal : right subtree -> root node -> left subtree



Result:



test_binary_tree.py

```
class Node:
    data = ''
    left = None
    right = None

    def __init__(self, data, left, right):
        self.data = data;
        self.left = left;
        self.right = right;

class BinaryTree:
    __root = None

    @property
    def root(self):
        return self.__root

    def postOrder(self, root):
        if root == None:
            return
        self.postOrder(root.left)  # Recursive Traversing the left
subtree
        self.postOrder(root.right) # Recursive Traversing the right
subtree
        print(root.data, ", ", end="")

    def insert(self, node, new_data):
        if self.__root == None:
            self.__root = Node(new_data, None, None)
            return
        compareValue = new_data - node.data
        #Recursive left subtree, continue to find the insertion position
```

```
        if compareValue < 0:
            if node.left == None:
                node.left = Node(new_data, None, None)
            else:
                self.insert(node.left, new_data)
        elif compareValue > 0: #Recursive right subtree, continue to find the insertion position
            if node.right == None:
                node.right = Node(new_data, None, None)
            else:
                self.insert(node.right, new_data)


def main():
    binary_tree = BinaryTree()
    #Constructing a binary search tree
    binary_tree.insert(binary_tree.root, 60)
    binary_tree.insert(binary_tree.root, 40)
    binary_tree.insert(binary_tree.root, 20)
    binary_tree.insert(binary_tree.root, 10)
    binary_tree.insert(binary_tree.root, 30)
    binary_tree.insert(binary_tree.root, 50)
    binary_tree.insert(binary_tree.root, 80)
    binary_tree.insert(binary_tree.root, 70)
    binary_tree.insert(binary_tree.root, 90)

    print("Post-order traversal binary search tree")
    binary_tree.postOrder(binary_tree.root)

if __name__ == "__main__":
    main()
```

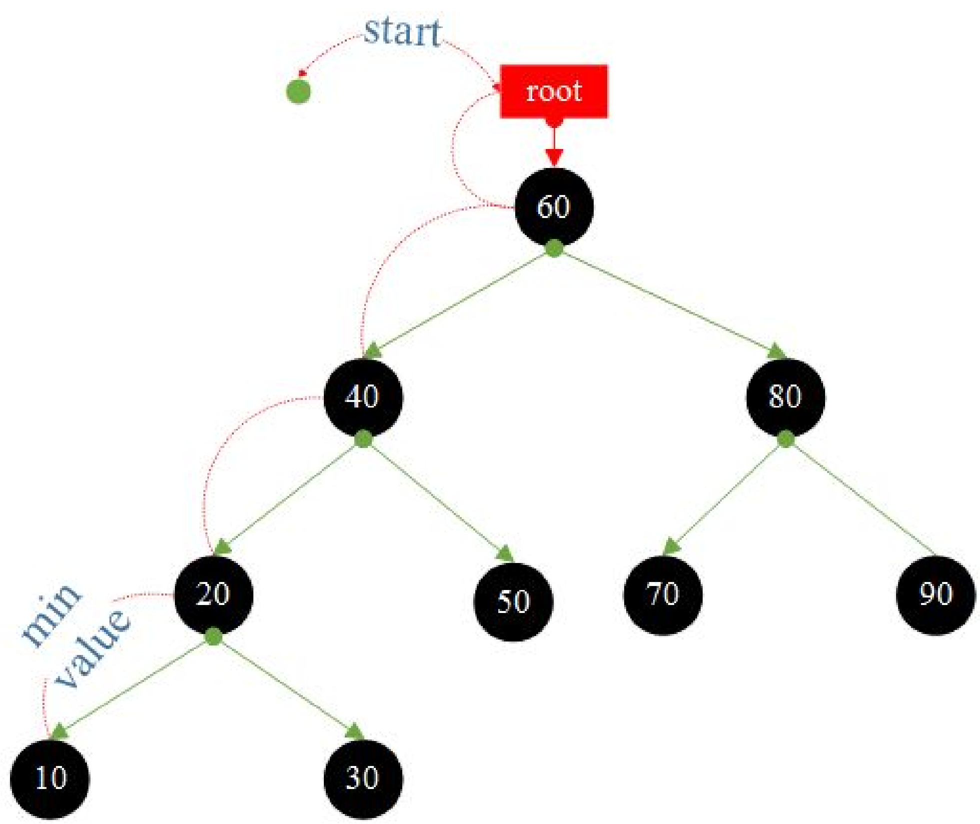
Result:

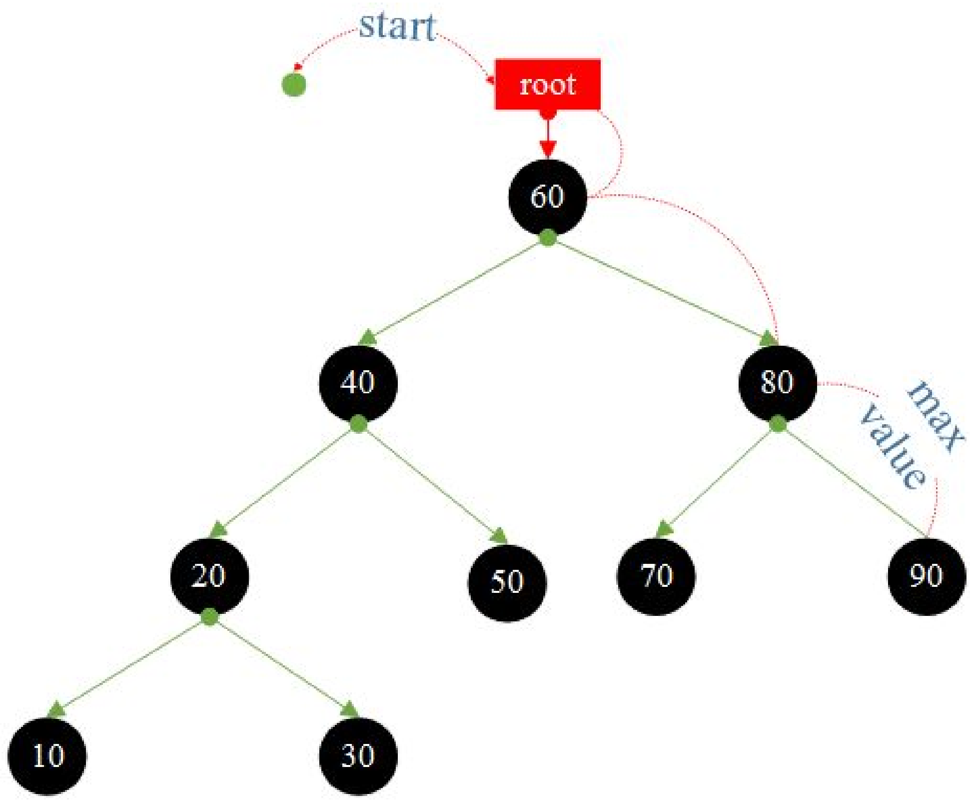
Post-order traversal binary search tree
10 , 30 , 20 , 50 , 40 , 70 , 90 , 80 , 60 ,

5. binary search tree Maximum and minimum

Minimum value: The small value is on the left child node, as long as the recursion traverses the left child until be empty, the current node is the minimum node.

Maximum value: The large value is on the right child node, as long as the recursive traversal is the right child until be empty, the current node is the largest node.





start
root
60
40
80
max value
20
50
70
90
10
30

test_binary_tree.py

```
class Node:
    data = ''
    left = None
    right = None

    def __init__(self, data, left, right):
        self.data = data;
        self.left = left;
        self.right = right;

class BinaryTree:
    __root = None

    @property
    def root(self):
        return self.__root

    #Minimum value
    def search_minimum_value(self, node):
        if node == None or node.data == 0:
            return None
        if node.left == None:
            return node
        # Recursively find the minimum from the left subtree
        return self.search_minimum_value(node.left)

    #Maximum value
    def search_maximum_value(self, node):
        if node == None or node.data == 0:
            return None
        if node.right == None:
            return node
```

```
        # Recursively find the minimum from the right subtree
        return self.search_maximum_value(node.right)






    def insert(self, node, new_data):
        if self.__root == None:
            self.__root = Node(new_data, None, None)
            return
        compareValue = new_data - node.data
        #Recursive left subtree, continue to find the insertion position
        if compareValue < 0:
            if node.left == None:
                node.left = Node(new_data, None, None)
            else:
                self.insert(node.left, new_data)
        elif compareValue > 0: #Recursive right subtree, continue to
find the insertion position
            if node.right == None:
                node.right = Node(new_data, None, None)
            else:
                self.insert(node.right, new_data)

def main():
    binary_tree = BinaryTree()
    binary_tree.insert(binary_tree.root, 60)
    binary_tree.insert(binary_tree.root, 40)
    binary_tree.insert(binary_tree.root, 20)
    binary_tree.insert(binary_tree.root, 10)
    binary_tree.insert(binary_tree.root, 30)
    binary_tree.insert(binary_tree.root, 50)
    binary_tree.insert(binary_tree.root, 80)
    binary_tree.insert(binary_tree.root, 70)
```

```
    binary_tree.insert(binary_tree.root, 90)

    print("\nMinimum Value")
    min_node=binary_tree.search_minimum_value(binary_tree.root)
    print(min_node.data)
    print("\nMaximum Value")

max_node=binary_tree.search_maximum_value(binary_tree.root)
    print(max_node.data)

if __name__ == "__main__":
    main()
```

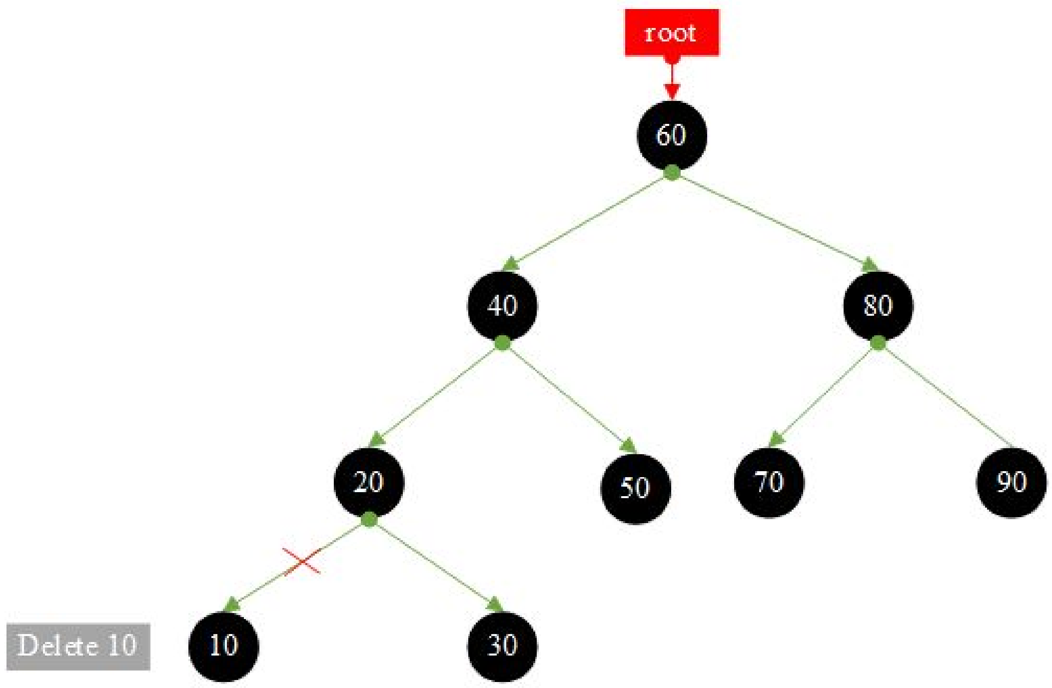
Result:

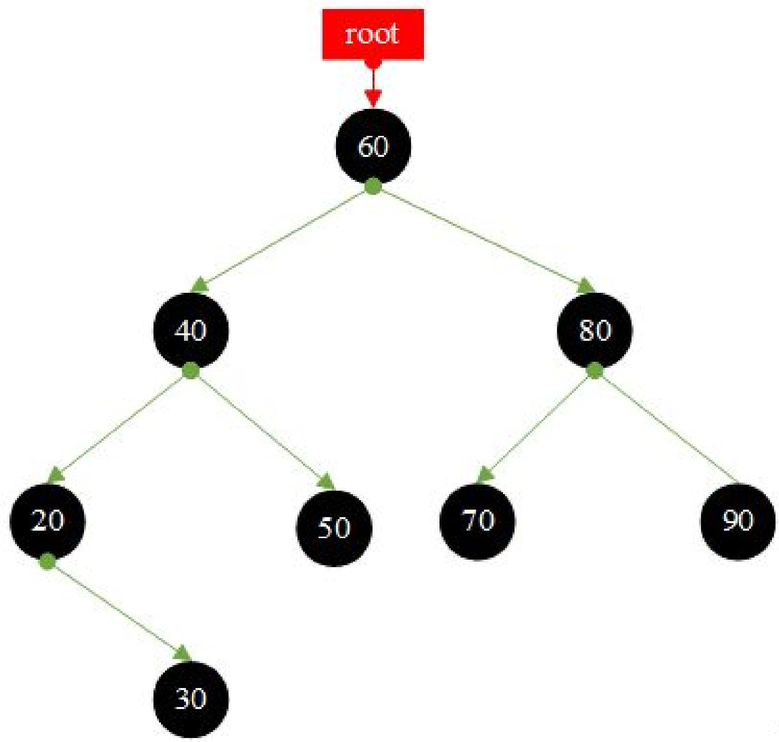
Minimum Value
10
Maximum Value
90

6. binary search tree Delete Node

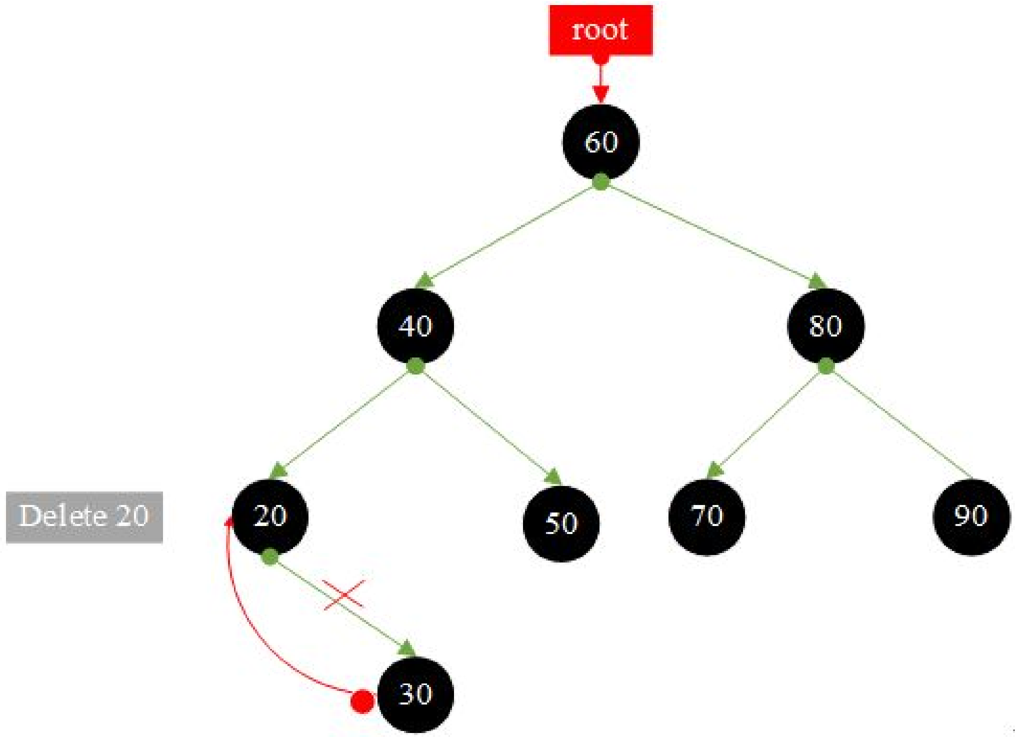
Binary search tree delete node 3 cases
1. If there is no child node, delete it directly
2. If there is only one child node, the child node replaces the current node, and then deletes the current node.
3. If there are two child nodes, replace the current node with the smallest node from the right subtree, because the smallest node on the right is also larger than the value on the left.

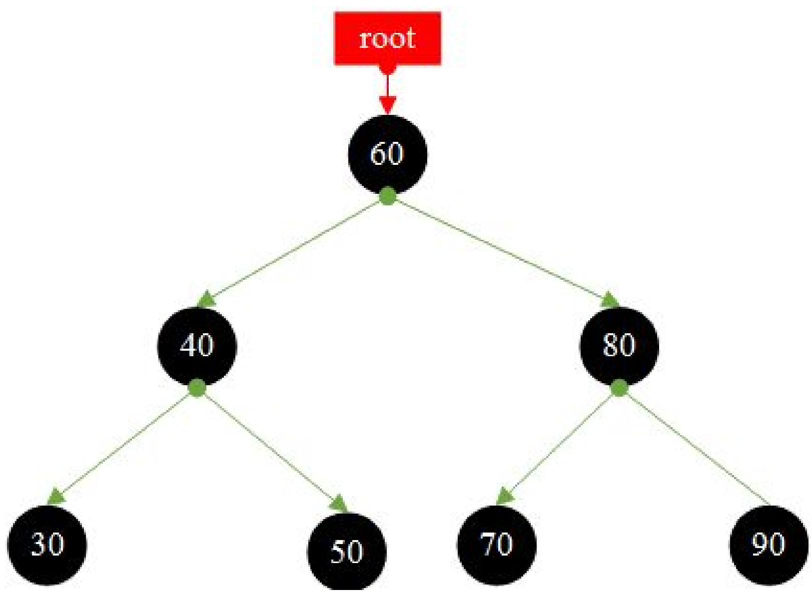
1. If there is no child node, delete it directly: delete node 10



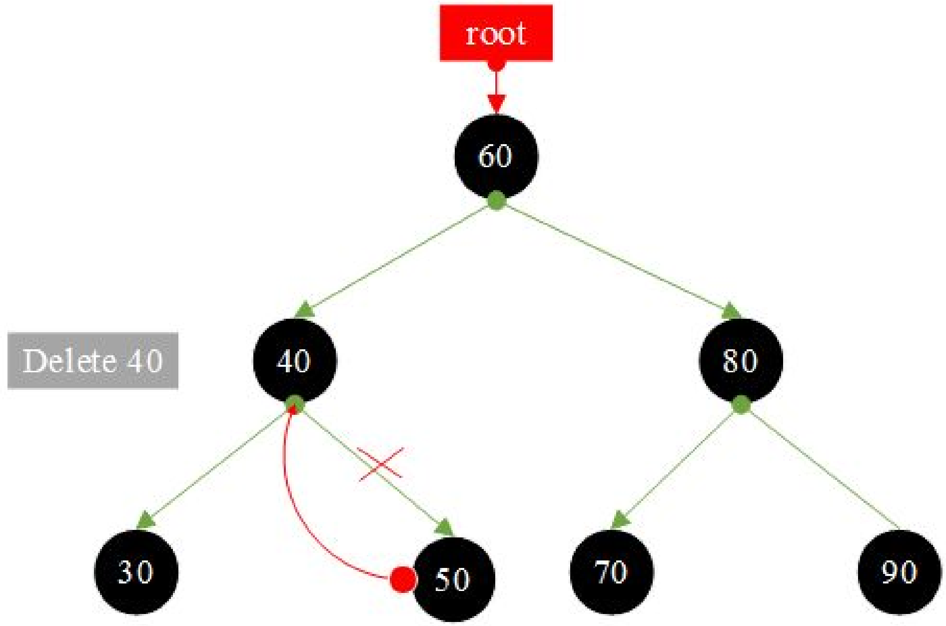

root
60
40
80
20
50
70
90
30

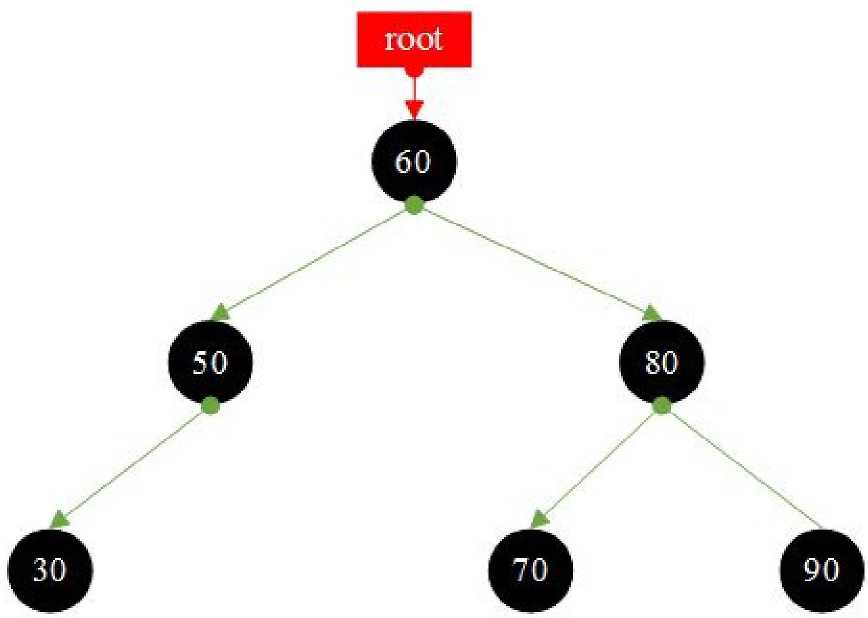
2. If there is only one child node, the child node replaces the current node, and then deletes the current node. **Delete node 20**





3. If there are two child nodes, replace the current node with the smallest node from the right subtree, Delete node 40





test_binary_tree.py

```
class Node:
    data = ''
    left = None
    right = None

    def __init__(self, data, left, right):
        self.data = data;
        self.left = left;
        self.right = right;

class BinaryTree:
    __root = None

    @property
    def root(self):
        return self.__root

    def inOrder(self, root):
        if root == None:
            return
        self.inOrder(root.left) # Traversing the left subtree
        print(root.data, ", ", end="")
        self.inOrder(root.right) # Traversing the right subtree

    def remove(self, node, new_data):
        if node == None:
            return node
        compareValue = new_data - node.data
        if compareValue > 0:
            node.right = self.remove(node.right, new_data)
        elif compareValue < 0:
            node.left = self.remove(node.left, new_data)
        elif node.left != None and node.right != None:
```

```
            # Find the minimum node of the right subtree to replace the
current node
            node.data =self.search_minimum_value(node.right).data
            node.right = self.remove(node.right, node.data)
        else:
            if node.left != None:
                node = node.left
            else:
                node = node.right
        return node

    def search_minimum_value(self, node):
        if node == None or node.data == 0:
            return None
        if node.left == None:
            return node
        # Recursively find the minimum from the left subtree
        return self.search_minimum_value(node.left)

    def insert(self, node, new_data):
        if self.__root == None:
            self.__root = Node(new_data, None, None)
            return
        compareValue = new_data - node.data

        #Recursive left subtree, continue to find the insertion position
        if compareValue < 0:
            if node.left == None:
                node.left = Node(new_data, None, None)
            else:
                self.insert(node.left, new_data)
        elif compareValue > 0: #Recursive right subtree find the
insertion position
            if node.right == None:
                node.right = Node(new_data, None, None)
            else:
                self.insert(node.right, new_data)
```

```python
def main():
    binary_tree = BinaryTree()
    #Constructing a binary search tree
    binary_tree.insert(binary_tree.root, 60)
    binary_tree.insert(binary_tree.root, 40)
    binary_tree.insert(binary_tree.root, 20)
    binary_tree.insert(binary_tree.root, 10)
    binary_tree.insert(binary_tree.root, 30)
    binary_tree.insert(binary_tree.root, 50)
    binary_tree.insert(binary_tree.root, 80)
    binary_tree.insert(binary_tree.root, 70)
    binary_tree.insert(binary_tree.root, 90)

    print("\ndelete node is:  10")
    binary_tree.remove(binary_tree.root, 10)
    print("\nIn-order traversal binary tree")
    binary_tree.inOrder(binary_tree.root)

    print("\n------------------------------------------")

    print("\ndelete node is:  20")
    binary_tree.remove(binary_tree.root, 20)
    print("\nIn-order traversal binary tree")
    binary_tree.inOrder(binary_tree.root)

    print("\n------------------------------------------")

    print("\ndelete node is:  40")
    binary_tree.remove(binary_tree.root, 40)
    print("\nIn-order traversal binary tree")
    binary_tree.inOrder(binary_tree.root)

if __name__ == "__main__":
    main()
```

Result:

delete node is: 10

In-order traversal binary tree
20 , 30 , 40 , 50 , 60 , 70 , 80 , 90 ,
----

delete node is: 20

In-order traversal binary tree
30 , 40 , 50 , 60 , 70 , 80 , 90 ,
----

delete node is: 40

In-order traversal binary tree
30 , 50 , 60 , 70 , 80 , 90 ,

Binary Heap Sorting

Binary Heap Sorting:
The value of the non-terminal node in the binary tree is not greater than the value of its left and right child nodes.

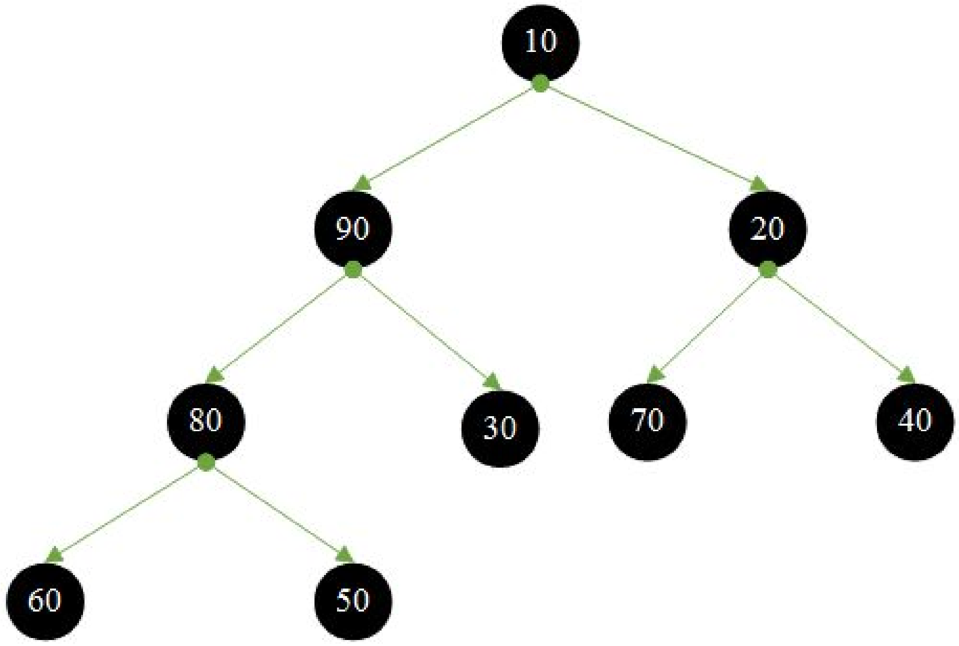
Small top heap : ki <= k2i and ki <= k2i+1
Big top heap :ki >= k2i and ki >= k2i+1

Parent node subscript = (i-1)/2
Left subnode subscript = 2*i+1
Right subnode subscript = 2*i+2

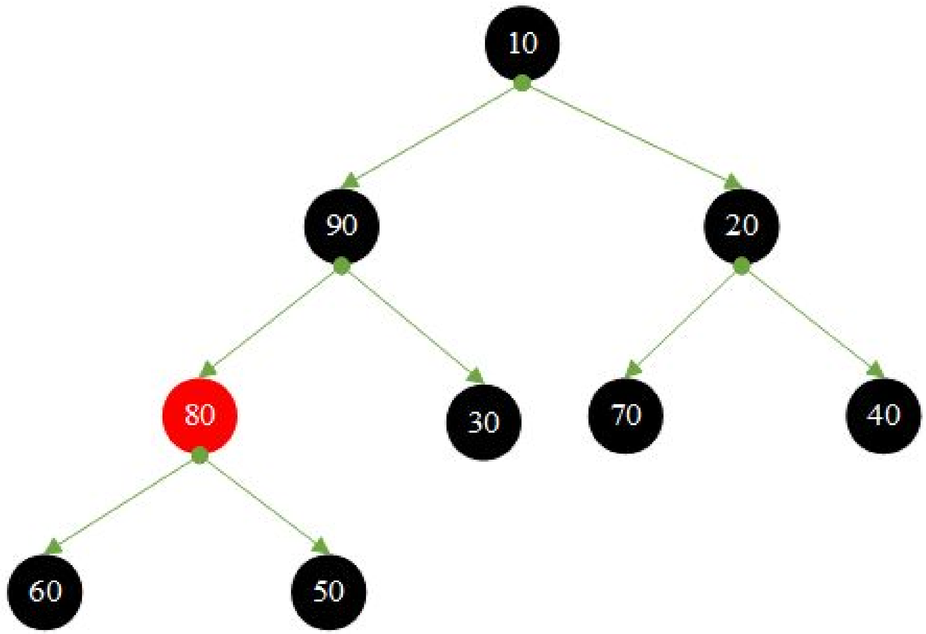
Heap sorting process:
1. Build a heap
2. After outputting the top element of the heap, adjust from top to bottom, compare the top element with the root node of its left and right subtrees, and swap the smallest element to the top of the heap; then adjust continuously until the leaf nodes to get new heap.

1. {10, 90, 20, 80, 30, 70, 40, 60, 50} build heap and then heap sort output.

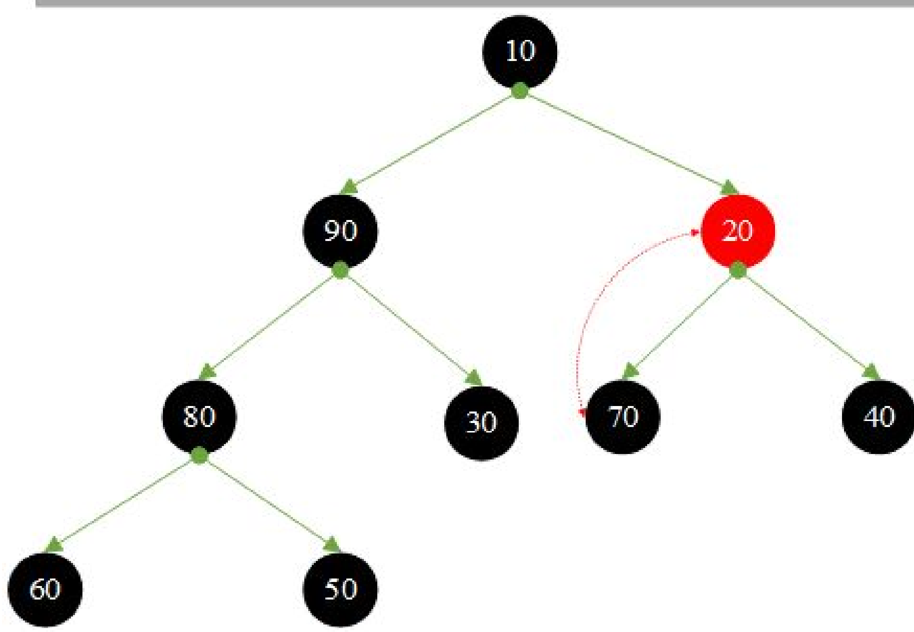
Initialize the heap and build the heap



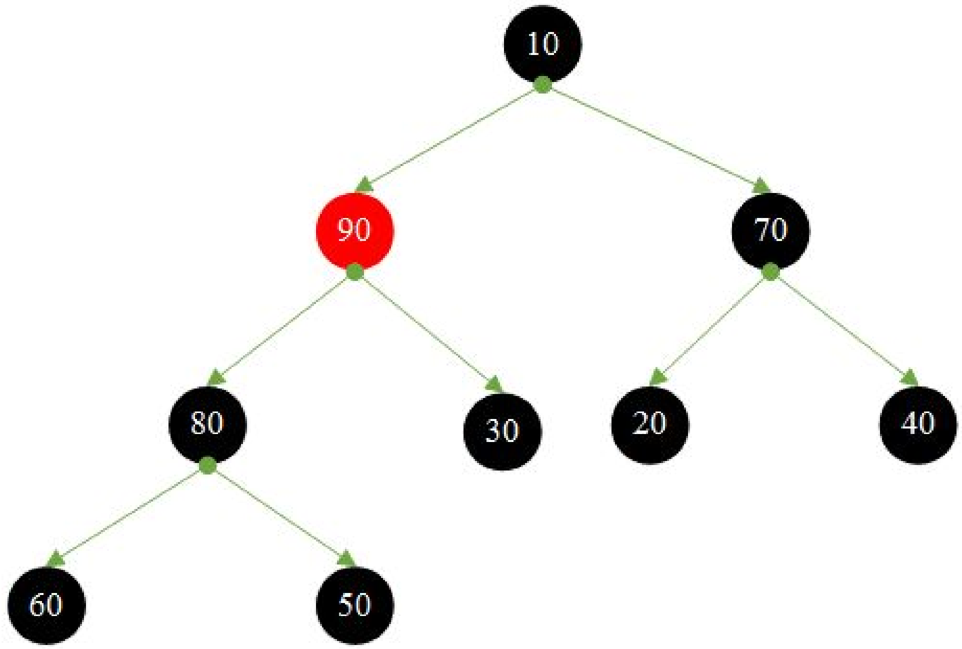
10
90
20
80
30
70
40
60
50



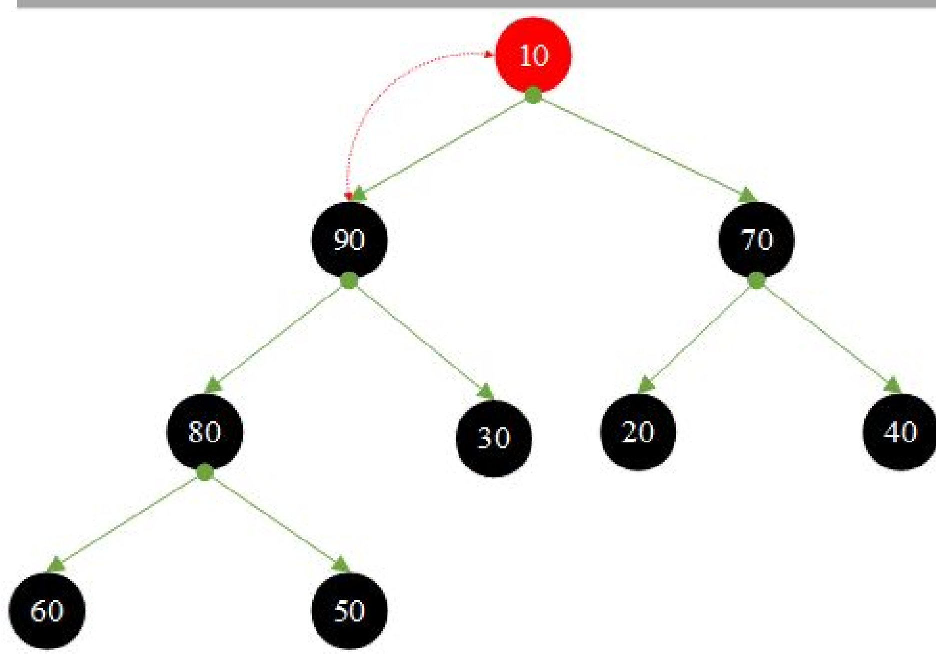
Not Leaf Node = 80 > left = 60， 80 > right = 50 No need to move



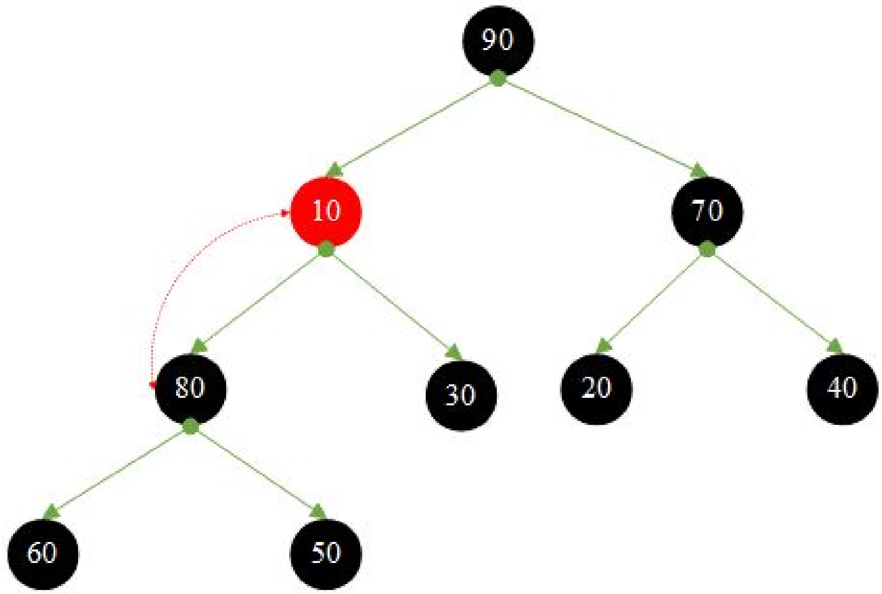
Not Leaf Node = 20 < left = 70， 70 > right = 40 ， 20 swap with 70



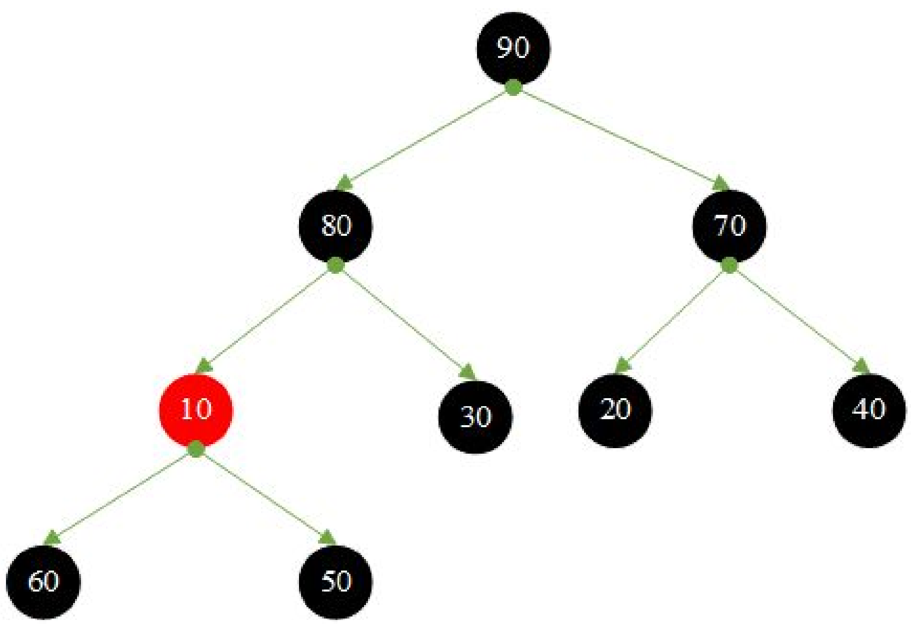
Not Leaf Node = 90 > left = 80， 80 > right = 30 No need to move



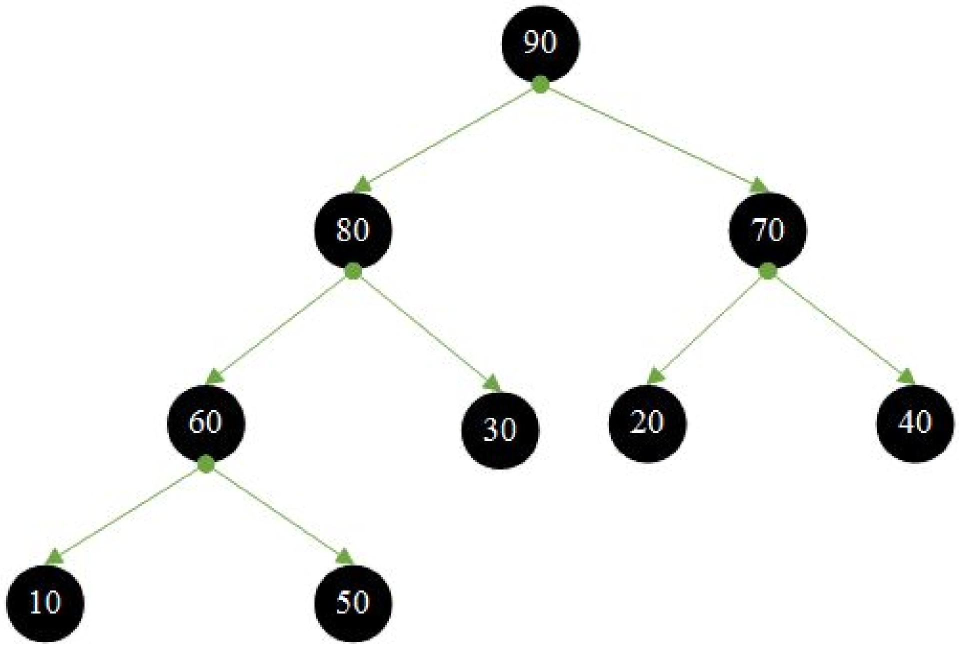
Not Leaf Node = 10 < left = 90， 90 > right = 70 ， 10 swap with 90



Still Not Leaf Node = 10 < left = 80 , 80 > right =30 , 10 swap with 80

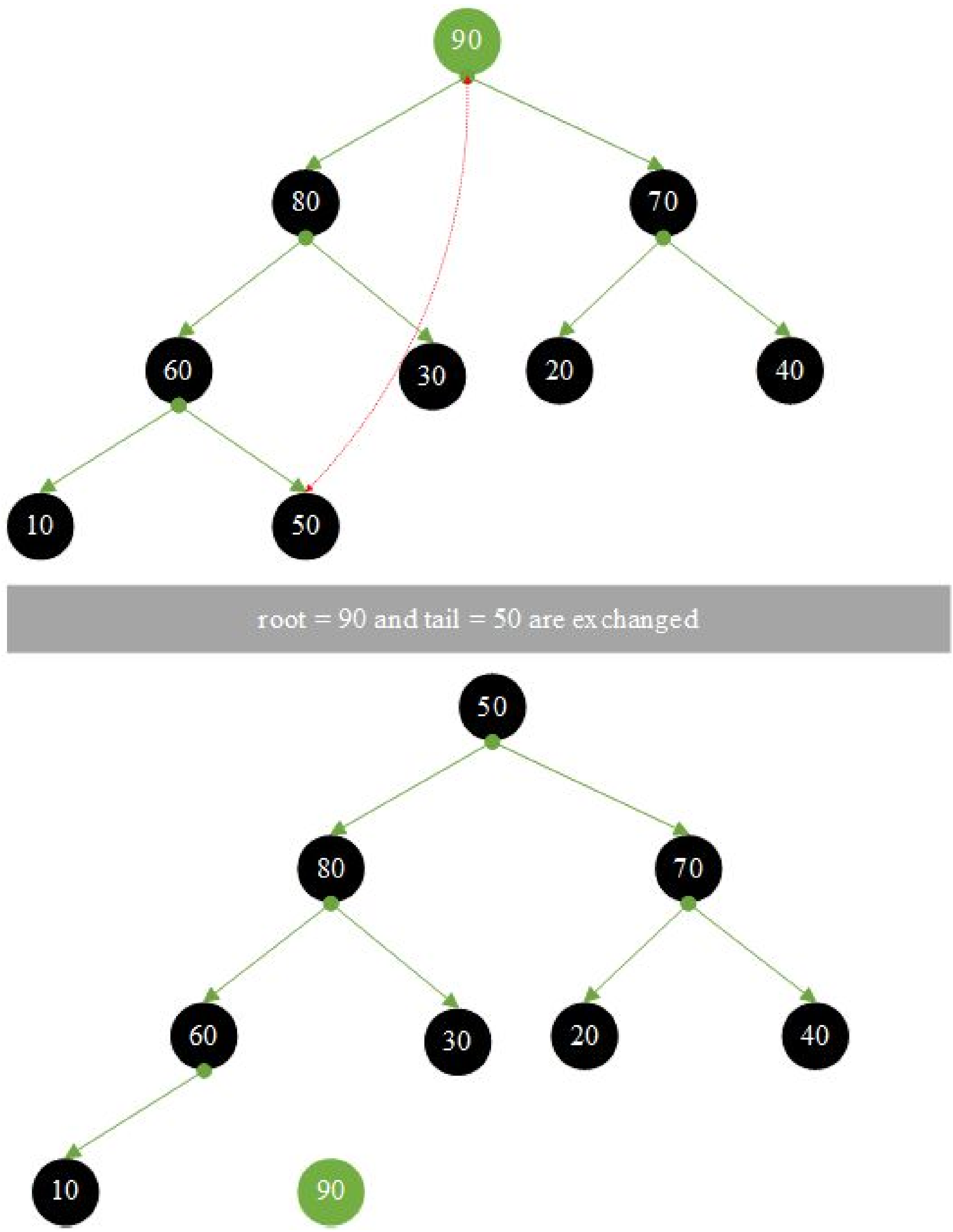


Still Not Leaf Node = 10 < left = 60 , 60 > right =50 , 10 swap with 60

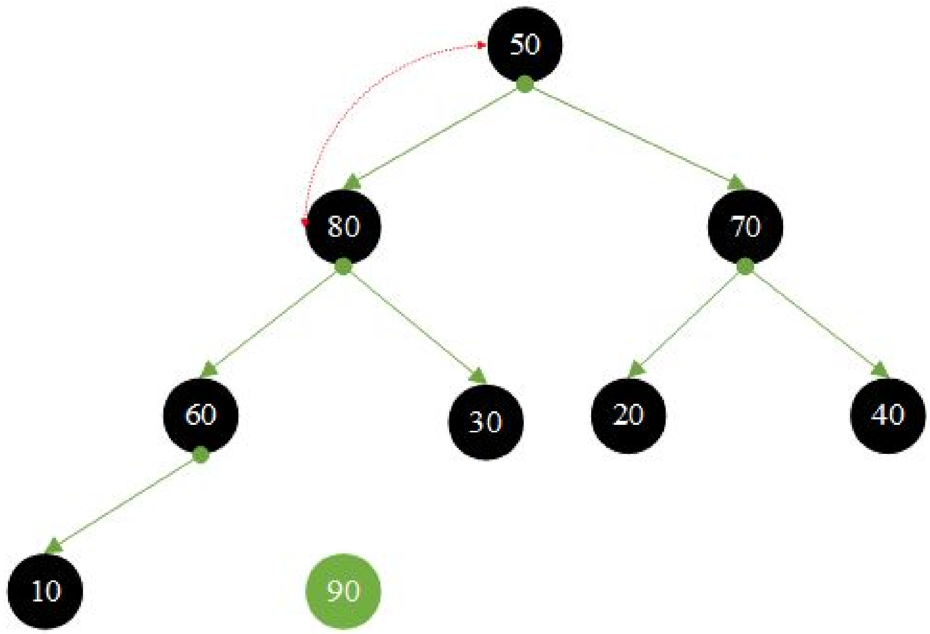


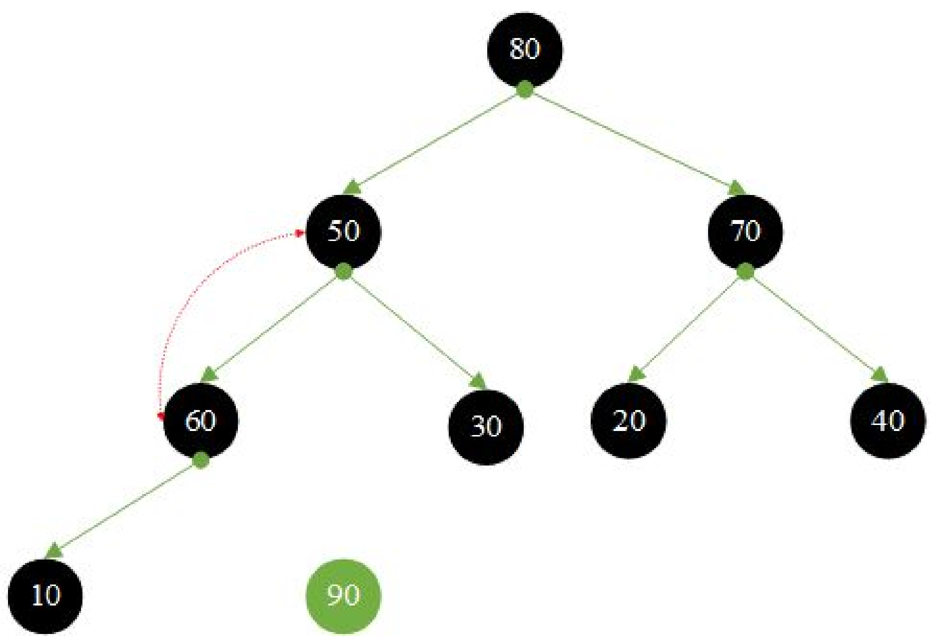
Create the heap finished

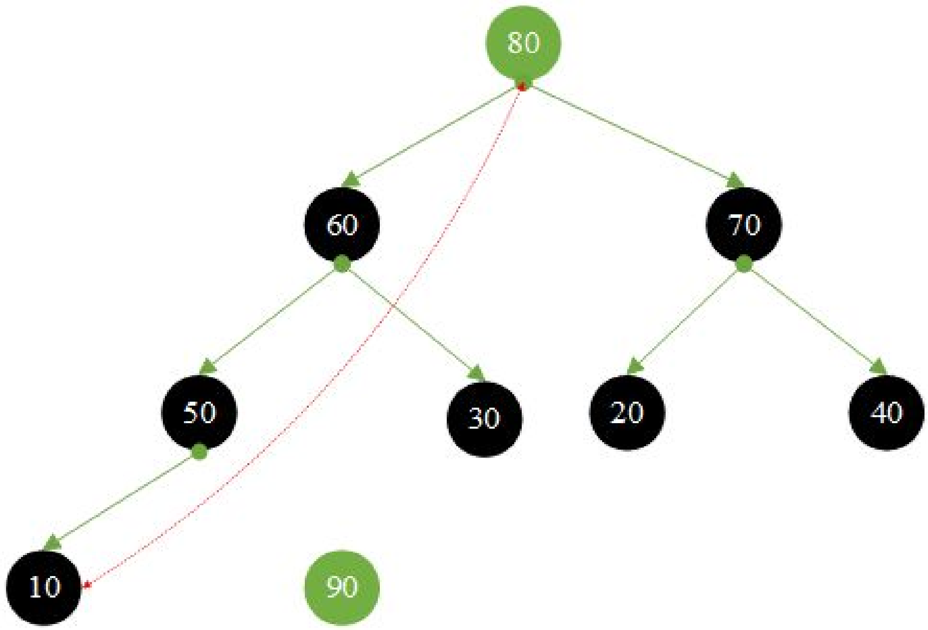
2. Start heap sorting



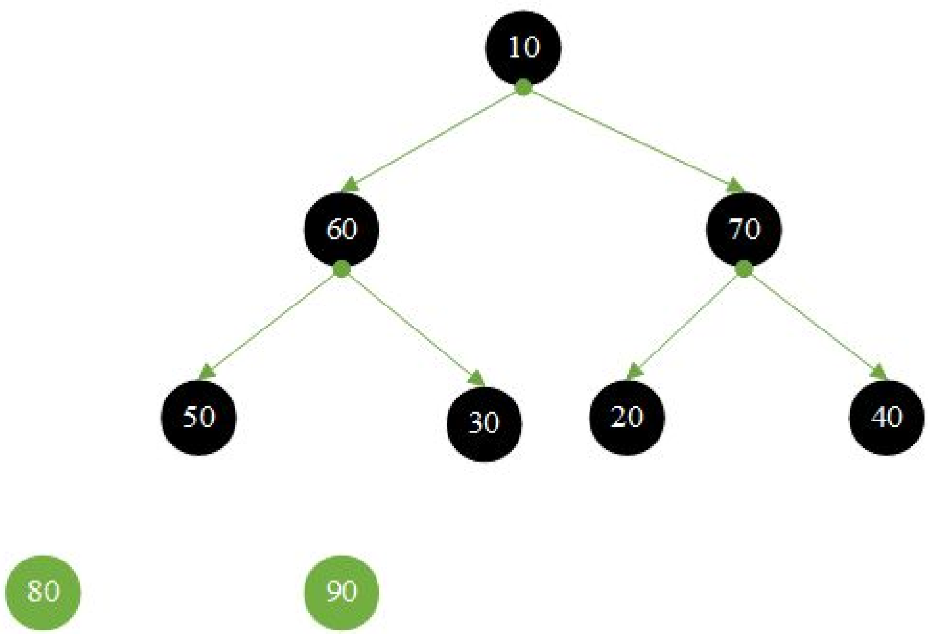
adjust the heap



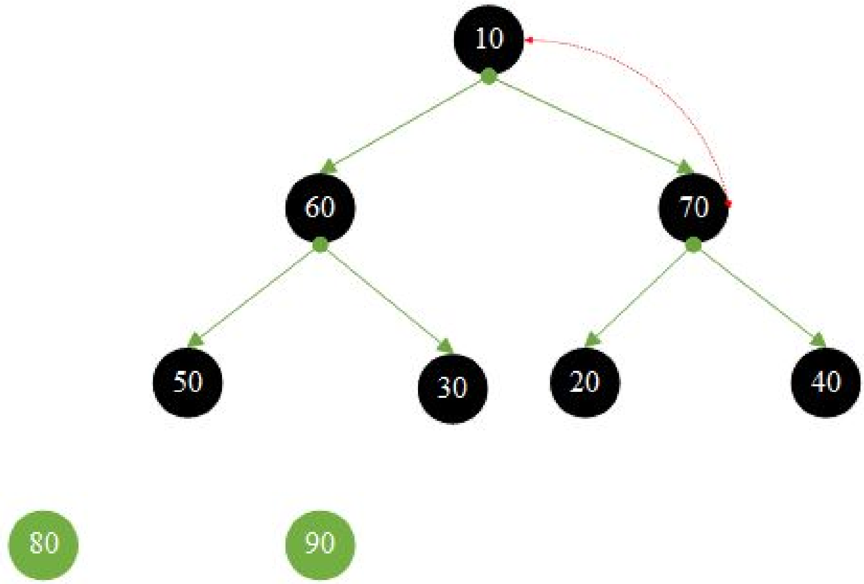


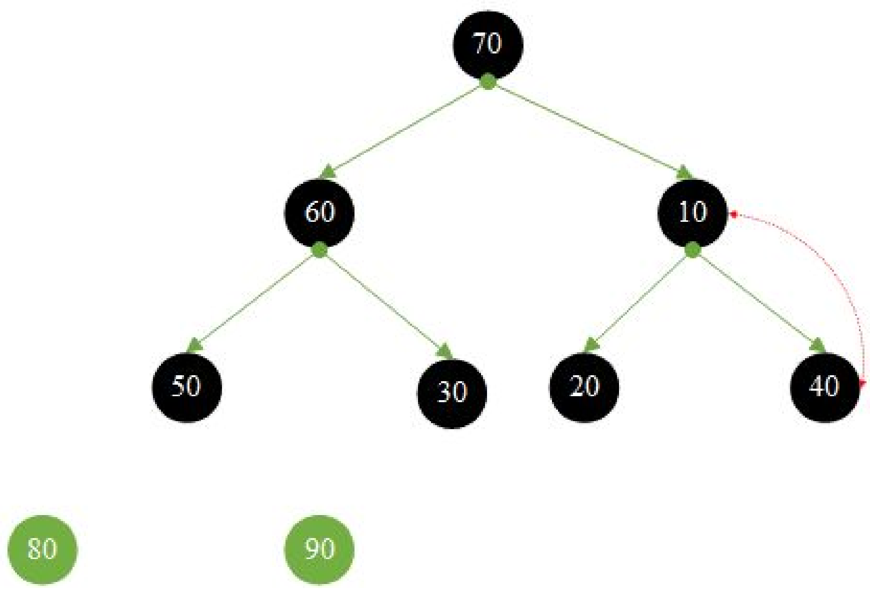


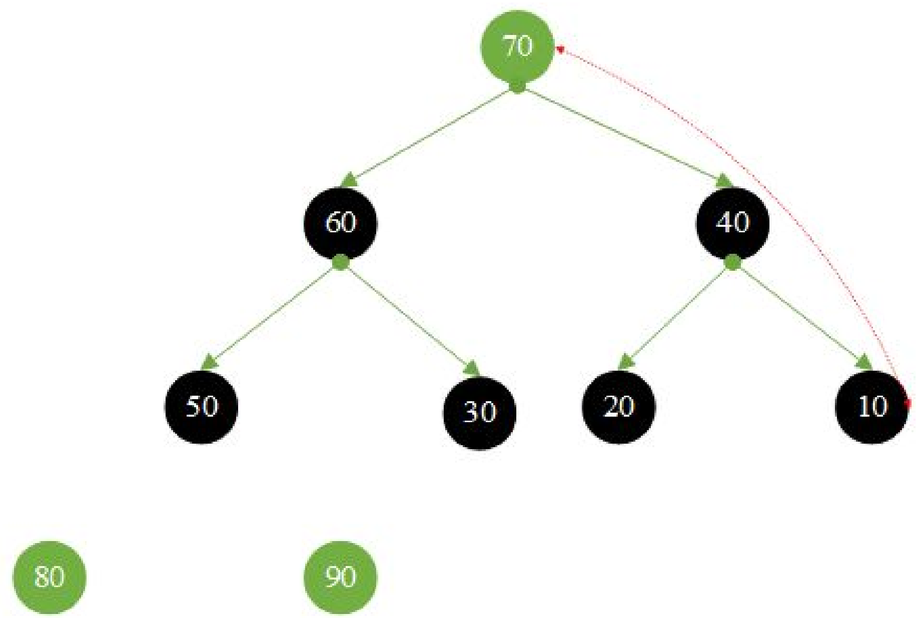
root = 80 and tail = 10 are exchanged



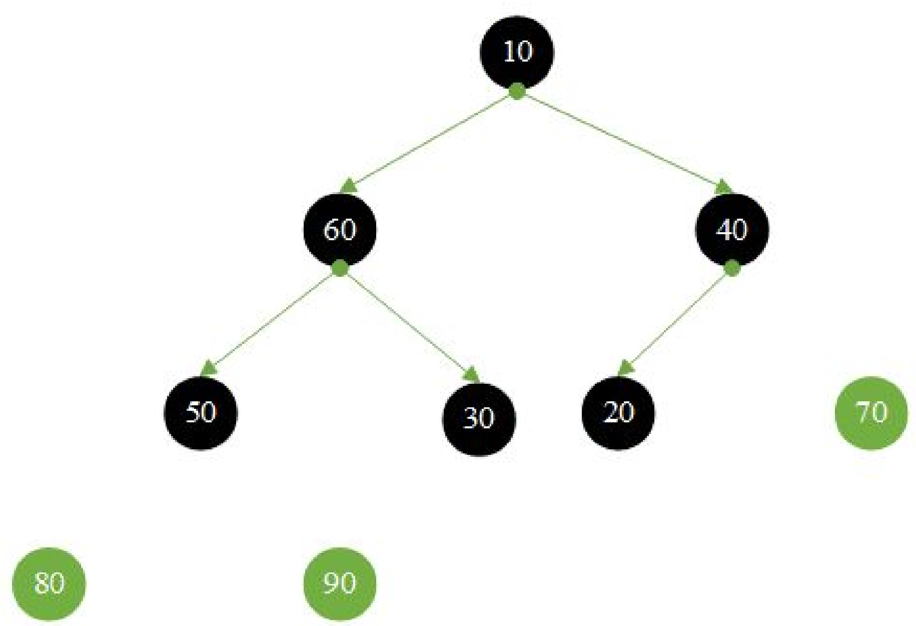
adjust the heap



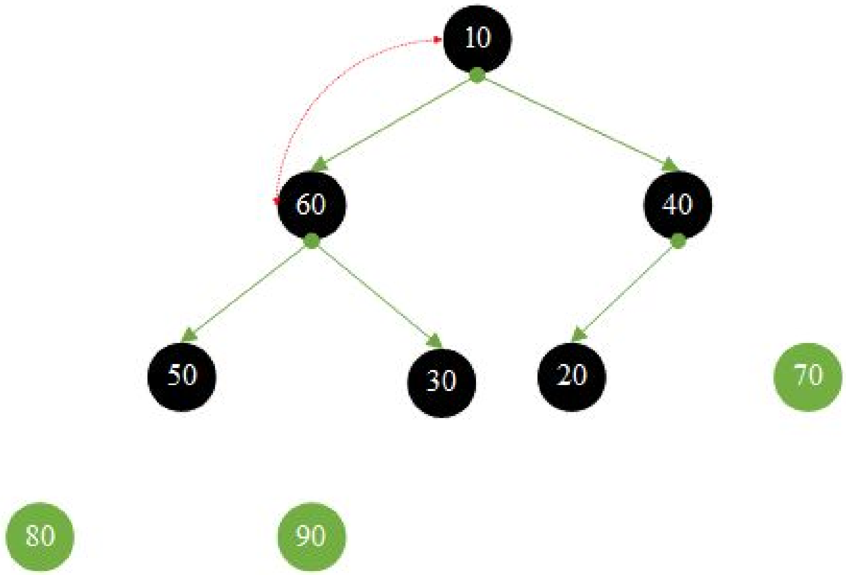


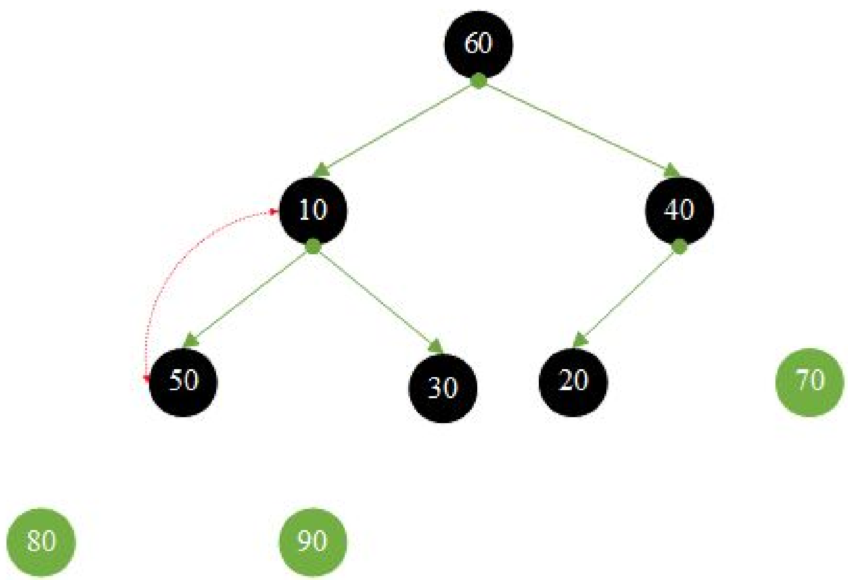


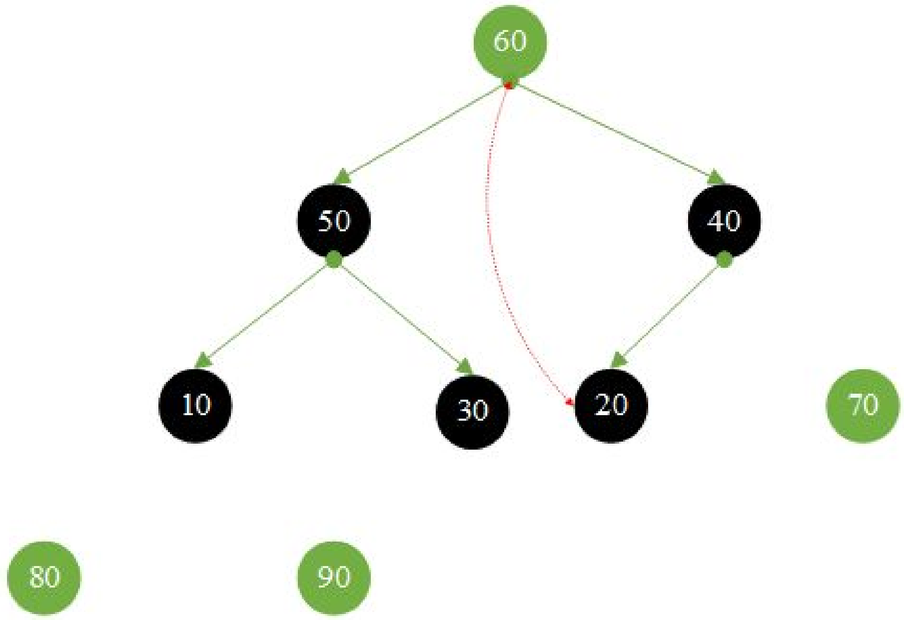
root = 70 and tail = 10 are exchanged



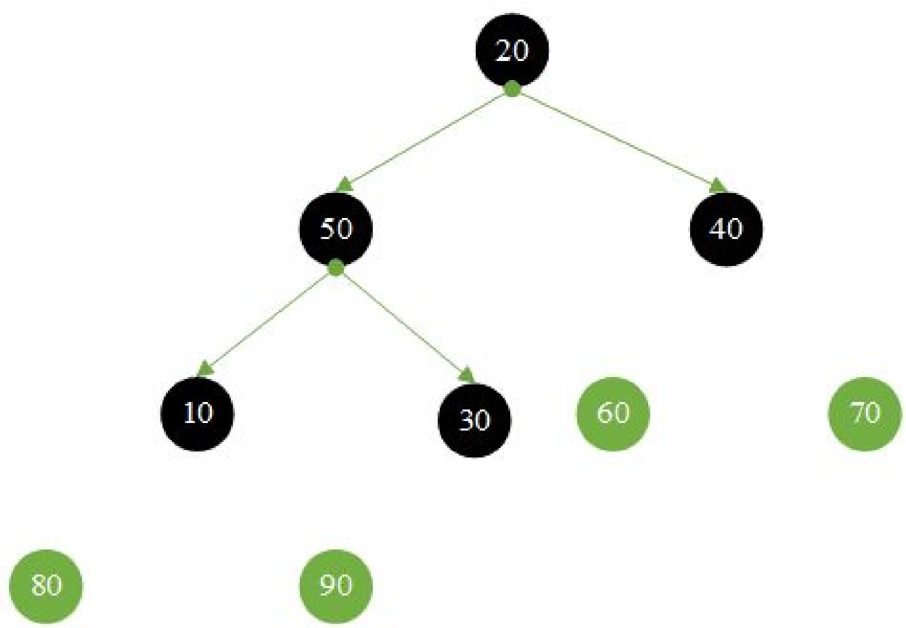
adjust the heap



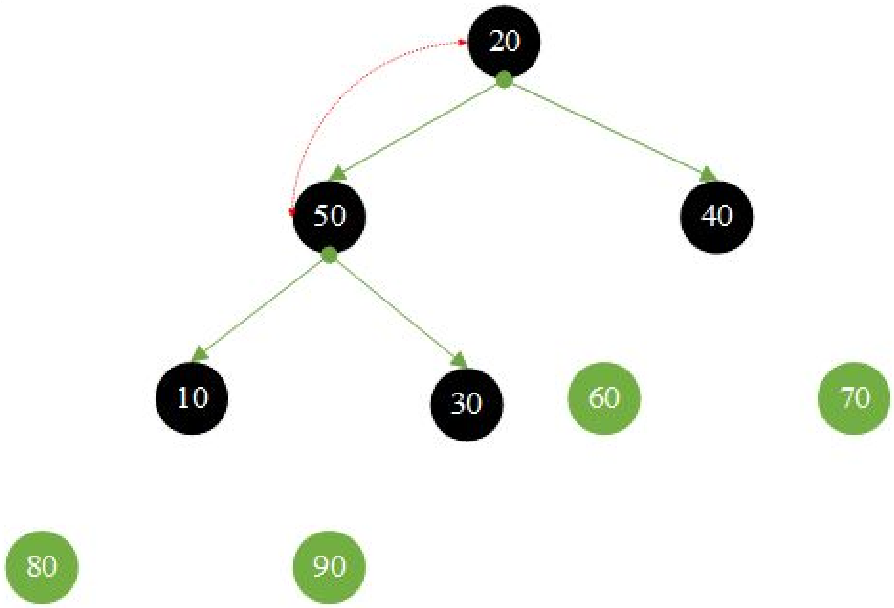


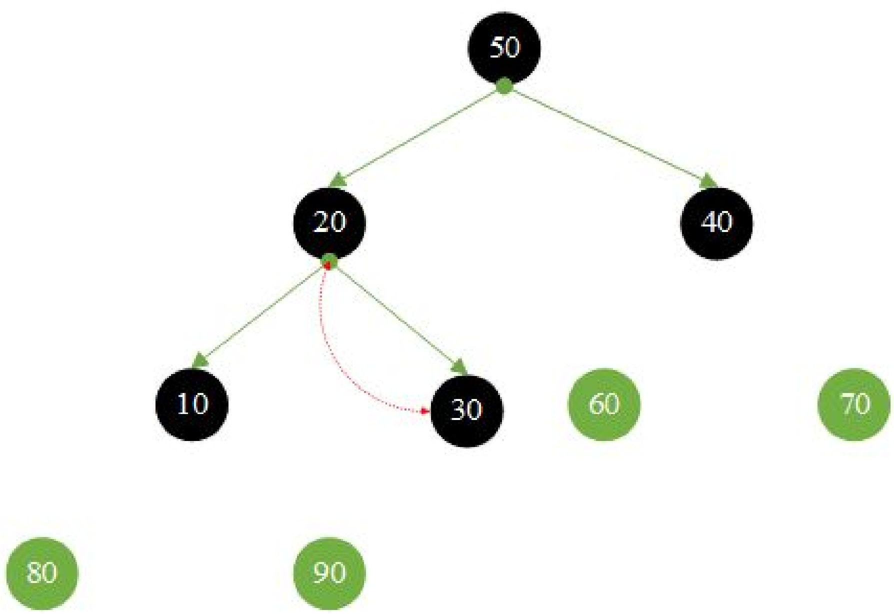


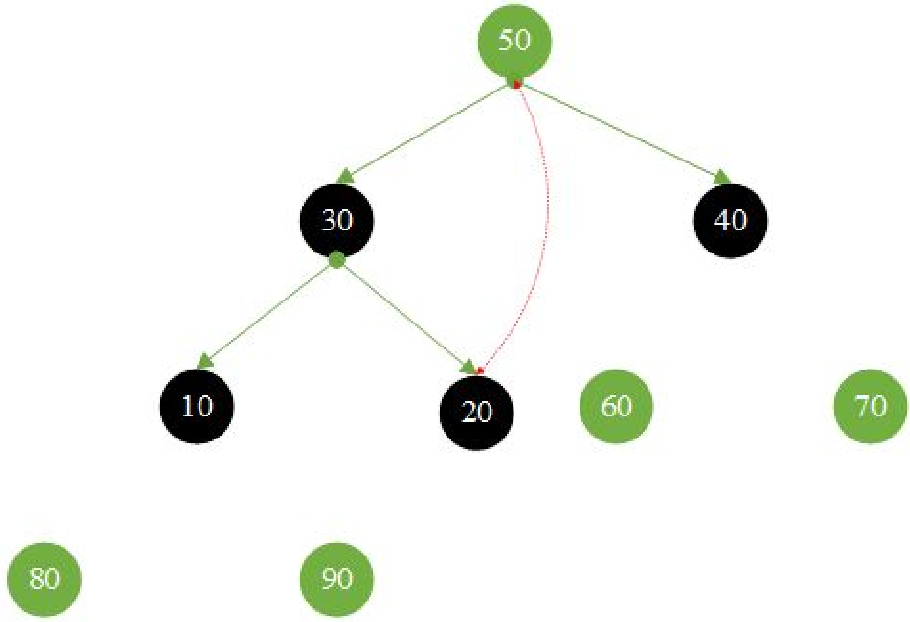
root = 60 and tail = 20 are exchanged



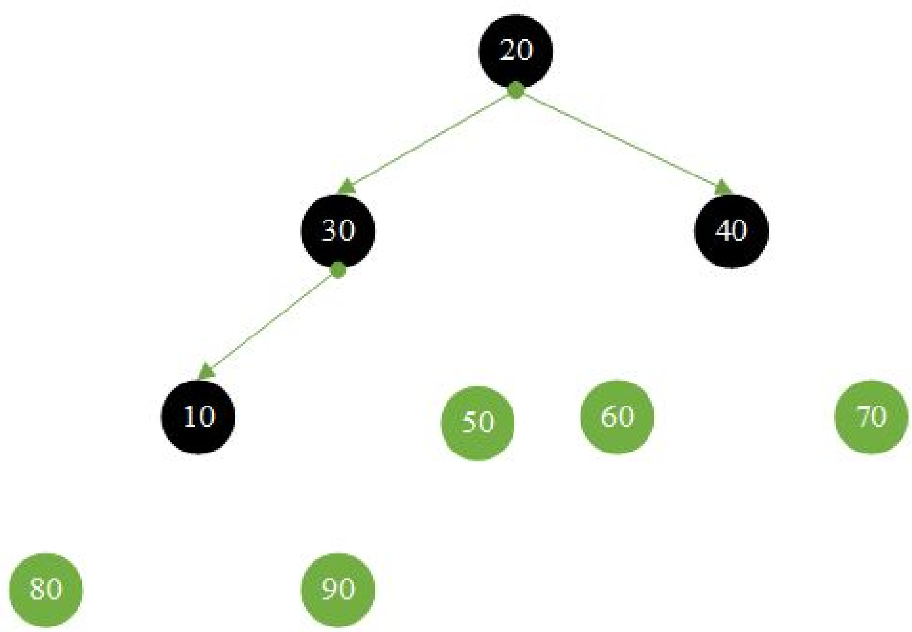
adjust the heap

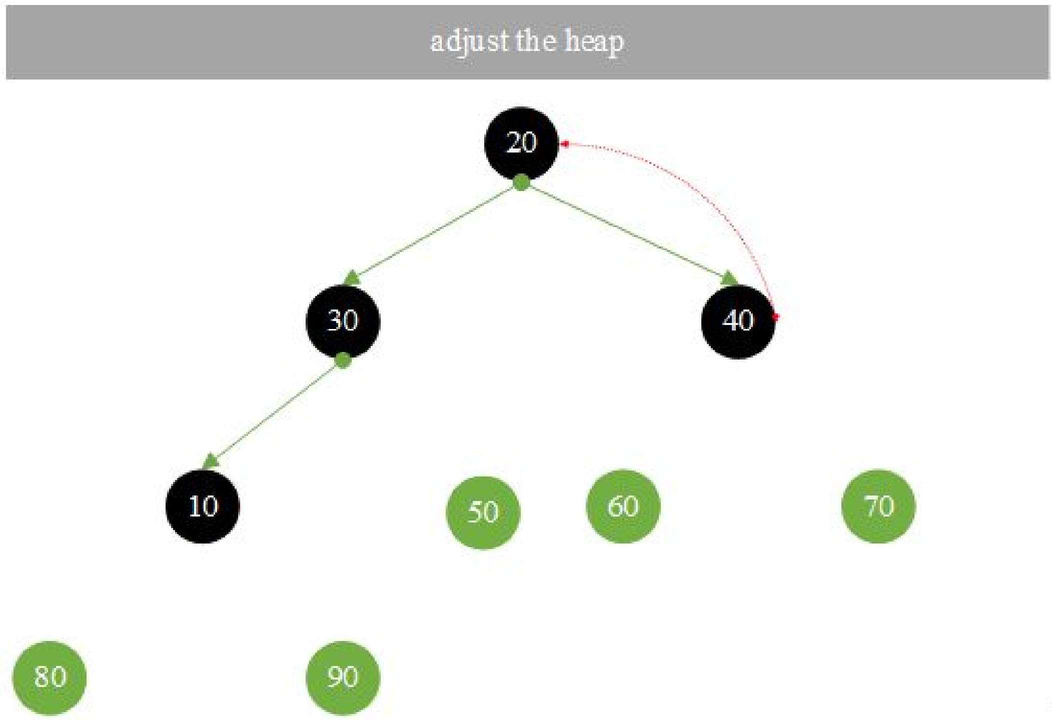


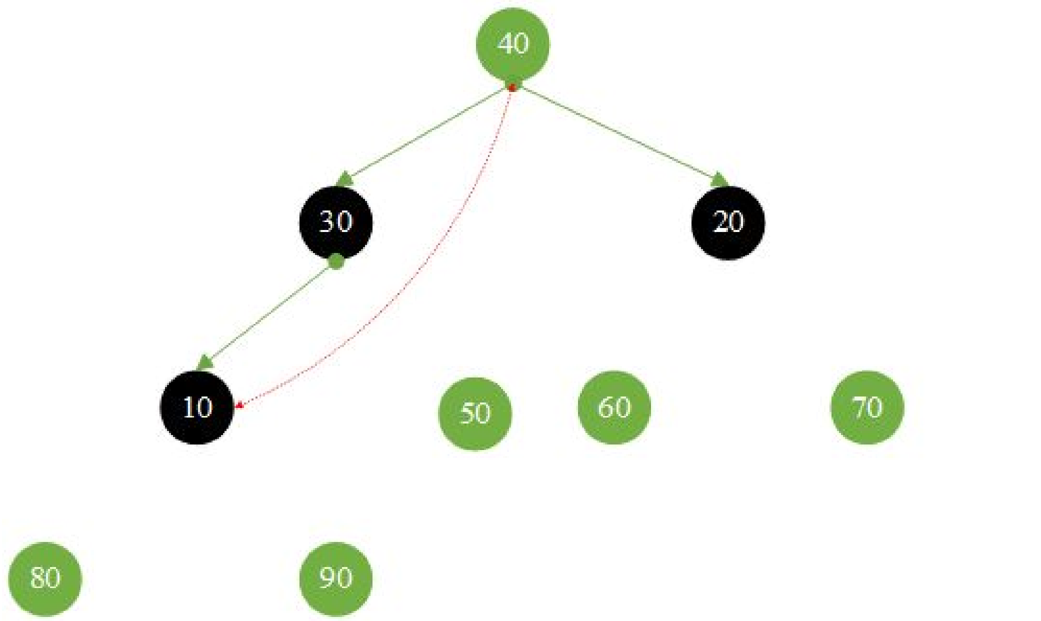




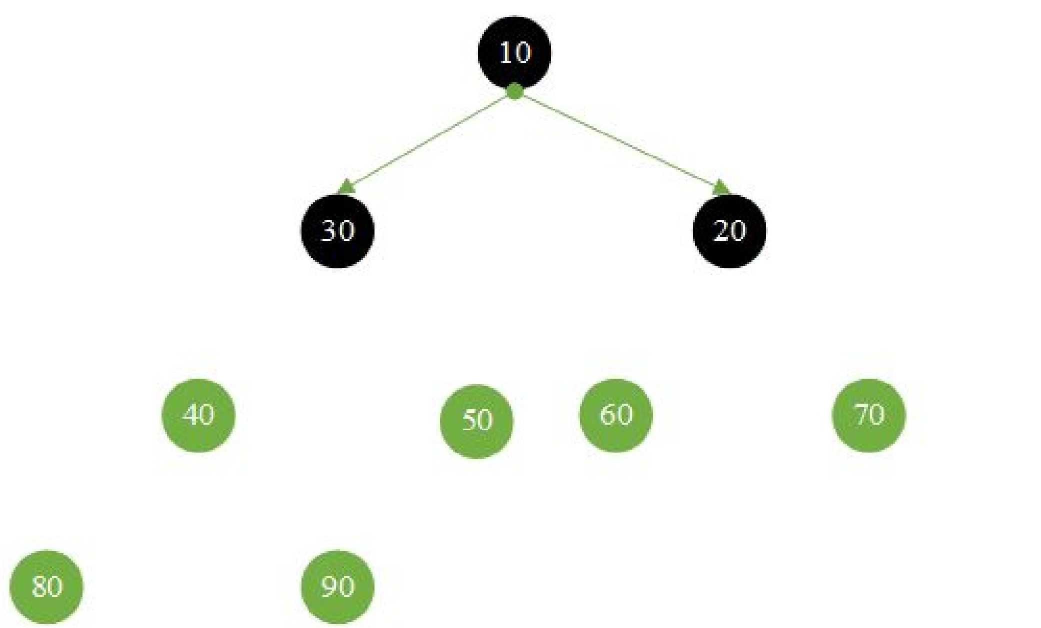
root = 50 and tail = 20 are exchanged

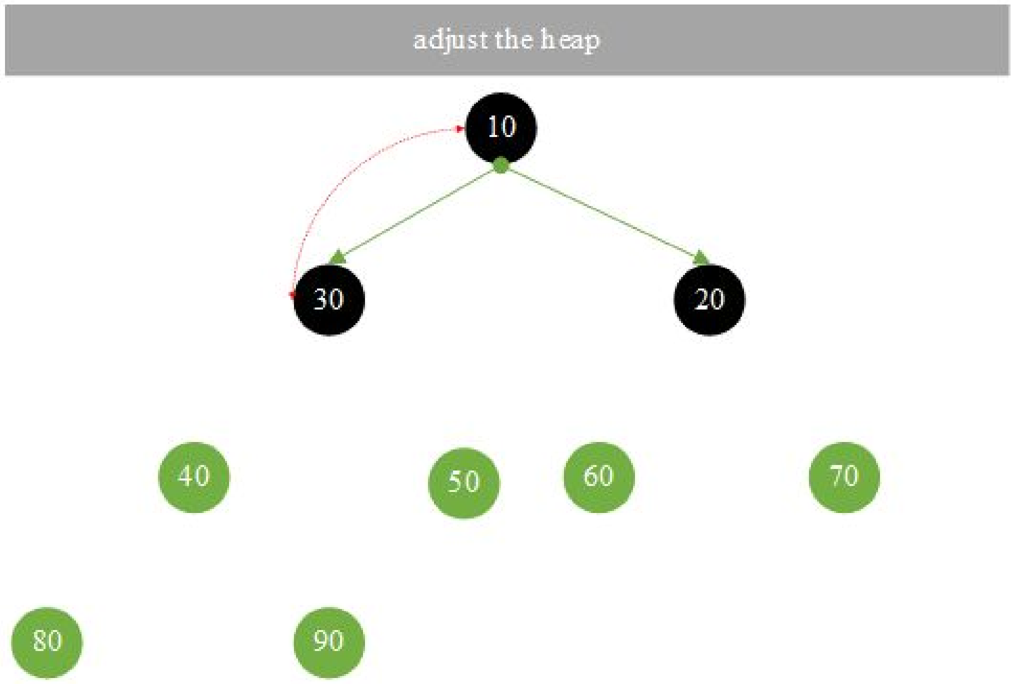


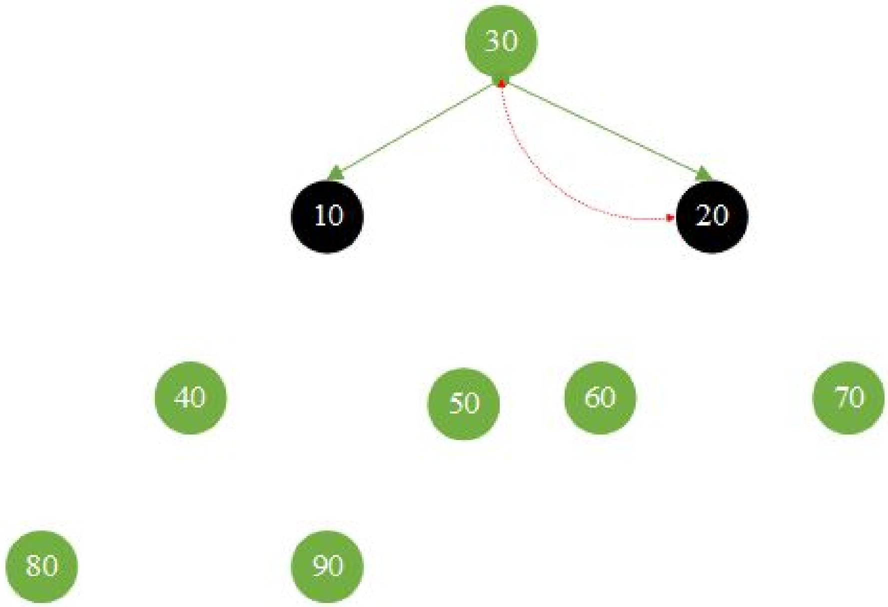

adjust the heap
20
30
40
10
50
60
70
80
90



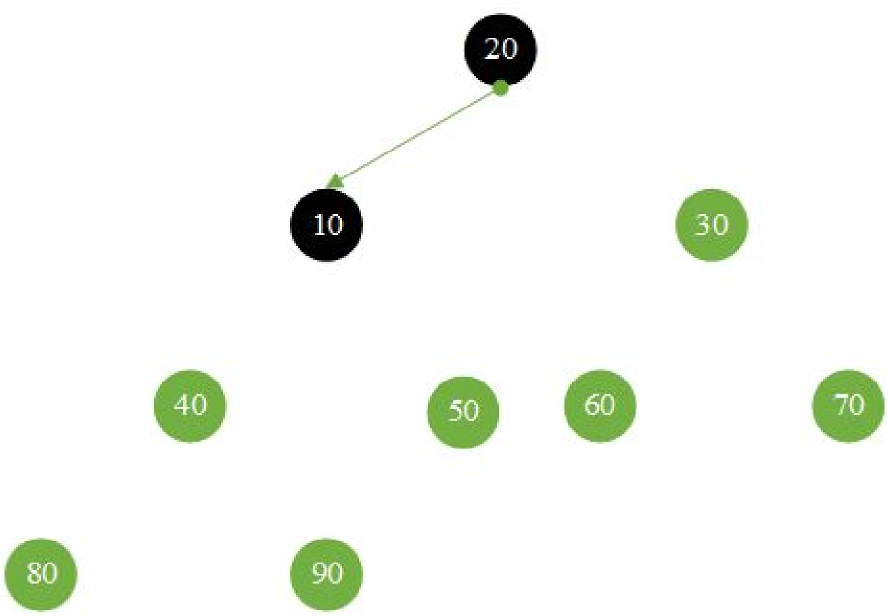
root = 40 and tail = 10 are exchanged



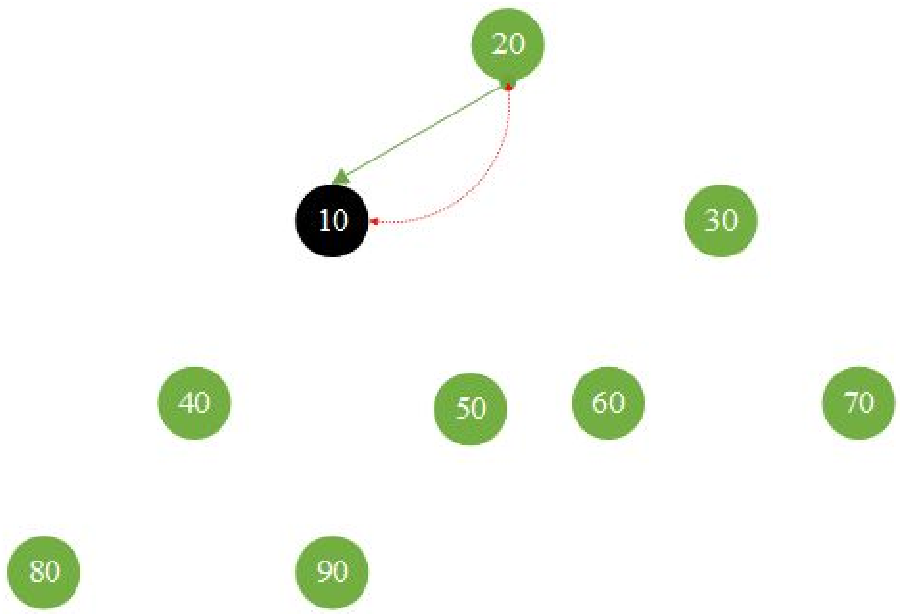

adjust the heap
10
30
20
40
50
60
70
80
90



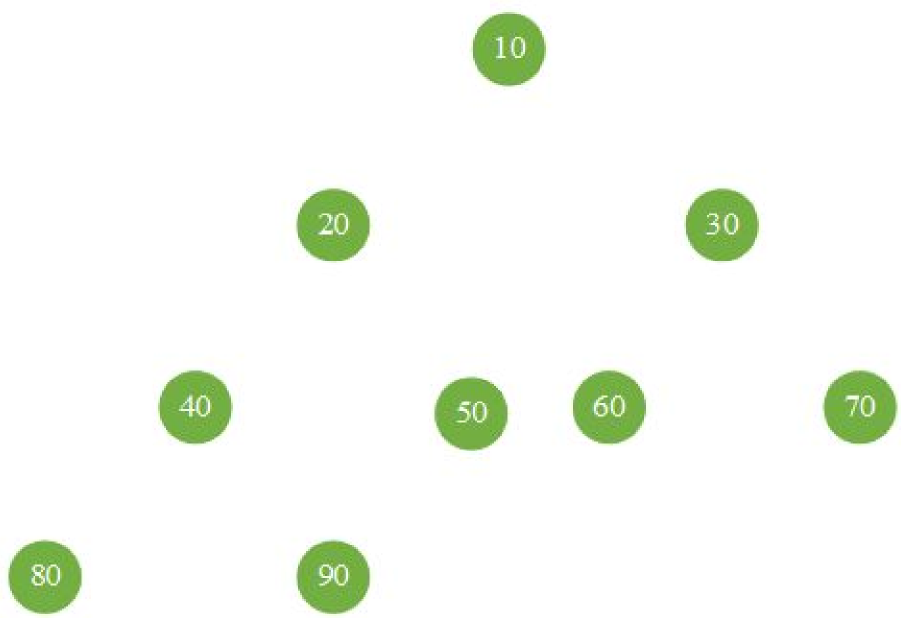
root = 30 and tail = 20 are exchanged



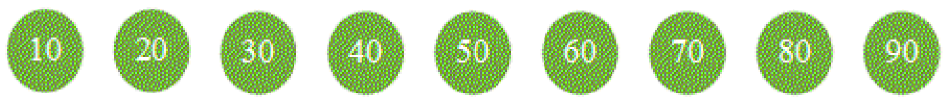
no need adjust the heap



root = 20 and tail = 10 are exchanged



Heap sort result



test_heap_sort.py

```
from array import array

class HeapSort:
    __array = []

    #Initialize the heap
    def create_heap(self, array):
        self.__array = array

        # Build a heap, (array.length - 1) / 2 scan half of the nodes with child nodes
        length = len(array)
        for i in range((length - 1) // 2, -1, -1):
            self.adjust_heap(i, length - 1)

    # Adjustment heap
    def adjust_heap(self, current_index, max_length):
        no_leaf_value = self.__array[current_index] # Current non-leaf node

        # 2 * current_index + 1  Current left subtree subscript
        j = 2 * current_index + 1
        while(j <= max_length):
            if (j < max_length and self.__array[j] < self.__array[j + 1]):
                j += 1 # j Large subscript

            if (no_leaf_value >= self.__array[j]):
```

```
            break

        self.__array[current_index] = self.__array[j] #  Move up to the parent node
        current_index = j

        j = current_index * 2 + 1

    self.__array[current_index] = no_leaf_value # To put in the position



  def heap_sort(self):
    length = len(self.__array)
    for i in range(length - 1, 0, -1):
        temp = self.__array[0]
        self.__array[0] = self.__array[i]
        self.__array[i] = temp
        self.adjust_heap(0, i - 1)

def main():
  heap_sort = HeapSort()
  scores = array('i') # Create an int type array
  scores = array( 'i',[10, 90, 20, 80, 30, 70, 40, 60, 50])
  print("Before building a heap : ")
  for  score in scores:
    print(score, ", ", end="")
  print("\n\n")

  print("After building a heap : ")
  heap_sort.create_heap(scores)
  for  score in scores:
    print(score, ", ", end="")
  print("\n\n")

  print("After heap sorting : ")
```

```
    heap_sort.heap_sort();
    for  score in scores:
        print(score, ", ", end="")

if __name__ == "__main__":
    main()
```

Result:

Before building a heap :
10 , 90 , 20 , 80 , 30 , 70 , 40 , 60 , 50 ,

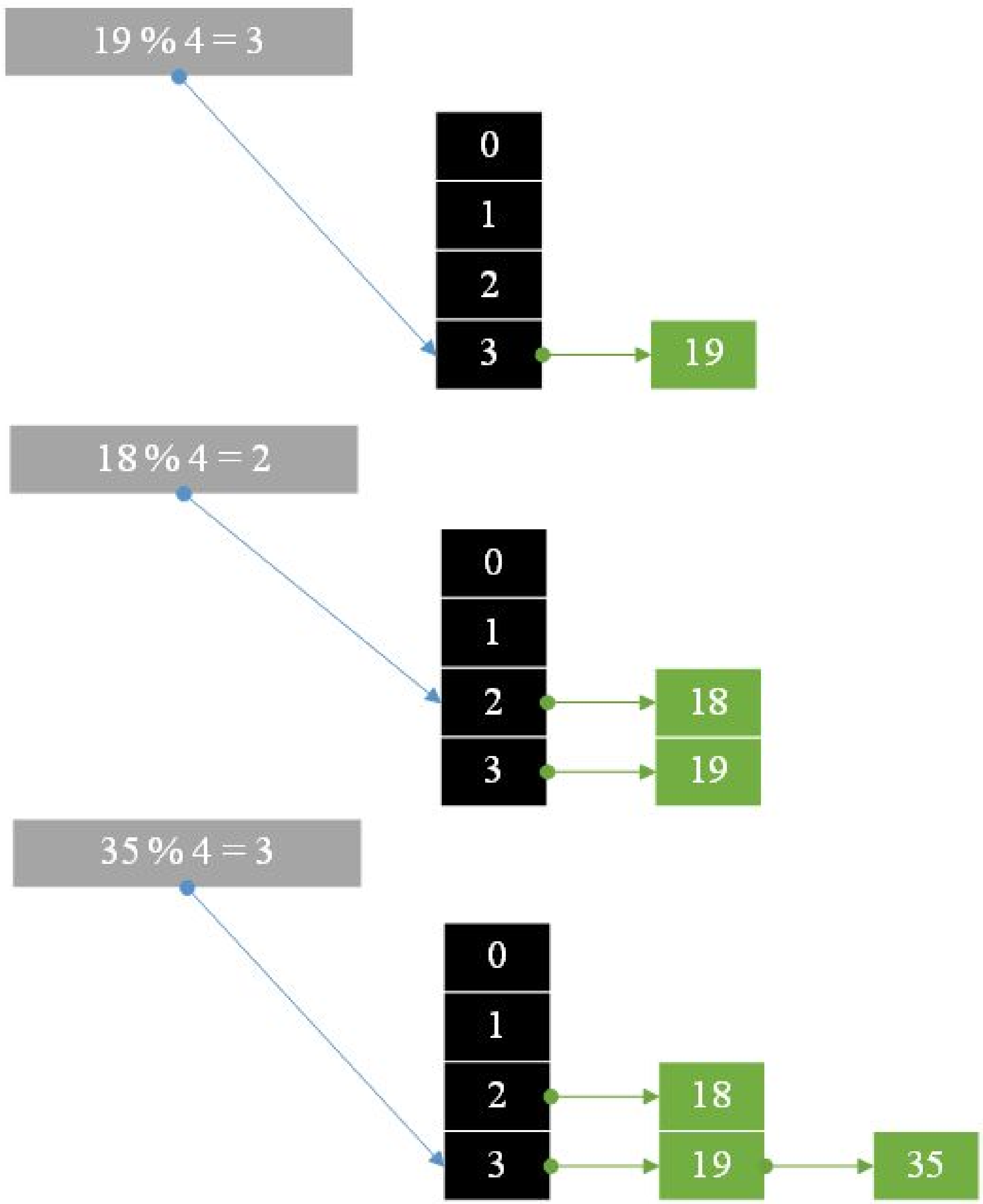
After building a heap :
90 , 80 , 70 , 60 , 30 , 20 , 40 , 10 , 50 ,

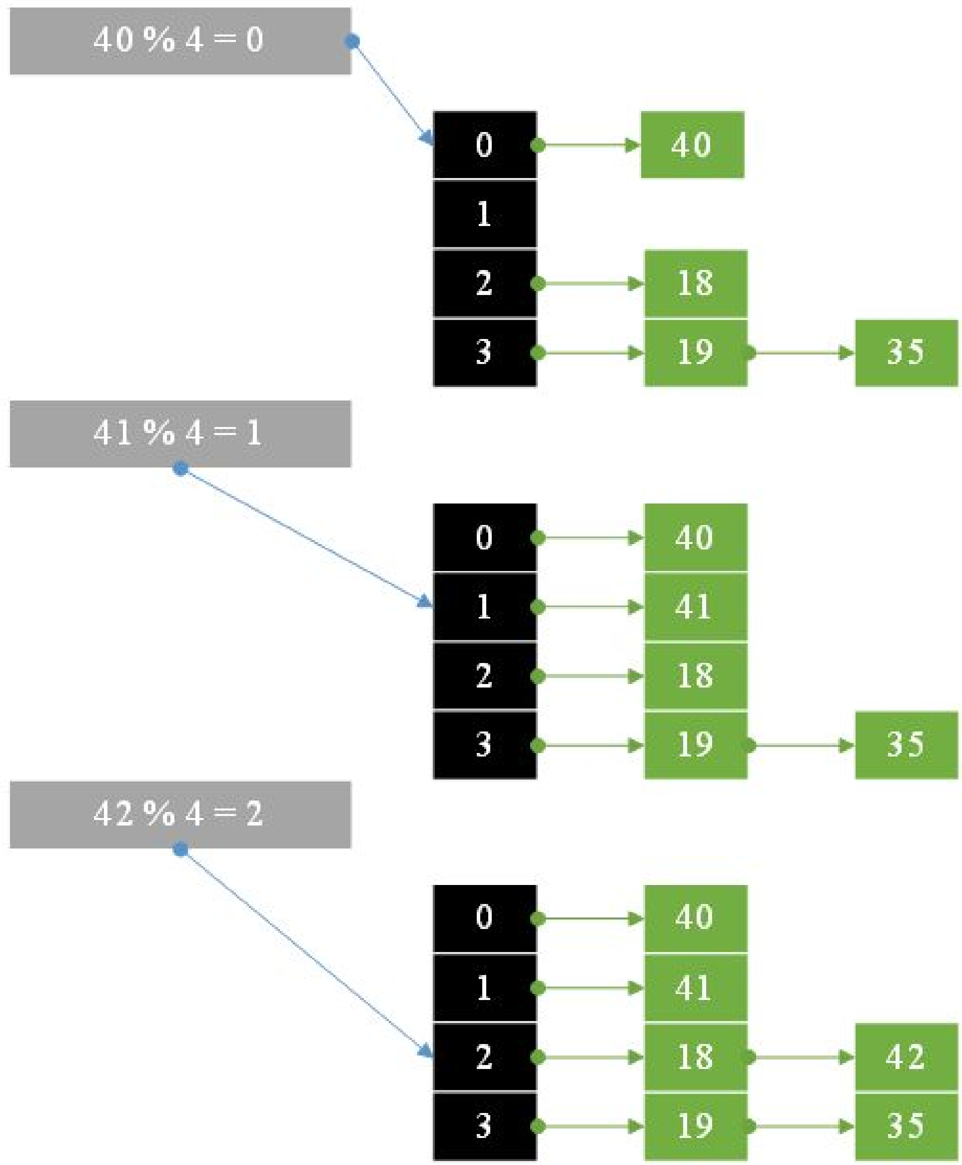
After heap sorting :
10 , 20 , 30 , 40 , 50 , 60 , 70 , 80 , 90 ,

Hash Table

Hash Table:
Access by mapping key => values in the table.

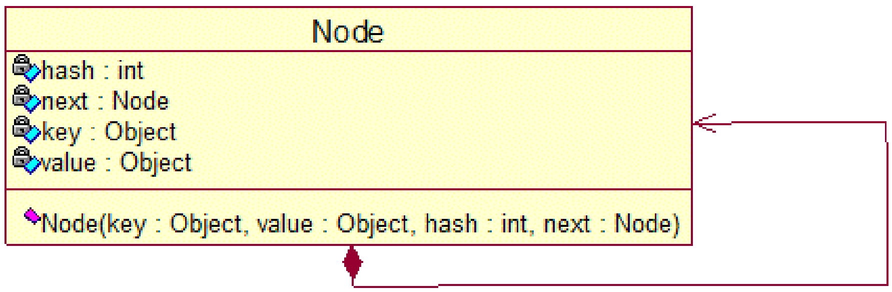
1. Map {19, 18, 35,40,41,42} to the HashTable mapping rule key % 4





40 % 4 = 0
0
1
2
3
40
18
19
35
41 % 4 = 1
0
1
2
3
40
41
18
19
35
42 % 4 = 2
0
1
2
3
40
41
18
42
19
35

2. Implement a Hashtable




```python
class Node:

    key = None
    value = None
    hash = 0
    next = None

    def __init__(self, key, value, hash, next):
        self.key = key
        self.value = value
        self.hash = hash
        self.next = next
```

hashtable.py

```
import math

class Node:
    key = None
    value = None
    hash = 0
    next = None

    def __init__(self, key, value, hash, next):
        self.key = key
        self.value = value
        self.hash = hash
        self.next = next

class Hashtable:

    __table = []
    __size = 0

    # Initialize the Hashtable size
    def __init__(self, capacity):
        self.__table = [[0 for i in range(capacity)]]
        self.__size = 0

    @property
    def size(self):
        return self.__size

    def is_empty(self):
        if self.__size == 0:
            return True
        else:
```

```
        return False

    #Calculate the hash value according to the key hash algorithm
    def hash_code(self, key):
        avg = hash(key) * (math.pow(5, 0.5) - 1) / 2 #hash policy for
middle-square method
        numeric = avg - math.floor(avg)
        return int(math.floor(numeric * len(self.__table)))



    def put(self, key, value):
        if key == None:
            return

        hash = self.hash_code(key)
        new_node = Node(key, value, hash, None)
        node = self.__table[hash]
        while node != None and hash != 0:
            if node.key == key:
                node.value = value
                return
            node = node.next

        new_node.next = self.__table[hash]
        self.__table[hash] = new_node
        self.__size+=1

    def get(self, key):
        if key == None:
            return None

        hash = self.hash_code(key)
        node = self.__table[hash]
        while node != None: #Get value according to key
            if node.key == key:
                return node.value
```

```python
            node = node.next
        return None

def main():
    table = Hashtable(16)
    table.put("david", "Good Boy Keep Going")
    table.put("grace", "Cute Girl Keep Going")

    print("david => ", table.get("david"))
    print("grace => ", table.get("grace"))

if __name__ == "__main__":
    main()
```

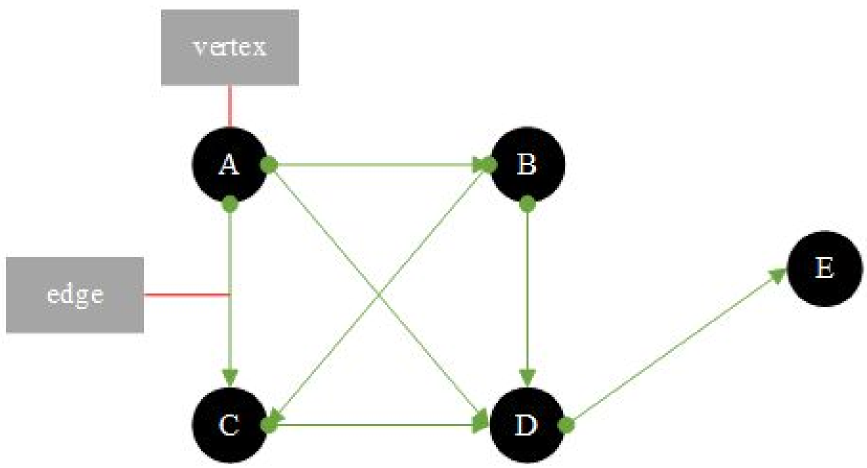
Result:

david => Good Boy Keep Going
grace => Cute Girl Keep Going

Directed Graph and Depth-First Search

Directed Graph:
The data structure is represented by an adjacency matrix (that is, a two-dimensional array) and an adjacency list. Each node is called a vertex, and two adjacent nodes are called edges.

Directed Graph has direction : A -> B and B -> A are different

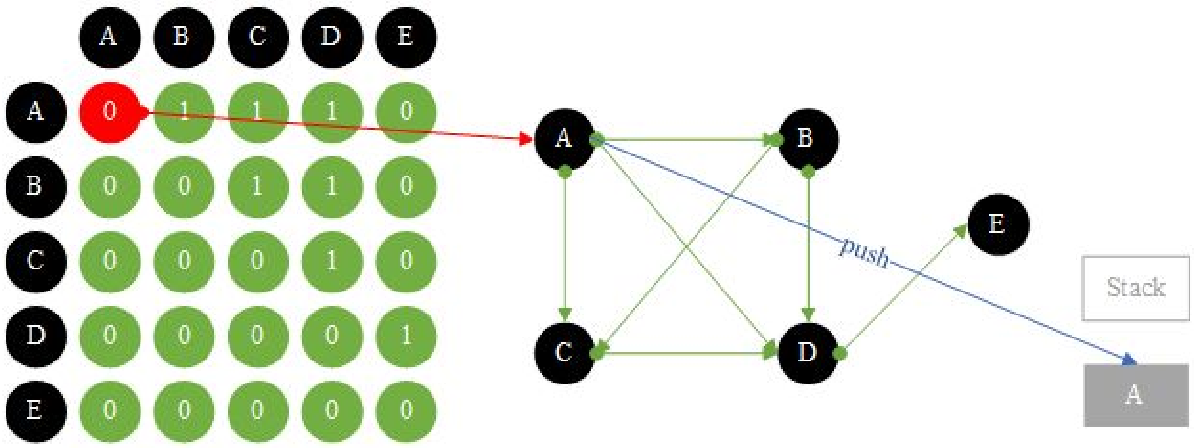


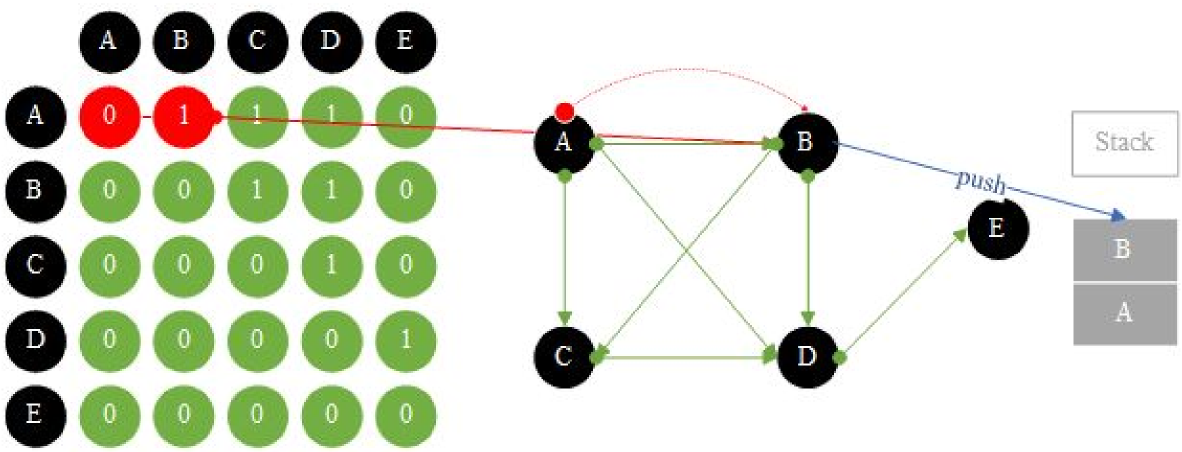
1. The adjacency matrix is described above:
The total number of vertices is a two-dimensional array size, if have value of the edge is 1, otherwise no value of the edge is 0.

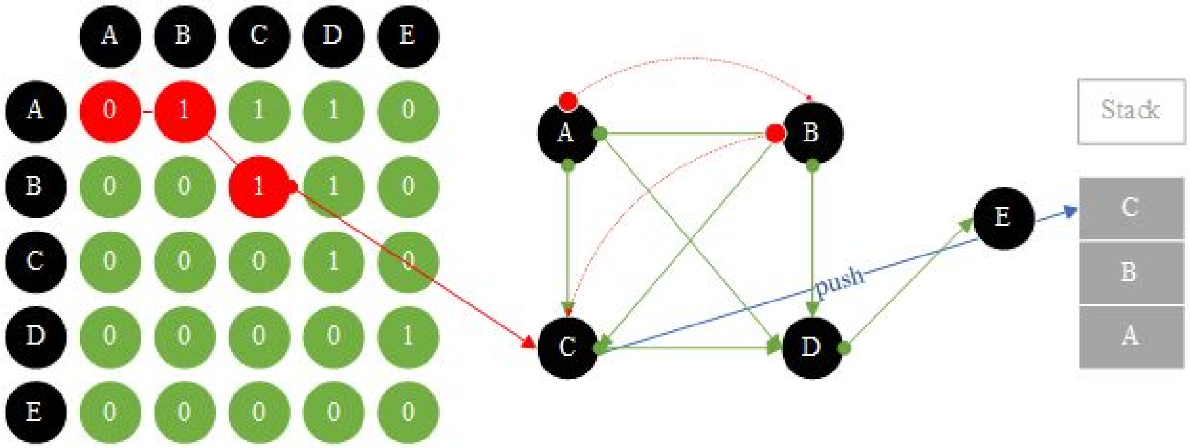
	A	B	C	D	E
A	0	1	1	1	0
B	0	0	1	1	0
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

2. Depth-First Search:

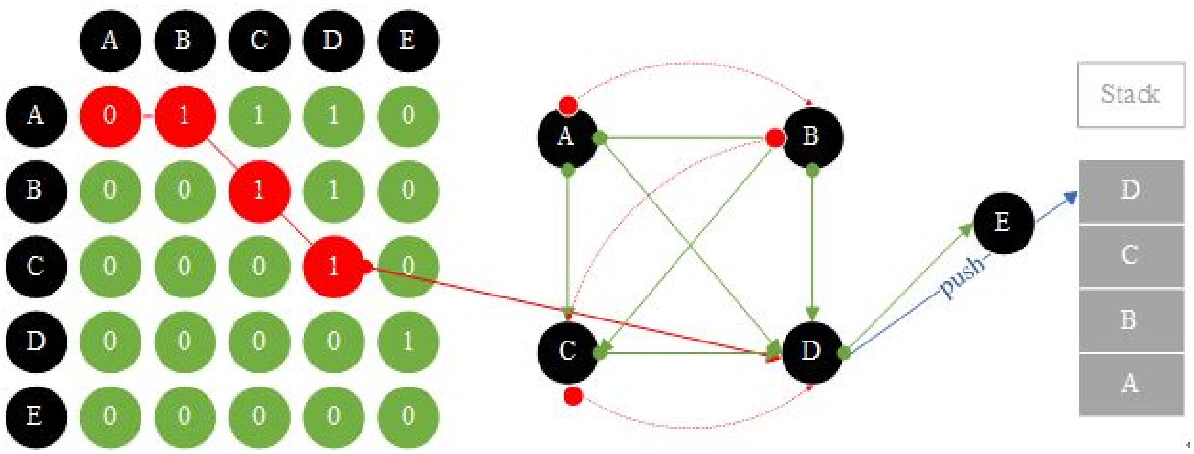
Look for the neighboring edge node B from A and then find the neighboring node C from B and so on until all nodes are found A -> B -> C -> D -> E.



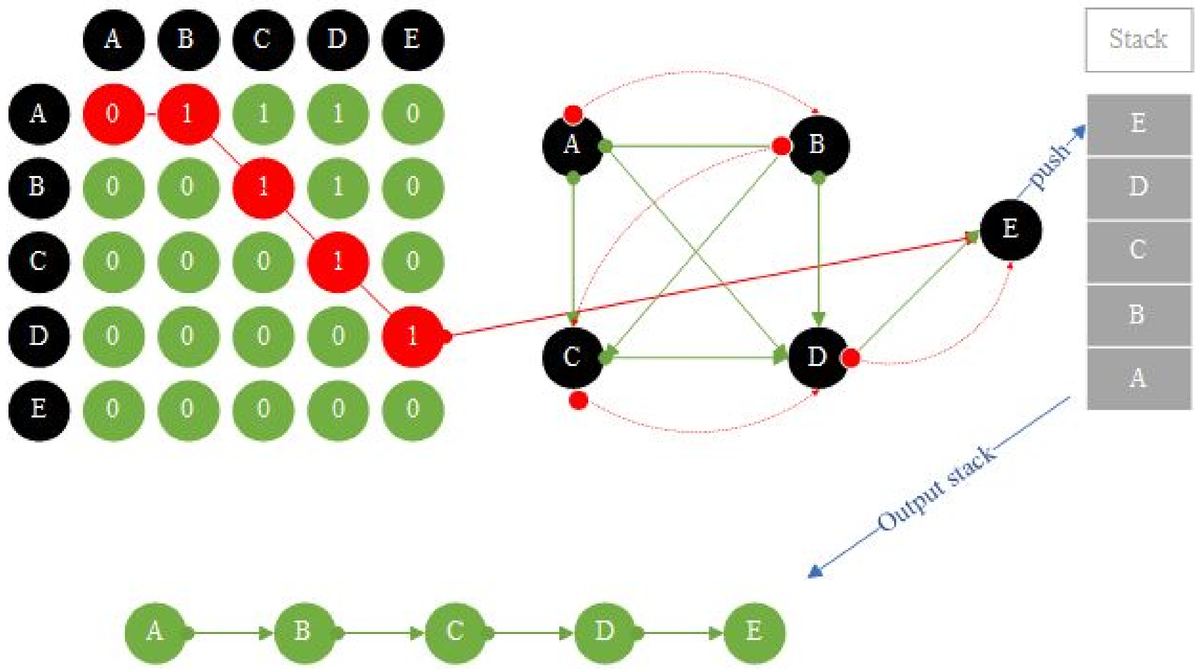




A B C D E
push
Stack
C
B
A



A B C D E
push
Stack
D
C
B
A



A
B
C
D
E
0
1
1
1
0
0
0
1
1
0
0
0
0
1
0
0
0
0
0
1
0
0
0
0
0
Stack
E
D
C
B
A
push
Output stack
A
B
C
D
E

node.py

```
class Node:
    data = ""
    prev = None
    next = None

    def __init__(self, data, prev, next):
        self.data = data
        self.prev = prev
        self.next = next
```

vertex.py

```
class Vertex:
    data = ""
    visited = False # Have you visited

    def __init__(self, data, visited):
        self.data = data
        self.visited = visited
```

stack.py

```
from node import Node

class Stack:
    __head = None
    __count = 0

    def push(self, element):
        if self.__head == None:
            self.__head = Node(element, None, None)
        else:
            new_node = Node(element, None, None)
            new_node.next =self.__head
            self.__head.prev = new_node
            self.__head = new_node
        self.__count += 1

    def pop(self):
        p = self.__head
        self.__head = self.__head.next

        # Delete node
        p.next = None
        p.prev = None

        self.__count -= 1
        return p.data

    def peek(self):
        return self.__head.data

    def size(self):
        return self.__count
```

```
def is_empty(self):
    if self.__count == 0:
        return True
    else:
        return False
```

graph.py

```
from stack import Stack
from vertex import Vertex

class Graph:
    __max_vertex___size = 0 # Two-dimensional array __size
    __size = 0 # Current vertex __size
    __vertexs = [] # Vertex array
    __adjacency_matrix = [[]] # Two-dimensional adjacency array

    __stack = None # Stack saves current vertices

    def __init__(self, __max_vertex___size):
        self.__max_vertex___size = __max_vertex___size
        self.__vertexs = []

        self.__adjacency_matrix = [[0 for i in
range(self.__max_vertex___size)]for i in
range(self.__max_vertex___size)]

        self.__stack = Stack()

    def add_vertex(self, data):
        vertex = Vertex(data, False)
        self.__vertexs.append(vertex)
        self.__size += 1

    def add_edge(self, from_position, to):
        # A -> B  and B -> A are different
        self.__adjacency_matrix[from_position][to] = 1
```

```python
def depth_first_search(self):
    # Start searching from the first vertex
    first_vertex = self.__vertexs[0]
    first_vertex.visited = True
    print(first_vertex.data, end="")
    self.__stack.push(0)

    while self.__stack.is_empty() == False:
        row = self.__stack.peek()
        # Get adjacent vertex positions that have not been visited
        col = self.find_adjacency_unvisited_vertex(row)
        if (col == -1):
            self.__stack.pop()
        else:
            self.__vertexs[col].visited = True
            print(" -> ", self.__vertexs[col].data, end="")
            self.__stack.push(col)
    self.clear()

# Get adjacent vertex positions that have not been visited
def find_adjacency_unvisited_vertex(self, row):
    for col in range(0, self.__size):
        if (self.__adjacency_matrix[row][col] == 1 and
self.__vertexs[col].visited == False):
            return col

    return -1
```

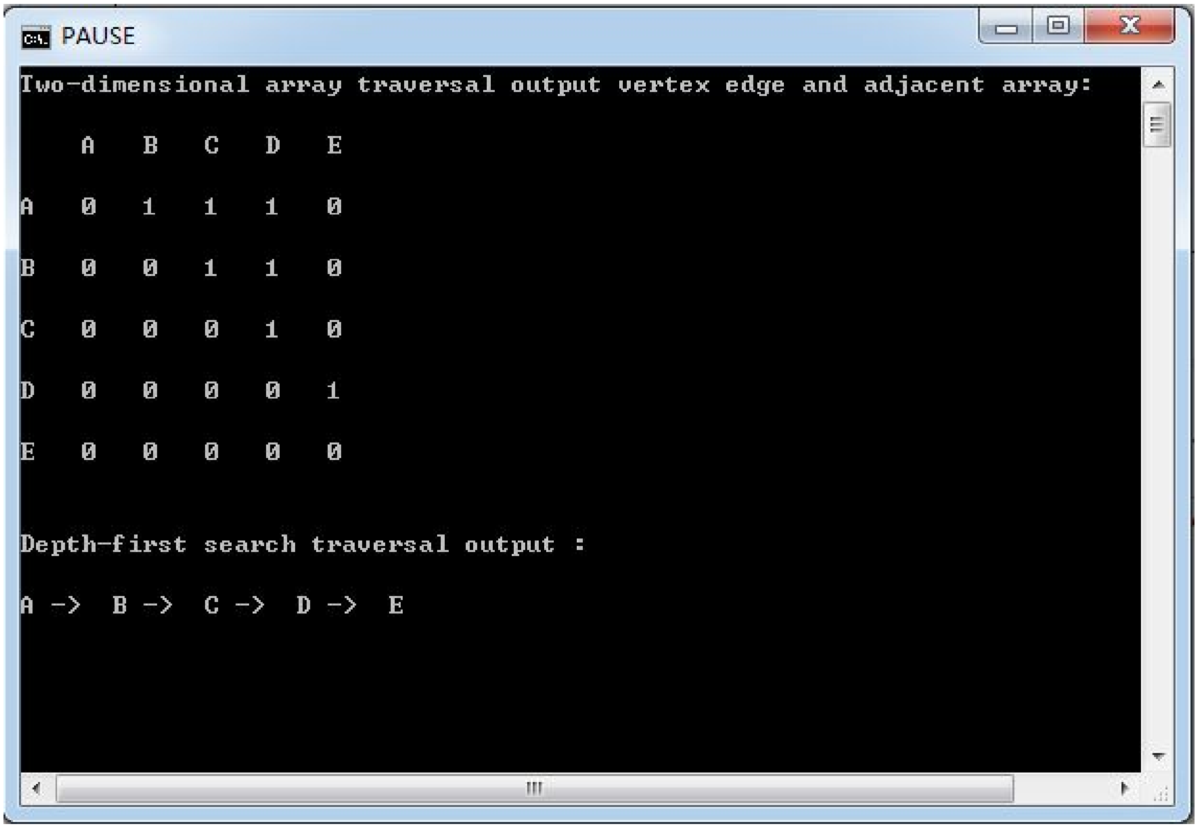
```
def clear(self):
    for i in range(0, self.__size):
        self.__vertexs[i].visited = False

def get_adjacency_matrix(self):
    return self.__adjacency_matrix

def get_vertexs(self):
    return self.__vertexs
```

test_graph.py

```
from graph import Graph

def print_graph(graph):
    print("Two-dimensional array traversal output vertex edge and adjacent array:  \n   ")
    length = len(graph.get_vertexs())
    print("    ", end="")
    for i in range(0, length):
        print(graph.get_vertexs()[i].data , "  ", end="")
    print("\n")

    length = len(graph.get_adjacency_matrix())
    for i in range(0, length):
        print(graph.get_vertexs()[i].data , "  ", end="")
        for j in range(0, length):
            print(graph.get_adjacency_matrix()[i][j] , "  ", end="")
        print("\n")

def main():
    graph = Graph(5)

    # Add Vertex
    graph.add_vertex("A")
    graph.add_vertex("B")
    graph.add_vertex("C")
    graph.add_vertex("D")
    graph.add_vertex("E")

    # Add adjacency edge
    graph.add_edge(0, 1)
    graph.add_edge(0, 2)
    graph.add_edge(0, 3)
```

```python
    graph.add_edge(1, 2)
    graph.add_edge(1, 3)
    graph.add_edge(2, 3)
    graph.add_edge(3, 4)






    # Two-dimensional array traversal output vertex edge and adjacent array
    print_graph(graph)

    # Depth-first search traversal output
    print("\nDepth-first search traversal output : \n")
    graph.depth_first_search()

if __name__ == "__main__":
    main()
```

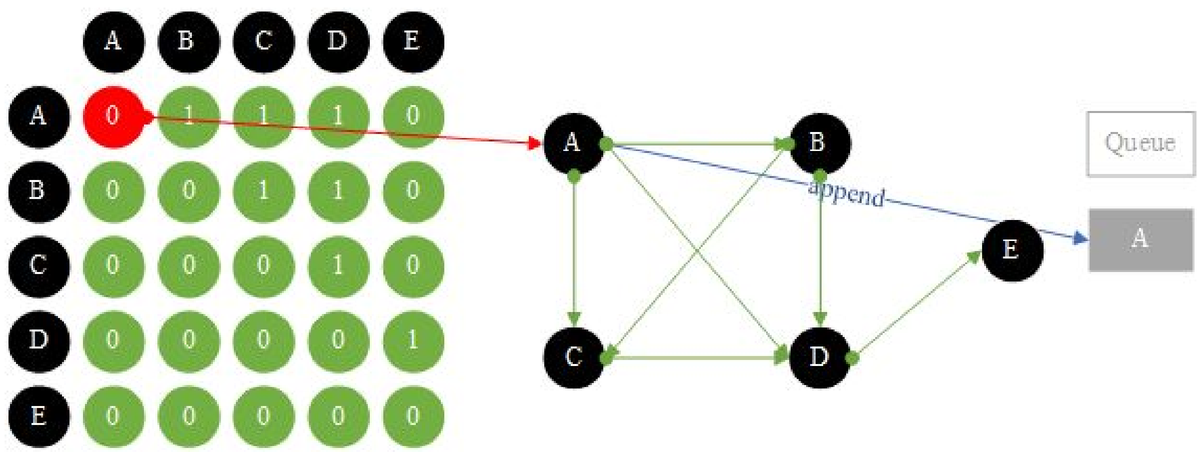
Result:

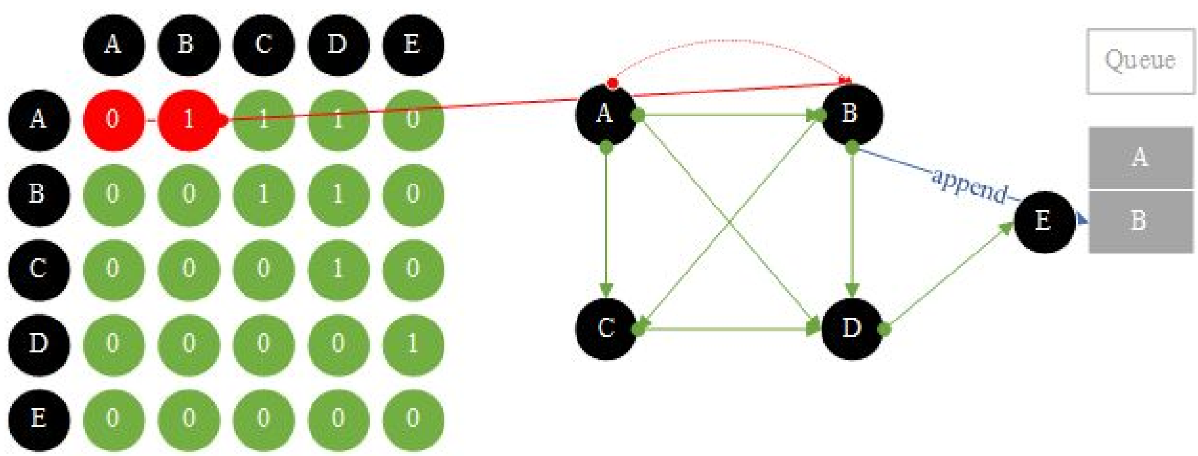


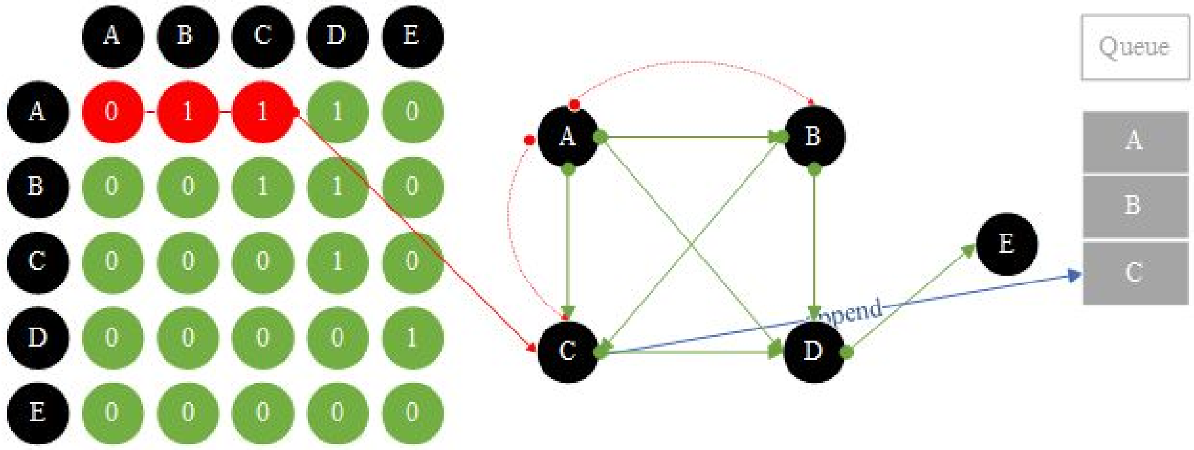

```
Two-dimensional array traversal output vertex edge and adjacent array:

    A   B   C   D   E

A   0   1   1   1   0

B   0   0   1   1   0

C   0   0   0   1   0

D   0   0   0   0   1

E   0   0   0   0   0


Depth-first search traversal output :

A ->  B ->  C ->  D ->  E
```

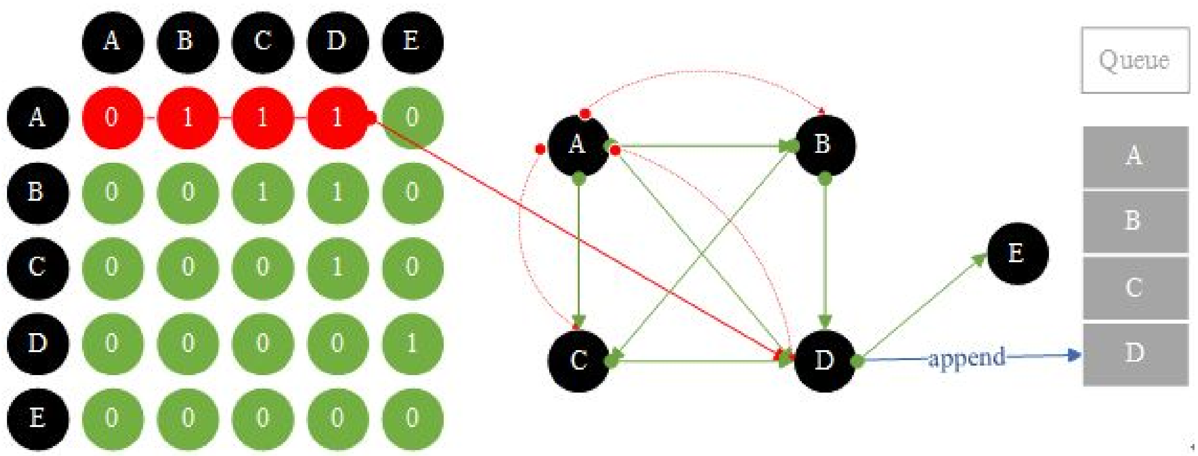
Directed Graph and Breadth-First Search

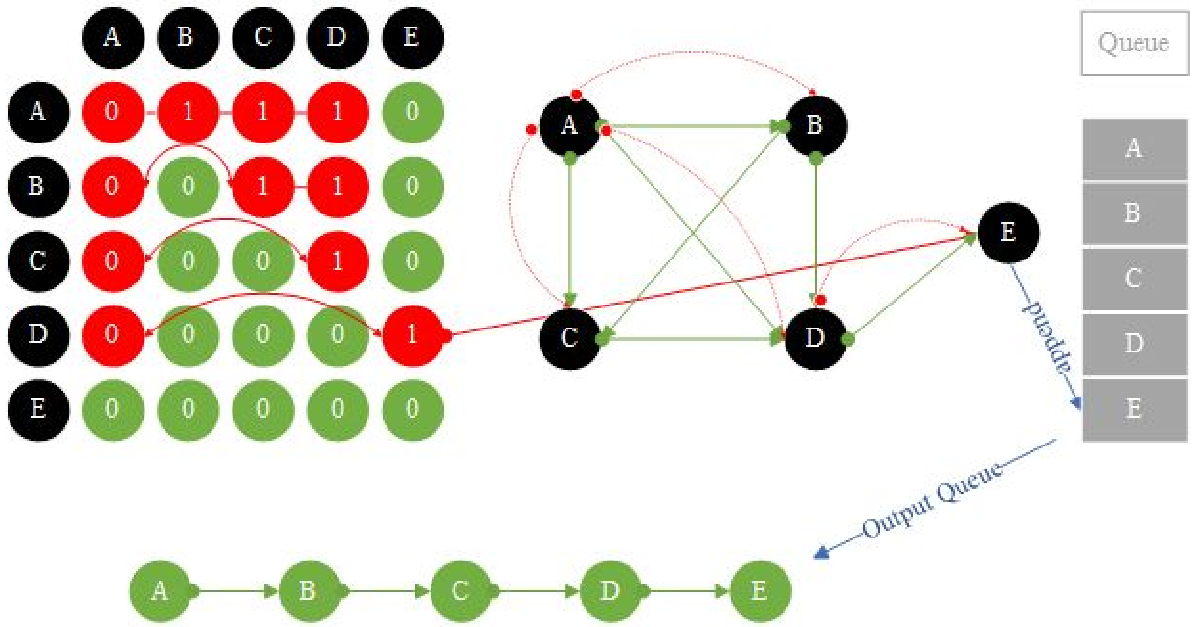
Breadth-First Search:
Find all neighboring edge nodes B, C, D from A and then find all neighboring nodes A, C, D from B and so on until all nodes are found A -> B -> C -> D -> E.






A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
append
Queue
A
B
C


A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
A
B
C
D
E
append
Queue
A
B
C
D



Queue
A
B
C
D
E
append
Output Queue

node.py

```
class Node:
    data = ""
    prev = None
    next = None

    def __init__(self, data, prev, next):
        self.data = data
        self.prev = prev
        self.next = next
```

vertex.py

```
class Vertex:
    data = ""
    visited = False # Have you visited

    def __init__(self, data, visited):
        self.data = data
        self.visited = visited
```

queue.py

```
from node import Node
class Queue:
    __head = None
    __tail = None
    __count = 0 # Number of elements

    def add(self, element):
        if self.__head == None:
            self.__head = Node(element, None, None)
            self.__tail = self.__head
        else:
            new_node = Node(element, None, None)
            self.__tail.next = new_node
            new_node.prev = self.__tail
            self.__tail = new_node
        self.__count += 1

    def poll(self):
        p = self.__head
        self.__head = self.__head.next

        p.next = None
        p.prev = None

        self.__count -= 1
        return p.data

    def peek(self):
        return self.__head.data

    def size(self):
        return self.__count
```

```
def is_empty(self):
    if self.__count == 0:
        return True
    else:
        return False
```

graph.py

```
from queue import Queue
from vertex import Vertex

class Graph:
    __max_vertex___size = 0 # Two-dimensional array __size
    __size = 0 # Current vertex __size
    __vertexs = [] # Vertex array
    __adjacency_matrix = [[]] # Two-dimensional adjacency array

    __queue = None # Queue saves current vertices

    def __init__(self, __max_vertex___size):
        self.__max_vertex___size = __max_vertex___size
        self.__vertexs = []

        self.__adjacency_matrix = [[0 for i in
range(self.__max_vertex___size)]for i in
range(self.__max_vertex___size)]

        self.__queue = Queue()

    def add_vertex(self, data):
        vertex = Vertex(data, False)
        self.__vertexs.append(vertex)
        self.__size += 1

    def add_edge(self, from_position, to):
        # A -> B  and B -> A are different
        self.__adjacency_matrix[from_position][to] = 1
```

```python
def breadth_first_search(self):
    # Start searching from the first vertex
    first_vertex = self.__vertexs[0]
    first_vertex.visited = True
    print(first_vertex.data, end="")
    self.__queue.add(0)

    col = 0
    while self.__queue.is_empty() == False:
        head = self.__queue.poll()

        # Get adjacent vertex positions that have not been visited
        col = self.find_adjacency_unvisited_vertex(head)
        # Loop through all vertices connected to the current vertex
        while(col != -1):
            self.__vertexs[col].visited = True
            print(" -> ", self.__vertexs[col].data, end="")
            self.__queue.add(col)
            col = self.find_adjacency_unvisited_vertex(head)
    self.clear()


# Get adjacent vertex positions that have not been visited
def find_adjacency_unvisited_vertex(self, row):
    for col in range(0, self.__size):
```

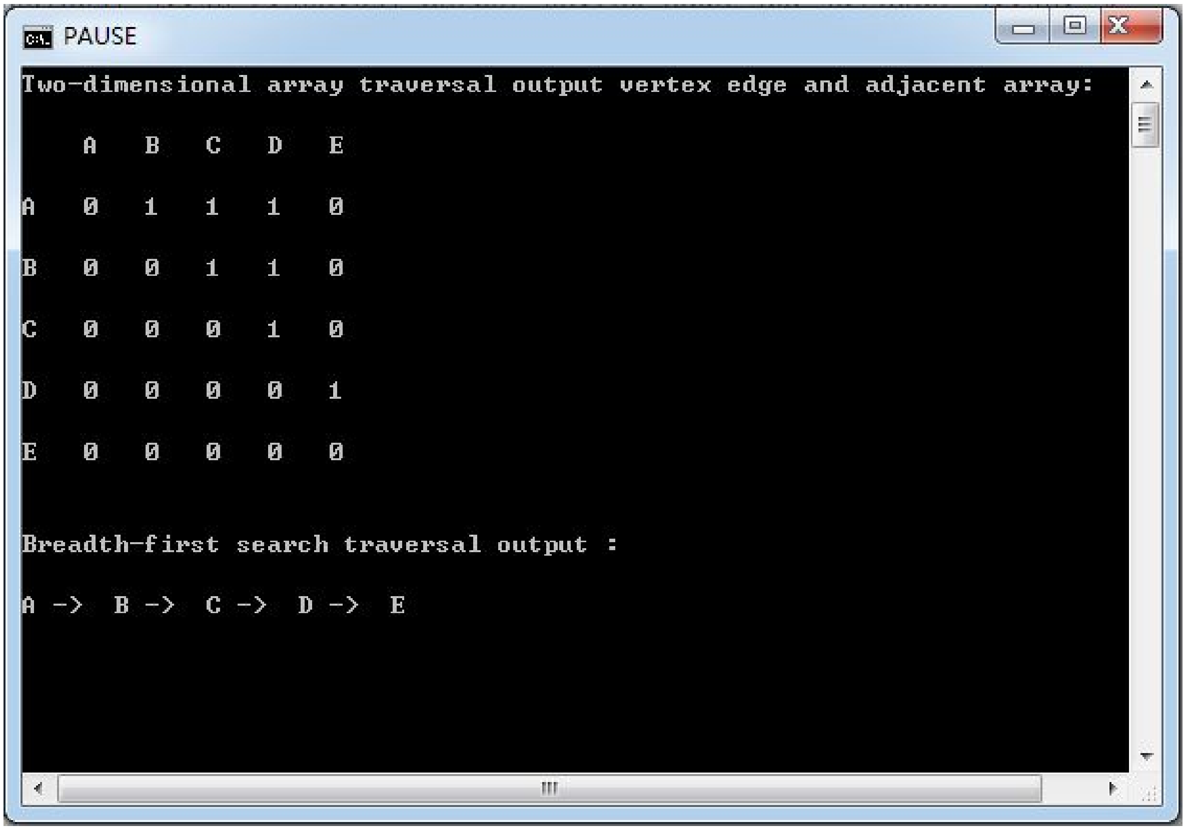
```
            if (self.__adjacency_matrix[row][col] == 1 and
self.__vertexs[col].visited == False):
                return col

        return -1

    def clear(self):
        for i in range(0, self.__size):
            self.__vertexs[i].visited = False

    def get_adjacency_matrix(self):
        return self.__adjacency_matrix

    def get_vertexs(self):
        return self.__vertexs
```

test_graph.py

```
from graph import Graph

def print_graph(graph):
    print("Two-dimensional array traversal output vertex edge and adjacent array:  \n   ")
    length = len(graph.get_vertexs())
    print("    ", end="")
    for i in range(0, length):
        print(graph.get_vertexs()[i].data , "  ", end="")
    print("\n")

    length = len(graph.get_adjacency_matrix())
    for i in range(0, length):
        print(graph.get_vertexs()[i].data , "  ", end="")
        for j in range(0, length):
            print(graph.get_adjacency_matrix()[i][j] , "  ", end="")
        print("\n")

def main():
    graph = Graph(5)

    # Add Vertex
    graph.add_vertex("A")
    graph.add_vertex("B")
    graph.add_vertex("C")
    graph.add_vertex("D")
    graph.add_vertex("E")

    # Add adjacency edge
    graph.add_edge(0, 1)
    graph.add_edge(0, 2)
    graph.add_edge(0, 3)
```

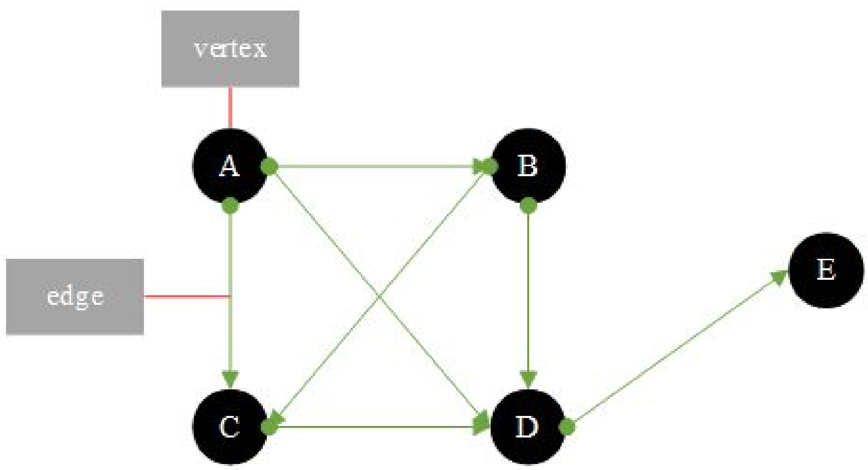
```python
    graph.add_edge(1, 2)
    graph.add_edge(1, 3)
    graph.add_edge(2, 3)
    graph.add_edge(3, 4)









    # Two-dimensional array traversal output vertex edge and adjacent array
    print_graph(graph)

    # Breadth-first search traversal output
    print("\nBreadth-first search traversal output : \n")
    graph.breadth_first_search()

if __name__ == "__main__":
    main()
```

Result:


PAUSE
Two-dimensional array traversal output vertex edge and adjacent array:
A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
Breadth-first search traversal output :
A -> B -> C -> D -> E

Directed Graph Topological Sorting

Directed Graph Topological Sorting:
Sort the vertices in the directed graph with order of direction
.
Directed Graph has direction : A -> B and B -> A are different

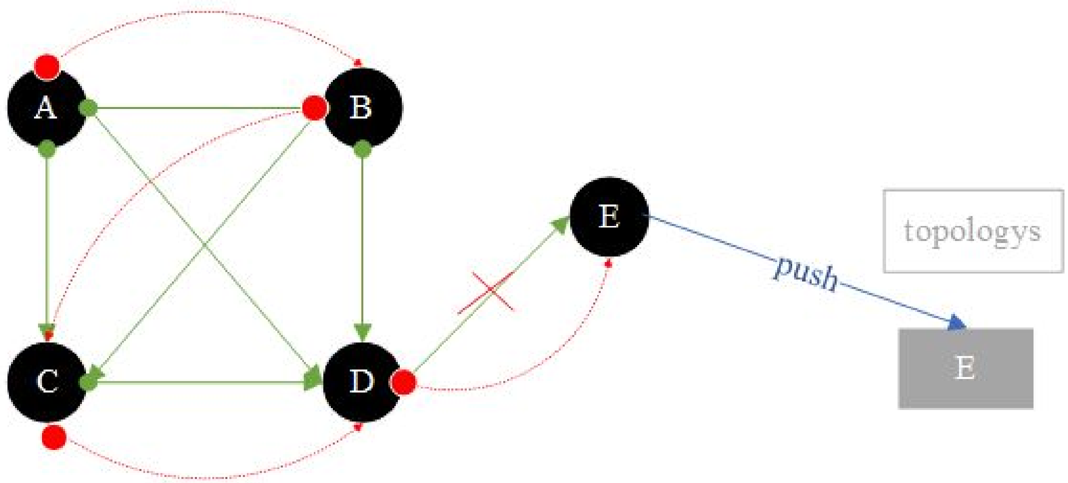


1. The adjacency matrix is described above:
The total number of vertices is a two-dimensional array size, if have value of the edge is 1, otherwise no value of the edge is 0.

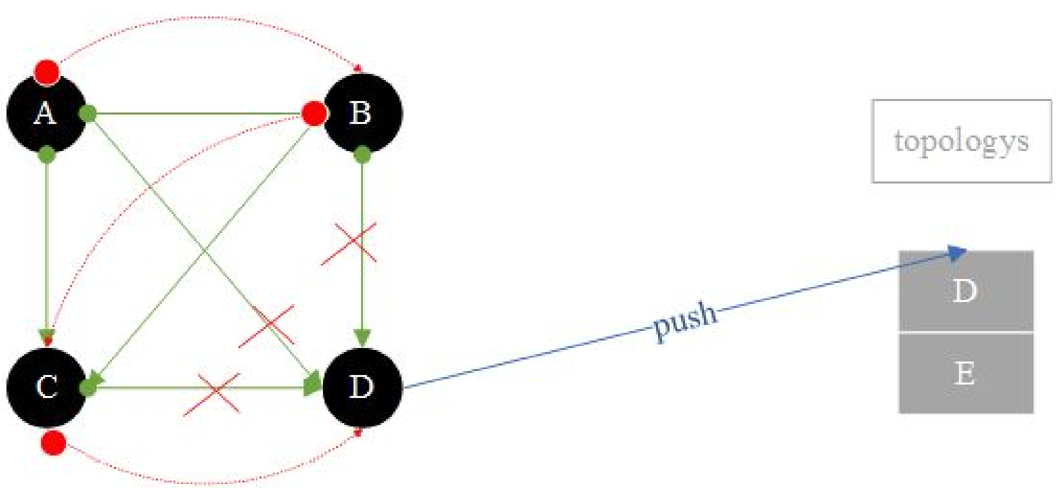
	A	B	C	D	E
A	0	1	1	1	0
B	0	0	1	1	0
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

Topological sorting from vertex A : A -> B -> C -> D -> E

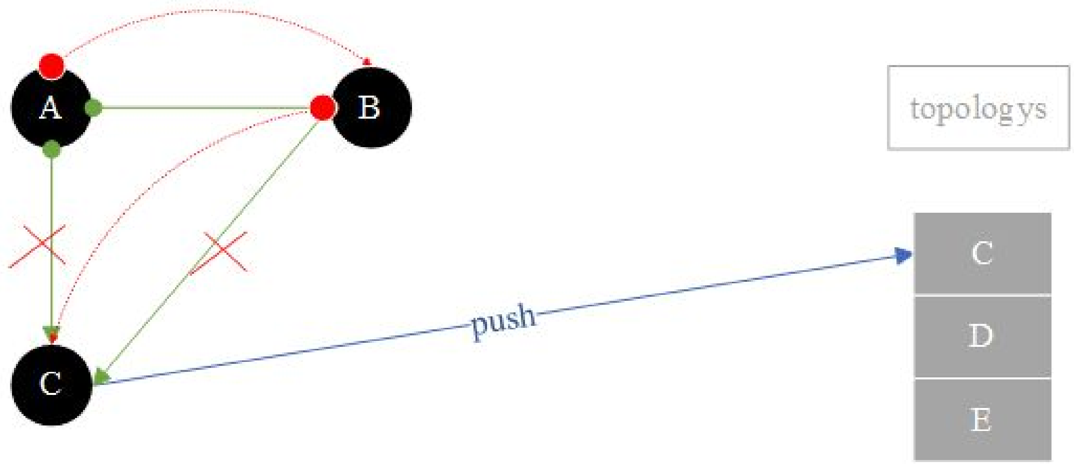
Find no successor vertices E then save to topologys, last E remove from the graph



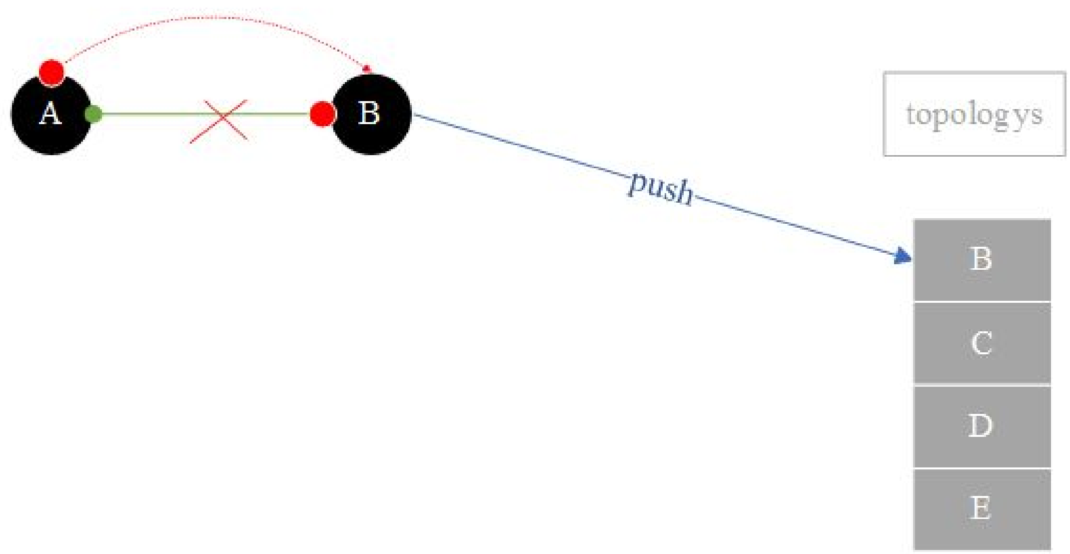
Find no successor vertices D then save to topologys, last D remove from the graph



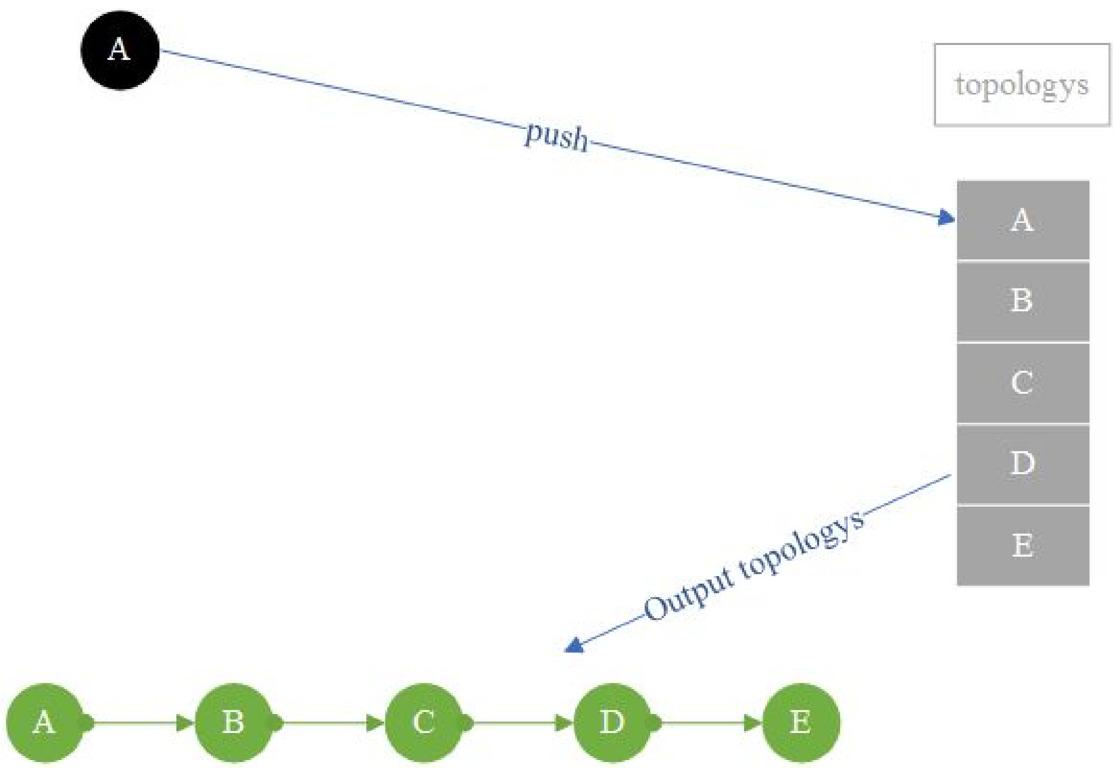
Find no successor vertices C then save to topologys, last C remove from the graph



Find no successor vertices C then save to topologys, last C remove from the graph



Find no successor vertices C then save to topologys, last C remove from the graph



node.py

```
class Node:
    data = ""
    prev = None
    next = None

    def __init__(self, data, prev, next):
        self.data = data
        self.prev = prev
        self.next = next
```

vertex.py

```
class Vertex:
    data = ""
    visited = False # Have you visited

    def __init__(self, data, visited):
        self.data = data
        self.visited = visited
```

queue.py

```
from node import Node

class Queue:
    __head = None
    __tail = None
    __count = 0 # Number of elements

    def add(self, element):
        if self.__head == None:
            self.__head = Node(element, None, None)
            self.__tail = self.__head
        else:
            new_node = Node(element, None, None)
            self.__tail.next = new_node
            new_node.prev = self.__tail
            self.__tail = new_node
        self.__count += 1

    def poll(self):
        p = self.__head
        self.__head = self.__head.next

        p.next = None
        p.prev = None

        self.__count -= 1
        return p.data

    def peek(self):
        return self.__head.data

    def size(self):
```

```python
        return self.__count

    def is_empty(self):
        if self.__count == 0:
            return True
        else:
            return False
```

graph.py

```
from queue import Queue
from vertex import Vertex

class Graph:
    __max_vertex___size = 0 # Two-dimensional array __size
    __size = 0 # Current vertex __size
    __vertexs = [] # Vertex array
    __adjacency_matrix = [[]] # Two-dimensional adjacency array
    __topologys = [] # An array of topological sort results, recording
the sorted sequence number of each node.
    __queue = None # Queue saves current vertices

    def __init__(self, __max_vertex___size):
        self.__max_vertex___size = __max_vertex___size
        self.__vertexs = []
        self.__topologys = [0 for i in
range(self.__max_vertex___size)]
        self.__adjacency_matrix = [[0 for i in
range(self.__max_vertex___size)]for i in
range(self.__max_vertex___size)]

        self.__queue = Queue()

    def add_vertex(self, data):
        vertex = Vertex(data, False)
        self.__vertexs.append(vertex)
        self.__size += 1

    def add_edge(self, from_position, to):
        # A -> B  and B -> A are different
        self.__adjacency_matrix[from_position][to] = 1
```

```
    def topology_sort(self):
        while (self.__size > 0):
            no_successor_vertex = self.get_no_successor_vertex() # Get a no successor node
            if (no_successor_vertex == -1):
                print("There are ring in graph")
                return

            # Copy the deleted node to the sorted array
            self.__topologys[self.__size - 1] = self.__vertexs[no_successor_vertex]
            self.remove_vertex(no_successor_vertex) # Delete no successor node


    def get_no_successor_vertex(self):
        exist_successor = False
        for row in range(0, self.__size):
            exist_successor = False
            # If the node has a fixed row, each column has a 1, indicating that the node has a successor, terminating the loop
            for col in range(0, self.__size):
                if (self.__adjacency_matrix[row][col] == 1):
                    exist_successor = True
                    break

            # If the node has no successor, return its subscript
            if (exist_successor == False):
                return row
        return -1

    def remove_vertex(self, vertex):
        if (vertex != self.__size - 1): #If the vertex is the last element, the end
            for i in range(vertex, self.__size - 1): # The vertices are removed
                self.__vertexs[i] = self.__vertexs[i + 1]
```

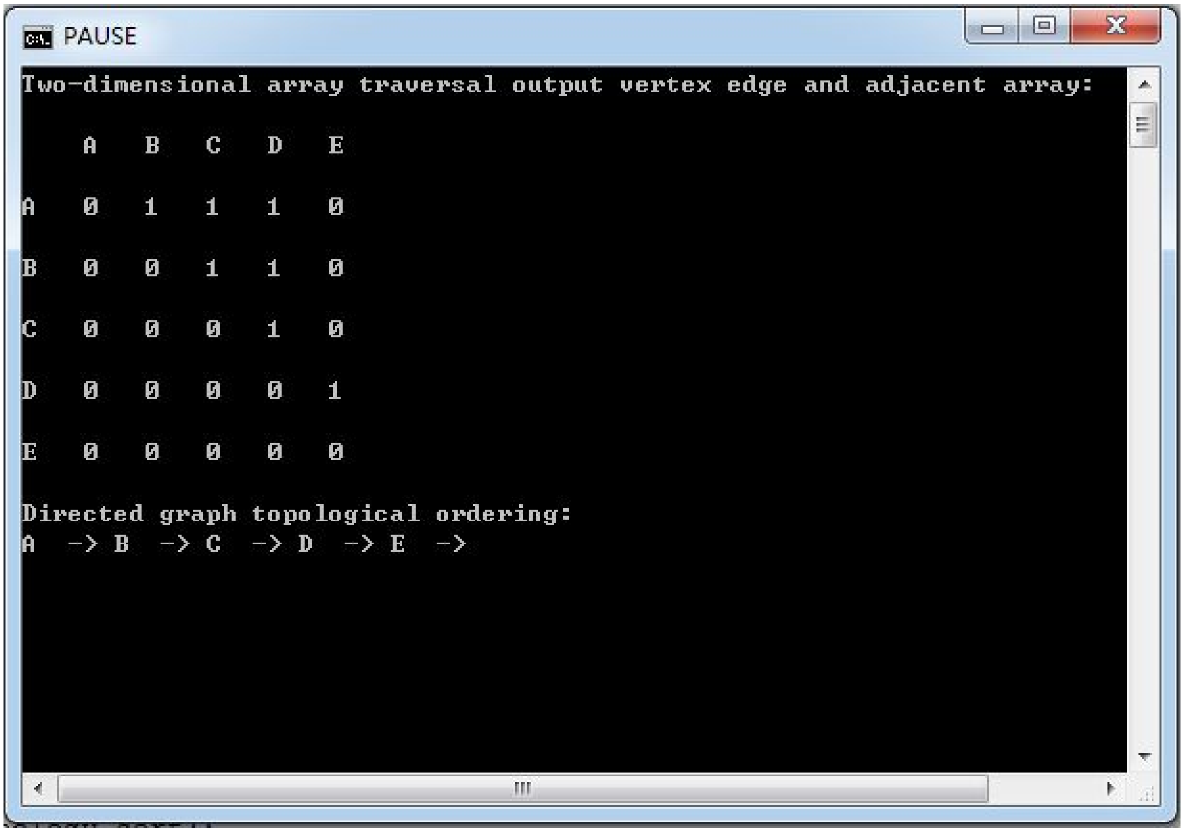
```
        for row in range(vertex, self.__size - 1): # Pan up one line
            for col in range(0, self.__size):
                self.__adjacency_matrix[row][col] =
self.__adjacency_matrix[row + 1][col]

        for col in range(vertex, self.__size - 1): # Pan one line to
the left
            for row in range(0, self.__size - 1):
                self.__adjacency_matrix[row][col] =
self.__adjacency_matrix[row][col + 1]
        self.__size -= 1 # Decrease the number of vertices

    def get_topologys(self):
        return self.__topologys

    def clear(self):
        for i in range(0, self.__size):
            self.__vertexs[i].visited = False

    def get_adjacency_matrix(self):
        return self.__adjacency_matrix

    def get_vertexs(self):
        return self.__vertexs
```

test_topology.py

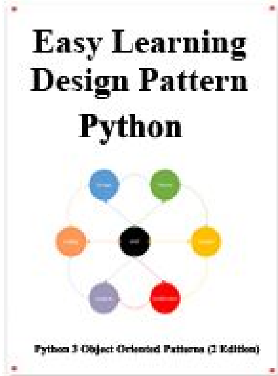
```
from graph import Graph

def print_graph(graph):
    print("Two-dimensional array traversal output vertex edge and adjacent array:  \n   ")
    length = len(graph.get_vertexs())
    print("    ", end="")
    for i in range(0, length):
        print(graph.get_vertexs()[i].data , "  ", end="")
    print("\n")

    length = len(graph.get_adjacency_matrix())
    for i in range(0, length):
        print(graph.get_vertexs()[i].data , "  ", end="")
        for j in range(0, length):
            print(graph.get_adjacency_matrix()[i][j] , "  ", end="")
        print("\n")

def main():
    graph = Graph(5)

    # Add Vertex
    graph.add_vertex("A")
    graph.add_vertex("B")
    graph.add_vertex("C")
    graph.add_vertex("D")
    graph.add_vertex("E")

    # Add adjacency edge
    graph.add_edge(0, 1)
    graph.add_edge(0, 2)
    graph.add_edge(0, 3)
```

```
    graph.add_edge(1, 2)
    graph.add_edge(1, 3)
    graph.add_edge(2, 3)
    graph.add_edge(3, 4)









    # Two-dimensional array traversal output vertex edge and
adjacent array
    print_graph(graph)

    print("Directed graph topological ordering:")
    graph.topology_sort()
    length = len(graph.get_topologys())
    for i in range(0, length):
        print(graph.get_topologys()[i].data, " -> ", end="")

if __name__ == "__main__":
    main()
```

Result:


PAUSE
Two-dimensional array traversal output vertex edge and adjacent array:
A B C D E
A 0 1 1 1 0
B 0 0 1 1 0
C 0 0 0 1 0
D 0 0 0 0 1
E 0 0 0 0 0
Directed graph topological ordering:
A -> B -> C -> D -> E ->

Thanks for learning, please study book
https://www.amazon.com/dp/B086PLBYSZ



If you enjoyed this book and found some benefit in reading this, I'd like to hear from you and hope that you could take some time to post a review on Amazon. Your feedback and support will help us to greatly improve in future and make this book even better.

You can follow this link now.

http://www.amazon.com/review/create-review?&asin=109873548X
Different country reviews only need to modify the amazon domain name in the link:
www.amazon.co.uk
www.amazon.de
www.amazon.fr
www.amazon.es
www.amazon.it
www.amazon.ca
www.amazon.nl
www.amazon.in
www.amazon.co.jp
www.amazon.com.br
www.amazon.com.mx
www.amazon.com.au

I wish you all the best in your future success!