**Q1.1:Q1:**

**A group of black and white text

Description automatically generated**

**def f1(self, linklist, name="", price=-1):**

* Hàm tạo một nút mới n với tên và giá trị được truyền vào làm đối số.

**if name.startswith("B") or price >100:**

**return**

* Nếu tên bắt đầu bằng chữ “B” hoặc giá lớn hơn 100, hàm sẽ trả về mà không làm gì cả.

n = Node(Car(name,price))

**if (linklist.isEmpty()):**

**linklist.head = linklist.tail = n**

* hàm sẽ kiểm tra xem danh sách liên kết có trống không. Nếu đúng như vậy thì phần đầu và phần cuối của danh sách liên kết được đặt thành n.

**else:**

**linklist.tail.next = n**

**linklist.tail = n**

* Ngược lại, n sẽ được thêm vào cuối danh sách liên kết bằng cách đặt tail.next thành n và cập nhật tail thành n.

**Q1.2:**

**A group of black text

Description automatically generated**

**def f2(self, linkList, name="", price=-1):**

**node = Node(Car(name,price))**

**if linkList ==None:**

**linkList.head = linkList.tail = node**

* Nếu Danh sách liên kết là Không có thì phần đầu và đuôi của Danh sách liên kết được đặt thành nút.

**else:**

**node.next = linkList.head**

**linkList.head = node**

* Nếu không, nút sẽ được thêm vào đầu linkList bằng cách đặt node.next thành linkList.head và cập nhật phần đầu thành nút.

**Q1.3:**

**A group of black letters

Description automatically generated**

**def f3(self, linkList):**

**if self.check(linkList.head.data.Price):**

**linkList.head = linkList.head.next**

* Nếu đúng, phần đầu của linkList sẽ được cập nhật sang nút tiếp theo.

**else:**

**cu = linkList.head**

**while(cu.next):**

**if (self.check(cu.next.data.Price)):**

**break**

**cu = cu.next**

* Nếu đúng thì vòng lặp bị hỏng. Nếu đến cuối linkList và không có nút nào thỏa mãn điều kiện thì hàm sẽ trả về.

**if (cu.next==None):**

**return**

**else:**

**if (cu.next==linkList.tail):**

**linkList.tail=cu**

**cu.next = cu.next.next**

* Ngược lại, con trỏ tiếp theo của nút đứng trước nút thỏa mãn điều kiện sẽ được cập nhật để bỏ qua nút thỏa mãn điều kiện. Nếu nút thỏa mãn điều kiện là phần đuôi của LinkList thì phần đuôi đó sẽ được cập nhật lên nút trước đó.

**Q1.4:**

**A group of black text

Description automatically generated with medium confidence**

**if linkList ==None:**

**return**

**i = linkList.head**

**while (i.next):**

**j = i.next**

**while j:**

**if i.data.Price > j.data.Price:**

**t = i.data**

**i.data = j.data**

**j.data = t**

**j= j.next**

**i = i.next**

* Nếu `Price` của nút đầu tiên lớn hơn `Price` của nút thứ hai, các đối tượng `Car` trong các nút sẽ được hoán đổi.

**Q2.1:**

**A group of black letters

Description automatically generated**

**def f1(self,tree,name, price=-1):**

**if name[0] =="B" or price >100: return**

**node = Node(Car(name,price))**

* Nếu chữ cái đầu tiên của tên là “B” hoặc giá lớn hơn 100, hàm sẽ trả về mà không làm gì cả.

**if tree.isEmpty():**

**tree.root = node**

**return**

**cur = tree.root**

**father = None**

* hàm sẽ kiểm tra xem cây có trống không. Nếu đúng như vậy thì gốc của cây được đặt thành nút.

**while cur:**

**if cur.data.Price ==price:**

**# print(f"key {data} is exist ");**

**Return**

* hàm sẽ lặp qua cây bằng vòng lặp while và kiểm tra xem giá của nút hiện tại có bằng giá được truyền vào dưới dạng đối số hay không. Nếu đúng như vậy, hàm sẽ trả về mà không làm gì cả.

**else:**

**father = cur**

**if cur.data.Price <price:**

**cur = cur.right**

**else:**

**cur = cur.left**

* Mặt khác, hàm cập nhật nút cha thành nút hiện tại và chuyển sang nút con bên trái nếu giá của nút hiện tại nhỏ hơn giá được truyền vào làm đối số hoặc chuyển sang nút con bên phải nếu ngược lại.

**if father.data.Price<price:**

**father.right = node**

**else:**

**father.left = node**

* Khi đến cuối cây, hàm sẽ thêm nút mới vào cây bằng cách đặt nút con bên phải của nút cha thành nút nếu giá của nút cha nhỏ hơn giá được truyền vào làm đối số hoặc đặt giá trị con trái của nút cha tới nút khác.

**Q2.2:**

**A close-up of a number

Description automatically generated**

**def preOrder(self,p):**

**if p==None:**

**return**

**if (self.check(p.data.Price)):**

**self.visit(p)**

**self.preOrder(p.left)**

**self.preOrder(p.right)**

**#end def**

**def visit(self,p):**

**if p==None:**

**return**

**print(f"{p.data}",end =" ")**

**def preVisit(self,tree):**

**self.preOrder(tree.root)**

**print("")**

**#end def**

**def check(self, x):**

**return x>=3 and x<=5**

**def f2(self,tree):**

**self.preVisit(tree)**

* Phương thức preOrder thực hiện việc duyệt cây nhị phân theo thứ tự trước có gốc tại p. Phương thức truy cập in dữ liệu được lưu trữ trong nút p. Phương thức preVisit thực hiện việc duyệt cây nhị phân theo thứ tự trước có gốc tại tree.root và in dữ liệu được lưu trữ trong mỗi nút. Phương thức kiểm tra trả về True nếu giá trị x nằm trong khoảng từ 3 đến 5 (bao gồm) và trả về Sai nếu ngược lại.

**Q2.3:**

**A close up of letters

Description automatically generated**

**flag = True**

**def check(self, x):**

**return x<7**

**deleteNode =None**

**count =0**

* Phương thức kiểm tra trả về True nếu giá trị x nhỏ hơn 7 và trả về Sai nếu ngược lại.

**def breadth\_first2(self,tree):**

**if tree.isEmpty():**

**return**

**my = MyQueue()**

**my.EnQueue(tree.root)**

* Phương thức breadth\_first2 thực hiện việc duyệt cây nhị phân có gốc tại tree.root. Phương thức này kiểm tra xem phần con bên trái và bên phải của mỗi nút có phải là Không hay không, Giá của nút có thỏa mãn điều kiện trong phương thức kiểm tra hay không và liệu thuộc tính cờ có đúng hay không. Nếu các điều kiện này được thỏa mãn, thuộc tính deleteNode được đặt thành nút hiện tại và thuộc tính count sẽ tăng lên.

**while not my.isEmpty():**

**p = my.DeQueue()**

**if (p.left!=None and p.right!=None and self.check(p.data.Price) and self.flag ):**

**# self.flag = False**

**self.deleteNode =p**

**self.count +=1**

**if self.count ==1:**

**self.flag = False**

**break;**

**# self.visit(p)**

**if p.left!=None:**

**my.EnQueue(p.left)**

**if p.right!=None:**

**my.EnQueue(p.right)**

**# print("")**

**#end def**

**def deleteByCopyingLeft(self, p):**

**if p==None or p.left ==None :**

**return**

**if p.left.right==None:**

**p.data = p.left.data**

**p.left = p.left.left**

**return**

**cur = p.left.right**

**father =p.left**

**while cur.right:**

**father = cur**

**cur = cur.right**

**p.data = cur.data**

**father.right = cur.left**

**def f3(self,tree):**

**self.breadth\_first2(tree)**

**self.deleteByCopyingLeft(self.deleteNode)**

* Nếu thuộc tính đếm bằng 1, thuộc tính cờ được đặt thành Sai và vòng lặp bị hỏng. Phương thức deleteByCopyingLeft xóa nút được lưu trữ trong thuộc tính deleteNode bằng cách sao chép dữ liệu từ nút con bên trái của nút cần xóa hoặc nút ngoài cùng bên phải trong cây con bên trái của nút sẽ bị xóa nếu nút con trái có nút con phải. Phương thức f3 gọi phương thức breadth\_first2 trên cây nhị phân được truyền vào dưới dạng đối số, sau đó gọi phương thức deleteByCopyingLeft trên nút được lưu trữ trong thuộc tính deleteNode.

**Q2.4:**

**A group of letters and numbers

Description automatically generated**

**def check(self, x):**

**return x<7**

**RotateNode =None**

**count=0**

* Phương thức kiểm tra trả về True nếu giá trị x nhỏ hơn 7 và trả về Sai nếu ngược lại. Thuộc tính RotateNode được khởi tạo thành Không và thuộc tính count được khởi tạo thành 0.

**def f4(self, tree):**

**self.breadth\_first3(tree)**

**if self.RotateNode ==None:**

**return**

**self.rightRotation(tree, self.RotateNode.data)**

**#end def**

* Phương thức kiểm tra trả về True nếu giá trị x nhỏ hơn 7 và trả về Sai nếu ngược lại. Thuộc tính RotateNode được khởi tạo thành Không và thuộc tính count được khởi tạo thành 0. Phương thức f4 gọi phương thức breadth\_first3 trên cây nhị phân được truyền vào dưới dạng đối số. Nếu thuộc tính RotateNode là Không có thì phương thức trả về mà không làm gì cả. Mặt khác, phương thức rightRotation được gọi trên cây nhị phân được truyền vào dưới dạng đối số, với thuộc tính dữ liệu của RotateNode làm đối số thứ hai.

**def breadth\_first3(self,tree):**

**if tree.isEmpty():**

**return**

**my = MyQueue()**

**my.EnQueue(tree.root)**

**while not my.isEmpty():**

**p = my.DeQueue()**

**if (p.left!=None and self.check(p.data.Price) and self.flag ):**

**# self.flag = False**

**self.RotateNode =p**

**self.count +=1**

**if self.count ==1:**

**self.flag = False**

**break;**

**# self.visit(p)**

**if p.left!=None:**

**my.EnQueue(p.left)**

**if p.right!=None:**

**my.EnQueue(p.right)**

**#end def**

**def findFather(self,tree, data):**

**if tree.root.data.Price == data:**

**return None**

**fa = None**

**cur = tree.root**

**while cur:**

**if cur.data.Price == data:**

**return fa**

**fa = cur**

**if cur.data.Price<data:**

**cur = cur.right**

**else:**

**cur = cur.left**

**return None**

**#end def**

**def rightRotate(self,p):**

**if not p or not p.left:**

**return**

**c = p.left**

**p.left = c.right**

**c.right = p**

**return c**

**def rightRotation(self, tree, data):**

**f = self.findFather(tree,data.Price)**

**p = None**

**if f == None:**

**if (tree.root.data !=data):**

**return**

**else:**

**p = tree.root**

**else:**

**if (f.data.Price>data.Price):**

**p = f.left**

**else:**

**p = f.right**

**newNode = self.rightRotate(p)**

**if f ==None:**

**self.root = newNode**

**else:**

**if (f.data.Price<data.Price):**

**f.right = newNode**

**else:**

**f.left = newNode**