Çünkü Board sınıfındaki her satırın kendisi bir liste olduğu için, matrisin satırları için doğrudan Python’un yerleşik list sınıfını kullanabiliriz. Matrisin her satırındaki her konum X, O veya bir Dummy nesnesi tutabilir. Dummy nesneleri, açık konumları temsil etmek için eklenmiş olup kullanım kolaylığı sağlar. equal, eval ve full metotları öğrenciler için bir alıştırma olarak bırakılmıştır. Her birinin amacı bir sonraki bölümde açıklanacaktır.

**4.8 İki Boyutlu Diziler**

Hem animasyonlu hem de animasyonsuz birçok oyun, Tkinter ve turtle grafikleri kullanılarak kolayca uygulanabilir. Bir oyundaki animasyonlu karakterler veya taşlar, ekranda gerektiği gibi hareket eden bir turtle nesnesi olarak temsil edilebilir. Tic-tac-toe oyununda X ve O’lar RawTurtle nesneleri olarak tanımlanabilir. RawTurtle, normal bir Turtle nesnesi gibidir, ancak hareket edeceği tuvalin (canvas) belirtilmesi gerekir. Bölüm 4.8.2’de, X, O ve tahtadaki açık konumlar için yer tutucu olan özel Dummy sınıfını tanımlayan üç sınıfın kodu bulunmaktadır.

**4.8.2 The X,O,and Dummy Classes**

1. Human = -1
2. Computer = 1
3. 3
4. 4 # This class is just for placeholder objects when no move has been made
5. 5 # yet at a position in the board. Having eval() return 0 is convenient when no 6 # move has been made.
6. **class** **Dummy**:
7. **def** \_\_init\_\_(self):
8. **pass**
9. 10
10. **def** eval(self):
11. **return** 0
12. 13
13. **def** goto(self,x,y):
14. **pass**
15. 16
16. # In the X and O classes below the constructor begins by initializing the
17. # RawTurtle part of the object with the call to super().\_\_init\_\_(canvas). The
18. # super() call returns the class of the superclass (the class above the X or O
19. # in the class hierarchy). In this case, the superclass is RawTurtle. Then, # calling \_\_init\_\_ on the superclass initializes the part of the object that is # a RawTurtle.
20. **class** **X**(RawTurtle):
21. **def** \_\_init\_\_(self, canvas):
22. super().\_\_init\_\_(canvas)
23. self.ht()
24. self.getscreen().register\_shape("X",((-40,-36),(-40,-44),(0,-4),(40,-44),\(40,-36), (4,0),(40,36),(40,44),(0,4),(-40,44),(-40,36),(-4,0),(-40,-36)))
25. self.shape("X")
26. self.penup()
27. self.speed(5)
28. self.goto(-100,-100)
29. 33
30. **def** eval(self):
31. **return** Computer
32. 36
33. **class** **O**(RawTurtle):
34. **def** \_\_init\_\_(self, canvas):
35. super().\_\_init\_\_(canvas)
36. self.ht()
37. self.shape("circle")
38. self.penup()
39. self.speed(5)
40. self.goto(-100,-100)
41. 45
42. **def** eval(self):
43. **return** Human

**4.9 The Minimax Algoritması**

Dummy, X ve O sınıflarının her biri, bir bilgisayar hamlesi için 1, bir insan hamlesi için -1 veya henüz yapılmamış bir hamle için 0 döndüren bir eval metoduna sahiptir. Bu değerler minimax adlı bir algoritmada kullanılır.

Minimax algoritması, iki oyunculu oyunlarda kullanılan ve genellikle bir oyuncunun bilgisayar olduğu durumlarda tercih edilen bir rekürsif algoritmadır. Oyuncular sırayla hamle yaparken, her hamlenin sonucu analiz edilir ve en iyi hamle seçilir. Algoritmanın mantığı oldukça basittir: Bilgisayarın sırası geldiğinde, kendisi için en iyi sonucu getirecek hamleyi seçmelidir. Her olası hamle değerlendirilerek bilgisayarın en iyi sonucu elde etmesini sağlayan hamle bulunur. En iyi hamle 1 ile, en kötü hamle ise -1 ile temsil edilir. Bilgisayar, mümkün olan en yüksek değeri getiren hamleyi seçer. İşte bu, minimax algoritmasının "max" kısmıdır.

En iyi hamleyi bulmak için bilgisayar, insanın da en iyi hamlesini yapacağını varsayarak oyunu simüle eder. Bir insanın yapabileceği en iyi hamle -1 ile gösterilir. İnsan sırası geldiğinde, bilgisayar insanın yapabileceği en iyi hamleyi yapacağını varsayar. İşte bu, minimax algoritmasının "min" kısmıdır.

Minimax fonksiyonu iki parametre alır: Oyuncu (insan için -1, bilgisayar için 1) ve oyunun mevcut tahtası. Rekürsif fonksiyonun temel durumu üç farklı durumda sona erer:

1. Eğer mevcut tahta bilgisayar için bir galibiyet sağlıyorsa, minimax 1 döndürür.
2. Eğer mevcut tahta insan için bir galibiyet sağlıyorsa, minimax -1 döndürür.
3. Eğer tahta dolu ve kimse kazanmamışsa, minimax 0 döndürür.

Rekürsif minimax fonksiyonunun çalışma şekli, oyuncu parametresine bağlıdır. Eğer oyuncu bilgisayarsa, fonksiyon tahtadaki her olası hamleyi dener ve bilgisayarın yapabileceği en iyi hamleyi belirlemek için minimax’i insanın hamle yapacağı şekilde çağırır. Bilgisayarın kazanma ihtimalini maksimize etmek için her olası hamlenin sonucunu değerlendirerek en yüksek değeri döndürür.

Eğer minimax fonksiyonu, insanın hamle yapacağı sırada çağrılırsa, aynı süreç tersine işler. Fonksiyon, tahtanın bir kopyasında insan hamlesi yapar ve ardından minimax’i bilgisayarın hamle yapacağı şekilde çağırır. Ancak bu sefer, algoritma insanın kazanma ihtimalini maksimize etmeye çalışır ve dönen en düşük değeri seçer.

Minimax algoritmasının en önemli noktalarından biri, yalnızca oyunun sonucunu değerlendiren bir sayı döndürmesidir, ancak hangi hamlenin en iyi olduğunu doğrudan belirtmez. Bunu çözmek için, bilgisayarın hamle yapmasını sağlayan kod, her olası hamleyi dener, minimax’i çağırarak sonuç değerlerini toplar ve en yüksek değeri üreten hamleyi seçer. En iyi hamle belirlendikten sonra, bilgisayar tahtada bu hamleyi yapar ve sıra insana geçer.

Minimax algoritması, dama ve dört taş gibi mükemmel bilgiye sahip iki oyunculu oyunlarda kullanılabilir. Mükemmel bilgi, her iki oyuncunun da oyunun mevcut durumunu tamamen görebilmesi anlamına gelir. Örneğin, poker gibi oyunlar eksik bilgi içerdiğinden minimax için uygun değildir. Tic-tac-toe gibi küçük oyun alanına sahip oyunlar, minimax algoritması tarafından tamamen çözülebilir. Yani, bilgisayar doğru oynarsa asla kaybetmez. Ancak, daha büyük arama alanlarına sahip oyunlar için (örneğin dört taş) minimax tam çözüm üretemez. Bu tür oyunlarda, minimax algoritması belirli bir derinliğe kadar çalıştırılır ve ardından bir tahmin fonksiyonu (heuristic) kullanılarak tahtanın durumu değerlendirilir.

Bu tür teknikler yapay zeka alanında incelenir. Yapay zeka, arama algoritmalarını kullanarak çok büyük arama alanlarında etkili kararlar verebilmeyi amaçlar. Bu metinde, 12. bölümde minimax algoritmasının dört taş oyununa uygulanmasını içeren bazı sezgisel arama algoritmalarına değinilmektedir.

**4.10 Bağlı Listeler**

Diziler farklı şekillerde organize edilebilir. PyList dizisi rastgele erişilebilen bir listedir. Bu, listedeki herhangi bir öğeye O(1) zaman karmaşıklığında erişilebileceği veya değer saklanabileceği anlamına gelir. Amortisman analizine göre bir öğe eklemek O(1) zaman alırken, belirli bir konuma öğe eklemek O(n) zaman alır. Burada n, ekleme noktasından sonraki öğelerin sayısını ifade eder.

Eğer bir programcı listenin başına çok sayıda öğe eklemek istiyorsa, farklı bir veri yapısı daha uygun olabilir. **Bağlı liste (linked list)**, her öğenin ayrı bir düğümde (node) saklandığı bir liste organizasyonudur. Bağlı listeler, zincir halkalarına benzer. Her halka (bağlantı), zincirdeki bir sonraki halkaya işaret eden bir referans ile bağlıdır. Bağlı listeyle çalışırken, zincirdeki her bağlantıya **düğüm (Node)** denir.

Her düğüm iki parçadan oluşur: **bir öğe (item)**, yani düğüme ait veri ve **bağlantı (next)**, yani bağlı listedeki bir sonraki düğüme işaret eden referans. **Bölüm 4.10.1**'de, bağlı listedeki düğümleri oluşturmak için kullanılabilecek **Node** sınıfı tanımlanmaktadır.

**4.10.1 Node Sınıfı**

1. **class** **Node**:
2. **def** \_\_init\_\_(self,item,next=None):
3. self.item = item
4. self.next = next

5

1. **def** getItem(self):
2. **return** self.item

8

1. **def** getNext(self):
2. **return** self.next

11

1. **def** setItem(self, item):
2. self.item = item

14

1. **def** setNext(self,next):
2. self.next = next

Node sınıfında iki temel bilgi bulunur: **item**, listedeki bir değere referans veren öğedir ve **next**, dizideki bir sonraki düğüme işaret eden referanstır. **Şekil 4.10**, üç elemanın eklendiği bir bağlı listeyi göstermektedir. Bu şekilde dört düğüm bulunmaktadır. İlk düğüm bir **dummy** düğümüdür. Bağlı listenin başında fazladan bir dummy düğümünün bulunması, özel durumları ele alma gereksinimini büyük ölçüde azaltır. **Boş bir bağlı liste bile bir dummy düğümüne sahip olur.**

Bağlı listedeki düğümler, iki bölüme ayrılmış yuvarlak dikdörtgenler olarak gösterilir. Düğümün **sol yarısı**, listedeki öğeye veya değere referans verirken, **sağ yarısı** dizideki bir sonraki düğüme veya eğer son düğümse **None** değerine işaret eder.

Şekil 4.10, üç öğeden oluşan bir bağlı listeyi gösterir. Listeyi gerektiğinde dolaşabilmek için ilk düğüme referans tutulur. Listenin son düğümüne yapılan referans, yeni bir öğeyi **O(1)** zaman karmaşıklığında eklemeyi mümkün kılar. Ayrıca, öğe sayısını takip ederek, biri liste boyutunu sorguladığında tekrar sayma gereksinimini ortadan kaldırır.

**Şekil 4.11’deki tablo**, bu bölümde tanıtılan **LinkedList** veri tipi için çeşitli liste işlemlerinin hesaplama karmaşıklığını içermektedir.

diyagram, ekran görüntüsü, çizgi, tasarım içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

**Fig.4.10** A Sample LinkedList Object

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Complexity | Usage | Method |
| List creation | O(len(y)) | x = LinkedList(y) | calls \_\_init\_\_(y) |
| indexed get | O(n) | a = x[i] | x.\_\_getitem\_\_(i) |
| indexed set | O(n) | x[i] = a | x.\_\_setitem\_\_(i,a) |
| concatenate | O(n) | z = x + y | z = x.\_\_add\_\_(y) |
| append | O(1) | x.append(a) | x.append(a) |
| insert | O(n) | x.insert(i,e) | x.insert(i,e)) |
| delete | O(n) | del x[i] | x.\_\_delitem\_\_(i) |
| equality | O(n) | x == y | x.\_\_eq\_\_(y) |
| iterate | O(n) | for a in x: | x.\_\_iter\_\_() |
| length | O(1) | len(x) | x.\_\_len\_\_() |
| membership | O(n) | a in x | x.\_\_contains\_\_(a) |
| sort | N/A | N/A | N/A |
|  |  |  |  |

**Fig.4.11** Complexity of LinkedList Operations

Şekil 4.11'de, **LinkedList** veri tipi üzerindeki çeşitli işlemlerin hesaplama karmaşıklıkları gösterilmektedir. Bu işlemler, birçok açıdan **Python list** veri tipiyle benzer görünebilir. Ancak, bazı önemli farklar vardır.

Bağlı liste (LinkedList) kullanımı, özellikle büyük veri yapılarını yönetirken veya listenin başına ve ortasına ekleme işlemlerini sıkça yaparken avantaj sağlar. İzleyen bölümlerde, bazı temel işlemlerin nasıl uygulandığı açıklanacak ve **Python list** veri tipi ile olan farklara dikkat çekilecektir.

4.10.2 The LinkedList Constructor

1 **class** **LinkedList**:

2

3 # This class is used internally by the LinkedList class. It is 4 # invisible from outside this class due to the two underscores

1. # that precede the class name. Python mangles names so that they
2. # are not recognizable outside the class when two underscores 7 # precede a name but aren’t followed by two underscores at the 8 # end of the name (i.e. an operator name).
3. **class** **\_\_Node**:
4. **def** \_\_init\_\_(self,item,next=None):
5. self.item = item
6. self.next = next

13

1. **def** getItem(self):
2. **return** self.item

16

1. **def** getNext(self):
2. **return** self.next

19

1. **def** setItem(self, item):
2. self.item = item

22

1. **def** setNext(self,next):
2. self.next = next

25

1. **def** \_\_init\_\_(self,contents=[]):
2. # Here we keep a reference to the first node in the linked list
3. # and the last item in the linked list. They both point to a 29 # dummy node to begin with. This dummy node will always be in
4. # the first position in the list and will never contain an item.
5. # Its purpose is to eliminate special cases in the code below.
6. self.first = LinkedList.\_\_Node(None,None)
7. self.last = self.first
8. self.numItems = 0

35

36 **for** e **in** contents: 37 self.append(e)

Bir **LinkedList** nesnesi oluşturmak, bir **list** nesnesi oluşturmakla aynı zaman karmaşıklığına sahiptir. Eğer boş bir liste oluşturulursa, işlem **O(1)** sürede tamamlanır. Ancak, eğer bir liste kopyalanıyorsa, bu işlem **O(n)** zaman alır.

Bir **LinkedList** nesnesi, bağlı listenin hem başına hem de sonuna referans içerir. Listenin başına işaret eden referans, bir **dummy** düğümüne yönlendirilir. **Dummy** düğümü kullanmak, eğer liste boş olsaydı ortaya çıkabilecek birçok özel durumu ortadan kaldırır.

**Node** sınıfı, **LinkedList** sınıfının içine tanımlanarak gizlenir. **Node** adının önüne iki alt çizgi (\_\_Node) eklenmesi, bu sınıfın **LinkedList** sınıfı dışındaki kodlar tarafından erişilememesini sağlar. Bu yaklaşım, yalnızca **LinkedList** sınıfının **Node** nesneleri hakkında bilgi sahibi olmasını sağlar.

Başlangıçta, **first** ve **last** referansları **dummy** düğümüne işaret eder. Eğer yapıcı metoda bir liste geçirilirse, bu öğeler **append** metodu kullanılarak **LinkedList** nesnesine eklenir.

**4.10.3 LinkedList Get ve Set Metotları**

1. **def** \_\_getitem\_\_(self,index):
2. **if** index >= 0 **and** index < self.numItems:
3. cursor = self.first.getNext()4 **for** i **in** range(index):

5 cursor = cursor.getNext()

6

7 **return** cursor.getItem()

8

9 **raise** IndexError("LinkedList index out of range")

10

1. **def** \_\_setitem\_\_(self,index,val):
2. **if** index >= 0 **and** index < self.numItems:
3. cursor = self.first.getNext()14 **for** i **in** range(index):

15 cursor = cursor.getNext()

16

1. cursor.setItem(val)
2. **return**

19

20 **raise** IndexError("LinkedList assignment index out of range")

Dizinlenmiş alma ve ayarlama işlemleri için uygulamalar, büyük ölçüde bağlantılı bir listede gezinme örneği olarak 4.10.4. Bölümde yer almaktadır. Bunların pratik değeri çok azdır. Bir listeye rastgele erişim isteniyorsa, liste sınıfı kullanılmalıdır. Bağlantılı listelere rastgele erişilemez. Listedeki belirli bir konuma erişmek için veri türünde doğrusal arama gerektirirler. Bu işlemlerin her biri, n dizinin değeri olmak üzere O(n)'dir.

**4.10.4 Bağlantılı Liste Birleştirme**

**def** \_\_add\_\_(self,other):

**if** type(self) != type(other):

1. **raise** TypeError("Concatenate undefined for " + \
2. str(type(self)) + " + " + str(type(other)))

5

6 result = LinkedList()

7

8 cursor = self.first.getNext()

9

1. **while** cursor != None:
2. result.append(cursor.getItem())
3. cursor = cursor.getNext()

13

14 cursor = other.first.getNext()

15

1. **while** cursor != None:
2. result.append(cursor.getItem())
3. cursor = cursor.getNext()

19

20 **return** result

Birleştirme, iki orijinal listeden oluşan yeni bir liste döndüren bir erişim yöntemidir. İşlem, Şekil 4.1'de sunulan PyList veri türünde olduğu gibi, bağlantılı listeler için bir kez daha O(n)'dir. Bu birleştirme kodunda, iki listenin düğümleri arasında adım adım ilerlemek için imleç adı verilen bir değişken kullanılır. Bu, bağlantılı listede adım adım ilerlemenin yaygın yöntemidir. İmleci bağlantılı listenin ilk öğesine ayarlayın. Ardından, imleç sona ulaştığında (yani özel değer Yok) sona eren bir while döngüsü kullanın. While döngüsünün her seferinde, imleç sıradaki bir sonraki düğüme ayarlanarak ilerletilir. Kodda, iki liste birleştirilirken her iki listedeki kukla düğümün (yani self ve diğer) atlandığını fark edin. Yeni listedeki kukla düğüm, kurucu çağrıldığında oluşturulmuştur.

**4.10.5 LinkedList Append**

1. **def** append(self,item):
2. node = LinkedList.\_\_Node(item)
3. self.last.setNext(node)
4. self.last = node
5. self.numItems += 1

4.10.6. Bölümdeki kod, bir LinkedList'in bir listeye göre küçük bir avantajını ilk kez gördüğümüz yerdir. Ekleme işleminin karmaşıklığı LinkedList'ler için O(1) iken, listelerde karmaşıklık amortize edilmiş bir O(1) karmaşıklığıydı. Bir LinkedList ile her ekleme her zaman aynı süreyi alacaktır. Bir LinkedList ayrıca olması gerekenden asla daha büyük değildir. Ancak, LinkedList'ler rastgele erişilebilir bir listenin yaklaşık iki katı kadar yer kaplar, çünkü hem öğeye referans hem de listedeki sonraki düğüme referans için yer olması gerekir.

Ekleme yönteminin kodu oldukça basittir. self.last referansı, yeni düğümü koymak istediğimiz yerden hemen önceki düğümü işaret ettiğinden, sadece yeni bir düğüm oluştururuz ve sonuncusunu onu işaret ederiz. Ardından yeni düğümü yeni self.last düğümü yaparız ve öğe sayısını 1 artırırız.

**4.10.6 LinkedList Insert**

1. **def** insert(self,index,item):
2. cursor = self.first

3

1. **if** index < self.numItems:
2. **for** i **in** range(index):
3. cursor = cursor.getNext()

7

1. node = LinkedList.\_\_Node(item, cursor.getNext())
2. cursor.setNext(node)
3. self.numItems += 111

**else**:

12 self.append(item)

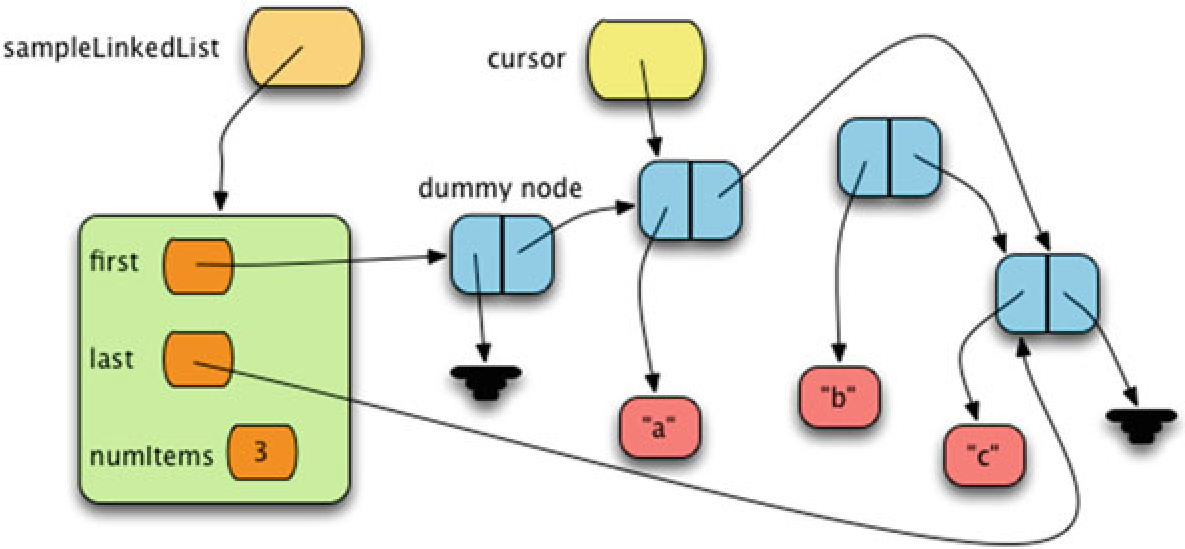
Ekleme işlemi, bir listeye ekleme ile aynı karmaşıklığa sahip olsa da, bağlantılı listeler için oldukça farklıdır. Bir listeye ekleme, n listenin ekleme noktasından sonraki eleman sayısı olmak üzere O(n) işlemidir, çünkü yeni öğe için yer açmak için hepsinin aşağı kaydırılması gerekir. Bir LinkedList ile çalışırken, ekleme noktasından önce görünen eleman sayısı n'dir çünkü doğru ekleme noktasını aramamız gerekir.

Bu, bir listenin başına ekleme O(n) işlemiyken, bir LinkedList'in başına ekleme O(1) işlemi olduğu anlamına gelir. Bir listenin başına yakın çok sayıda ekleme yapmanız gerekiyorsa, rastgele erişilebilir bir liste yerine bağlantılı bir liste kullanmak daha iyi olabilir.

4.10.7 Diğer Bağlantılı Liste İşlemleri

Diğer bağlantılı liste işlemleri okuyucuya alıştırma olarak bırakılmıştır. Çoğu durumda, bağlantılı bir liste ile çalışmanın anahtarı, çalışmak istediğiniz konumdan önceki düğüme bir referans elde etmektir. Örneğin, bir bağlantılı listeden bir düğümü silmek için, önceki düğümün sonraki alanını silmek istediğiniz düğümü izleyen düğümü işaret ettirmeniz yeterlidir. Şekil 4.10'daki örnek bağlantılı listeyi göz önünde bulundurun. Bu listeden ikinci öğeyi (yani "b") silmek için, "b"ye referans içeren düğümü kaldırmak istiyoruz. Bunu yapmak için, "b"ye referans veren düğümden önceki düğümü bulmak için bir imleç kullanabiliriz. Bulunduktan sonra, imlecin sonraki alanı, Şekil 4.12'de gösterildiği gibi "b"den sonraki düğümü işaret edecek şekilde yapılabilir.

İmlecin düğümünün sonraki işaretçisini "b"den sonraki düğümü işaret edecek şekilde değiştirmek, "b"yi içeren düğümün listeden düşmesine neden olur. "b"yi içeren düğüme ve aslında "b"nin kendisine başka referans olmadığı için, iki nesne de çöp olarak toplanır.



**Fig.4.12** Deleting a Node from a Linked List

Son olarak, sıralama işlemi bağlantılı listelerde uygulanamaz. Verimli sıralama algoritmaları, bir listeye rastgele erişim gerektirir. O(n²) algoritması olan ekleme sıralaması çalışır, ancak oldukça verimsiz olurdu. Sıralama gerekliyse, bağlantılı listeyi rastgele erişilebilir bir listeye kopyalamak, sıralamak ve ardından sıralanmış listeden yeni bir sıralanmış bağlantılı liste oluşturmak çok daha verimli olacaktır.

4.11 Yığınlar ve Kuyruklar Bilgisayar programlamada çok yaygın olan iki sıralı veri yapısı daha vardır. Yığın, erişimin yalnızca dizinin bir ucunda olduğu bir veri yapısıdır. Yeni değerler diziye eklemek için yığına gönderilir ve diziden çıkarmak için yığından çıkarılır. Bölüm 3'te açıklanan çalışma zamanı yığını, yığının böyle bir örneğiydi. Yığınlar, bilgisayar bilimindeki birçok algoritmada kullanılır. Yığınlar, sayısal ifadeleri değerlendirmek için kullanılabilir. Bilgileri ayrıştırırken kullanışlıdırlar. Programlarda ve ifadelerde parantezleri eşleştirmek için kullanılabilirler. Bir yığındaki işlemler Şekil 4.13'te verilmiştir.

Yığınlara Son Giren/İlk Çıkan veya LIFO veri yapıları denir. Gönderilen son öğe, çıkarılan ilk öğedir. Bir Yığın sınıfı en az birkaç farklı şekilde uygulanabilir.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Complexity | Usage | Description |
| Stack Creation | O(1) | s=Stack() | calls the constructor |
| pop | O(1) | a=s.pop() | returns the last item pushed and removes it from s |
| push | O(1) | s.push(a) | pushes the item, a, on the stack, s |
| top | O(1) | a=s.top() | returns the top item without popping s |
| isEmpty | O(1) | s.isEmpty() | returns True if s has no pushed items |

Bu tabloda özetlenen hesaplama karmaşıklıklarına ulaşmanın birkaç yolu vardır. Bir liste veya bağlantılı liste yeterli olacaktır. 4.11.2. Bölümdeki kod, bir listeyle bir yığın uygulamasının bir örneğidir. Uygulama oldukça basittir. 4.11.2. Bölümdeki ana program, kodun doğru çalıştığından emin olmak için birkaç testle yığın veri türünü test eder.

**4.11.1 The Stack Class Code**

1. **class** **Stack**:
2. **def** \_\_init\_\_(self):
3. self.items = []

4

1. **def** pop(self):
2. **if** self.isEmpty():
3. **raise** RuntimeError("Attempt to pop an empty stack")

8

1. topIdx = len(self.items)-1
2. item = self.items[topIdx]
3. **del** self.items[topIdx]
4. **return** item

13

1. **def** push(self,item):
2. self.items.append(item)

16

1. **def** top(self):
2. **if** self.isEmpty():
3. **raise** RuntimeError("Attempt to get top of empty stack")

20

1. topIdx = len(self.items)-1
2. **return** self.items[topIdx]

23

1. **def** isEmpty(self):
2. **return** len(self.items) == 0

26

1. **def** main():
2. s = Stack()
3. lst = list(range(10))
4. lst2 = []

31

1. **for** k **in** lst:
2. s.push(k)

34

1. **if** s.top() == 9:
2. **print**("Test 1 Passed") 37

**else**:

38 **print**("Test 1 Failed")

39

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

1. lst2.append(s.pop())

42

43 lst2.reverse()

44

**if** lst2 != lst:

1. **print**("Test 2 Failed") 47

**else**:

48 **print**("Test 2 Passed")

49

**try**:

1. s.pop()
2. **print**("Test 3 Failed")

53

54 **except** RuntimeError: 55 **print**("Test 3 Passed") 56 **except**:

57 **print**("Test 3 Failed")

58

**try**:

s.top()

**print**("Test 4 Failed")

62

63 **except** RuntimeError: 64 **print**("Test 4 Passed") 65 **except**:

66 **print**("Test 4 Failed")

67

1. **if** \_\_name\_\_=="\_\_main\_\_":

main()

"stack.py" adlı bir dosyaya kaydedilirse, bu kod diğer modüllere içe aktarılabilir. Bu modül kendi başına çalıştırıldığında, test ana işlevi yürütülür. Bu modül başka bir programa içe aktarıldığında, \_\_name\_\_ değişkeni "**main**"e eşit olmayacağından ana işlev yürütülmez.

Bir kuyruk, LIFO veri yapısı olmak yerine FIFO veya İlk Giren/İlk Çıkan veri yapıları olması dışında birçok yönden bir yığına benzer. Gönderilen ilk öğe, çıkarılan ilk öğedir. Bir kuyrukla çalışırken, bir öğeyi göndermek yerine kuyruğa eklemekten bahsederiz. Bir öğeyi kuyruktan çıkarırken, bir yığından yaptığımız gibi onu çıkarmak yerine kuyruktan çıkarmaktan bahsederiz. Şekil 4.14'teki tablo, kuyruk işlemlerinin ve karmaşıklıklarının ayrıntılarını sağlar.

Bu tabloda verilen karmaşıklıklarla bir kuyruk uygulamak, yığını uygulamaktan biraz daha karmaşıktır. Bu karmaşıklıklarla bir kuyruk uygulamak için, bir dizinin bir ucuna ekleyebilmeli ve dizinin diğer ucundan O(1) süresinde kaldırabilmeliyiz. Bu, bağlantılı bir listenin kullanımını düşündürür. Elbette, bağlantılı bir liste istenen karmaşıklıkları elde etmek için işe yarayacaktır. Ancak, kuyruktan çıkarma işlemi için amortize edilmiş O(1) karmaşıklığını kabul etmeye istekli olursak, yine de bir liste kullanabiliriz. Bu Kuyruk sınıfı kodu, bir listeyle bir kuyruk uygular ve kuyruktan çıkarma işlemi için amortize edilmiş O(1) karmaşıklığı elde eder

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Complexity | Usage | Description |
| Queue Creation | O(1) | q=Queue() | calls the constructor |
| dequeue | O(1) | a=q.dequeue() | returns the first item enqueued and removes it from q |
| enqueue | O(1) | q.enqueue(a) | enqueues the item, a, on the queue, q |
| front | O(1) | a=q.front() | returns the front item without dequeueing the item |
| isEmpty | O(1) | q.isEmpty() | returns True if q has not enqueued items |