Структуры и алгоритмы

Линейные структуры данных

Типы данных в Python

В Python типы данных можно разделить на встроенные в интерпретатор (built-in) и не встроенные, которые можно использовать при импортировании соответствующих модулей.

Встроенные типы данных

1. *None* (неопределенное значение переменной)
2. Логические переменные (*Boolean Type*)
3. Числа (*Numeric Type*)
   - *int* – целое число
   - *float* – число с плавающей точкой
   - *complex* – комплексное число

Встроенные типы данных

4. Списки (*Sequence Type*)
   - *list* – список
   - *tuple* – кортеж
   - *range* – диапазон
5. Строки (*Text Sequence Type*)
   - *str*
6. Бинарные списки (*Binary Sequence Types*)
   - *bytes* – байты
   - *bytearray* – массивы байт
   - *memoryview* – специальные объекты для доступа к внутренним данным объекта через protocol buffer

Встроенные типы данных

7. Множества (*Set Types*)
   - *set* – множество
   - *frozenset* – неизменяемое множество
8. Словари (*Mapping Types*)
   - *dict* – словарь

Изменяемые и неизменяемые типы данных

К **неизменяемым** (*immutable*) типам относятся: целые числа (*int*), числа с плавающей точкой (*float*), комплексные числа (*complex*), логические переменные (*bool*), кортежи (*tuple*), строки (*str*) и неизменяемые множества (*frozen set*).

См. пример 1

К **изменяемым** (*mutable*) типам относятся: списки (*list*), множества (*set*), словари (*dict*).

См. пример 2

Абстрактные структуры данных

Абстрактные структуры данных предназначены для удобного хранения и доступа к информации. Они предоставляют удобный интерфейс для типичных операций с хранимыми объектами, скрывая детали реализации от пользователя.

Абстрактные структуры данных иногда делят на две части:

1. интерфейс, набор операций над объектами, который называют АТД (*абстрактный тип данных*);
2. реализация.

Абстрактные структуры данных

Например, операции с числами.

Языки программирования высокого уровня предоставляют удобный интерфейс для чисел: операции +, *, = и т.п.

При этом скрывается сама реализация этих операций, конкретные машинные команды.

Линейные структуры данных

Элементы *линейных структур* упорядочены в зависимости от того, как они добавлялись или удалялись. Единожды добавленный, элемент остаётся на одном и том же месте по отношению к остальным, пришедшим раньше и позже него.

О линейных структурах можно думать, как об имеющих два конца («левый» и «правый», «голова» и «хвост», «вершина» и «основание»).

Отличие одной линейной структуры от другой в способе, каким добавляются или удаляются их элементы, в частности - в месте, где происходят данные операции.

Например, добавление только в хвост. Или удаление элемента с любого края.

Стек

Стек - это упорядоченная коллекция элементов, где добавление нового или удаление существующего всегда происходит только на «вершине». Соответствует принципу **LIFO, last-in, first-out** (англ. *«последним пришёл — первым вышел»*).

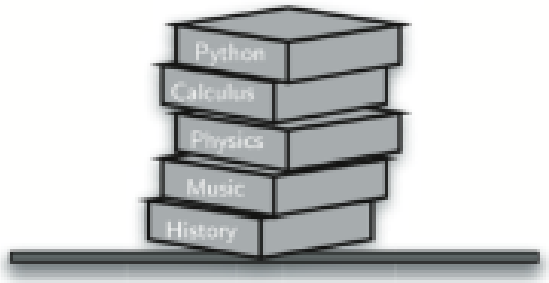
Он предоставляет упорядочение по времени нахождения в коллекции. Более новые элементы расположены ближе к вершине, более старые - ближе к основанию.

Стеки фундаментально важны, поскольку их можно использовать для реверсирования порядка элементов.

Стек

Одна из наиболее часто используемых идей, связанных со стеком, пришла из простого наблюдения за тем, как добавляются и удаляются его элементы.

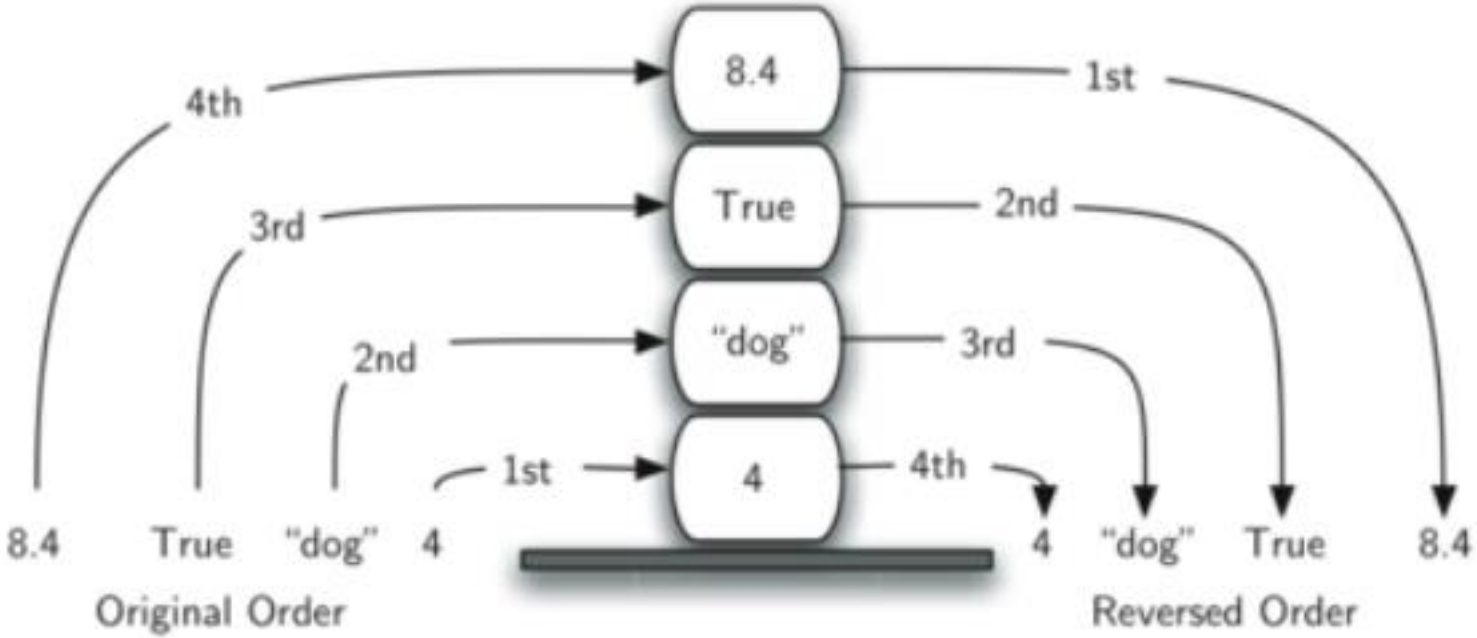
Предположим, вы начинаете с чистого стола. Теперь кладите книги по одной за раз друг поверх друга. Вы конструируете стек. При этом когда вы начнёте их удалять очерёдность, в которой это будет происходить, в точности противоположна тому, как они укладывались.



Стек

Стеки фундаментально важны, поскольку их можно использовать для реверсирования порядка элементов.

Последовательность вставок противоположна последовательности удалений.



Стек

Рассматривая это реверсивное свойство, вы, возможно, подумаете о примерах стека, имеющих место при работе с компьютером.

Например, каждый веб-браузер имеет кнопку “Назад”. Когда вы перемещаетесь от одной веб-страницы к другой, они помещаются в стек (точнее, в стек помещаются их URL’ы). Текущая страница, которую вы просматриваете, находится на вершине, а самая первая из просмотренных - в основании. Если вы нажмёте кнопку “Назад”, то начнёте двигаться по страницам в обратном порядке.

Абстрактный тип данных «стек»

Абстрактный тип данных для стека определяется следующими операциями:

1. **Stack()** создаёт новый пустой стек. Параметры не нужны, возвращает пустой стек.
2. **push(item)** добавляет новый элемент на вершину стека. В качестве параметра выступает элемент; функция ничего не возвращает.
3. **pop()** удаляет верхний элемент из стека. Параметры не требуются, функция возвращает элемент. Стек изменяется.
4. **peek()** возвращает верхний элемент стека, но не удаляет его. Параметры не требуются, стек не модифицируется.
5. **isEmpty()** проверяет стек на пустоту. Параметры не требуются, возвращает булево значение.
6. **size()** возвращает количество элементов в стеке. Параметры не требуются, тип результата - целое число.

Абстрактный тип данных «стек»

Операция над стеком	Содержание стека	Возвращаемое значение
`s.isEmpty()`	`[]`	`True`
`s.push(4)`	`[4]`	
`s.push('dog')`	`[4,'dog']`	
`s.peek()`	`[4,'dog']`	`'dog'`
`s.push(True)`	`[4,'dog',True]`	
`s.size()`	`[4,'dog',True]`	`3`
`s.isEmpty()`	`[4,'dog',True]`	`False`
`s.push(8.4)`	`[4,'dog',True,8.4]`	
`s.pop()`	`[4,'dog',True]`	`8.4`
`s.pop()`	`[4,'dog']`	`True`
`s.size()`	`[4,'dog']`	`2`

Реализация стека на Python

Создадим новый класс. Стековые операции воплотятся в его методах. А за основу возьмем стандартный список.

```
1  class Stack:
2      def __init__(self):
3          self.items = []
4
5      def isEmpty(self):
6          return self.items == []
7
8      def push(self, item):
9          self.items.append(item)
10
11     def pop(self):
12         return self.items.pop()
13
14     def peek(self):
15         return self.items[len(self.items)-1]
16
17     def size(self):
18         return len(self.items)
19
```

См. пример 3

Реализация стека на Python

Важно отметить, что мы можем выбрать реализацию стека через список, где вершиной считается первый, а не последний элемент.

В этом случае предыдущие методы append и pop работать не будут.

Мы должны будем явно использовать pop и insert для позиции с индексом 0 (первый элемент в списке).

Реализация стека на Python

Эта возможность изменять физическое воплощение абстрактного типа данных при поддержке логических характеристик - пример того, как работает абстракция.

Хотя обе реализации ведут себя аналогично и поддерживают единый интерфейс, но их производительность принципиально отличается.

Операции append и pop обе являются O(1). Это означает, что первая реализация будет выполнять добавление и выталкивание за постоянное время, независимо от количества элементов в стеке.

Производительность второго варианта страдает, поскольку и insert(0), и pop(0) для списка размером n составляет O(n).

Реализация стека на Python

Дана следующая последовательность стековых операций. Что будет на вершине стека, когда она завершится?

```
m = Stack()
m.push('x')
m.push('y')
m.pop()
m.push('z')
m.peek()
```

a) 'x'

b) 'y'

c) 'z'

d) Стек пуст

Реализация стека на Python

Дана следующая последовательность стековых операций. Что будет на вершине стека, когда она завершится?

```
m = Stack()
m.push('x')
m.push('y')
m.push('z')
while not m.isEmpty():
    m.pop()
    m.pop()
```

a) 'x'

b) стек пуст

c) возникнет ошибка

d) 'z'

Простые сбалансированные скобки

Сбалансированность скобок означает, что каждый открывающий символ имеет соответствующий ему закрывающий, и пары скобок правильно вложены друг в друга.

Корректно

```
()()()()
(((())))
(()((())()))
```

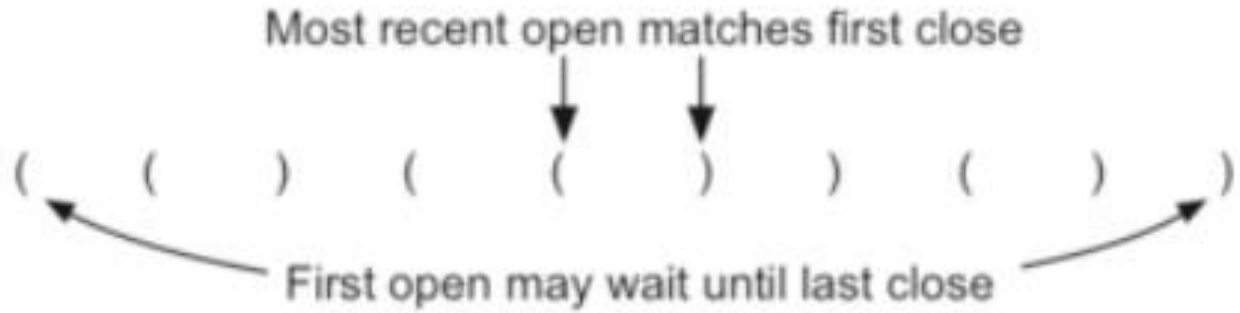
Не корректно

```
(((((((())
)))
()()(()
```

Простые сбалансированные скобки

Способность различать, какие скобки сбалансированы корректно, а какие нет - важная часть распознавания структур во многих языках программирования.

Таким образом, задача заключается в создании алгоритма, читающего строку из скобок слева направо и определяющего, являются ли они сбалансированными.



Простые сбалансированные скобки

Начиная с пустого стека, строка скобок обрабатывается слева направо. Если символ - открывающая скобка, то она кладётся в стек, как напоминание, что соответствующий закрывающий знак должен появиться позже. С другой стороны, если символ - закрывающая скобка, то из стека выталкивается верхний элемент.

До тех пор, пока будет происходить выталкивание для соотнесения каждого закрывающего символа, скобки будут сбалансированными. Если в какой-то момент в стеке не окажется открывающей скобки для связи с закрывающим символом, то строка является несбалансированной.

В конце, когда будут обработаны все символы, стек должен быть пуст.

См. пример 4

Общий случай сбалансированных скобок

Обобщённая проблема балансировки и вложенности различных типов открывающих и закрывающих символов возникает очень часто. Например, в Python квадратные скобки, [и], используются для списков, фигурные скобки, { и }, - для словарей, а круглые (и) – для кортежей и арифметических выражений. Можно сколько угодно перемешивать символы до тех пор, пока каждый из них поддерживает свои открывающие и закрывающие отношения.

Корректно

```
{ { ( [ ] [ ] ) } ( ) }

[ [ { { ( ( ) ) } } ] ]

[ ] [ ] [ ] ( ) { }
```

Не корректно

```
( [ ) ]

( ( ( ) ] ) )

[ { ( ) ]
```

См. пример 5

Очередь

Очередь – это упорядоченная коллекция элементов, в которой добавление новых происходит с одного конца, называемого «хвост очереди», а удаление существующих – с другого, «головы очереди». Соответствует принципу **FIFO, first-in first-out** (англ. *“первым пришёл - первым вышел”*).

Как только элемент добавляется в конец очереди, он начинает свой путь к её началу, ожидая удаления предыдущих.

Очередь

Простейший пример такой структуры данных – это обычная очередь, в которой все мы иногда стоим: в кинотеатре, перед кассой в бакалейной лавке, в закусочной (где мы, кстати, можем “выталкивать” поднос из стопки/стека).

Очереди ограничены тем, что имеют только один путь движения элементов. Для них не предусмотрены прыжки в середину или выход до того, как они дойдут до конца.

Абстрактный тип данных «очередь»

Абстрактный тип данных для очереди определяется следующими операциями:

1. **Queue()** создаёт новую пустую очередь. Не нуждается в параметрах, возвращает пустую очередь.
2. **enqueue(item)** добавляет новый элемент в конец очереди. Требует элемент в качестве параметра, ничего не возвращает.
3. **dequeue()** удаляет из очереди передний элемент. Не нуждается в параметрах, возвращает элемент. Очередь изменяется.
4. **isEmpty()** проверяет очередь на пустоту. Не нуждается в параметрах, возвращает булево значение.
5. **size()** возвращает количество элементов в очереди (целое число). Не нуждается в параметрах.

Абстрактный тип данных «очередь»

Оператор	Содержимое	Возвращаемое значение
`q.isEmpty()`	`[]`	`True`
`q.enqueue(4)`	`[4]`	
`q.enqueue('dog')`	`['dog',4]`	
`q.enqueue(True)`	`[True,'dog',4]`	
`q.size()`	`[True,'dog',4]`	`3`
`q.isEmpty()`	`[True,'dog',4]`	`False`
`q.enqueue(8.4)`	`[8.4,True,'dog',4]`	
`q.dequeue()`	`[8.4,True,'dog']`	`4`
`q.dequeue()`	`[8.4,True]`	`'dog'`
`q.size()`	`[8.4,True]`	`2`

Реализация очереди на Python

Создадим новый класс. Для построения внутреннего представления очереди используем список.

Нам надо определиться, какой конец списках считать головой, а какой - хвостом. Пусть последний элемент очереди находится на его нулевой позиции. Это позволяет использовать функцию **insert** для добавления новых элементов в конец очереди. Операция **pop** будет использоваться для удаления переднего элемента (последнего элемента в списке).

Это означает постановку в очередь за O(n), а извлечение - за O(1) времени.

См. пример 6

Реализация очереди на Python

Предположим, что у вас есть следующая последовательность операций с кодом. Какие элементы остались в очереди?

```
q = Queue()
q.enqueue('hello')
q.enqueue('dog')
q.enqueue(3)
q.dequeue()
```

a) 'hello', 'dog'

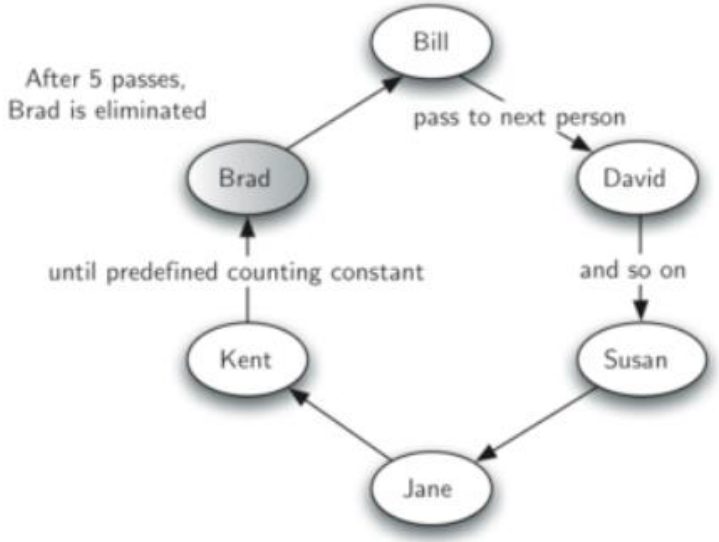
b) 'dog', 3

c) 'hello', 3

d) 'hello', 'dog', 3

Симуляция: Hot Potato

Детская игра Hot Potato. В этой игре дети выстраиваются в круг и перебрасывают предмет от соседа к соседу так быстро, как только могут. В некоторый момент игры действие останавливается, и ребёнок, у которого в руках остался предмет (картошка), выбывает из круга. Игра продолжается до тех пор, пока не останется единственный победитель.



Симуляция: Hot Potato

Мы реализуем общую симуляцию игры Hot Potato.

Наша программа будет получать на входе список имён и константу num, используемую для подсчёта.

Она будет возвращать имя последнего человека, оставшегося после повторяющегося отсчёта num.

Симуляция: Hot Potato

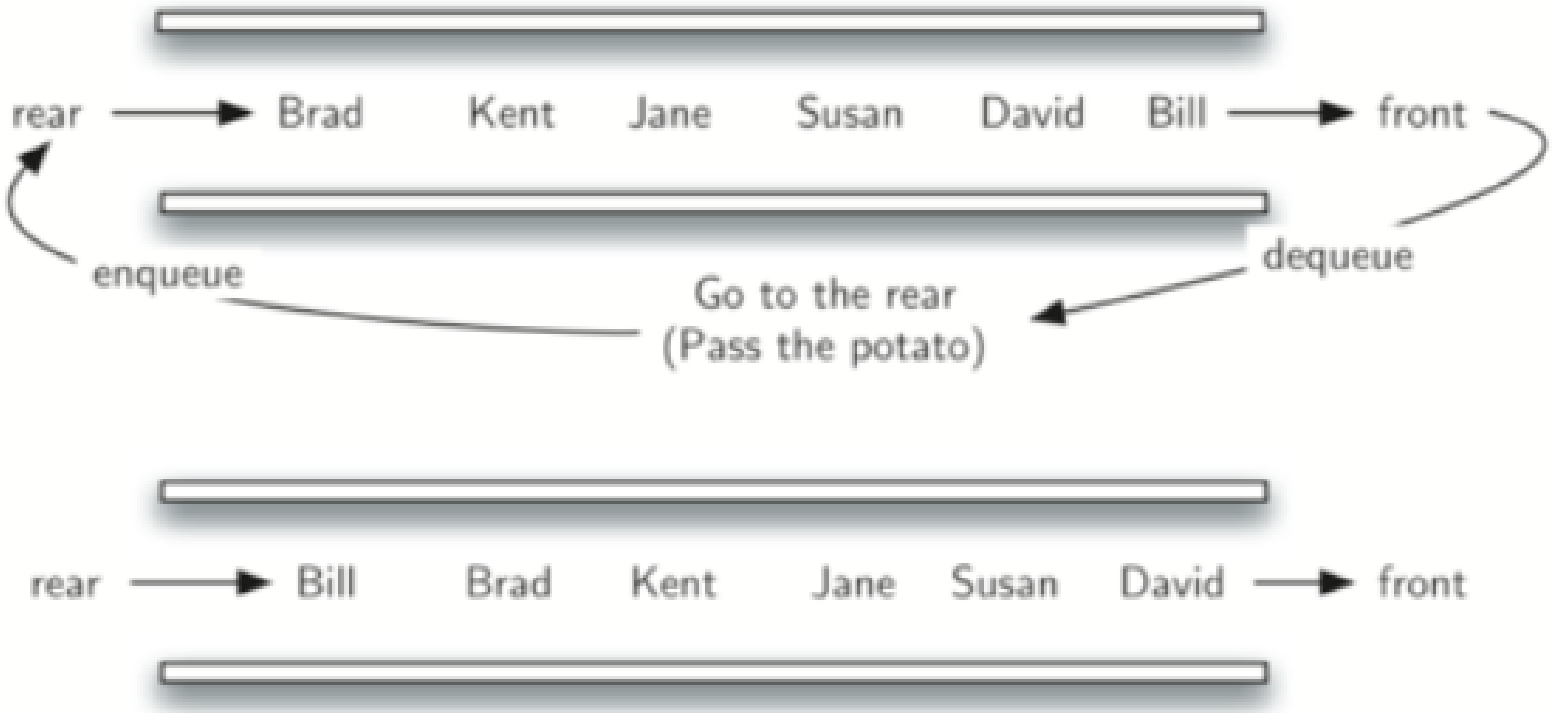
Для симуляции круга мы будем использовать очередь.

Предположим, что игрок, держащий картошку, - первый в очереди.

После переброса картошки мы просто извлечём его оттуда и тут же поставим обратно, но уже в хвост. Игрок будет ждать, пока все, кто перед ним, побудут первыми, а затем вернётся на это место снова.

После num операций извлечений/постановок участник, стоящий впереди, будет удалён окончательно, и цикл начнётся заново. Этот процесс будет продолжаться до тех пор, пока не останется всего одно имя.

Симуляция: Hot Potato



См. пример 7

Симуляция: Задания на печать

Рассмотрим поведение очереди на печать на принтере.

Студенты отправляют документы для печати на общем принтере, их документы помещаются в очередь, чтобы быть обработанными в манере “первым пришёл - первым обслужен”.

Способен ли принтер обработать заданное количество документов?

Симуляция: Задания на печать

В любой среднестатистический день в любой час с принтером работает порядка 10 студентов.

В течение этого времени они обычно печатают дважды, причём длина задания варьируется от одной до двадцати страниц.

Принтер в лаборатории стар и в черновом качестве способен обрабатывать всего 10 страниц в минуту. Можно переключить его на лучшее качество печати, но тогда производительность упадёт до 5 страниц/мин.

Какую скорость для страниц следует использовать?

Симуляция: Задания на печать

Мы можем определить это, построив симуляционную модель лаборатории.

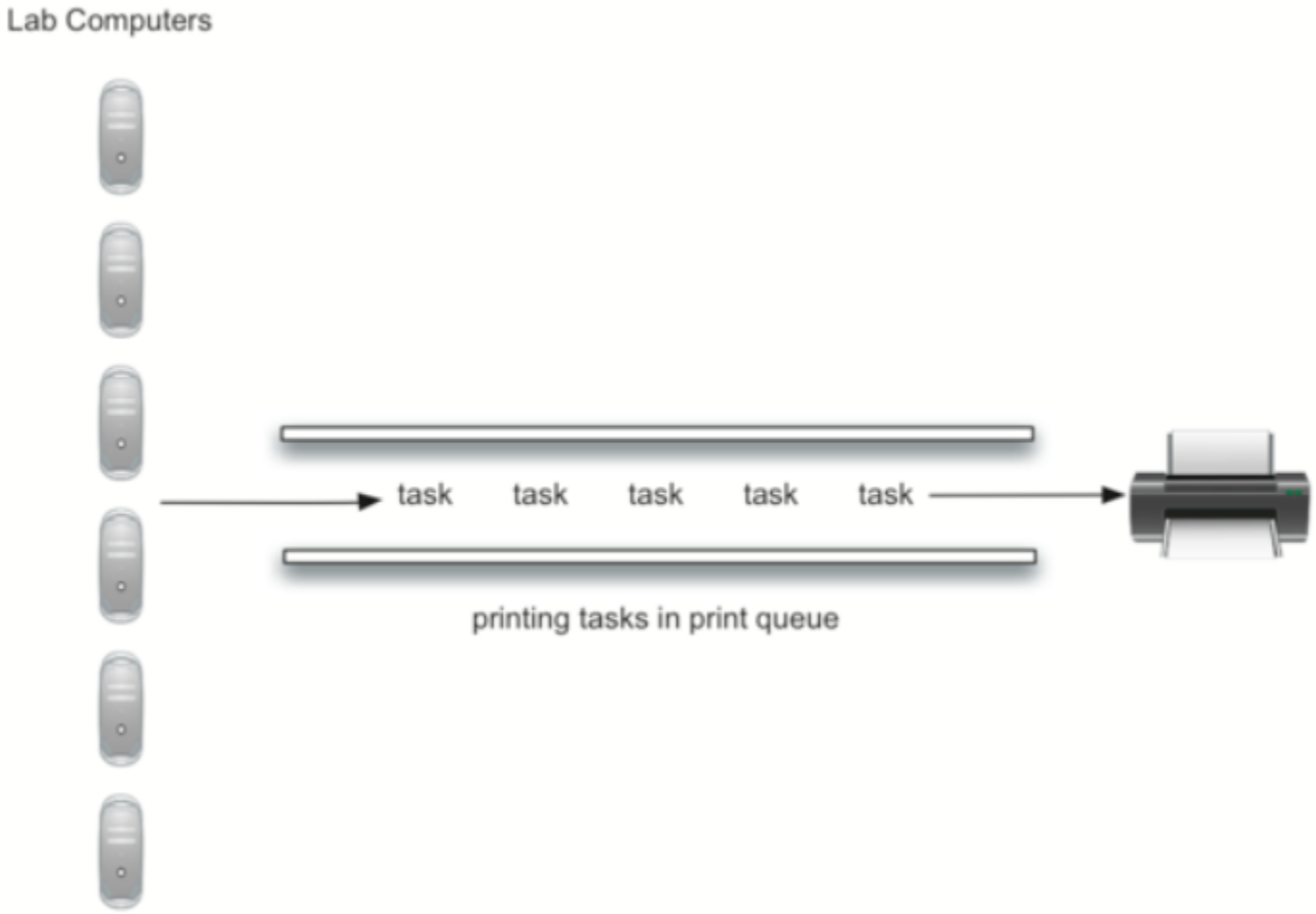
Нам потребуется создать представления студентов, заданий на печать и принтера.

Когда студенты подают документы на печать, мы будем добавлять их в список ожидания - очередь заданий на печать, прикреплённую к принтеру.

Когда принтер заканчивает очередное задание, он смотрит в очередь на предмет наличия оставшихся документов для обработки.

Интерес для нас представляет среднее время ожидания задачи в очереди.

Симуляция: Задания на печать



Симуляция: Задания на печать

Вот основная модель.

1. Создать очередь из заданий на печать. Каждое из них будет иметь отметку о времени постановки в очередь. В самом начале очередь пуста.
2. Для каждой секунды (`currentSecond`):
    - Создано ли новое задание на печать? Если да, то добавить его в очередь с `currentSecond` в качестве отметки времени.
    - Если принтер не занят и есть ожидающее задание, то
        - Удалить следующее задание из очереди на печать и передать его принтеру.
        - Извлечь отметку о времени из `currentSecond` чтобы вычислить время ожидания для данного задания.
        - Добавить время ожидания этой задачи в список для дальнейшей обработки.
        - Основываясь на количестве страниц в задании на печать, вычислить, сколько для него потребуется времени.
    - Если необходимо, принтер тратит одну секунду печати. Также он вычитает её из времени, необходимого для выполнения задачи.
    - Если задание завершено (другими словами, требуемое на его выполнение время равно нулю), то принтер более не занят.
3. После завершения симуляции вычисляется среднее время ожидания из сгенерированного списка времён ожидания.

См. пример 8

Дек

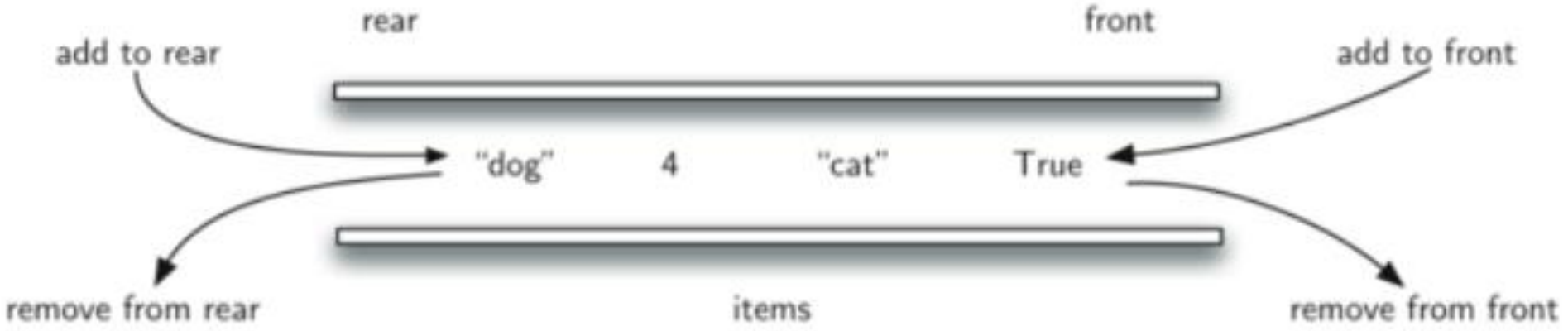
Дек, также называемый двусторонней очередью, - это упорядоченная коллекция элементов, подобная очереди. Он имеет два конца (голову и хвост), и его элементы остаются позиционированными.

Дек отличает нестрогая природа добавления и удаления его составляющих. Новые элементы могут быть добавлены как в голову, так и в хвост. Аналогично, существующие компоненты могут удаляться из обоих концов.

В каком-то смысле, этот гибрид линейной структуры объединяет все возможности стеков и очередей.

Дек

Важно отметить, что несмотря на обладание деком многих характеристик стеков и очередей, он не поддерживает LIFO или FIFO упорядочение, которые воплощаются в жизнь этими структурами данных. Только от вас зависит, какого типа операции добавления или удаления использовать.



Абстрактный тип данных «дек»

Абстрактный тип данных дека определяется следующими операциями:

1. **Deque()** создаёт новый пустой дек. Не нуждается в параметрах и возвращает пустой дек.
2. **addFront(item)** добавляет новый элемент в голову дека. Параметр (элемент) необходим; ничего не возвращает.
3. **addRear(item)** добавляет новый элемент в хвост дека. Параметр (элемент) необходим; ничего не возвращает.
4. **removeFront()** удаляет первый элемент из дека. Не нуждается в параметрах и возвращает элемент. Дек модифицируется.
5. **removeRear()** удаляет последний элемент из дека. Не нуждается в параметрах и возвращает элемент. Дек модифицируется.
6. **isEmpty()** проверяет дек на пустоту. Не нуждается в параметрах и возвращает булево значение.
7. **size()** возвращает количество элементов в деке. Не нуждается в параметрах и возвращает целое число.

Абстрактный тип данных «дек»

Операция	Содержимое дека	Возвращаемое значение
`d.isEmpty()`	`[]`	`True`
`d.addRear(4)`	`[4]`	
`d.addRear('dog')`	`['dog',4,]`	
`d.addFront('cat')`	`['dog',4,'cat']`	
`d.addFront(True)`	`['dog',4,'cat',True]`	
`d.size()`	`['dog',4,'cat',True]`	`4`
`d.isEmpty()`	`['dog',4,'cat',True]`	`False`
`d.addRear(8.4)`	`[8.4,'dog',4,'cat',True]`	
`d.removeRear()`	`['dog',4,'cat',True]`	`8.4`
`d.removeFront()`	`['dog',4,'cat']`	`True`

Реализация дека в Python

Создадим новый класс. С помощью списка построим детализацию этой структуры данных. Наша реализация будет предполагать, что хвост дека находится в нулевой позиции списка.

В removeFront мы используем метод pop для удаления последнего элемента из списка. Однако, в removeRear метод pop(0) должен удалять первый из них.

Также нам нужно использовать метод insert в addRear, поскольку append предполагает добавление нового элемента в конец списка.

См. пример 9

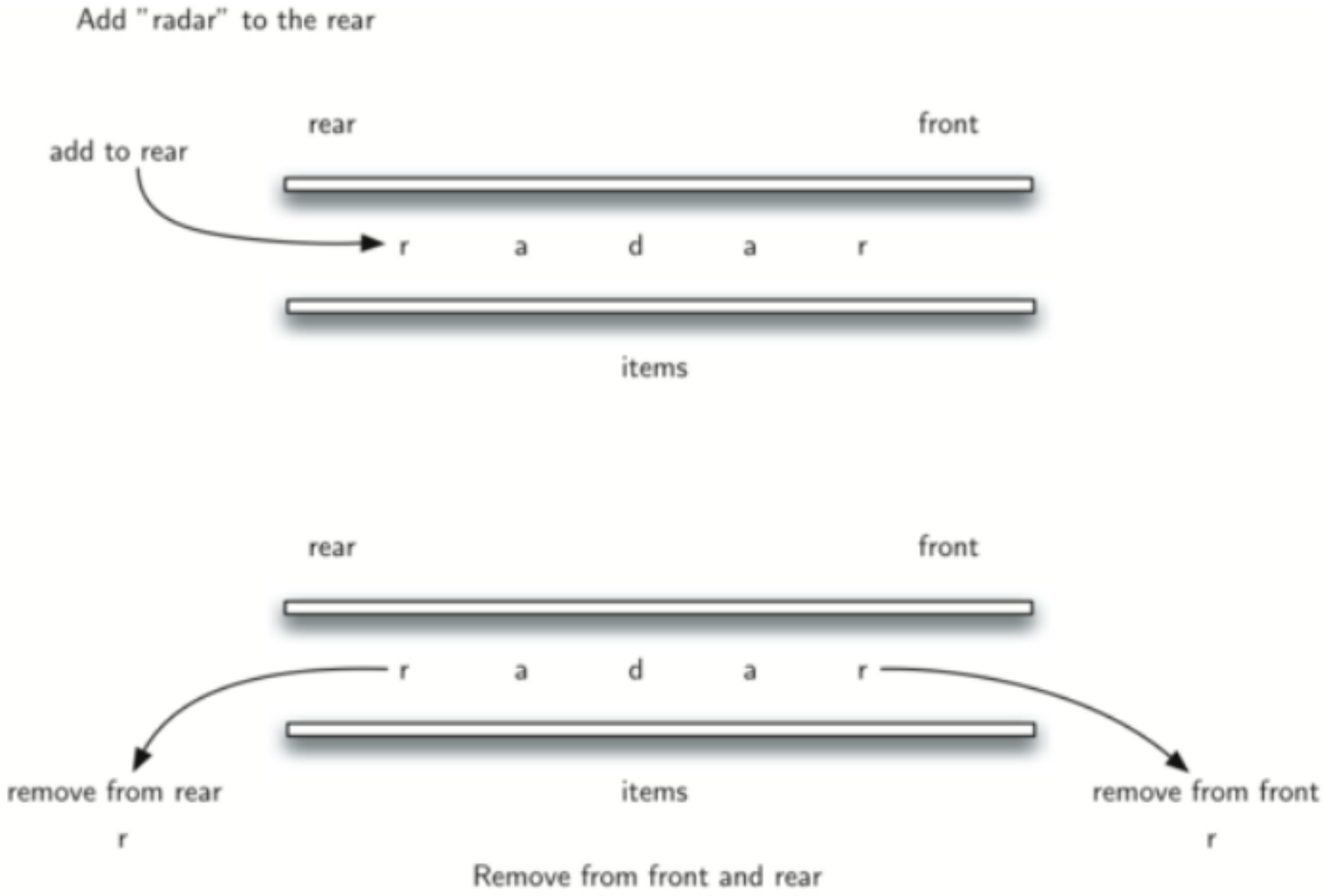
Проверка палиндрома

Палиндромом называют строку, которая одинаково читается справа налево и слева направо. Мы хотим создать алгоритм, принимающий на вход строку символов и проверяющий, является ли она палиндромом.

Для решения данной задачи используем дек в качестве хранилища строковых символов. Мы будем обрабатывать строку слева направо и добавлять каждый её элемент в хвост дека.

В этот момент он будет работать очень схоже с обычной очередью. Однако, теперь мы можем использовать дуальную функциональность дека. Его голова будет хранить первый символ строки, а хвост – последний.

Проверка палиндрома



Проверка палиндрома

Поскольку мы способны удалять оба элемента сразу, то можно производить сравнение и продолжать только в случае, если символы совпадают.

Если соответствие первого и последнего элементов будет сохраняться, то в конечном итоге мы придём или к отсутствию символов, или останемся с деком размером 1 - в зависимости от того, была ли длина исходной строки чётным или нечётным числом.

Но в обоих случаях входная последовательность будет палиндромом.

См. пример 10

Списки

Список - это коллекция элементов, каждый из которых хранится на соответствующей позиции по отношению к остальным. Точнее, список такого рода называется неупорядоченным списком.

Можно сделать вывод, что список имеет первый элемент, второй элемент, третий и так далее.

Мы также можем ссылаться на начало списка (первый элемент) или его конец (последний элемент).

Для простоты будем полагать, что списки не содержат дублирующихся данных.

Абстрактный тип данных «неупорядоченный список»

Структура неупорядоченного списка представляет из себя коллекцию элементов, каждый из которых находится на определённой позиции по отношению к остальным.

Некоторые из возможных операций над неупорядоченными списками представлены далее.

Абстрактный тип данных «неупорядоченный список»

1. **List()** создаёт новый пустой список. Не нуждается в параметрах и возвращает пустой список.
2. **add(item)** добавляет в список новый элемент. Требует значение в качестве аргумента, ничего не возвращает. Предполагает, что до этого элемент в списке отсутствовал.
3. **remove(item)** удаляет элемент из списка. Требует значение элемента и изменяет список. Предполагает первоначальное наличие элемента в списке.
4. **search(item)** ищет элемент в списке. Требует элемент и возвращает булево значение.
5. **isEmpty()** проверка списка на пустоту. Не нуждается в параметрах, возвращает булево значение.
6. **size()** возвращает количество элементов в списке. Не нуждается в параметрах, возвращает целое число.

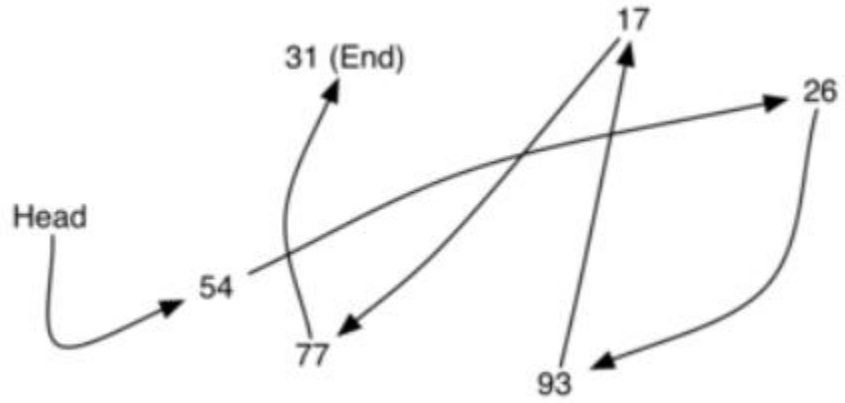
Абстрактный тип данных «неупорядоченный список»

7. **append(item)** добавляет новый элемент в конец списка. Требует значение в качестве аргумента, ничего не возвращает. Предполагает, что до сих пор такой элемент в списке отсутствовал.
8. **index(item)** возвращает позицию элемента в списке. Требует значение в качестве аргумента, возвращает его индекс. Предполагает, что элемент присутствует в списке.
9. **insert(pos,item)** вставляет новый элемент в заданную позицию pos списка. Требует элемент, ничего не возвращает. Предполагает, что до сих пор такой элемент в списке отсутствовал и существующий размер списка позволяет задать индекс pos.
10. **pop()** удаляет и возвращает последний элемент списка. Не требует аргументов, возвращает элемент. Предполагает, что в списке есть хотя бы один элемент.
11. **pop(pos)** удаляет и возвращает элемент из позиции pos. Требует позицию в качестве аргумента и возвращает элемент. Предполагает, что такой элемент присутствует в списке.

Реализация неупорядоченного списка: связанные списки

Для реализации неупорядоченного списка мы создадим то, что обычно называют **связанным списком**.

Мы должны быть уверены, что сможем поддерживать порядок взаимного расположения элементов. Однако, такое позиционирование не обязательно должно быть на смежных участках памяти.



Реализация неупорядоченного списка: связанные списки

Важно отметить, что положение первого элемента должно быть определено явно. Поскольку мы знаем, где он находится, то можем найти местоположение второго и так далее.

Внешнюю ссылку часто называют **головой** списка.

Аналогично, последнему элементу нужно знать, что за ним больше ничего нет.

Реализация неупорядоченного списка: связанные списки

Основным строительным блоком в реализации связанного списка является **узел**.

Каждый такой объект должен обладать как минимум двумя информационными составляющими.

Во-первых, узел должен содержать сам элемент списка. Мы назовём это **полем данных** узла.

Дополнительно он должен хранить ссылку на следующий узел.

См. пример 11

Абстрактный тип данных «упорядоченный список»

Если бы список целых, показанный выше, был упорядоченным (по возрастанию), то он записался как 17, 26, 31, 54, 77 и 93. 17 - наименьший элемент, поэтому он ставится на первую позицию, а 93 - наибольший, так что он занимает последнее место.

По структуре упорядоченный список представляет собой коллекцию элементов, каждый из которых занимает положение в зависимости от некой общей для всех характеристики.

Порядок упорядочения или возрастающий, или убывающий, и мы полагаем, что для элементов списка существует и определена операция сравнения.

Многие из операций для упорядоченного списка аналогичны методам неупорядоченного.

Реализация упорядоченного списка

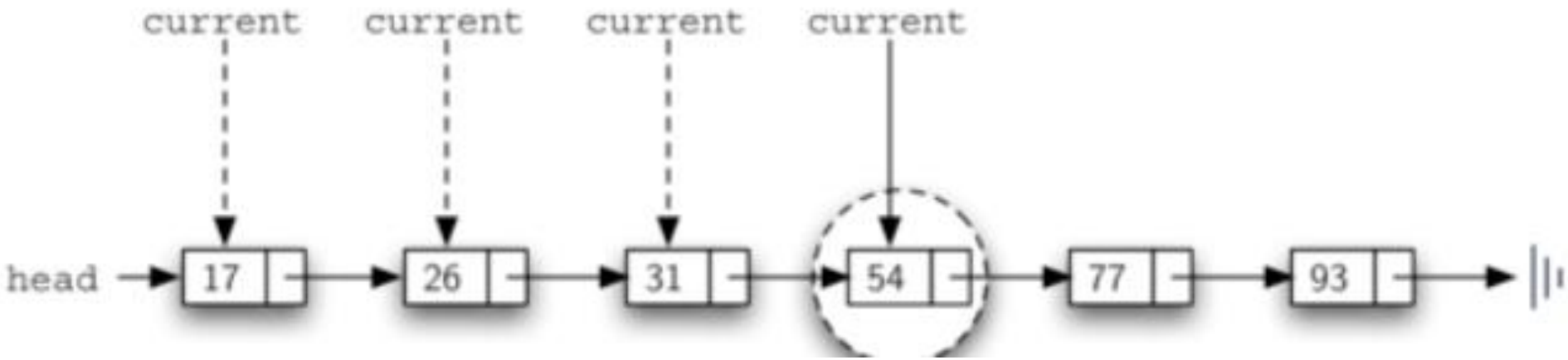
Для реализации класса OrderedList мы будем использовать ту же технику, что и для неупорядоченного списка. Пустой список вновь будет обозначаться ссылкой head на None.

Рассматривая операции для упорядоченного списка, стоит отметить, что методы isEmpty и size могут быть реализованы аналогично неупорядоченному списку, поскольку имеют дело только с количеством узлов безотносительно их содержимого.

Также хорошо будет работать метод remove, потому что нам по-прежнему надо искать элемент, а затем окружающие узел ссылки, чтобы удалить его. Два оставшихся метода - search и add - потребуют некоторой модификации.

Реализация упорядоченного списка

Например, произведем поиск значения 54.



См. пример 12

Анализ сложности

Чтобы проанализировать сложность операций для связанных списков, нам нужно выяснить, требуют ли они обход.

Метод isEmpty – O(1), поскольку нужен всего один шаг, чтобы проверить, ссылается ли head на None.

Метод size – O(n), поскольку не существует способа узнать количество узлов в связанном списке, не обойдя его от головы до конца.

Добавление элемента add в неупорядоченный список всегда будет O(1), ведь мы просто помещаем новый узел в голову связанного списка.

Методы search и remove, а так же add для упорядоченных списков, требуют процесса обхода. Хотя в среднем им потребуется обойти только половину списка, все они имеют O(n) исходя из наихудшего случая с обработкой каждого узла в списке.

Отметим, что для списков существуют и другие реализации.