МІНІСТЕРСТВО ОСВІТИ І НАУКИ
ДЕРЖАВНИЙ ЗАКЛАД
„ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА”

Укладачі: Г.А. Могильний, М.А. Семенов, О.О. Смагіна, С. О. Переяславська

АЛГОРИТМІ І СТРУКТУРИ ДАНИХ

Курс лекцій до вивчення освітнього компонента для здобувачів освіти спеціальності
122 – „Комп'ютерні науки”

Полтава
ДЗ „ЛНУ імені Тараса Шевченка”
2024

УДК004.421(075.8)
М74

Р е ц е н з е н т и :

Козуб Ю.Г. – доктор технічних наук, професор, в. завідувача кафедри математики та інформатики ДЗ „Луганський національний університет імені Тараса Шевченка”.

Лахно В.А. – доктор технічних наук, професор кафедри комп'ютерних систем, мереж та кібербезпеки, Національний університет біоресурсів та природокористування України, м. Київ.

М74 Алгоритми та структури даних: Курс лекцій до вивчення освітнього компонента для здобувачів освіти спеціальності 122 – Комп'ютерні науки / укладачі А. Могильний, М. А. Семенов, О. О. Смагіна, С. О. Переяславська; держ. закл. „Луган. нац. ун-т імені Тараса Шевченка”. – Полтава: ДЗ „ЛНУ імені Тараса Шевченка”, 2024. – 110 с.

Курс лекцій структуровано відповідно до розділів робочої програми курсу „Алгоритми та структури даних”для спеціальності122 – Комп'ютерні наукикафедри математики та інформатики ДЗ ЛНУ імені Тараса Шевченка.знання про різноманітність структур даних, галузі їх використання, способи їх програмної обробки; формування умінь та навичок програмно обробляти статичні та динамічні дані з використанням різних методів та алгоритмів, у т.ч. розв'язування завдань на пошук, сортування, обробку динамічних структур тощо.

Курс лекцій призначений для здобувачів освіти спеціальності 122 – Комп'ютерні науки.

УДК 004.421(075.8)

Рекомендовано до друку Вченою радою Луганського національного університету імені Тараса Шевченка (протокол №10 від 20 грудня 2024 р.)

ЗМІСТ

Лекція 1. Аналіз алгоритмів. Нотація Big-O

Потрібно згадати про важливу різницю між власне програмою та алгоритмом, який вона втілює. Алгоритм - це універсальна покрокова інструкція з розв'язання задачі. Це спосіб вирішення будь-якого окремого її випадку, який за заданих вхідних значеннях завжди видає необхідний результат. З іншого боку, програма - це те, як алгоритм перекладено певною мовою програмування. Може існувати безліч програм, що реалізують один і той же алгоритм залежно від програміста та мови, яку він використовує.

Для подальшого дослідження цієї відмінності розглянемо функцію вActiveCode 1. Вона вирішує всім знайоме завдання обчислення суми перших nn цілих чисел. Алгоритм використовує ідею змінної-акумулятора, яка ініціалізується нулем.

```
def sumOfN(n):
  theSum = 0
  for i in range(1,n+1):
    theSum = theSum + i

  return theSum

print(sumOfN(10))
```

Функція уActiveCode 2. На перший погляд вона може здатися дивною, але при ближчому розгляді ви побачите, що тут робиться в точності те саме, що й у попередній. Причина, через яку це не очевидно, у поганій якості коду. Ми не використовуємо зрозумілі імена для ідентифікаторів, щоб підвищити читабельність, і робимо надмірне присвоєння на етапі акумуляції, що зовсім не є необхідним. def foo(tom):

```
  fred = 0
  for bill in range(1,tom+1):
    barney = bill
    fred = fred + barney

  return fred

print(foo(10))
```

Функція sumOf, природно, краще, ніж foo, якщо ви турбуєтеся про читабельність. Фактично, ви, можливо, бачили безліч подібних прикладів у ваших вступних курсах з програмування, оскільки однією з їх цілей є бажання допомогти вам навчитися писати програми, які легко читати та розуміти. Однак у цьому курсі ми зацікавлені в тому, щоб характеризувати алгоритм сам по собі.

Аналіз алгоритмів ґрунтується на порівнянні витрачених кожним з них об'ємів обчислювальних ресурсів. точки зору дві функції вище виглядають дуже схожими.

На даний момент стає важливим подумати над тим, що ми маємо на увазі під

"обчислювальними ресурсами". Існує два різні підходи до цього питання. Перший розглядає обсяг простору чи пам'яті, необхідний алгоритму на вирішення завдання. Ця величина зазвичай залежить від її конкретного окремого випадку. Однак часто зустрічаються алгоритми, що мають специфічні вимоги до обсягу, і тоді нам треба дуже акуратно підходити до пояснення варіантів.

Альтернативою вимог до простору є аналіз та порівняння алгоритмів за часом, який необхідний їм для обчислень. Цю величину іноді називають "часом виконання" алгоритму. Одним із способів виміряти час виконання функції sumOfN є проведення порівняльного аналізу. Він має на увазі, що ми засічемо реальний час, необхідний програмі на обчислення результату. У Python можна виконати цю операцію, відзначивши час початку та час закінчення роботи програми щодо системи, що використовується нами. У модулі time є функція time, яка повертає поточний системний час у секундах, що минув з довільно обраного початкового моменту. Викликавши цю функцію двічі - на початку і в кінці, - і потім порахувавши різницю, ми отримаємо точну кількість секунд (дрібна в більшості випадків), витрачених на виконання.

```
import time

def sumOfN2(n):
  start = time.time()

  theSum = 0
  for i in range(1,n+1):
    theSum = theSum + i

  end = time.time()

  return theSum,end-start
```

Лістинг 1демонструє оригінальну функцію sumOfN із викликами часу, вбудованими до та після підсумовування. Вона повертає кортеж, що складається з результату та кількості витраченого на обчислення часу (у секундах). Якщо ми виконаємо п'ять функцій, у кожному з яких буде обчислюватися сума перших 10000 цілих чисел, то ми отримаємо наступне:

```
>>>for i in range(5):
      print("Sum is %d required %10.7f seconds"%sumOfN(10000))
Sum is 50005000 required 0.0018950 seconds
Sum is 50005000 required 0.0018620 seconds
Sum is 50005000 required 0.0019171 seconds
Sum is 50005000 required 0.0019162 seconds
Sum is 50005000 required 0.0019360 seconds
```

Ми з'ясували, що результат добре повторюємо і що на виконання коду витрачається приблизно 0,0019 секунд.

```
>>>for i in range(5):
      print("Sum is %d required %10.7f seconds"%sumOfN(100000))
```

```
Sum is 5000050000 required 0.0199420 seconds
Sum is 5000050000 required 0.0180972 seconds
Sum is 5000050000 required 0.0194821 seconds
Sum is 5000050000 required 0.0178988 seconds
Sum is 5000050000 required 0.0188949 seconds
>>>
```

Знову часи, необхідні для кожного запуску, лежать близько один до одного, але стають довшими – приблизно вдесятеро. Для n, що дорівнює 1000000 ми отримаємо:

```
>>>for i in range(5):
      print("Sum is %d required %10.7f seconds"%sumOfN(1000000))
Sum is 500000500000 required 0.1948988 seconds
Sum is 500000500000 required 0.1850290 seconds
Sum is 500000500000 required 0.1809771 seconds
Sum is 500000500000 required 0.1729250 seconds
Sum is 500000500000 required 0.1646299 seconds
>>>
```

Середнє значення знову зросло приблизно в десять разів порівняно з попереднім.

А тепер розглянемоActiveCode 3, що демонструє інший спосіб вирішення задачі підсумовування Ця функція sumOfN3 використовує перевагу замкнутої формули ∑ni=1i=(n)(n+1)2∑i=1ni=(n)(n+1)2 для обчислення суми перших nn цілих без виконання ітерацій.

```
def sumOfN3(n):
  return (n*(n+1))/2

print(sumOfN3(10))
```

Якщо ми проведемо аналогічні контрольні виміри для sumOfN3, використовуючи п'ять різних значень для n (10000, 100000, 1000000, 10000000 і 100000000), то отримаємо наступні результати:

```
Sum is 50005000 required 0.00000095 seconds
Sum is 5000050000 required 0.00000191 seconds
Sum is 500000500000 required 0.00000095 seconds
Sum is 50000005000000 required 0.00000095 seconds
Sum is 5000000050000000 required 0.00000119 seconds
```

Є два важливі моменти, пов'язані з цими вихідними даними, на які варто звернути увагу. Перший - витрачений час набагато менший, ніж у будь-якому з попередніх прикладів. І другий - всі тимчасові величини дуже близькі одна до одної, незалежно від значення n.

Але про що цей тест свідчить нам насправді? Інтуїтивно ми здогадуємося, що ітеративне рішення виконуватиме більше роботи через повторення набору програмних кроків. Це, швидше за все, причина, через яку воно займає більше часу. Так само схоже, що час, необхідний ітеративному рішенню, зростає зі збільшенням

значення n. Тут, однак, постає проблема. Якщо ми запустимо ту саму функцію на різних комп'ютерах або використовуємо різні мови програмування, то цілком ймовірно, що отримаємо різні результати. Обчислення sumOfN3 займе тим більше часу, чим старший комп'ютер.

Нам потрібен найкращий спосіб характеризувати алгоритми щодо часу виконання. Тестова методика обчислює дійсний час виконання. Вона не надає нам дійсно корисного результату вимірювань, оскільки він залежить від конкретної машини, програми, часу дня, компілятора та мови програмування. Натомість ми хотіли б мати характеристику, незалежну від програми чи комп'ютера. Такий вимір було б корисно для оцінки алгоритму самого по собі, і його можна було б використовувати для порівняння алгоритмів у різних реалізаціях.

Лекція 2. Структура даних "стек".

Стек(іноді говорять "магазин" - за аналогією з магазином вогнепальної зброї) - це впорядкована колекція елементів, де додавання нового або видалення існуючого завжди відбувається лише на одному кінці. Цей кінець зазвичай називають "вершиною", а протилежний йому - "підставою".

Значимість підстави стека полягає в тому, що елементи, що зберігаються ближче до нього, являють собою ті, які знаходяться в стеку найдовше. Елемент, доданий останнім, розташований на позиції, з якої буде видалено в першу чергу. Такий принцип організації іноді називається LIFO, last-in, first-out (англ. «останнім прийшов – першим вийшов»). Він надає впорядкування за часом перебування у колекції. Нові елементи розташовані ближче до вершини, більш старі - ближче до основи.

З прикладами стеку ми стикаємось щодня. Чи не кожна закусочна має стопку з підносів або тарілок, де вам потрібно брати одну зверху, відкриваючи нову тацю для наступного відвідувача в черзі. Уявіть стек із книг на столі (*малюнок 1*).Єдиною книгою, чия обкладинка видна, є найвища.*Малюнок 2*демонструє інший стек, який містить кілька простих об'єктів даних Python.

Одна з найчастіше використовуваних ідей, пов'язаних зі стеком, прийшла з простого спостереження за тим, як додаються та видаляються його елементи. Припустимо, що ви починаєте з чистого столу. Тепер кладіть книги по одній один раз за одним. Ви конструюєте стек. Подивимося, що станеться, коли ви почнете видаляти їх. Черговість, в якій це відбуватиметься, протилежна тому, як вони клалися. Стеки є фундаментально важливими, оскільки їх можна використовувати для реверсування порядку елементів. Послідовність вставок протилежна послідовності видалень.*Малюнок 3*показує стек з об'єктів даних Python у процесі його створення та видалення з нього елементів. Зверніть увагу до порядку об'єктів.

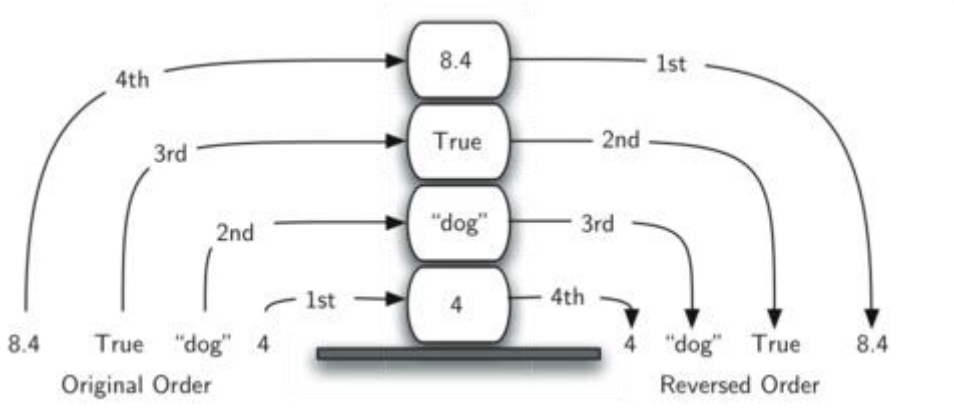


Рисунок 3: Властивість реверсування у стеків

Розглядаючи цю реверсивну властивість, ви, можливо, подумаєте про приклади стека, що мають місце під час роботи з комп'ютером. Наприклад, кожен веб-браузер має кнопку “Назад”. Коли ви переміщуєтеся від однієї веб-сторінки до іншої, вони розміщуються в стек (точніше, в стек розміщуються їх URL'и). Поточна сторінка, яку ви переглядаєте, знаходиться на вершині, а найперша з переглянутих - на підставі.

Якщо ви натиснете кнопку “Назад”, то почнете рухатися сторінками у зворотному порядку.

Абстрактний тип даних для стека визначається наступними структурою та операціями. Як писалося вище, стек організований як упорядкована колекція елементів, що додаються та видаляються з кінця, званого “вершина” (LIFO-упорядкованість). Нижче наведені операції над стеком.

- Stack()створює новий порожній стек. Параметри не потрібні, повертає порожній стек.
- push(item)додає новий елемент до вершини стека. Як параметр виступає елемент; функція нічого не повертає.
- pop()видаляє верхній елемент зі стека. Установки не потрібні, функція повертає елемент. Стек змінюється.
- peek()повертає верхній елемент стека, але не видаляє його.
- isEmpty()перевіряє стек на порожнечу. Параметри не потрібні, повертає значення бульова.
- size()повертає кількість елементів у стеку. Параметри не потрібні, тип результату – ціле число.

Наприклад, якщоs- свіжостворений порожній стек, то в*таблиці 1*показано результати проведення над ним певної послідовності операцій. Для узгодження з сутністю стека верхній елемент ставитиме на праве місце в списку.

Таблиця 1: Прості стікові операції

Операція над стеком	Зміст стеку	Значення, що повертається
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

Тепер, коли у нас є чітке визначення стека АТД, звернемо свою увагу на використання Python для його реалізації. Нагадаємо, що коли ми даємо фізичну реалізацію абстрактного типу даних, то маємо на увазі реалізацію структури даних.
В Python (як і в будь-якій об'єктно-орієнтованій мові) реалізація обраного абстрактного типу даних (наприклад, стека) - це створення нового класу. суть - колекція елементів, що має сенс скористатися потужністю та простотою примітивних колекцій, що надаються Python. Ми будемо використовувати список.
Клас списку Python надає механізм і набір методів для впорядкованої колекції. Наприклад, якщо у нас є список [2, 5, 3, 6, 7, 4], то нам потрібно лише визначитися, який із його кінців прийняти за вершину стека, а який – за базу. Як тільки рішення ухвалено, можна починати реалізовувати операції, використовуючи такі спискові методи, якappendipop
Нижченаведена реалізація стека (*ActiveCode 1*) Припускає, що верхній елемент стека розташований в кінці списку. У міру зростання стека (має місце операціяpush), нові елементи будуть додаватися туди. Їм же маніпулюватиме операціяpop

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

З натисканням кнопкиrunне буде нічого, крім оголошення класу.Stack, а потім використовувати його.*ActiveCode 2*демонструє класStacky дії, яку ми представили послідовністю операцій з*таблиці 1*

```
від pythonds.basic.stack import Stack

s = Stack ()

print(s.isEmpty())
s.push(4)
s.push('dog')
```

```
print(s.peek())
s.push(True)
print(s.size())
print(s.isEmpty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())
```

Важливо, що ми можемо вибрати реалізацію стека через список, де вершиною вважається перший, а чи не останній елемент. У цьому випадку попередні методиappendipopпрацювати не будуть. Ми повинні будемо явно використовуватиpopiinsertдля позиції з індексом 0 (перший елемент у списку).

Лекція 3. Структура даних "черга"

Черга - це впорядкована колекція елементів, у якій додавання нових відбувається з кінця, званого “хвіст черги”, а видалення існуючих - з іншого, “голови черги”. Як тільки елемент додається до кінця черги, він починає свій шлях до її початку, очікуючи видалення попередніх.

Останні з доданих у чергу одиниць повинні чекати наприкінці колекції. Елемент, який пробув у черзі найдовше, знаходиться на її початку. Такий принцип упорядкування іноді називають FIFO, first-in first-out (англ. "першим прийшов - першим вийшов"). Ще він відомий, як "першим прийшов - першим обслужений".

Найпростіший приклад такої структури даних - це звичайна черга, в якій усі ми іноді стоїмо: у кінотеатрі, перед касою в бакалійній лавці, в закусочній (де ми, до речі, можемо "виштовхувати" тацю зі стопки/стека). Правильні черги дуже обмежені тим, що мають тільки один шлях у та один шлях. Для них не передбачені стрибки в середину або вихід до того, як пройде достатньо часу, щоб перейти на початок.*Малюнок 1*показує просту чергу з даних Python.

Рисунок 1: Черга з даних Python

В інформатиці також є поширені приклади черг. У нашій комп'ютерній лабораторії 30 комп'ютерів, підключених по мережі до одного принтера. Коли студенти хочуть щось роздрукувати, вони набирають завдання “встати у чергу” разом з усіма іншими завданнями, що очікують друку. Перше завдання – наступне, яке буде виконано. Якщо ви останній у черзі, то повинні чекати, поки надрукуються всі документи, що стоять перед вами. Пізніше ми досліджуємо цей цікавий приклад докладніше.

Крім черги на друк, оперативна система використовує кілька різних черг для контролю процесів усередині комп'ютера. Планування, що робити наступним, зазвичай ґрунтується на алгоритмі черги, який намагається запускати програми на виконання так швидко, як це можливо, та обслуговувати максимальну кількість користувачів. Також іноді під час друку на клавіатурі натискання клавіш випереджають появу символів на екрані. Це відбувається через те, що комп'ютер виконує іншу роботу. Натискання клавіш поміщаються в чередоподібний буфер, щоб зрештою відобразитись на екрані в правильному порядку.

Абстрактний тип даних для черги визначається наступними структурою та операціями. Як говорилося вище, черга структурована як упорядкована колекція елементів, які додаються з одного кінця (хвоста), а видаляються з іншого (голови). Черга підтримує властивість упорядкування FIFO. Операції для черги подано нижче.

- Queue()створює нову порожню чергу. Не потребує параметрів, повертає порожню чергу.
- enqueue(item)додає новий елемент до кінця черги. Вимагає елемент як параметр, нічого не повертає.

- dequeue()видаляє із черги передній елемент. Не потребує параметрів, повертає елемент. Черга змінюється.
- isEmpty()перевіряє чергу на порожнечу. Не потребує параметрів, повертає булеве значення.
- size()повертає кількість елементів у черзі (ціле число). Не потребує параметрів.

Як приклад, якщо ми припустимо, щоq- це черга, яка створена і поки що порожня, то*таблиця 1*показує результати послідовності операцій з неї. Вміст черги показаний таким чином, що голова знаходиться праворуч. Число 4 було першим елементом, що очікує обробки, тому воно буде першим і прибрано з черги.

Таблиця 1: Приклад операцій для черги

Оператор	Вміст	Значення, що повертається
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

Для реалізації АТД черги відповідним рішенням знову стане створення нового класу. Як і раніше, для побудови внутрішньої вистави черги ми будемо використовувати міць і простоту колекції “список”.

Нам треба визначитися, який кінець списків вважати головою, а який – хвостом. Реалізація, показана в*лістингу 1*припускає, що останній елемент черги знаходиться на його нульовій позиції. Це дозволяє використовувати функцію`insert`для додавання нових елементів до кінця черги.`pop`використовуватиметься для видалення переднього елемента (останнього елемента у списку). Також це означає постановку в чергу за O(n), а витяг - за O(1) часу.

```
клас Queue:
   def __init__(self):
      self.items=[]
```

```
    def isEmpty(self):
        return self.items==[]

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

CodeLens 1 демонструє класQueueу дії для послідовності операцій з*таблиці 1*

```
>>>q.size()
3
>>>q.isEmpty()
False
>>>q.enqueue(8.4)
>>>q.dequeue()
4
>>>q.dequeue()
'dog'
>>>q.size()
2
```

Лекція 4. Структура даних "Дек"

Грудень, також званий двосторонньою чергою, - це впорядкована колекція елементів, подібна до черги. Він має два кінці (голову та хвіст), і його елементи залишаються позиціонованими. Що відрізняє дек, так це несувора природа додавання та видалення його складових. Нові елементи можуть бути додані як в голову, так і хвіст. Аналогічно, існуючі компоненти можуть видалятися з обох кінців. В якомусь сенсі цей гібрид лінійної структури поєднує всі можливості стеків і черг.*Малюнок 1*демонструє дек із об'єктів даних Python.
Важливо, що незважаючи на володіння деком багатьох характеристик стеків і черг, він не підтримує LIFO або FIFO впорядкування, які втілюються в життя цими структурами даних. Тільки від вас залежить, який тип операції додавання чи видалення використовувати.

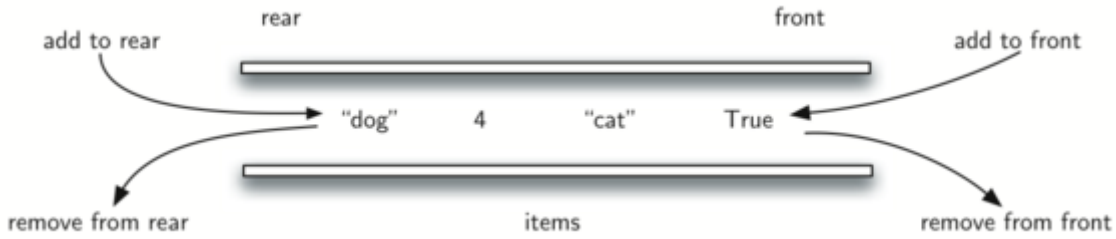


Рисунок 1: Дек з даних Python

Абстрактний тип даних дека визначається наступними структурою та операціями. Як було описано вище, він має структуру впорядкованої колекції елементів, які можуть додаватися та видалятися з будь-якого кінця – і з голови, і з хвоста. Операції для дека представлені нижче:

- Deque()створює новий порожній грудень. Не потребує параметрів і повертає порожній дек.
- addFront(item)додає новий елемент до голови дека. Параметр (елемент) необхідний; нічого не повертає.
- addRear(item)додає новий елемент у хвіст дека. Параметр (елемент) необхідний; нічого не повертає.
- removeFront()видаляє перший елемент із дека. Не потребує параметрів та повертає елемент. Грудень модифікується.
- removeRear()видаляє останній елемент із дека. Не потребує параметрів та повертає елемент. Грудень модифікується.
- isEmpty()перевіряє гру на порожнечу. Не потребує параметрів і повертає значення бульова.
- size()повертає кількість елементів у колоді. Не потребує параметрів і повертає ціле число.

Як приклад, якщо ми припустимо, щоd- це дек, який був створений і все ще порожній, таблиця 1 демонструє результати серії операцій над ним. Зверніть увагу, що вміст голови розташований праворуч. Дуже важливо відстежувати, де голова, а де

хвіст дека, коли ви переміщаєте елементи в та з колекції, оскільки ці речі можуть бути дещо заплутаними.

Таблиця 1: Приклад операцій для дека

Операція	Вміст дека	Значення, що повертається
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

Створимо новий клас для реалізації АТД "дек", як ми неодноразово робили у попередніх розділах. Список Python знову надасть дуже гарний набір методів, за допомогою яких ми збудуємо деталізацію цієї структури даних. Наша реалізація (*листинг 1*) припускатиме, що хвіст дека знаходиться в нульовій позиції списку.

```
клас Deque:
    def __init__(self):
        self.items=[]

    def isEmpty(self):
        return self.items==[]

    def addFront(self, item):
        self.items.append(item)

    def addRear(self, item):
        self.items.insert(0,item)

    def removeFront(self):
        return self.items.pop()
```

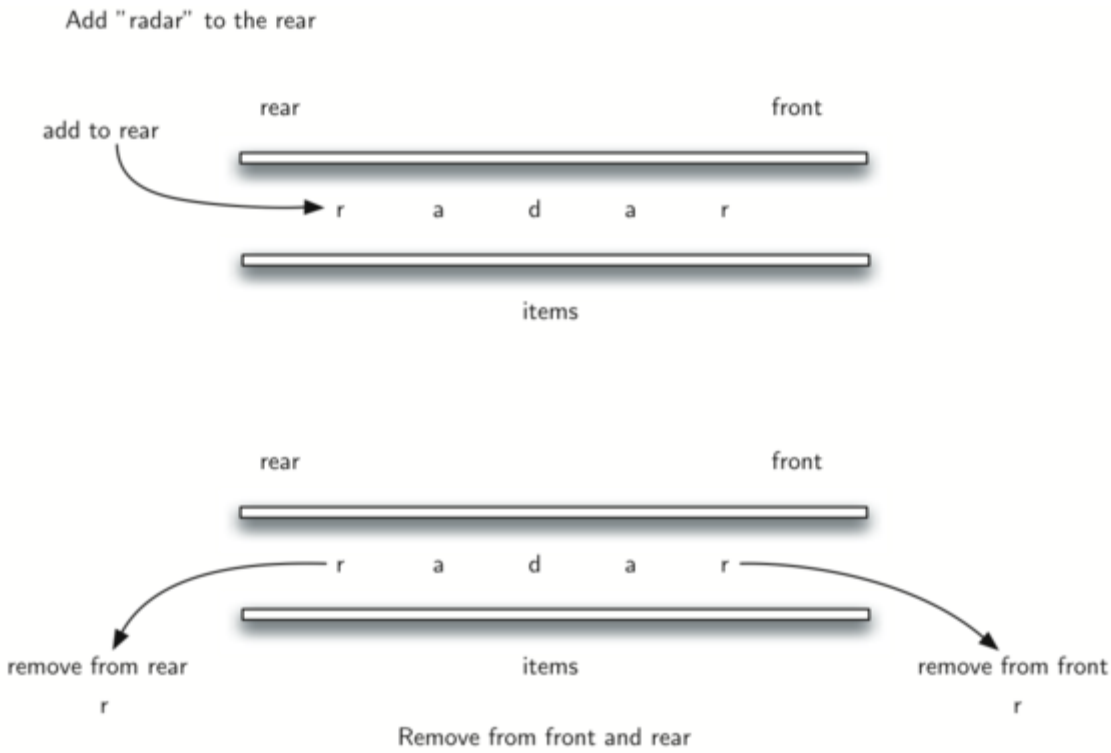
```
def removeRear(self):
    return self.items.pop(0)

def size(self):
    return len(self.items)
```

У removeFront ми використовуємо метод pop для видалення останнього елемента зі списку. Однак у removeRear метод pop(0) повинен видаляти перший з них. Також нам потрібно використовувати метод insert (рядок 12) в addRear, оскільки append передбачає додавання нового елемента до кінця списку.

Можна знайти багато подібності в коді на Python, що описує стеки та черги. Ймовірно, ви також помітили, що в цій реалізації додавання та видалення елементів із голови має O(1), в той час як ті ж операції для хвоста – O(n). Це слід було очікувати, враховуючи які поширені методи використовувалися цієї мети. Знову ж таки, головне, в чому ми повинні бути впевнені, то це в тому, що в нашій реалізації призначено хвостом, а головою дека.

Цікава задача, яка може бути легко вирішена з використанням структури даних "дек" - це класичне завдання паліндрому. Паліндромом називають рядок, який однаково читається праворуч наліво та зліва направо. Наприклад, radar, toot або madam. Ми хочемо створити алгоритм, який приймає на вхід рядок символів і перевіряє, чи він є паліндромом.

Для вирішення цього завдання використовуємо дек як сховище рядкових символів. Ми оброблятимемо рядок зліва направо і додаватимемо кожен її елемент у хвіст дека. У цей момент він працюватиме дуже схоже зі звичайною чергою. Однак тепер ми можемо використовувати дуальну функціональність дека. Його голова зберігатиме перший символ рядка, а хвіст – останній (seемалюнок 2).



Малюнок 2: Грудень

Оскільки ми здатні видаляти обидва елементи відразу, можна порівняти і продовжувати лише тоді, коли символи збігаються. Якщо відповідність першого і останнього елементів буде зберігатися, то зрештою ми прийдемо або до відсутності символів, або залишимося з деком розміром 1 - залежно від того, чи довжина вихідного рядка була парним чи непарним числом. Але в обох випадках вхідна послідовність буде паліндромом. Повністю функція перевірки представлена вActiveCode 1.

```
from pythonds.basic.deque import Deque

def palchecker(aString):
    chardeque = Deque()

    for ch in aString:
        chardeque.addRear(ch)

    stillEqual = True

    while chardeque.size() > 1 and stillEqual:
        first = chardeque.removeFront()
        last = chardeque.removeRear()
        if first != last:
            stillEqual = False
```

```
    return stillEqual

print(palchecker("lsdkjfskf"))
print(palchecker("radar"))
```

Лекція 5. Структура даних "Список"

У процесі обговорення основних структур даних ми використовували списки Python для реалізації представлених абстрактних типів даних. Списки – потужний, але простий механізм колекцій, що дарує програмісту велику різноманітність операцій. Однак, не всі мови програмування мають списки як інтегровані структури даних. У разі програмісту доводиться створювати їх вручну.

Список- це колекція елементів, кожен із яких зберігається на відповідній позиції стосовно решти. Точніше, список такого роду ми називатимемо невпорядкованим списком. Можна зробити висновок, що список має перший елемент, другий елемент, третій тощо. Ми можемо також посилатися на початок списку (перший елемент) або його кінець (останній елемент). Для простоти будемо вважати, що списки не містять даних, що дублюються.

Наприклад, колекція цілих чисел 54, 26, 93, 17, 77 і 31 може бути простим невпорядкованим списком екзаменаційних оцінок. Зауважте, що ми повинні записувати їх як розділені комами значення - загальноприйнятим способом зображення структури списку. Python, звісно, покаже список так: [54,26,93,17,77,31][54,26,93,17,77,31].

Структура неупорядкованого списку, описаного вище, є колекцією елементів, кожен із яких перебуває в певної позиції стосовно іншим. Деякі з можливих операцій над невпорядкованими списками наведені нижче.

- List()створює новий пустий список. Не потребує параметрів та повертає порожній список.
- add(item)додає новий елемент до списку. Потребує значення як аргумент, нічого не повертає. Припускає, що раніше елемент у списку був відсутній.
- remove(item)видаляє елемент зі списку. Вимагає значення елемента та змінює список. Передбачає початкову наявність елемента у списку.
- search(item)шукає елемент у списку. Вимагає елемент і повертає значення бульова.
- isEmpty()перевірка списку порожнечі. Не потребує параметрів, повертає булеве значення.
- size()повертає кількість елементів у списку. Не потребує параметрів, повертає ціле число.
- append(item)додає новий елемент до кінця списку. Потребує значення як аргумент, нічого не повертає. Припускає, що досі такого елемента у списку не було.
- index(item)повертає позицію елемента у списку. Потребує значення як аргумент, повертає його індекс. Припускає, що елемент є у списку.
- insert(pos,item)вставляє новий елемент у задану позиціюпосписок. Потребує елемента, нічого не повертає. Припускає, що до цього часу такий елемент у списку був відсутній і існуючий розмір списку дозволяє задати індекспос.
- pop()видаляє та повертає останній елемент списку. Чи не вимагає аргументів, повертає елемент. Припускає, що у списку є хоч один елемент.
- pop(pos)видаляє та повертає елемент з позиціїпос. Вимагає позицію як аргумент і повертає елемент. Припускає, що такий елемент є у списку.

Для реалізації невпорядкованого списку ми створимо те, що зазвичай називають пов'язаним списком. Нагадаємо: ми повинні бути впевнені, що зможемо підтримувати порядок взаємного розташування елементів.*малюнку 1*. Схоже, ці значення було розміщено випадковим чином. Якщо ми зможемо зберегти в кожному елементі певну інформацію про розміщення його сусіда (див.*малюнок 2*), то відповідна позиція кожного з них може бути виражена простим посиланням від одного до іншого.

Рисунок 1: Елементи не обмежені у своєму фізичному розміщенні.

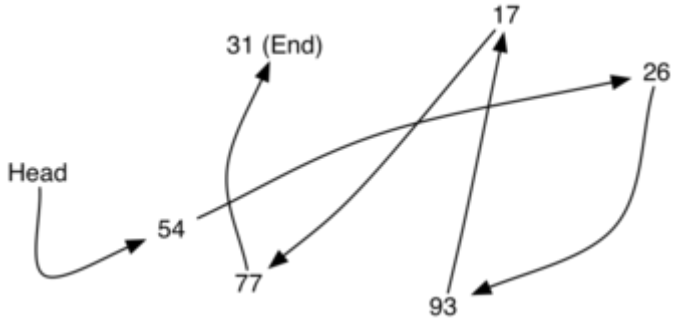


Рисунок 2: Відповідні позиції визначаються явними посиланнями.

Важливо відзначити, що положення першого елемента має бути явно. Оскільки ми знаємо, де він знаходиться, то можемо знайти місце розташування другого і так далі Зовнішнє посилання часто називають головою списку.

КласNode

Основним будівельним блоком у реалізації пов'язаного списку є вузол. Кожен такий об'єкт повинен мати як мінімум дві інформаційні складові. По-перше, вузол повинен містити самий елемент списку. Ми назвемо це полем даних вузла. Додатково він повинен зберігати посилання наступного вузол.*Лістинг 1*демонструє реалізацію цієї ідеї на Python. Щоб створити вузол, необхідно надати початкове значення його даних. Обчислення оператора привласнення нижче дасть об'єкт "вузол", що містить значення 93 (див.*малюнок 3*). Слід зазначити, що ми представляємо вузол-об'єкт, як це показано на*малюнку 4*. КласNodeтакож включає звичайні методи для доступу та модифікації полів даних та посилання на наступний вузол.

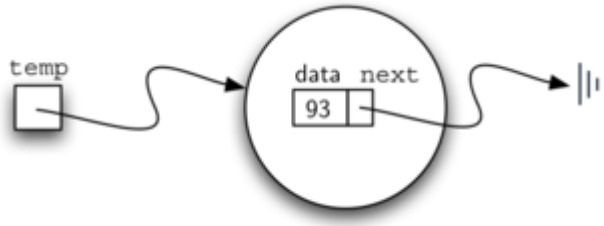
```
клас Node:
   def __init__(self,initdata):
      self.data=initdata
      self.next= None
```

```
    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self,Newdata):
        self.data=newdata

    def setNext(self,Newnext):
        self.next=newnext
```

Ми створюємо об'єктNodeзвичайним способом.

```
>>>temp=Node(93)
>>>temp.getData()
93
```

Спеціальне посилання PythonNoneвідіграє важливу роль і в класіNode, і в найповнішому списку. Зверніть увагу, що конструктор спочатку створює вузол зnext, встановленим уNone. Явне присвоєннняNoneПочаткове значення посилання на наступний елемент - незмінно хороша ідея.



Малюнок 3: Об'єктNode, що містить значення та посилання на наступний вузол

Малюнок 4: Типове представлення вузла

КласUnordered List

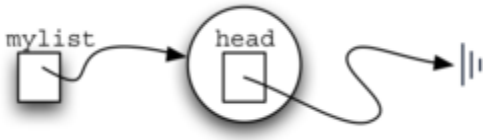
Як ми пропонували вище, невпорядкований список буде колекцією вузлів, кожен з яких пов'язаний з наступним за допомогою явного посилання. Поки ми знаємо, як знайти перший вузол (що містить перший елемент), кожен наступний буде виявлено за допомогою успішного прямування за посиланнями. Маючи це на увазі, класUnorderedListповинен містити посилання першому вузол. У*лістингу 2*показаний конструктор такого класу. Зауважте, що кожен об'єкт списку буде підтримувати єдине посилання на свою голову.

```
клас UnorderedList:

    def __init__(self):
        self.head= None
```

Спочатку ми створимо порожній список. Оператор присвоєння

```
>>>mylist=UnorderedList()
```

створить уявлення пов'язаного списку, показаного на*малюнку 5*. Як ми вже обговорювали для класуNode, спеціальне посиланняNoneзнову буде використовуватися як стан, коли голова списку ні на що не посилається.*малюнку 6*. Голова списку посилається перший вузол, що містить перший елемент списку. У свою чергу цей вузол містить посилання на наступний вузол (наступний елемент) і так далі. Дуже важливо відзначити, що клас списку сам по собі не містить будь-яких об'єктів-вузлів. Він має лише єдине посилання на перший вузол пов'язаної структури.



Малюнок 5: Порожній список

Малюнок 6: Пов'язаний список цілих чисел

МетодisEmpty, показаний у*лістингу 3*, просто перевіряє, чи посилається голова списку наNone. Результат булевого виразуself.head==Noneбуде дійсним лише якщо у зв'язаному списку немає вузлів. Оскільки новий список порожній, конструктор і перевірка на порожнечу повинні узгоджуватися один з одним. Це демонструє переваги використання посиланняNoneвизначення “кінця” пов'язаної структури. У PythonNoneможна порівнювати з будь-яким посиланням. Два посилання дорівнюють, якщо вони обидві посилаються на один об'єкт. У методах, що залишилися, ми будемо часто використовувати цей факт.

```
def isEmpty(self):
   return self.head== None
```

Отже, як нам помістити елементи в наш список? Для цього потрібно реалізувати методadd. Однак, перед цим варто відповісти на важливе запитання: де в списку розміщувати новий елемент? Оскільки, він невпорядкований, то конкретизації розташування нового елемента стосовно вже існуючим немає необхідності. Тобто. вставку можна здійснити будь-де. Зважаючи на це, має сенс помістити нове значення в найдоступнішу позицію.

Нагадаємо, що структура зв'язаного списку надає нам лише одну точку входу – голову списку. Всі інші вузли можна досягти лише через доступ до першого вузла по ланцюжку з посиланьnext. Це має на увазі, що найпростіше місце для додавання

нового вузла – це голова, початок списку. Іншими словами, ми будемо створювати новий елемент як перший у списку, а існуючі елементи потрібно буде зв'язати так, щоб вони йшли за ним.
Пов'язаний список, показаний на*малюнку 6*, був побудований за допомогою методу виклику`add`кілька разів.

```
>>>mylist.add(31)
>>>mylist.add(77)
>>>mylist.add(17)
>>>mylist.add(93)
>>>mylist.add(26)
>>>mylist.add(54)
```

Зверніть увагу: оскільки 31 - перший з доданих до списку елементів, то він, зрештою, і останній вузол пов'язаного списку, так як всі інші елементи додаються перед ним. Аналогічно, 54 - останній доданий елемент і він буде значенням першого вузла пов'язаного списку.
Метод`add`показаний у*лістингу 4*. Кожен елемент повинен бути обгорнутий в об'єкт`Node`. Рядок 2 створює новий вузол і розміщує в полі даних заданий елемент. Тепер потрібно завершити процес, зв'язавши новий вузол із існуючою структурою. Це вимагає двох кроків, показаних на*малюнку 7*. Крок 1 (рядок 3) змінює посилання.`next`нового вузла, щоб вона вказувала на попередній. Тепер, коли залишок відповідним чином приєднано до нового вузла, ми можемо змінити голову списку, щоб вона також посилалася на новий вузол. Цим займається оператор присвоєння у рядку 4.
Порядок двох описаних вище кроків дуже важливий. Що станеться, якщо поміняти місцями рядки 3 та 4? Якщо першим відбудеться зміна голови списку, результат можна побачити на*малюнку 8*. Оскільки голова - єдине зовнішнє посилання на список вузлів, всі початкові дані будуть втрачені і отримати до них доступ більше не вдасться.
Лістинг 4

```
def add(self,item):
   temp=Node(item)
   temp.setNext(self.head)
   self.head=temp
```

Наступними методами, які ми реалізуємо, будуть`size`,`search`і`remove`. Усі вони ґрунтуються на техніці обходу пов'язаного списку. Вона має на увазі процес чергового відвідування кожного вузла. Щоб зробити це, використовуємо зовнішнє посилання як стартове. Оскільки ми відвідуємо кожен вузол, то вона переміщатиметься`next`-ам елементів списку
Для реалізації методу`size`досить просто обійти пов'язаний список і підрахувати кількість вузлів, що зустрінуться.*Лістинг 5*демонструє код Python, що виконує цю роботу. Зовнішнє посилання називається`current`та ініціалізується головою списку у рядку 2. На початку обробки ми не бачимо будь-яких вузлів, тому рахунок дорівнює

нулю. Рядки 4-6 реалізують власне обхід. Поки поточне посилання не схоже на кінець списку (None), ми переміщаємо її на наступний вузол за допомогою оператора присвоювання у рядку 6. Зазначимо ще раз, що можливість порівнювати посилання зNoneдуже корисна. Щоразу переміщаючись на новий вузол, ми додаємо доcountодиницю. Після закінчення ітераційcountповертається як результат.*Малюнок 9*показує, як відбувається обробка з просуванням вниз по списку.

Лістинг 5

```
1  def size(self):
2      current= self.head
3      count= 0
4      whilecurrent!= None:
5          count=count+ 1
6          current=current.getNext()
7
8      returncount
```

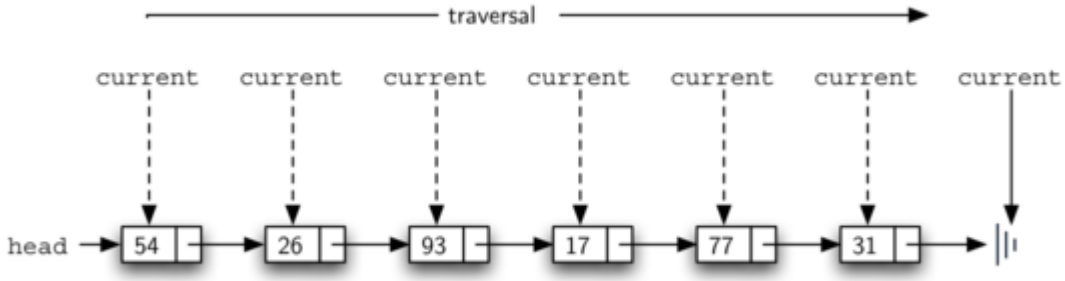


Рисунок 9: Обхід пов'язаного списку від голови до кінця

Реалізація пошуку значення пов'язаного списку для неупорядкованого списку також використовує техніку обходу. У процесі відвідування кожного вузла ми запитуємо, як його дані співвідносяться з тим елементом, який шукаємо. Однак у цьому випадку нам може і не знадобитися обходити весь список до кінця. Фактично, якщо ми дісталися кінця списку, то шуканого елемента він містить. А якщо ми знайшли потрібне, то нема рації продовжувати обхід.

*Лістинг 6*показує реалізацію методуsearch. Як і у методіsizeобхід починається з голови списку (рядок 2). Ми також використовуємо бульову змінноїfoundщоб пам'ятати, чи знайшли ми шукане. Оскільки на початку обходу ще нічого не знайдено, тофoundвстановлюється вFalse(Рядок 3). Цикл у рядку 4 приймає обидва описані вище умови. Доки є вузли для відвідування і шуканий елемент не знайдено, ми продовжуємо перевірку. Рядок 5 запитує: чи не в цьому вузлі те, що нам потрібно? Якщо відповідь “так”, тофoundвиставляється вTrue.

Лістинг 6

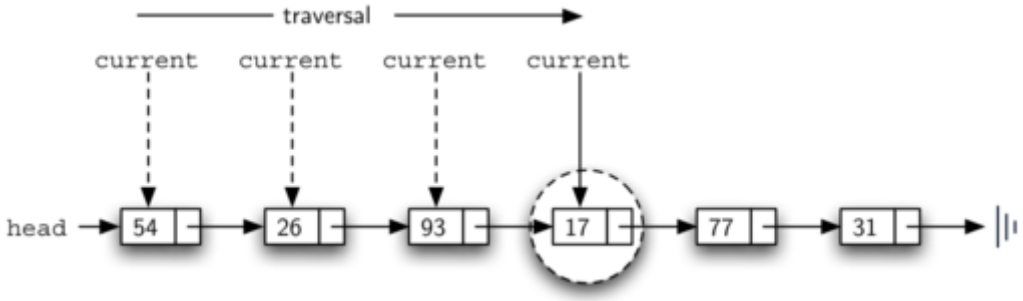
```
1  def search(self,item):
2      current= self.head
3      found= False
4      whilecurrent!= None and notfound:
5          ifcurrent.getData()==item:
6              found= True
```

```
7           else:
8               current=current.getNext()
9
10      returnfound
```

Як приклад розглянемо виклик методуsearchдля пошуку елемента 17.

```
>>>mylist.search(17)
True
```

Оскільки 17 у списку присутній, то обхід відбуватиметься доти, доки не досягне вузла, що його містить. У цей момент змінноїfoundбуде встановлена вTrueта умовawhileпорушиться, що призведе до повернення значення, як ми бачили вище. Цей процес можна розглянути на*малюнку 10*.



Малюнок 10: Успішний пошук значення 17

Методremoveлогічно потребує двох стадій. Спочатку треба обійти список, щоб знайти елемент, що видаляється. Як тільки це станеться (нагадаємо, що ми припускаємо наявність шуканого у списку), буде видалено. Перший крок дуже схожийsearch. Починаючи із зовнішнього посилання, встановленого на голову списку, ми обходимо посилання доти, доки знайдемо шуканий елемент. Оскільки передбачається його присутність, цикл зупиниться до того, якcurrentотримаєNone. Отже, за умови можна обмежитися однією булевою змінноюfound.

Колиfoundнабуде значенняTrue,currentбуде посилатися на вузол, який містить елемент, який потрібно видалити. Але як це зробити? Одним із способів є заміна значення елемента деяким маркером, що означає відсутність елемента у списку. Проблемою за такого підходу і те, що тепер кількість вузлів не збігається з кількістю елементів. Було б краще видаляти елемент за допомогою видалення всього вузла.

Для того щоб видалити потрібний вузол, нам потрібно змінити посилання його попередника таким чином, щоб воно посилалося на вузол післяcurrent. На жаль, немає можливості переміщатися по зв'язаному списку у зворотному напрямку. Оскількиcurrentпосилається на вузол наступний за тим, у якому ми хотіли б провести зміни, то для модифікації вже надто пізно.

Вирішенням цієї дилеми стане використання двох зовнішніх посилань у процесі обходу списку.currentповодитиметься як раніше, відзначаючи поточне положення обходу. Нове посилання, яке ми назвемоprevious, завжди буде йти на один вузол позадуcurrent. Таким чином, колиcurrentзупиниться на вузлі, що видаляється,previousбуде вказувати місце для відповідної модифікації пов'язаного списку.

*Лістинг 7*демонструє методremoveповністю. У рядках 2-3 двом посиланням надаються початкові значення. Зверніть увагу, щоcurrentпочинає з голови списку, як у попередніх прикладах. Однак, передбачається, щоpreviousзавжди на вузол за поточним. З цієї причини їй надається значенняNone, адже немає вузла перед головою списку (див.*малюнок 11*). Бульова змінnafoundзнову відповідає контроль ітерацій.

У рядках 6-7 ми запитуємо, чи збігається елемент, що зберігається у вузлі, з тим, який хочемо видалити. Якщо так, тоfoundвстановлюється вTrue. Якщо ні, topreviousicurrentпереміщаються однією вузол вперед. Порядок двох присвоювань знову дуже важливий. Насамперед потрібно перемістити`Previous, а потімcurrent. Цей процес часто називають "черв'яком", так якpreviousповинен наздогнатиcurrentдо того, як той знову піде вперед.*Малюнок 12*показує переміщенняpreviousicurrentу процесі спуску за списком у пошуках вузла, що містить значення 17.

Лістинг 7

```
1  def remove(self,item):
2      current= self.head
3      previous= None
4      found= False
5      while notfound:
6          ifcurrent.getData()==item:
7              found= True
8          else:
9              previous=current
10             current=current.getNext()
11
12     ifprevious== None:
13         self.head=current.getNext()
14     else:
15         previous.setNext(current.getNext())
```

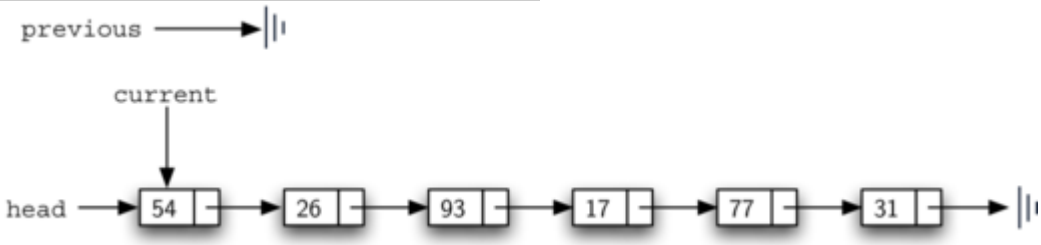
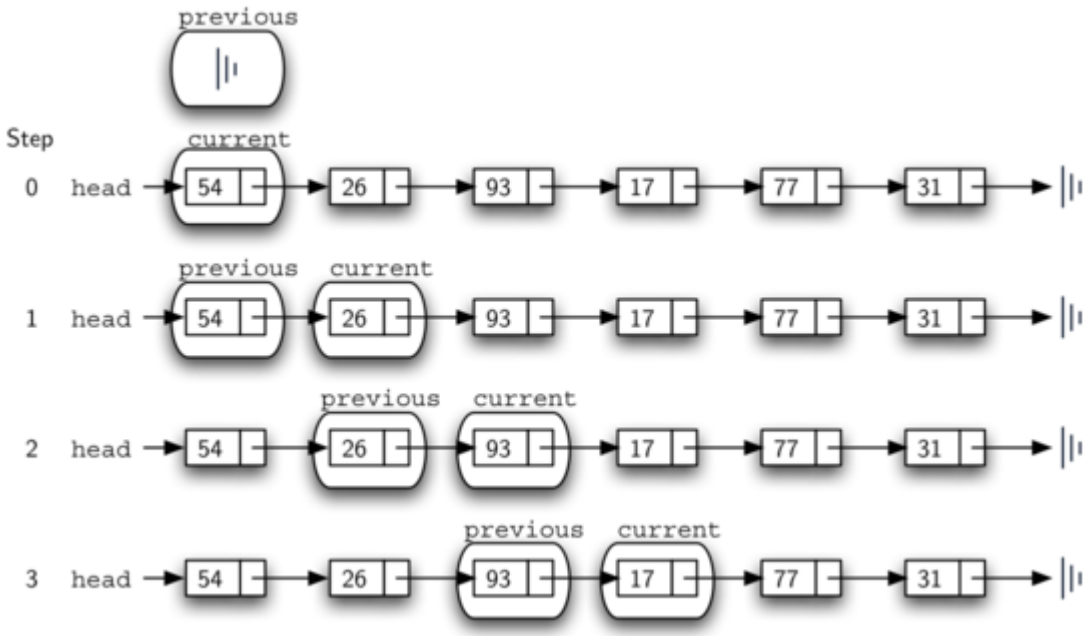
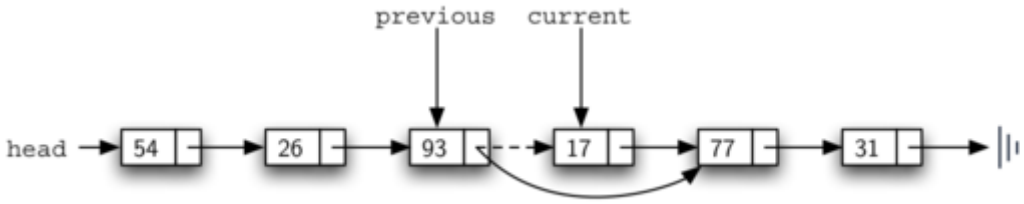


Рисунок 11: Початкові значення посиланьpreviousicurrent

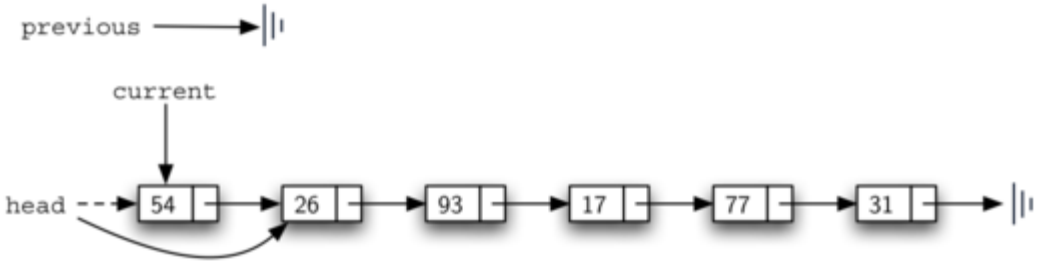


Малюнок 12:previousicurrent: вниз за списком

Після завершення пошукового кроку методуremoveнам потрібно видалити вузол зі зв'язаного списку. на*малюнку 13*показано посилання, яке має бути змінено. Однак є особливий випадок, за яким потрібно прийняти окреме рішення. Якщо елемент, що видаляється - перший у списку, тоcurrentбуде посилатися на перший вузол, апreviousдорівнюєNone. Раніше ми говорили, що вказівкою на вузол, чиє посилання вимагатиме зміни для завершення операції видалення, служитьprevious. Але цього випадку модифікації потребує голова списку (див.*малюнок 14*).



Малюнок 13: Видалення елемента з середини списку



Малюнок 14: Видалення першого вузла у списку

У рядку 12 перевіряється, чи маємо справу зі спеціальним випадком, описаним вище. Якщоpreviousне переміщався, то він, як і раніше, матиме значенняNoneприfoundрівномуTrue. У цьому випадку (рядок 13) голова списку змінюється так, щоб посилатися на вузол, що йде за поточним, ефект видалення першого вузла зі зв'язаного списку. Однак, якщоpreviousнеNone, що видаляється вузол знаходиться десь нижче у зв'язаному списку. У цьому випадку слід

дотримуватися основного алгоритму. Рядок 15 використовує методsetNextдляprevious, щоб видалити.current.getNext().

А тепер розглянемо такий підвид списків як упорядкований список. Наприклад, якби список цілих, показаний вище, був упорядкованим (за зростанням), то він записався як 17, 26, 31, 54, 77 і 93. 17 - найменший елемент, тому він ставиться на першу позицію, а 93 - найбільший, так що він посідає останнє місце.

По структурі впорядкований перелік є колекцію елементів, кожен із яких займає становище залежно від певної загальної всім характеристики. Порядок упорядкування зазвичай або зростаючий, або спадаючий, і ми вважаємо, що для елементів списку існує і визначена операція порівняння. Багато операцій для впорядкованого списку аналогічні методам невпорядкованого.

- OrderedList()створює новий упорядкований перелік. Не потребує параметрів, повертає пустий список.
- add(item)додає до списку новий елемент, попередньо переконавшись, що порядок зберігається. Вимагає елемент як аргумент, нічого не повертає. Передбачається, що раніше елемент у списку був відсутній.
- remove(item)видаляє елемент зі списку. Вимагає елемент та змінює список. Передбачається, що елемент є у списку.
- search(item)шукає елемент у списку. Вимагає елемент і повертає значення бульова.
- isEmpty()перевіряє список на порожнечу. Не потребує параметрів і повертає значення бульова.
- size()повертає кількість елементів у списку. Не потребує параметрів та повертає ціле число.
- index(item)повертає позицію елемента у списку. Вимагає елемент та повертає індекс. Передбачається, що елемент у списку є.
- pop()виштовхує останній елемент списку. Не вимагає параметрів і повертає елемент.
- pop(pos)виштовхує елемент із позиціїpos. Потребує індекс як аргумент, повертає елемент. Передбачається наявність елемента у списку.

Перед початком реалізації впорядкованого списку незайвим буде згадати, що положення елементів щодо один одного ґрунтується на певній базовій характеристиці. Упорядкований список цілих чисел, наведений вище (17, 26, 31, 77 і 93), може бути виражений пов'язаною структурою, показаною на*малюнку 15*Знову ж таки, вузол і посилання ідеально підходять для представлення взаємного розташування елементів.

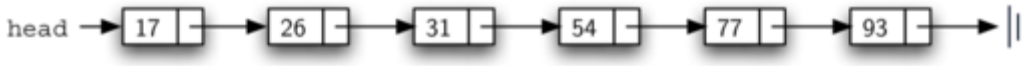


Рисунок 15: Упорядкований пов'язаний список

Для реалізації класуOrderedListми використовуватимемо ту ж техніку, що й для неупорядкованого списку. Порожній список знову позначатиметься посиланнямheadнаNone(Див.*лістинг 8*).

Лістинг 8

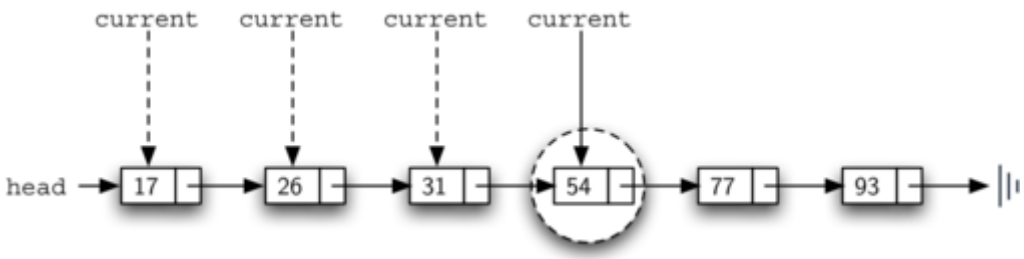
```
клас OrderedList:
   def __init__(self):
```

```
        self.head= None
```

Розглядаючи операції для впорядкованого списку, слід зазначити, що методиisEmptyisizeможуть бути реалізовані аналогічно до неупорядкованого списку, оскільки мають справу лише з кількістю вузлів безвідносно їхнього вмісту. Також добре працюватиме методremove, тому що нам, як і раніше, потрібно шукати елемент, а потім навколишні вузол посилання, щоб видалити його. Два методи, що залишилися -searchiadd- Вимагають деякої модифікації.

Пошук у неупорядкованому списку вимагає, щоб ми обходили вузли по одному за раз, поки не знайдемо шуканий елемент або не вийдемо за межі списку (None). Такий підхід працюватиме і для впорядкованого списку. У тому випадку, коли елемент знайдеться, він буде тим, що нам потрібний. Однак, якщо елемент не міститься у списку, ми можемо скористатися перевагою впорядкування, щоб зупинити пошук якомога раніше.

Наприклад,*малюнок 16*показує впорядкований пов'язаний список, в якому шукається значення 45. У процесі обходу ми починаємо з його голови і спочатку перевіряємо на відповідність 17. Оскільки 17 – не те, що ми шукаємо, то зміщуємося до наступного вузла – 26. Це знову не те, переміщаємось до 31, а потім до 54. І тут дещо змінюється. Оскільки 54 не той елемент, що ми шукаємо, наша попередня стратегія має полягати у просуванні вперед. Проте, з урахуванням упорядкованості списку, у цьому більше немає необхідності. Якщо значення у вузлі більше, ніж шукане, то пошук можна зупиняти і повертатиFalse. Не існує способу значення опинитися серед залишку впорядкованого списку.



Малюнок 16: Пошук в упорядкованому списку

*Лістинг 9*показує закінчений методsearch. Нову умову, що обговорюється вище, можна вставити дуже легко: додати ще одну булеву змінну -stop- та ініціалізувати їїFalse(Рядок 4). Поки щостоодно`False`ми можемо продовжувати пошук у списку (рядок 5). Якщо будь-якому з вузлів виявиться значення більше шуканого, тостовстановиться вTrue(Рядки 9-10). Рядки, що залишилися, ідентичні пошуку в неупорядкованому списку.

Лістинг 9

```
def search(self,item):
    current= self.head
    found= False
    stop= False
    whilecurrent!= None and notfoundand notstop:
        ifcurrent.getData()==item:
```

```
            found= True
        else:
            ifcurrent.getData()>item:
                stop= True
            else:
                current=current.getNext()

    returnfound
```

Найбільш значна модифікація торкнеться методуadd. Нагадаємо, що у невпорядкованих списках він просто поміщав новий елемент у голову списку – найдоступнішу точку. На жаль, із упорядкованим списком це більше не спрацює. Тепер нам треба шукати спеціальне місце, де розміщуватиметься новий елемент серед тих, що вже існують у впорядкованому списку.

Припустимо, що є впорядкований список із 17, 26, 54, 77 та 93, і ми хочемо додати до нього значення 31. Методaddповинен вирішити, що новий елемент слід розташувати між 26 та 54.*Малюнок 17*показує потрібну вставку. Як ми пояснювали раніше, потрібно обійти пов'язаний список у пошуках місця, куди буде вставлено новий елемент. Ми знаємо, що місце знайдено, якщо ми або вийшли за межі списку (currentодноNone), або значення поточного вузла стало більше, ніж елемент, що додається. У прикладі нас змусить зупиниться поява значення 54.

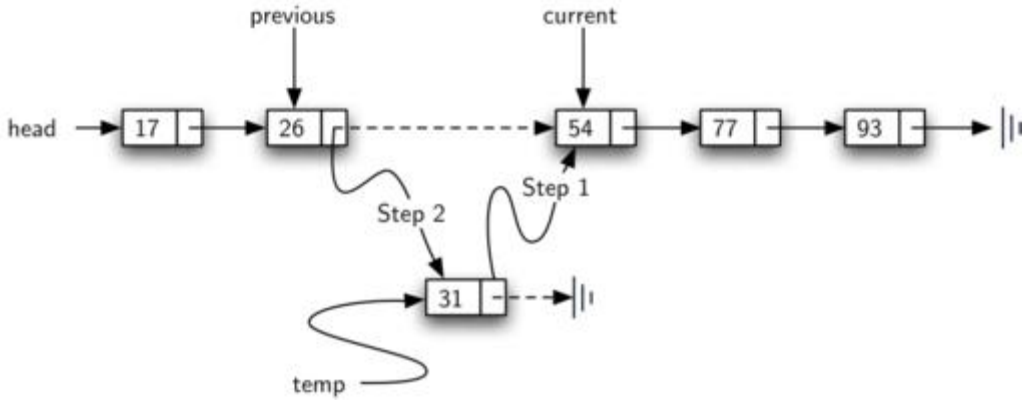


Рисунок 17: Додавання елемента до впорядкованого пов'язаного списку

Як ми вже бачили для невпорядкованих списків, тут знадобиться додаткове посилання (previous), оскількиcurrentне зможе надати доступ до вузла, який потрібно змінити.*Лістинг 10*показує закінчений методadd. Рядки 2-3 встановлюють два зовнішні посилання, а рядки 9-10 знову дозволяютьpreviousслідувати на один вузол післяcurrentпід час кожної ітерації. Умова рядку 5 дозволяє ітераціям продовжуватися доти, доки залишаються непереглянуті вузли і значення поточного не перевищує шукане. Неприємний випадок - коли ітерація зазнає невдачі - означає, що ми знайшли місце для нового вузла.

Залишок методу завершує двокроковий процес, показаний на*малюнку 17*. Оскільки для елемента був створений новий вузол, то залишається єдине питання: куди його буде додано - на початок або в середину списку? Для відповіді на нього знову використовуєтьсяprevious == None.

Лістинг 10

```
def add(self,item):
    current= self.head
    previous= None
    stop= False
    whilecurrent!= None and notstop:
        ifcurrent.getData()>item:
            stop= True
        else:
            previous=current
            current=current.getNext()

    temp=Node(item)
    ifprevious== None:
        temp.setNext(self.head)
        self.head=temp
    else:
        temp.setNext(current)
        previous.setNext(temp)
```

КласOrderedListз реалізованими методами можна знайти в ActiveCode 4.

Щоб проаналізувати складність операцій для пов'язаних списків, нам потрібно з'ясувати, чи вони вимагають обхід. Розглянемо пов'язаний список із nn вузлів МетодisEmptyмає O(1)O(1), оскільки потрібен лише один крок, щоб перевірити, чи посилаєтьсяheadнаNone. З іншого боку,sizeзавжди вимагає nn кроків, оскільки немає способу дізнатися кількість вузлів у зв'язаному списку, не обійшовши його від голови до кінця. Таким чином,sizeмає O(n)O(n). Додавання елемента до невпорядкованого списку завжди буде O(1)O(1), адже ми просто поміщаємо новий вузол у голову зв'язаного списку. Однак,searchiremove, а так самоaddдля впорядкованих списків вимагають процесу обходу. Хоча в середньому їм потрібно обійти лише половину списку, всі вони мають O(n)O(n) - виходячи з найгіршого випадку з обробкою кожного вузла у списку.
Ви також можете помітити, що представлення цієї реалізації відрізняється від існуючої раніше, що дається для списків Python. Це передбачає, що там списки ґрунтуються не на пов'язаній моделі, а на чомусь ще. Справді, в основі існуючої реалізації списків у Python лежить поняття масиву.

Лекція 6. Рекурсія

Рекурсія- це спосіб вирішення завдань за допомогою розбиття їх на менші і менші підзавдання до тих пір, поки результат не зможе бути знайдений тривіальним способом. Зазвичай рекурсія включає виклик функцією себе. Хоча з першого погляду це непомітно, але рекурсія дозволяє нам писати елегантні розв'язання задач, які інакше було б складно запрограмувати.

Почнемо наше дослідження з простого завдання, рішення для якого ви вже знаєте і без використання рекурсії. Припустимо, ви хочете підрахувати суму списку чисел [1,3,5,7,9][1,3,5,7,9]. Рішення у вигляді ітеративної функції показано в*ActiveCode 1*. Вона використовує змінноуtheSumяк акумулятор, чиє початкове значення дорівнює нулю і до якого додаються всі числа зі списку.

```
def listsum(numList):
    theSum = 0
    for i in numList:
        theSum = theSum + i
    return theSum

print(listsum([1,3,5,7,9]))
```

Уявіть на хвилину, що ви не можете використовувати циклиwhileабоfor. Як підрахувати суму чисел у списку? Якби ви були математиками, то могли б почати з того, що додавання – це функція, яка приймає два параметри (пару чисел). Щоб перевизначити завдання від додавання значень у списку до складання пар чисел, ми перепишемо список у вигляді виразу з повною розстановкою дужок. Виглядатиме воно приблизно так:

((((1+3)+5)+7)+9)((((1+3)+5)+7)+9)

У принципі, дужки можна розставити і у зворотному порядку:

(1+(3+(5+(7+9))))(1+(3+(5+(7+9)))))

Зверніть увагу, що самий внутрішній вираз у дужках - (7+9)(7+9) - це завдання, яке можна вирішити без використання циклів або якихось спеціальних конструкцій. Фактично ми можемо використовувати наступну послідовність спрощень для обчислення підсумкової суми:

total= (1+(3+(5+(7+9))))total= (1+(3+(5+16)))total= (1+(3+21))total= (1+ 24) total = 25 total = (1 + (3 + (5 + (7 + 9)))) total = (1 + (3 + (5 + 16))) total = (1 + (3 +21)) total = (1+24) total = 25

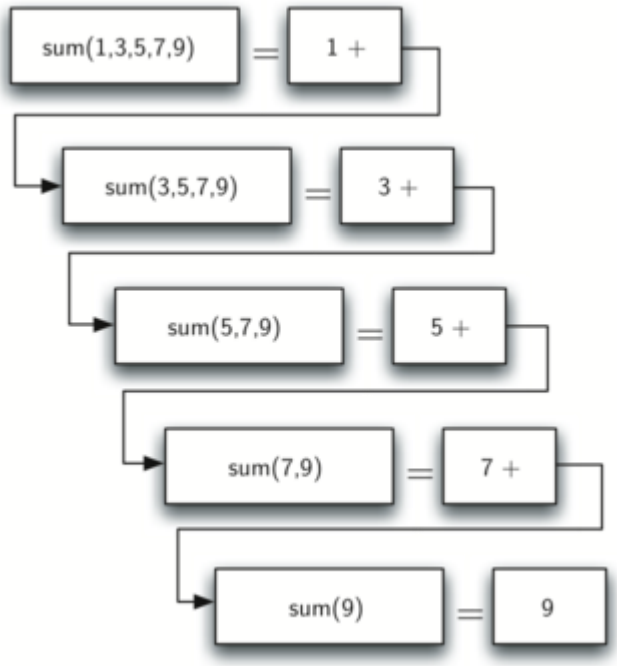
Залишилося лише переписати цю ідею у вигляді програми на Python. Для початку, давайте заново сформулюємо завдання додавання в термінах списків Python. Ми можемо сказати, що сума спискуnumList- це сума першого його елемента (numList[0]) та вже порахованої суми залишку списку (numList[1:]). У вигляді функції виглядає так:

listSum(numList)=first(numList)+listSum(rest(numList))listSum(numList)=first(numList)+ listSum(rest(numList))

У цьому виразі first(numList)first(numList) повертає перший елемент списку, а rest(numList)rest(numList) - список з чисел, що залишилися. Це легко виявляється у коді (див.*ActiveCode 2*):

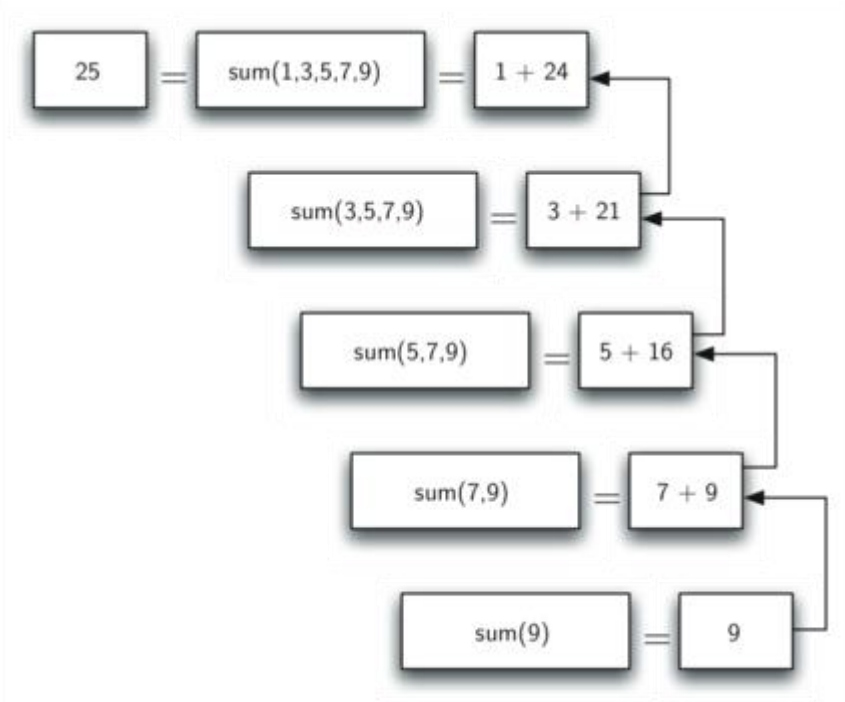
```
def listsum(numList):
  if len(numList) == 1:
      return numList[0]
  else:
      return numList[0] + listsum(numList[1:])

print(listsum([1,3,5,7,9]))
```

З цього лістингу можна отримати кілька ключових моментів. По-перше, у рядку 2 ми перевіряємо, чи список є одиничним. Ця перевірка має вирішальне значення і є лазівкою з функції. Знаходження суми одиничного списку – тривіальне завдання. Нею буде значення єдиного його елемента. По-друге, у рядку 5 функція викликає саму себе! Ось чому ми називаємо алгоритмlistsumрекурсивним. Рекурсивна функція - це функція, що викликає саму себе.
на*малюнку 1*показано послідовність рекурсивних викликів, які потрібні для підрахунку суми списку [1,3,5,7,9][1,3,5,7,9]. Ви можете думати про неї як про серію спрощень. Щоразу, коли ми робимо рекурсивний виклик, ми вирішуємо завдання меншого розміру до того часу, поки досягнемо точки, де її не можна буде зменшити.



Малюнок 1: Послідовність рекурсивних дзвінків для додавання списку чисел.
Коли ми досягаємо точки максимального спрощення завдання, то починаємо збирати разом шматочки розв'язання кожної з маленьких підзавдань доти, доки вони не зіллються у вирішення початкового завдання.*Малюнок 2*показує операції додавання, які виконуються під час роботиlistsumу зворотному напрямку за послідовністю

дзвінків. Колиlistsumповерне відповідь найвищого завдання, ми матимемо підсумкове рішення.



Малюнок 2: Послідовність рекурсивних повернень для складання списку чисел.

Подібно до робіт Азімова, всі рекурсивні алгоритми повинні підкорятися трьом важливим законам:

1. Рекурсивний алгоритм повинен мати основний випадок.
2. Рекурсивний алгоритм повинен змінювати свій стан і рухатися до базового випадку.
3. Рекурсивний алгоритм має викликати себе.

Давайте розглянемо кожен із цих законів докладніше і знайдемо приклади їх застосування в алгоритміlistsum. Перший – базовий випадок – це умова, яка дозволяє алгоритму зупинити рекурсію. Він являє собою завдання настільки мале, що його можна вирішити без застосування будь-яких додаткових засобів. Для алгоритмуlistsumбазовий випадок – це список довжиною 1.

Щоб дотриматись другого закону, ми повинні організувати зміни стану таким чином, щоб алгоритм рухався у напрямку базового випадку. Зміна стану означає модифікацію даних, використовуваних алгоритмом. Зазвичай обсяг даних, поданих у задачі, зменшується будь-яким чином. В алгоритміlistsumнаша первісна структура даних - це список, тому слід сфокусувати зусилля зі зміни стан саме на ньому. Оскільки базовий випадок – список одиничної довжини, природним прогресом у його бік стане скорочення списку. Це точно те, що відбувається в рядку 5*ActiveCode 2*, коли ми викликаємоlistsumз більш коротким списком.

Останній закон у тому, що алгоритм має викликати себе. Власне, у цьому полягає визначення рекурсії. Рекурсія - бентежна концепція для багатьох новачків-програмістів. Як програміст-початківець ви вчили, що функції хороші тим, що дозволяють взяти велике завдання і розбити її на дрібніші частини. Їх можна вирішити, написавши функцію кожної. Коли ж ми говоримо про рекурсію, то

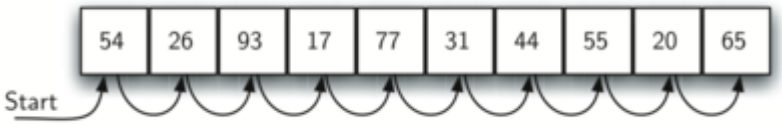
здається, що ми збираємося зациклитися. У нас є завдання, яке вирішується за допомогою функції, але для цього їй необхідно викликати саму себе! Однак, логічно тут нічого не замкнуто: логіка рекурсії в елегантному вираженні розв'язання задачі за допомогою розбиття її на дрібніші та легші підзавдання.

Лекція 7. Алгоритми пошуку.

Послідовний пошук

Коли елементи даних зберігаються колекцією у вигляді списку, ми говоримо, що між ними є лінійні або послідовні відносини. Кожен елемент зберігається на певній позиції щодо інших. У списках Python вона задається індексом цього елемента. Оскільки значення індексів упорядковано, ми маємо можливість послідовно проходити ними. Цей процес призводить до нашої першої пошукової техніки.**послідовному пошуку**.

*Малюнок 1*демонструє, як працює такий пошук. Починаючи з першого елемента у списку, ми просто рухаємося від значення до значення, дотримуючись внутрішнього порядку послідовності, поки або не знайдемо те, що шукаємо, або не досягнемо останнього елемента. Другий випадок означає, що послідовність не містить.



Малюнок 1: Послідовний пошук у списку цілих чисел

Реалізація цього алгоритму на Python показана в*CodeLens 1*. Функція вимагає список та елемент, який ми шукаємо, а повертає логічне значення, яке говорить про його присутність. Бульова змінна`found`ініціалізується значенням`False`, і якщо елемент виявляємо у списку, їй присвоюється`True`.

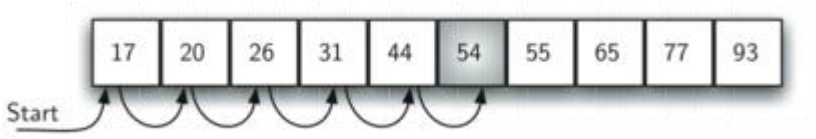
```
def sequentialSearch(alist, item):
    pos = 0
    found = False

    while pos < len(alist) and not found:
        if alist[pos] == item:
            found = True
        else:
            pos = pos+1

    return found

testlist = [1, 2, 32, 8, 17, 19, 42, 13, 0]
print(sequentialSearch(testlist, 3))
print(sequentialSearch(testlist, 13))
```

Для аналізу алгоритму пошуку необхідно визначитися з базовим блоком обчислень. Нагадаємо, що зазвичай ним є поширений крок, який необхідно повторювати, щоб вирішити завдання. Для пошуку є сенс підраховувати кількість вироблених порівнянь. Кожне порівняння може (або може) виявити шуканий елемент. На додаток, ми зробимо ще одне припущення. Список буде невпорядкований, тобто. елементи розміщуються у ньому випадковим чином. Інакше кажучи, можливість знайти шукане цій позиції однакова всім індексів.
Якщо елемент над списку, єдиний спосіб дізнатися звідси - порівняти його з усіма наявними значеннями. Якщо є nn елементів, то послідовний пошук вимагатиме nn порівнянь, щоб відкрити відсутність елемента. Якщо ж елемент у списку є, то аналіз вже не такий очевидний. Взагалі, є три різні сценарії. У кращому випадку ми знайдемо елемент на першій позиції, яку розглянемо, - на самому початку списку. Нам буде потрібно лише одне порівняння. У гіршому випадку ми будемо шукати елемент, поки не дійдемо до останнього - n-го - порівняння.
Що можна сказати про середній випадок? У ньому ми знайдемо елемент приблизно на середині списку. що складність послідовного пошуку дорівнює O(n)O(n).*Таблиця 1*підсумовує результати цих міркувань:

Таблиця 1: Порівняння, які використовуються у послідовному пошуку в неупорядкованому списку

Варіант	Найкращий випадок	Найгірший випадок	Усереднений випадок
Елемент є	11	nn	n2n2
Елемент відсутня	nn	nn	nn

Раніше ми припускали, що елементи в нашій послідовності розміщені довільно, тому між ними немає відносного впорядкування. Що станеться з послідовним пошуком, якщо елементи будуть упорядковані.
Припустимо, список значень був побудований таким чином, що вони розташовані в ньому за зростанням від найменшого до найбільшого. Якщо елемент є у списку, то ймовірність для нього бути на будь-якій з nn позицій така ж, як і раніше. Нам, як і раніше, необхідна та ж кількість порівнянь для пошуку. Однак для випадку, коли елемент у списку відсутній, у нас є невелика перевага.*Малюнок 2*показує процес пошуку алгоритмом числа 50. Зверніть увагу, що елементи порівнюються послідовно аж до 54. У цей момент у нас є якась додаткова інформація. Не тільки те, що 54 - не той елемент, який ми шукаємо, але й що елементи за 54 однозначно не підійдуть, оскільки список відсортований. У цьому випадку алгоритму немає сенсу йти далі і переглядати все до кінця, щоб сказати, що шукане не знайдено. Він негайно зупиниться.*CodeLens 2*показує цей варіант функції послідовного пошуку.



Малюнок 2: Послідовний пошук у списку цілих чисел.

```
def orderedSequentialSearch(alist, item):
2                                   pos = 0
3                                   found = False
4                                   stop = False
5                                   while pos < len(alist) and not found and not stop:
6                                   if alist[pos] == item:
7                                   found = True
8                                   else:
9                                   if alist[pos] > item:
10                                  stop = True
11                                  else:
12                                  pos = pos+1
13
14                                  return found
15
16                                  testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
17                                  print(OrderedSequentialSearch(testlist, 3))
18                                  print(OrderedSequentialSearch(testlist, 13))
```

*Таблиця 2*підсумовує ці результати. Зауважте, що в кращому випадку ми виявимо, що елемент не в списку, переглянувши лише одне значення. У середньому ми це знатимемо, переглянувши n2n2 елементів. Однак, ця методика, як і раніше, має O(n)O(n). Резюме: Послідовний пошук покращується при впорядкуванні списку тільки для випадку, коли в ньому відсутній елемент.

Таблиця 2: Кількість порівнянь під час послідовного пошуку в упорядкованому списку

Таблиця 2: Кількість порівнянь під час послідовного пошуку в упорядкованому списку

Елемент є	11	nn	n2n2
Елемент відсутня	11	nn	n2

Бінарний пошук

Існує можливість максимально використовувати переваги впорядкованого списку, якщо порівняти з розумом. У послідовному пошуку, коли ми порівнюємо перший елемент, може бути доn−1n−1елемента, які потрібно ще переглянути, якщо перший – не те, що ми шукаємо. Замість того, щоб шукати у списку по порядку,**бінарний пошук**починає перевіряти елементи з середини. Якщо це те, що нам потрібно, все готове. Якщо ні, то можна використовувати впорядковану природу списку, виключивши половину елементів, що залишилися. Коли шукане більше, ніж елемент, що знаходиться посередині, то з подальшого розгляду можна сміливо виключити “нижню” частину, що містить менші величини. Шукане значення (якщо воно є у списку) перебуватиме у верхній частині.

Це міркування можна повторити для половини, що залишилася. Почнемо із середнього елемента та порівняємо його з шуканим. Знову ми або виявимо його, або розіб'ємо список навпіл, викресливши половину простору для пошуку.*Малюнок 3*показує, як цей алгоритм швидко знаходить значення 54. Повністю функція показана в*CodeLens 3*.

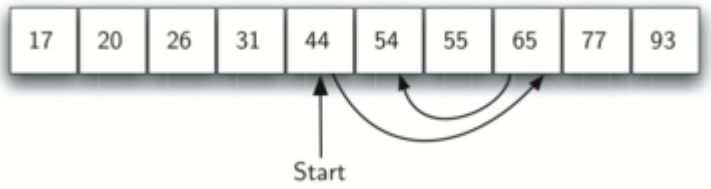


Рисунок 3: Бінарний пошук у впорядкованому списку цілих чисел

```
def binarySearch(alist,
item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
    midpoint = (first + last)//2
    if alist[midpoint] == item:
```

```
        found = True
    else:
        if item < alist[midpoint]:
            last = midpoint-1
        else:
            first = midpoint+1

    return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

Перед тим, як перейти до аналізу, відзначимо, що цей алгоритм - чудовий приклад стратегії "розділяй і володарюй". "Поділяй і владарюй" означає, що завдання ділиться на маленькі ділянки, які вирішуються якимось чином, а потім відповіді компонуються в підсумковий результат. Коли ми робимо бінарний пошук у списку, то на початку перевіряємо середній елемент. Якщо шукане менше його, можна просто виконати бінарний пошук для лівої частини оригінального списку. Аналогічно, якщо більше шукане, то ми проводимо бінарний пошук для правої половини. В обох випадках є рекурсивний виклик функції бінарного пошуку на меншому списку.*CodeLens 4*демонструє цю рекурсивну версію.

```
def binarySearch(alist,
item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist)//2
        if alist[midpoint]==item:
            return True
        else:
            if item<alist[midpoint]:
                return binarySearch(alist[:midpoint],item)
            else:
```

```
    return binarySearch(alist[midpoint+1:],item)

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

Аналіз бінарного пошуку

Для аналізу алгоритму бінарного пошуку, нам необхідно згадати, що кожне порівняння виключає з розгляду близько половини елементів, що залишилися. Яка максимальна кількість порівнянь, які вимагатиме алгоритм для перевірки списку? Якщо ми починаємо зnелементів, то після першого порівняння відкинеться близько$\frac{n}{2}$елементів. Після другого – порядку.$\frac{n}{4}$. Потім$\frac{n}{8}$,$\frac{n}{16}$і таке інше. Скільки разів ми будемо розділяти список?*Таблиця 3*допоможе знайти відповідь.

Таблиця 3: Табличний аналіз для бінарного пошуку

Порівняння	**Приблизна кількість відкинутих елементів**
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
...	
i	$\frac{n}{2^i}$

Процес розбиття закінчиться на списку, що містить лише один елемент. Їм може виявитися чи не виявитись те, що ми шукаємо. У будь-якому випадку справа зроблена. Кількість порівнянь, необхідних до потрапляння до цієї точки дорівнює ii, де $\frac{n}{2^i}=1$$\frac{n}{2^i}=1$. Розв'язавши рівняння для ii, отримуємо $i=\log n$$i=\log n$. Максимальна кількість порівнянь є логарифмом щодо кількості елементів у списку. Таким чином, бінарний пошук буде $O(\log n)$$O(\log n)$.

Але потрібно розібратися з ще одним додатковим питанням щодо аналізу. У рекурсивному рішенні, показаному вище, рекурсивний виклик`binarySearch(alist[:midpoint],item)`використовує оператор зрізу, щоб створити ліву половину списку, яку потім буде передано наступному виклику (аналогічно у разі правої половини). Аналіз, проведений вище, передбачає, що це операція займає константний час. Однак, ми знаємо, що оператор зрізу в Python взагалі $O(k)$$O(k)$. Таким чином, бінарний пошук, що використовує зріз, не виконуватиметься за суворо логарифмічний час. На щастя, це можна виправити, передаючи крім списку початковий та кінцевий індекси його елементів. Індекси можуть бути обчислені способом з*лістингу 3*. Реалізацію цієї ідеї ми залишаємо як вправу.

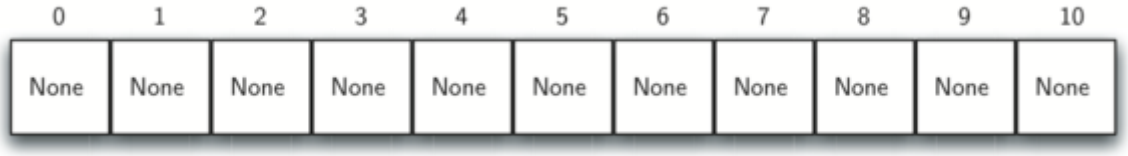
Незважаючи на те, що бінарний пошук в цілому краще послідовного, важливо зазначити, що при малих nn додаткові витрати на сортування менше не стають. відсортувати список один раз, а потім шукати в ньому багато разів, то ціна сортування значення не має. одиничне сортування може бути настільки затратним, що просто провести послідовний пошук від початку може стати найкращим рішенням.

Лекція 8. Хешування.

У попередніх розділах ми змогли вдосконалити наші алгоритми пошуку, використовуючи переваги інформації про те, де елементи зберігаються один до одного. Наприклад, знаючи, що список упорядкований, ми можемо шукати логарифмічний час, використовуючи бінарний алгоритм. У цьому розділі ми спробуємо піти ще на крок далі: побудувати таку структуру даних, у якій можна буде здійснювати пошук за часO(1)O(1). Цю концепцію називають**хешуванням**.

Тепер нам треба знати більше, ніж просто розташування елемента, коли ми шукаємо його в колекції. Якщо кожен елемент знаходиться там, де йому слід бути, то пошук може використовувати лише порівняння для виявлення присутності шуканого. Однак далі ми побачимо, що це, як правило, не єдиний вихід.

Хеш-таблиця- це колекція елементів, які зберігаються таким чином, щоб їх пізніше було легко знайти. Кожна позиція у хеш-таблиці (часто званас**лотом**) може містити власне елемент і ціле число, що починається з нуля. Наприклад, у нас є слот 0, слот 1, слот 2 і таке інше. Спочатку хеш-таблиця не містить елементів, тому кожен з них порожній. Ми можемо зробити реалізацію хеш-таблиці, використовуючи список, в якому кожен елемент ініціалізований спеціальним значенням PythonNone.*Малюнок 4*демонструє хеш-таблицю розміромm=11m=11. Іншими словами, в ній єmmслотів, пронумерованих від 0 до 10



Малюнок 4: Хеш-таблиця з 11 порожніми слотами

Зв'язок між елементом та слотом, у який він кладеться, називається хеш-функцією. Вона приймає будь-який елемент із колекції та повертає ціле число з діапазону імен слотів (від 0 до m−1m−1). Припустимо, що у нас є набір цілих чисел 54, 26, 93, 17, 77 і 31. Наша перша хеш-функція, іноді звана "методом залишків", просто бере елемент і ділить його на розмір таблиці, повертаючи залишок як хеш- значення (h(item)=item%11h(item)=item%11). У*таблиці 4*представлені всі хеш-значення чисел із нашого прикладу. Зверніть увагу: метод залишків (модульна арифметика) зазвичай присутня у тій чи іншій формі у всіх хеш-функціях, оскільки результат повинен лежати в діапазоні імен слотів.

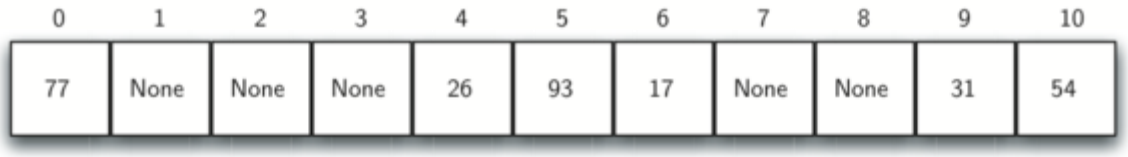
Таблиця 4: Проста хеш-функція, яка використовує залишки

Елемент	Хеш-значення
54	10
26	4
93	5

Таблиця 4: Проста хеш-функція, яка використовує залишки

Елемент	Хеш-значення
17	6
77	0
31	9

Оскільки хеш-значення можуть бути пораховані, ми можемо вставити кожен елемент у хеш-таблицю на певне місце, як це показано на*малюнку 5*. Зверніть увагу, що тепер зайнято 6 із 11 слотів. Це називається**фактором завантаження**і зазвичай позначаєтьсяλ=numberofitemstablesizeλ=numberofitemstablesize. У цьому прикладіλ=611λ=611.



Малюнок 5: Хеш-таблиця із шістьма елементами

Тепер, коли ми хочемо знайти елемент, ми просто використовуємо хеш-функцію, щоб обчислити ім'я слота елемента, а потім перевірити по таблиці його наявність. Ця операція пошуку має O(1)O(1), оскільки на обчислення хеш-значення потрібен константний час, як і на перехід за знайденим індексом. Якщо все знаходиться там, де йому належить, ми отримуємо алгоритм пошуку за константний час.

Можливо, ви вже помітили, що така техніка працює тільки якщо кожен елемент відображається на унікальну позицію в хеш-таблиці. 11==0) А оскільки 77 теж має хеш-значення 0, то у нас проблеми. хеш-функцією два або більше елементів повинні мати один слот. Це називається колізією.

Хеш-функції

Для заданої колекції елементів хеш-функція, що зв'язує кожен із них з унікальним слотом, називається ідеальною хеш-функцією. Якщо ми знаємо, що жоден елемент колекції ніколи не зміниться, можливо створити ідеальну хеш-функцію (див. вправи, щоб дізнатися про це більше). На жаль, для довільного набору елементів немає систематичного способу сконструювати ідеальну хеш-функцію. На щастя, для ефективної роботи вона нам і не потрібна.

Один із способів завжди мати ідеальну хеш-функцію полягає у збільшенні розміру хеш-таблиці таким чином, щоб у ній могло бути розміщене кожне з можливих значень елементів. Таким чином, гарантується унікальність слотів. Хоча такий підхід практичний для малого числа елементів, при зростанні їх кількості він перестає бути здійсненним. Наприклад, для дев'ятизначних індексів соціального страхування потрібно близько мільярда слотів. Навіть якщо ми захочемо лише зберігати дані для класу з 25 студентів, то витратимо на цю жахливу кількість пам'яті.

Наша мета: створити хеш-функцію, яка б мінімізувала кількість колізій, легко

вважалася і рівномірно розподіляла елементи в хеш-таблиці. Існує кілька поширених способів розширити найпростіший метод залишків. Розглянемо деякі з них.

Метод згорткидля створення хеш-функцій починає з поділу елемента на складові однакової величини (крім останнього, який може мати різний розмір). Ці шматочки складаються разом і дають результуючу хеш-значення. Наприклад, якщо наш елемент – телефонний номер 436-555-4601, то ми можемо взяти цифри та робити їх на групи по дві (43, 65, 55, 46, 01). Після складання 43+65+55+46+0143+65+55+46+01 ми отримаємо 210. Якщо припустити, що хеш-таблиця має 11 слотів, потрібно виконати додатковий крок, поділивши це число на 11 і взявши залишок. В даному випадку 210% 11210% 11 дорівнює 1, так що телефонний номер 436-555-4601 хешується в слот 1. Деякі методи згортки йдуть на крок далі і перед складанням перевертають кожен шматочок розбиття. Наприклад вище ми отримали 43+56+55+64+01=21943+56+55+64+01=219, що дає 219 % 11=10219 % 11=10.

Інша числова техніка до створення хеш-функцій називається методом середніх квадратів. Спочатку значення елемента зводиться в квадрат, а потім з цифр, що вийшли в результаті, виділяється деяка порція. Наприклад, якщо елемент дорівнює 44, ми раніше обчислимо 442=1,936442=1,936. Виділивши дві середні цифри (93) та виконавши крок отримання залишку, ми отримаємо 5 (93 % 1193 % 11).*Таблиця 5*показує елементи, до яких застосували обидва методи: залишків та середніх квадратів.

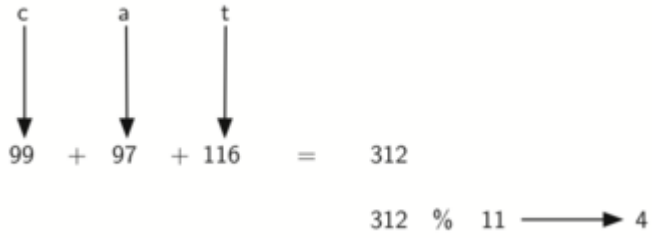
Таблиця 5: Порівняння методів залишків та середніх квадратів

Елемент	Залишок	Середній квадрат
54	10	3
26	4	7
93	5	9
17	6	8
77	0	4
31	9	6

Ми також можемо створити хеш-функцію для символьних елементів (наприклад, рядків). Слово “cat” можна як послідовність кодів його букв.

```
>>>ord('c')
99
>>>ord('a')
97
>>>ord('t')
116
```

Потім можна взяти ці три коди, скласти їх і використати метод залишків, щоб отримати хеш-значення (див.*малюнок 6*).*Лістинг 1*демонструє функціюhash, що приймає рядок та розмір таблиці та повертає хеш-значення з діапазону від 0 дотablesize-1.

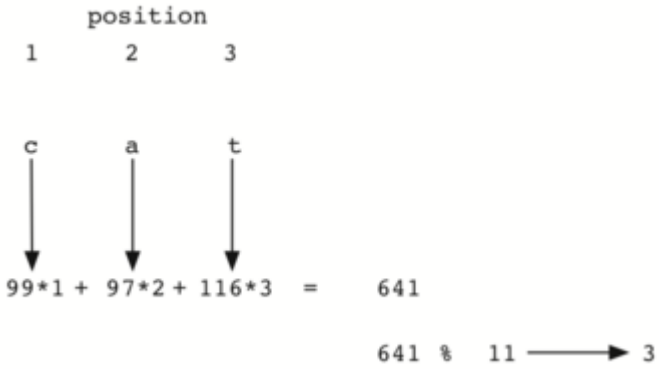


Малюнок 6: Хешування рядка з використанням кодів символів

Лістинг 1

```
def hash(astring,tablesize):
   sum = 0
   forposin range(len(astring)):
      sum = sum + ord(astring[pos])

   return sum%tablesize
```

Цікаве спостереження: коли ми використовуємо цю хеш-функцію, анаграми завжди матимуть однакову хеш-значення. Щоб виправити це, слід використовувати позицію символу як вагу.*Малюнок 7*показує один з варіантів використання позиційного значення як ваговий фактор. Модифікацію функціїhashми залишаємо як вправу.



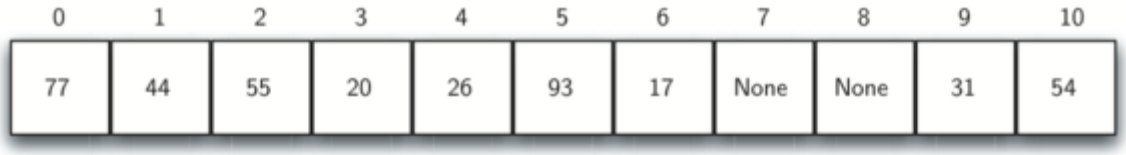
Малюнок 7: Хешування рядка з використанням кодів символів та ваг

Ви можете вигадати інші числові способи обчислення хеш-значень для елементів колекції. Важливо лише пам'ятати, що ефективна хеш-функція не повинна бути домінуючою частиною процесів зберігання та пошуку. Якщо вона дуже складна, то вимагає багато роботи на обчислення імені слота. У цьому випадку простіше було б використовувати послідовний чи бінарний пошук, описані вище. Таким чином, сама ідея хешування зазнає поразки.

Дозвіл колізій

Повернемося до проблеми колізій. Коли два елементи хешуються в один слот, нам потрібний систематичний метод розміщення в хеш-таблиці другого елемента. Цей

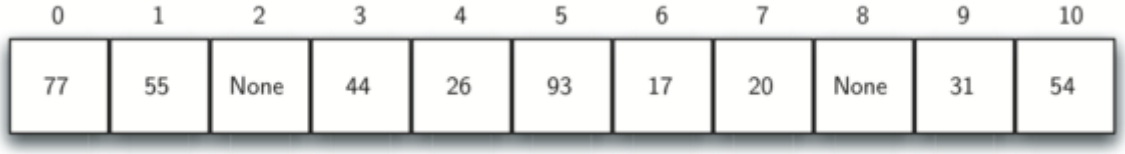
процес називається дозволом колізії. Як ми стверджували раніше, якщо хеш-функція ідеальна, то колізії ніколи не станеться. Однак, оскільки часто такий стан справ неможливий, вирішення колізій стає важливою частиною хешування.
Одним із методів вирішення колізій є перегляд хеш-таблиці та пошук іншого вільного слота для розміщення в ньому елемента, що створив проблему. Простий спосіб зробити це - почати з оригінальної позиції хеш-значення та переміщатися по слотах певним чином доти, доки не буде знайдено порожнім. Зверніть увагу: нам може знадобитися повернутися назад до першого слоту (циклічно), щоб охопити хеш-таблицю цілком. Цей процес вирішення колізій називається відкритою адресацією, оскільки намагається знайти наступний вільний слот (або адресу) у хеш-таблиці. Систематично відвідуючи кожен слот по одному разу, ми діємо відповідно до техніки відкритої адресації, що називається лінійним пробуванням.
*Малюнок 8*показує розширений набір цілих елементів після застосування простої хеш-функції методу залишків (54,26,93,17,77,31,44,55,20). У*таблиці 4*вище зібрані хеш-значення оригінальних елементів, а на*малюнку 5*представлений початковий вміст хеш-таблиці. Коли ми намагаємося помістити 44 у слот 0, виникає колізія. При лінійному пробуванні ми послідовно - слот за слотом - переглядаємо таблицю, доки знайдемо відкриту позицію. У разі це виявився слот 1.
Наступного разу 55, яке має розміститися в слоті 0, буде покладено в слот 2 наступну незайняту позицію. Останнє значення 20 хешується в слот 9. Але оскільки він зайнятий, ми робимо лінійне пробування. Ми відвідуємо слоти 10, 0, 1, 2 і, нарешті, знаходимо порожній слот на позиції 3.



Малюнок 8: Дозвіл колізій шляхом лінійного пробування
Оскільки ми побудували хеш-таблицю за допомогою відкритої адресації (або лінійного пробування), важливо використовувати той самий метод під час пошуку елемента. Припустимо, ми хочемо знайти число 93. Врахування його хеш-значення дасть 5. Виявивши в п'ятому слоті 93 ми повернемоTrueАле якщо ми шукаємо 20? Тепер хеш-значення дорівнює 9, а слот 9 містить 31. Не можна просто повернути.False, оскільки тут могла бути колізія. Тож ми змушені провести послідовний пошук, починаючи з десятої позиції, яка закінчиться, коли знайдеться число 20 або порожній слот.
Недоліком лінійного пробування є його схильність до кластеризації: елементи таблиці групуються. Це означає, що якщо виникає багато колізій з одним хеш-значенням, то слоти, що його оточують, при лінійному пробуванні будуть заповнені. Це почне впливати на вставку інших елементів, як ми спостерігали вище при спробі вставити в таблицю число 20. У підсумку, кластер значень, що хешуються в 0, має бути пропущений, щоб знайти вакантне місце. Цей кластер показано на*малюнку 9*.

Малюнок 9: Кластер елементів для слота 0

Одним із способів мати справу з кластеризацією є розширення лінійного пробування таким чином, щоб замість послідовного пошуку наступного вільного місця ми пропускали слоти, отримуючи таким чином більш рівномірний розподіл елементів колізії. Потенційно це зменшить кластеризацію, що виникає.*Малюнок 10*показує елементи після дозволу колізій з використанням пробування плюс 3. Це означає, що у разі колізії, ми розглядаємо кожен третій слот до того часу, поки знайдемо порожній.



Малюнок 10: Дозвіл колізій з використанням методики плюс 3

Загальна назва для такого процесу пошуку іншого слота після колізії - повторне хешування. rehash(pos)=(pos+1)%sizeoftablerehash(pos)=(pos+1)%sizeoftable. (pos+3)%sizeoftable.

rehash(pos)=(pos+skip)%sizeoftablerehash(pos)=(pos+skip)%sizeoftable. виявиться невикористаною. Для забезпечення цієї умови часто передбачається, що розмір таблиці є простим числом.

Ще одним варіантом лінійного пробування є квадратичне пробування. Замість використання константного значення "перепустки", ми використовуємо повторну хеш-функцію, яка інкрементує хеш-значення на 1, 3, 5, 7, 9 і так далі. Це означає, якщо перше хеш-значення дорівнює hh, то наступними будуть h+1h+1, h+4h+4, h+9h+9, h+16h+16 тощо. Іншими словами, квадратичне пробування використовує перепустку, що складається з наступних один за одним повних квадратів.*Малюнок 11*демонструє значення нашого прикладу після використання цієї методики.

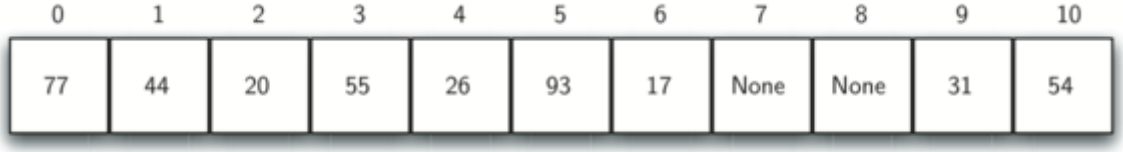


Рисунок 11: Дозвіл колізій за допомогою квадратичного пробування

Альтернативним методом вирішення проблеми колізій є дозвіл кожному слоту містити посилання на колекцію (або ланцюжок) значень. Ланцюжки дозволяють безлічі елементів займати ту саму позицію в хеш-таблиці. Чим більше елементів хешуються в одне місце, тим складніше знайти елемент у колекції.*Малюнок 12*показує, як елементи додаються до хеш-таблиці з використанням ланцюжків для вирішення колізій.

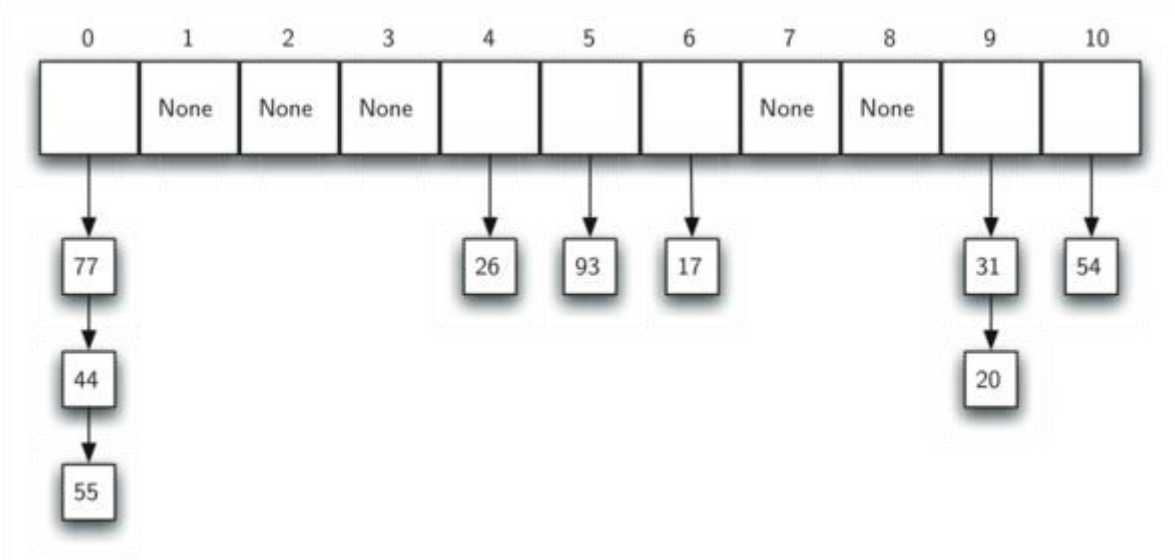


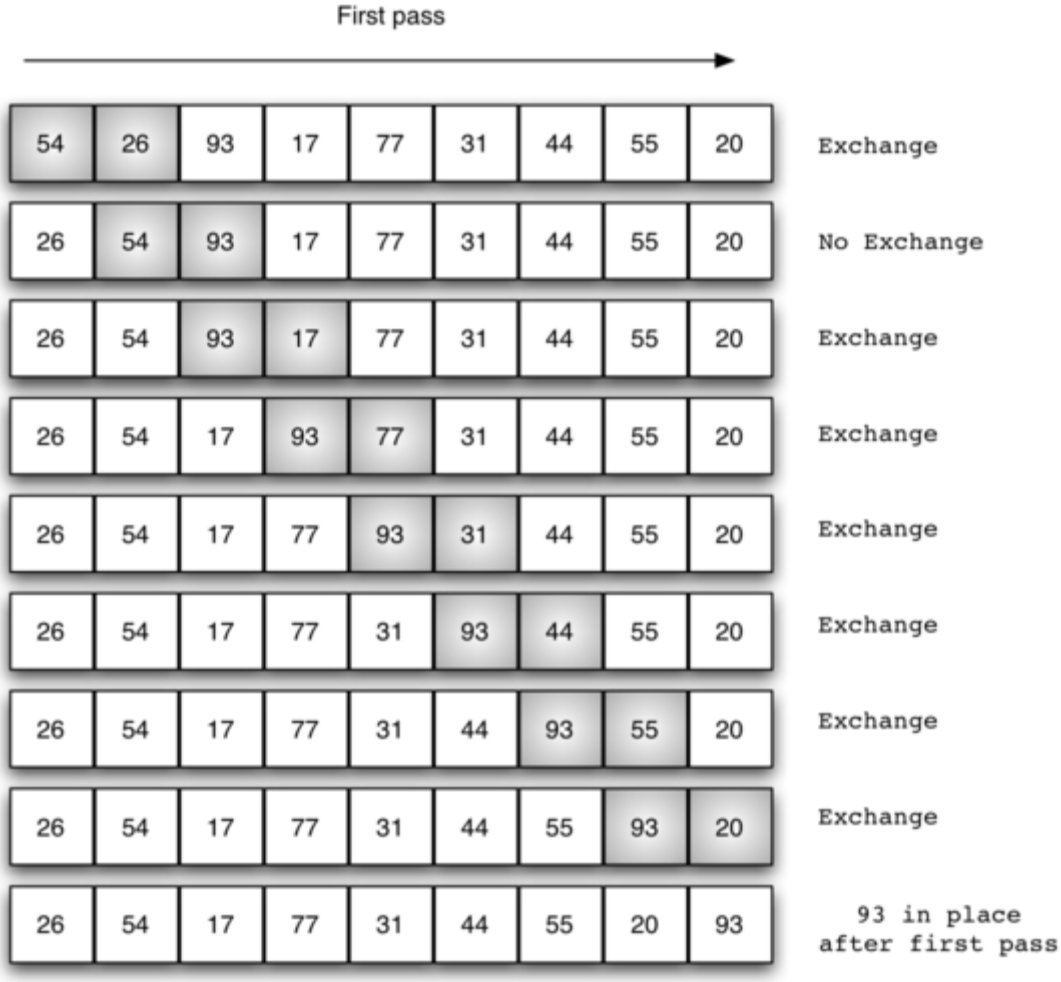
Рисунок 12: Дозвіл колізій за допомогою ланцюжків

Коли ми хочемо знайти елемент, ми використовуємо хеш-функцію для генерації номера слота, де він повинен розміщуватися. Оскільки кожен слот містить колекцію, ми використовуємо різні техніки пошуку, щоб визначити, чи представлений він у ній. Перевагою даного підходу є можливість отримати набагато менше елементів у кожному слоті, так що пошук буде більш ефективним.

Лекція 9. Алгоритми сортування.

Пухирцеве сортуванняробить за списком кілька проходів. Вона порівнює елементи, що стоять поруч, і змінює місцями ті з них, що знаходяться в неправильному порядку. Кожен прохід по списку містить наступне найбільше значення на його правильну позицію. По суті, кожен елемент "бульбашкою" спливає на своє місце.

*Малюнок 1*показує перший прохід бульбашкового сортування. Затінені елементи будуть порівнюватися для визначення чи правильному порядку вони стоять. Якщо списку nn елементів, то за перший прохід потрібно порівняти n−1n−1 пару. Важливо, що оскільки найбільше значення - частина пари, воно буде переміщатися вздовж списку до завершення проходу.



Малюнок 1:bubbleSort: перший прохід

На початок другого проходу найбільше значення стоїть своєму місці. Залишається число n−1n−1 для сортування, або n−2n−2 пари. Оскільки кожен прохід поміщає наступне найбільше значення на місце, то загальна кількість проходів дорівнює n−1n−1. Після завершення проходу n−1n−1 найменший елемент буде на вірній позиції без додаткових обчислень.*ActiveCode 1*демонструє функціюbubbleSortцілком. Вона приймає список як параметр і за необхідності змінює його за допомогою перестановок елементів.

Операція перестановки, іноді звана “обміном”, у Python дещо простіше, ніж у більшості інших мов програмування.

temp=alist[i]
alist[i]=alist[j]

alist[j]=temp
змінює місцями i-й та j-й елементи списку.
У Python можливе одночасне присвоєння. Операторa, b = b, aдасть той самий результат, як і два присвоєння, зроблених одночасно і той самий час (див.*малюнок 2*). З використанням одночасного присвоєння операція обміну займе лише один рядок.
Рядки 5-7*ActiveCode 1*проводять обмін ii-го та (i+1)(i+1)-го елементів, використовуючи триступінчасту операцію, описану вище.

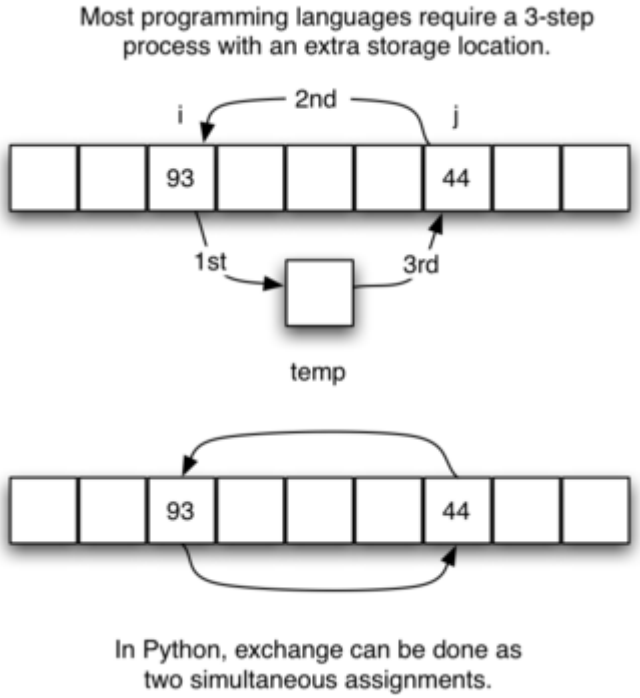


Рисунок 2: Перестановка місцями двох значень у Python
Наступний приклад ActiveCode демонструє завершену функціюbubbleSort, що працює зі списком, наведеним вище.

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

При аналізі бульбашкового сортування варто відзначити, що, незалежно від початкового порядку елементів, для списку з nn елементів буде зроблено n-1n-1 прохід.*Таблиця 1*показує кількість порівнянь кожного проходу. Загальна кількість - сума перших n−1n−1 чисел. Нагадаємо, що сума перших nn цілих дорівнює 12n2+12n12n2+12n. Сума перших n−1n−1 чисел дорівнює 12n2+12n−n12n2+12n−n,

або після скорочення 12n2−12n12n2−12n. Тобто. це, як і раніше, O(n2)O(n2) порівнянь. У кращому разі, коли список вже відсортовано, не буде зроблено жодної перестановки. Однак, для найгіршого випадку кожне порівняння спричинить обмін. У середньому обмін займе половину часу.

Таблиця 1: Порівняння при кожному проході бульбашкового сортування

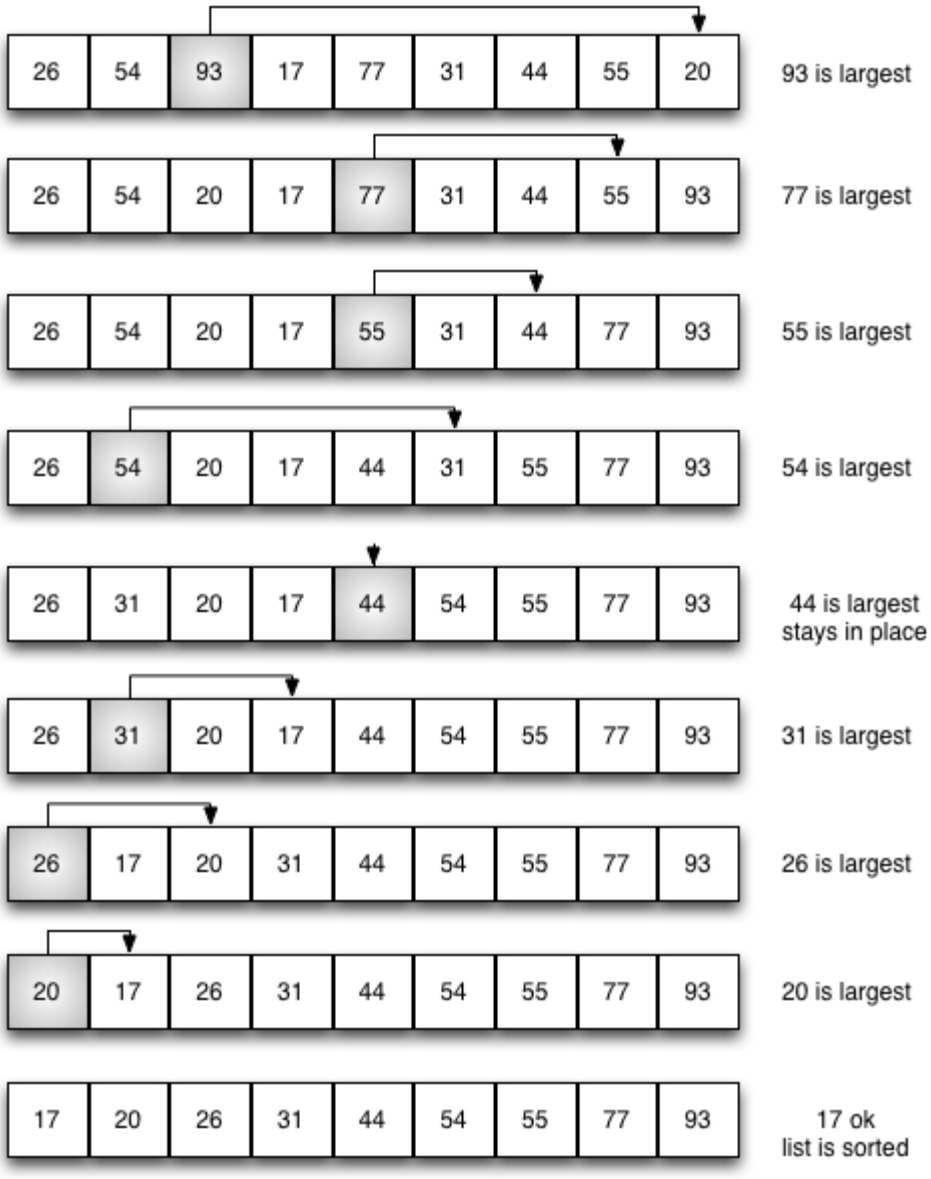
Прохід	Кількість порівнянь
1	n−1n−1
2	n−2n−2
3	n−3n−3
...	...
n−1n−1	11

Пухирцеве сортування часто розглядається як найбільш неефективний сортувальний метод, оскільки воно має переставляти елементи до того, як стане відома їхня остаточна позиція. Ці “порожні” операції обміну дуже затратні. Однак, оскільки бульбашкове сортування робить прохід по всій несортованій частині списку, вона вміє те, що не можуть більшість сортувальних алгоритмів. Зокрема, якщо під час проходу не було зроблено жодної перестановки, ми знаємо, що список вже відсортований. Таким чином, алгоритм може бути модифікований, щоб зупинятися раніше, якщо виявляє, що завдання виконане. Тобто. для списків, яким потрібно всього кілька проходів, бульбашкова сортування має перевагу, оскільки вміє розпізнати сортований список та зупинитися.*ActiveCode 2*демонструє цю модифікацію, яку часто називають короткою бульбашкою.

```
def shortBubbleSort(alist):
    exchanges = True
    passnum = len(alist)-1
    while passnum > 0 and exchanges:
       exchanges = False
       for i in range(passnum):
           if alist[i]>alist[i+1]:
               exchanges = True
               temp = alist[i]
               alist[i] = alist[i+1]
               alist[i+1] = temp
       passnum = passnum-1

alist = [20,30,40,90,50,60,70,80,100,110]
shortBubbleSort(alist)
```

print(alist)

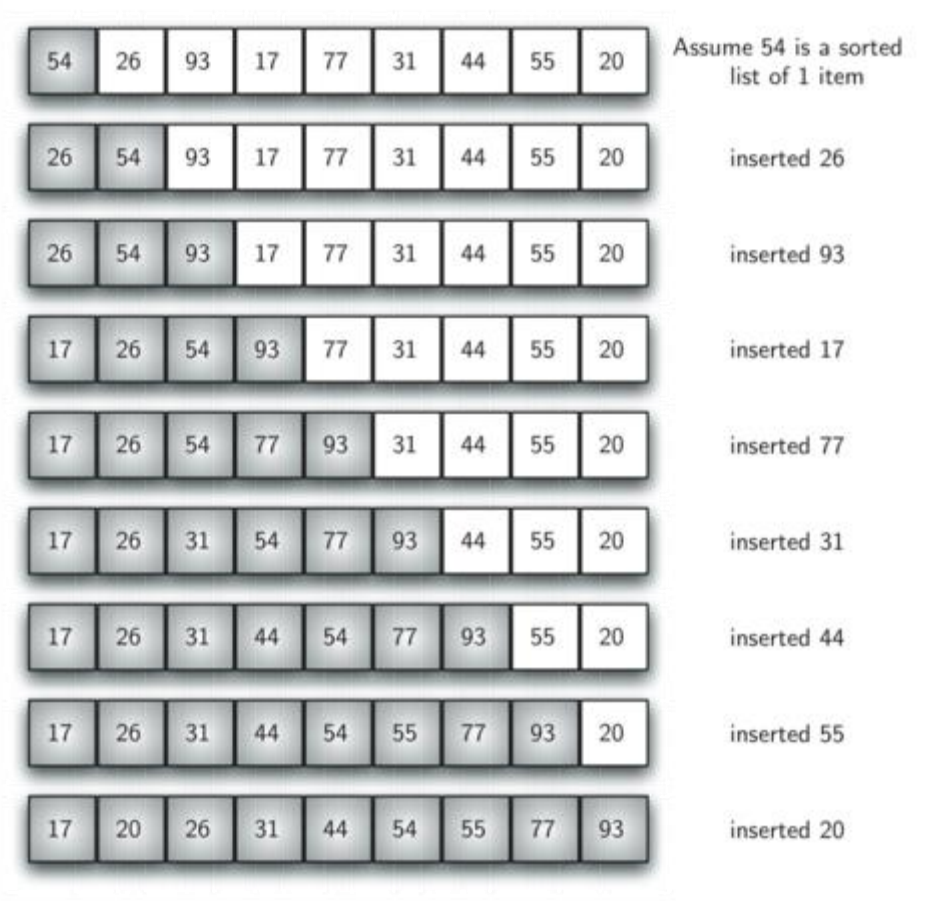
Сортування виборомпокращує бульбашкове сортування, здійснюючи лише один обмін за кожен прохід за списком. Щоб це зробити, вона шукає найбільший елемент і поміщає його на відповідну позицію. Як і для бульбашкового сортування, після першого проходу найбільший елемент знаходиться на правильному місці. Після другого - на місце стає наступний найбільший елемент. Процес триває, вимагаючи n−1n−1 прохід для сортування nn елементів, оскільки останній автоматично виявляється своєму місці.
*Малюнок 3*демонструє сортувальний процес повністю. На кожному проході вибирається найбільший з несортованих елементів і поміщається на відповідну позицію.



def selectionSort(alist):

```
    для fillslot в range(len(alist)-1,0,-1):
        positionOfMax=0
        для розміщення в межах(1,fillslot+1):
            if alist[location]>alist[positionOfMax]:
                positionOfMax = location

        temp = alist[fillslot]
        alist[fillslot] = alist[positionOfMax]
        alist[positionOfMax] = temp

alist = [54,26,93,17,77,31,44,55,20]
selectionSort(alist)
print(alist)
```

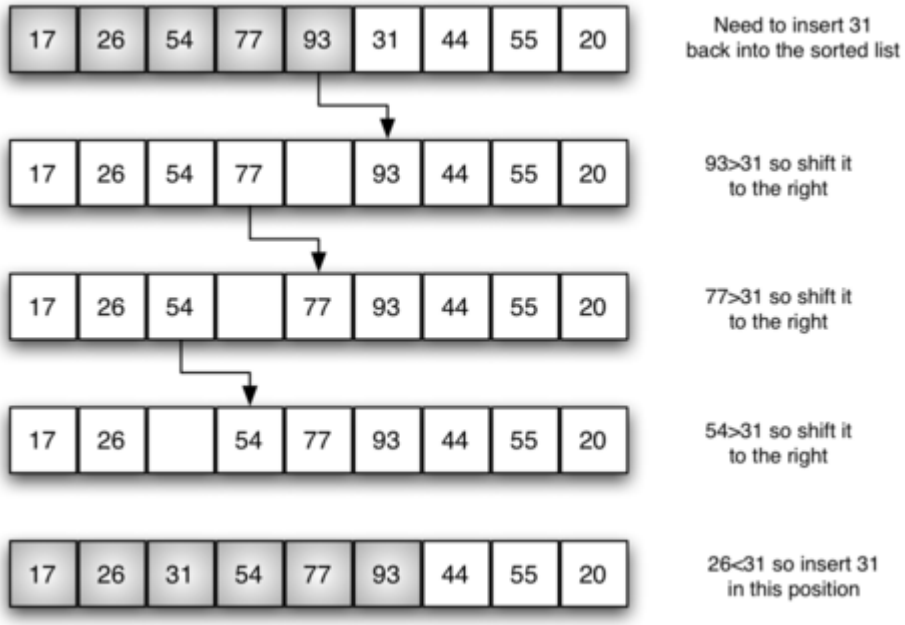
Сортування вставками, Маючи, як і раніше, O(n2)O(n2), працює дещо інакше. Вона завжди підтримує у сортованому вигляді підсписок на нижніх індексах списку. Кожен новий елемент "вставляється" в упорядкований на минулій ітерації підсписок так, щоб той залишився сортованим і став на один елемент більшим.*Малюнок 4*демонструє процес сортування вставками.



Малюнок 4:insertionSort

Почнемо з припущення, що список одного елемента (під номером 0) вже відсортований. На кожному проході для кожного елемента з першого по n−1n−1 поточний елемент порівнюється з уже відсортованими. Оскільки ми переглядаємо

відсортований підсписок у зворотному порядку, зсуваємо великі елементи вправо. При виявленні меншого елемента або кінця підпису поточний елемент вставляється на відповідну позицію.
на*малюнку 5*детально показано п'ятий прохід. У цій точці алгоритму сортований список із п'яти елементів містить 17, 26, 54, 77 та 93. Ми хочемо вставити в нього 31. Перше порівняння з 93 призводить до того, що 93 зрушується вправо. Так само зсуваються 77 і 54. Коли ми натрапляємо на елемент 26, процес зміщення припиняється, і 31 стає на вільне місце. Тепер у нас є відсортований список із шести елементів.

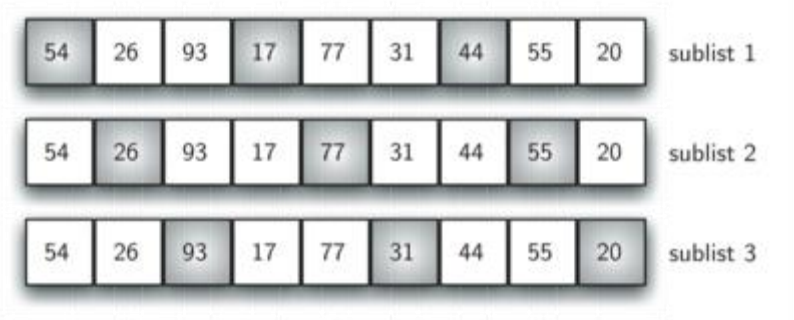


Малюнок 5:insertionSort: п'ятий прохід сортування
РеалізаціяinsertionSort(*ActiveCode 4*) демонструє, що знову виконується n−1n−1 прохід для сортування елементів nn. Ітерації починаються з індексу 1 і рухаються до позиції n−1n−1, оскільки всі ці елементи повинні бути вставлені у відсортований список. У рядку 8 відбувається операція зсуву, яка переміщає значення на одну позицію вгору за списком, створюючи місце для вставки. Пам'ятайте, що повноцінний обмін, як і в попередніх алгоритмах, тут не відбувається.
Максимальна кількість порівнянь для сортування дорівнює сумі перших n-1n-1 цілих Т.е. .
Важливе зауваження про протиставлення зсуву та обміну: загалом операція зсуву вимагає приблизно третини від обчислювальної роботи обміну, оскільки робиться лише одне привласнення. У контрольних дослідженнях сортування вставками має дуже хорошу продуктивність.

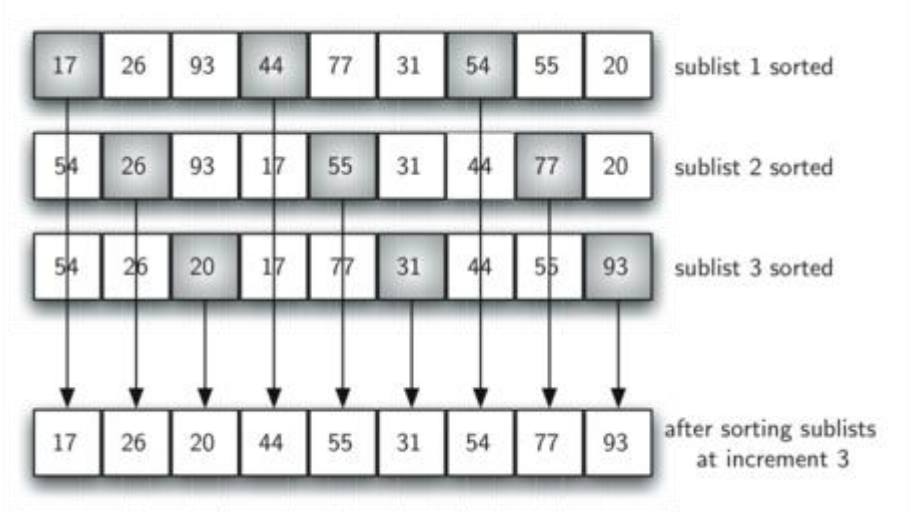
```
def insertionSort(alist):
   для index в range(1,len(alist)):

     currentvalue = alist[index]
     position = index
```

```
    while position>0 and alist[position-1]>currentvalue:
        alist[position]=alist[position-1]
        position = position-1

    alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
insertionSort(alist)
print(alist)
```

Сортування Шеллаіноді називають "сортуванням із зменшенням інкременту". Вона покращує сортування вставками, розбиваючи початковий список на кілька підсписків, кожен із яких сортується окремо. Оригінальний спосіб вибору таких підписів – ключова ідея сортування Шелла. Замість того, щоб виділяти підписки з елементів, що стоять поруч, сортування Шелла використовує інкременті(приріст), щоб створювати підписки з значень, що знаходяться на відстаніодин від одного.
Це можна побачити на*малюнку 6*. Зображений на ньому список складається із дев'яти елементів. Якщо ми використовуємо трійку як інкремент, то отримаємо три підписки, кожен з яких можна відсортувати вставками. Після завершення цих сортувань ми отримаємо список з*малюнку 7*. Хоча це ще не повністю відсортовано, сталося щось дуже цікаве. Сортуючи ці підписки, ми перемістили елементи ближче один до одного щодо того, де вони були раніше.

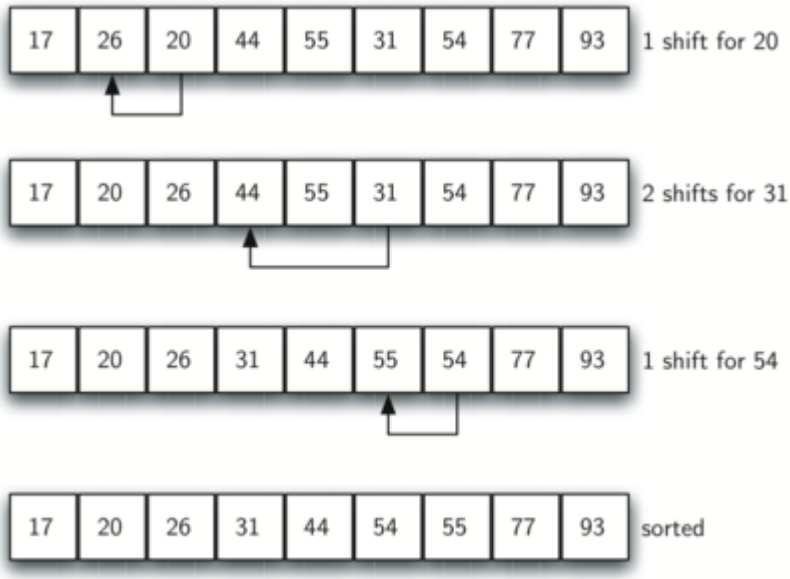


Малюнок 6: Сортування Шелла з інкрементом 3

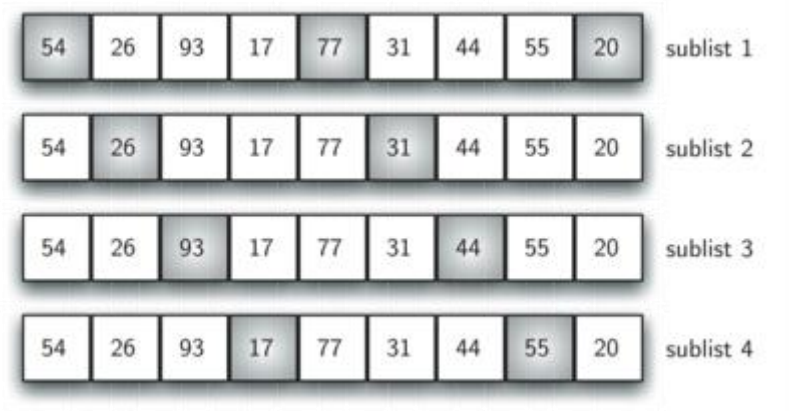


Малюнок 7: Сортування Шелла після сортування кожного підпису

*Малюнок 8*демонструє остаточне сортування вставками з використанням збільшення 1 - іншими словами, стандартне сортування вставками. Зверніть увагу, що завдяки проведенню попередніх сортувань підсписів зараз ми скоротили загальну кількість операцій зсуву, необхідних для розташування елементів списку в правильному порядку. У нашому випадку потрібно всього чотири зрушення, щоб завершити процес.



Малюнок 8: Сортування Шелла: підсумкове сортування вставками з інкрементом 1



Малюнок 9: Початковий підписки для сортування Шелла

Раніше ми вже говорили, що спосіб, яким вибирається інкремент, - унікальна властивість сортування Шелла.*ActiveCode 5*, використовує різні набори інкрементів. Ми починаємо з n2n2 підписів. На наступному проході сортуються підписки n4n4. Зрештою, єдиний список сортується за допомогою базового сортування вставками.*Малюнок 9*показує перші підписки нашого прикладу, використовують цей инкремент.

Наступний виклик функціїshellSortдемонструє списки, частково відсортовані після кожного збільшення, та фінальне сортування з інкрементом 1.

```
def shellSort(alist):
```

```
    sublistcount = len(alist)//2
    при sublistcount > 0:

      для startposition в range(sublistcount):
        gapInsertionSort(alist,startposition,sublistcount)

      print("After increments of size", sublistcount,
                                  "The list is",alist)

      sublistcount = sublistcount // 2

def gapInsertionSort(alist,start,gap):
    for i in range(start+gap,len(alist),gap):

        currentvalue = alist[i]
        position = i

        while position>=gap and alist[position-gap]>currentvalue:
            alist[position]=alist[position-gap]
            position = position-gap

        alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
shellSort(alist)
print(alist)
```

На перший погляд може здатися, що сортування Шелла не краще за попереднє, оскільки робить повне сортування вставками на останньому кроці. Однак, виходить так, що це підсумкове сортування не вимагає великої кількості порівнянь (або зрушень), оскільки список уже був частково відсортований (як описано вище). Іншими словами, кожний прохід робить список “більш сортованим” по відношенню до попереднього. Тому фінальний прохід виходить таким ефективним.
Хоча загальний аналіз сортування Шелла виходить далеко за рамки цього тексту, ми можемо сказати, що вона знаходиться між O(n)O(n) та O(n2)O(n2), залежно від поведінки, описаної вище. Для інкрементів, показаних у*лістингу 5*, продуктивність буде O(n2)O(n2). Змінивши інкремент (наприклад, на 2k−12k−1 (1, 3, 7, 15, 31 тощо)) отримаємо продуктивність сортування Шелла O(n32)O(n32).

Сортування злиттям
Тепер звернемо нашу увагу на використання стратегії "розділяй і владарюй", як способу покращити продуктивність сортувальних алгоритмів.**сортування злиттям**. Це рекурсивний алгоритм, який постійно розбиває список навпіл. Якщо список порожній або складається з одного елемента, він відсортований за визначенням

(базовий випадок). Якщо у списку більше, ніж один елемент, ми розбиваємо його та рекурсивно викликаємо сортування злиттям для кожної половини. Після того, як вони вже відсортовані, виконується основна операція, званазлиттям.Злиття - це процес комбінування двох менших сортованих списків в один новий, але теж відсортований.*Малюнок 10*демонструє як приклад наш старий знайомий список, який починають розбивати за допомогоюmergeSort.*Малюнок 11*показує відсортовані списки та їх злиття разом.

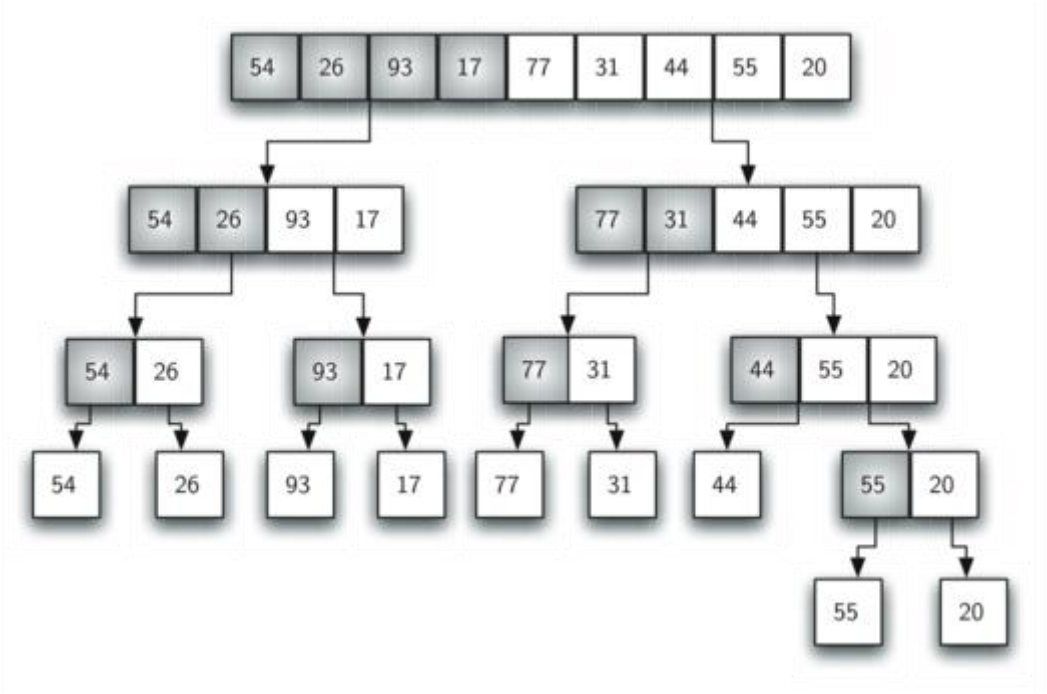
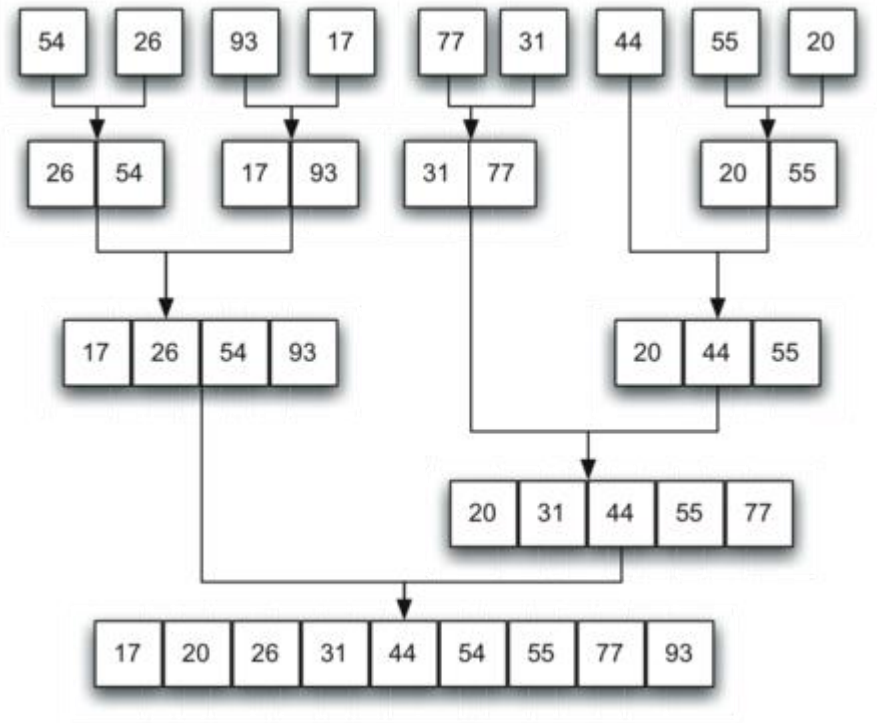


Рисунок 10: Розбиття списку у сортуванні злиттям



Малюнок 11: Списки, які з'єднуються разом

Функція mergeSort, показана в *ActiveCode 6* починає з перевірки базової умови. Якщо довжина списку менша чи дорівнює одиниці, він вже відсортований, й у подальшій обробці немає необхідності. З іншого боку, якщо довжина більше одиниці, ми використовуємо операцію Python slice, щоб витягти праву та ліву частини. Важливо, що список може мати непарну кількість елементів. Для алгоритму це важливо, оскільки довжини відрізнятимуться максимум на одиницю.

```
def mergeSort(alist):
    print("Splitting",alist)
    if len(alist)>1:
        mid = len (alist) / / 2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0
        while i<len(lefthalf) and j<len(righthalf):
            if lefthalf[i]<righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i<len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j<len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging",alist)

alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

Після того, як функціяmergeSortбула викликана для правої та лівої частин (рядки 8 - 9), передбачається, що вони відсортовані. Залишок функції (рядки 11 - 31) відповідає за злиття двох менших сортованих списків більший. Зверніть увагу, що операція злиття поміщає елементи назад до оригінального списку (alist) по одному за раз за допомогою вибору найменшого елемента, що повторюється, з двох сортованих списків.

функціюmergeSortдоповнює операторprint(рядок 2), який виводить вміст сортованого списку на початок кожного дзвінка. Також є операторprint(Рядок 32), що показує процес злиття. Результат обчислення функції у прикладі списку виводиться на екран. Зверніть увагу, що список з 44, 55 та 20 не ділиться порівну. Перша його частина дорівнює [44], а друга – [55, 20]. Легко побачити, як процес розбивки зрештою призводить до того, що список може бути негайно злитий з іншими сортованими списками.

Щоб проаналізувати функціюmegreSort, нам треба розглянути два різні процеси, які в ній відбуваються. По-перше, список розбивається навпіл. Ми вже обчислювали (для бінарного пошуку), що розділяти список на дві половини можна lognlog⁡f0n разів, де nn – довжина списку. Другий процес – це злиття. Кожен елемент буде оброблений та поміщений у сортований список. Таким чином, операція злиття, чий результат - список з nn елементів, вимагатиме nn операцій. Підсумок даного аналізу: lognlog⁡f0n розбиття, кожне вартістю nn, що в сумі дасть nlognnlog⁡f0n операцій. Таким чином, сортування злиттям - O(nlogn)O(nlog⁡f0n) алгоритм.

Нагадаємо, що вбудований оператор розбиття має O(k)O(k), де k - розмір розбиття. Щоб гарантувати для сортування злиттям O(nlogn)O(nlog⁡f0n), нам потрібно його позбутися. Нагадаємо, що це можливо, якщо просто поміщати початковий та кінцевий індекси разом зі списком як аргументи рекурсивного виклику. Ми залишаємо це вам як вправу.

ФункціяmergeSortвимагає додаткового місця для зберігання двох вилучених операцією розбиття частин. Цей додатковий простір може стати критичним фактором, якщо список є великим, і зробити цей спосіб сортування проблемним для великих наборів даних.

Швидке сортуваннявикористовує техніку "поділяй і владарюй", щоб отримати ті ж переваги, що і сортування злиттям, але при цьому не використовувати додаткове місце. Однак ціною за це стане те, що список може не поділитися навпіл, призводячи до зменшення продуктивності.

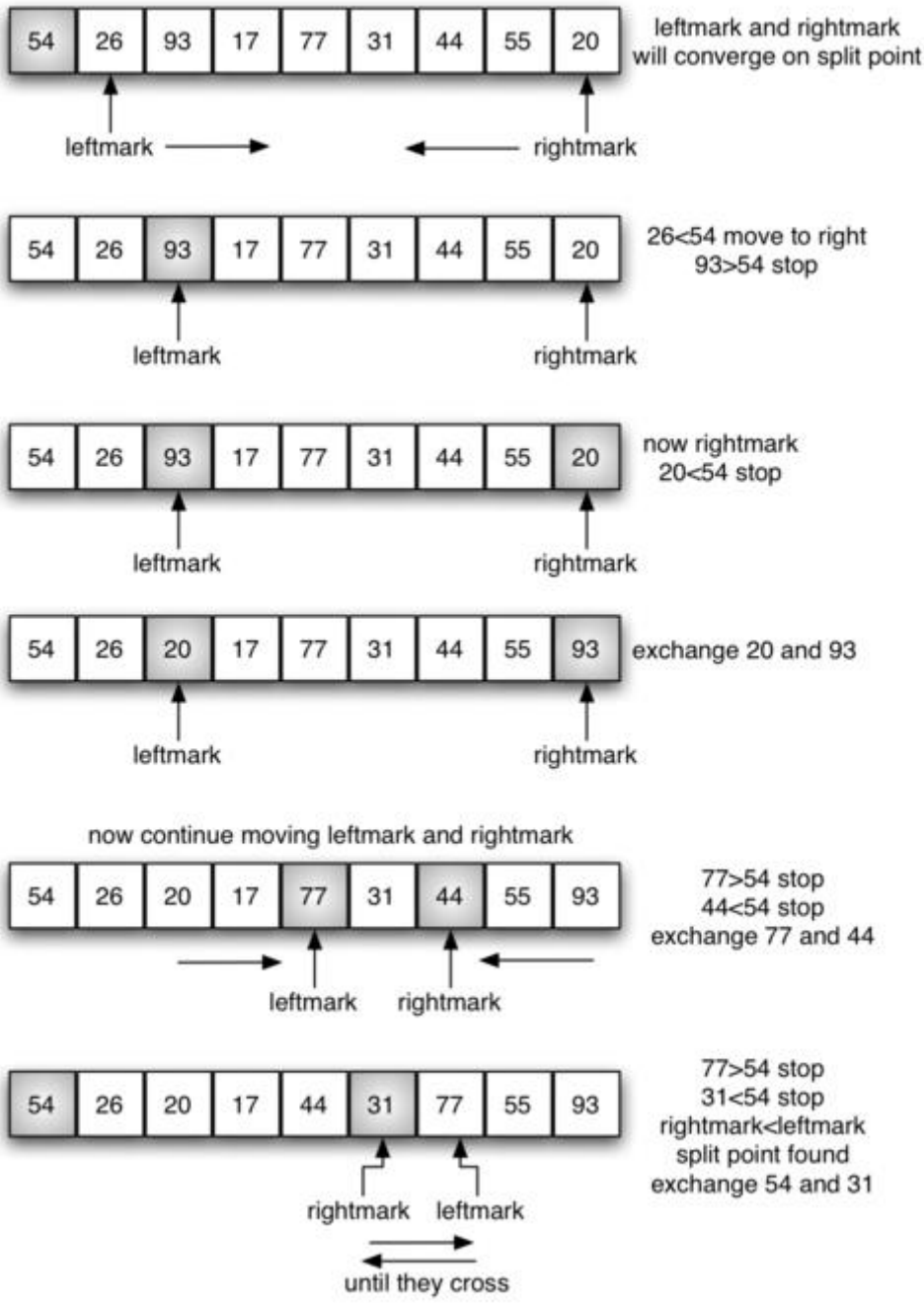
Спочатку швидке сортування вибирає значення, яке називається опорним елементом. Незважаючи на те, що є багато способів вибрати його, ми просто використовуватимемо перше значення в списку. Роль опорного елемента полягає у допомозі під час розбиття списку. Позиція, на якій він опиниться в підсумковому сортованому списку, яка зазвичай називається точкою розбиття, буде використовуватися для розділення списку при наступних викликах швидкого сортування.

*Малюнок 12*показує, як 54 виступає у ролі першого опорного значення. Оскільки ми вже розглядали цей приклад кілька разів, то знаємо, що 54 зрештою опиниться на

позиції, зайнятій зараз 31. Далі відбувається процес поділу. Він знаходить точку поділу і одночасно переміщає елементи по відповідним сторонам списку, залежно від того, більше вони або менше опорної величини.

Рисунок 12: Перша опорна величина для швидкого сортування

Розбиття починається з визначення двох маркерів положення - назвемо їхleftmarkirightmark- на початку і в кінці елементів списку, що залишилися (позиції 1 і 8 на*малюнку 13*). У процесі розбиття елементи, що лежать по неправильних сторонах від опорного, повинні переміщатися, поки маркери не зійдуться в точці поділу.*Малюнок 13*показує процес для опорного значення 54.



Малюнок 13: Пошук точки поділу для 54

Ми починаємо зі збільшення на одиницюleftmark, Доки не знаходимо значення, більше опорного. Тоді ми зменшуємо на одиницюrightmark, Доки не знаходимо значення, менше опорного. У цей момент ми маємо два елементи, що не на своїх місцях щодо підсумкової точки розбиття. У прикладі такими є 93 і 20. Тепер можна поміняти їх місцями і повторити процес заново.

Колиrightmarkстає меншеleftmark, ми зупиняємось.rightmarkзараз - точка розбиття. Опорне значення слід поміняти місцями з її вмістом, і тоді воно стоятиме на своєму місці (*малюнок 14*). На додаток, всі елементи ліворуч від точки розбиття тепер менше опорного значення, а праворуч – більше. Список поділений на дві частини, і швидке сортування може бути рекурсивно застосовано до кожної з них.

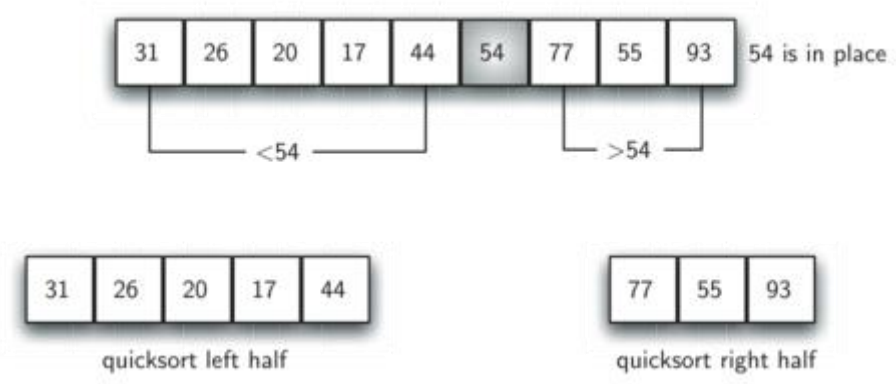


Рисунок 14: Завершення процесу з метою пошуку точки розбиття для 54

Функціяquick Sort, показана в*CodeLens 7*, викликає іншу рекурсивну функцію - quick Sort Helper. Вона починає працювати з базового випадку, аналогічного сортуванню злиттям. Якщо довжина списку менша або дорівнює одиниці, він вже відсортований. Якщо більше, він може бути розділений і рекурсивно відсортований. Функціяpartitionвтілює описаний раніше процес.

```
def quickSort(alist):
   quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
   if first<last:

       splitpoint = partition(alist,first,last)

       quickSortHelper(alist,first,splitpoint-1)
       quickSortHelper(alist,splitpoint+1,last)


def partition(alist,first,last):
   pivotvalue = alist[first]

   leftmark = перша+1
   rightmark = last

   done = False
   while not done:

       while leftmark <= rightmark and \
             alist[leftmark] <= pivotvalue:
          leftmark = leftmark + 1

       while alist[rightmark] >= pivotvalue and \
             rightmark >= leftmark:
          rightmark = rightmark -1
```

```
        if rightmark < leftmark:
            done = True
        else:
            temp = alist[leftmark]
            alist[leftmark] = alist[rightmark]
            alist[rightmark] = temp

    temp = alist[first]
    alist[first] = alist[rightmark]
    alist[rightmark] = temp


    return rightmark

alist = [54,26,93,17,77,31,44,55,20]
quickSort(alist)
print(alist)
```

Для аналізу функціїquickSortСлід зазначити, що з списку довжиною nn (якщо розподіл посідає його середину) ми знову отримаємо lognlog⁡n поділів. Щоб знайти точку розбиття, кожен з nn елементів потребує порівняння з опорним значенням. Результатом стане nlognnlog⁡n. При цьому не потрібно додаткової пам'яті, як у сортуванні злиттям.

На жаль, у найгіршому випадку точка розбиття може бути не посередині, а скакати зліва направо, роблячи поділ дуже нерівномірним. У цьому випадку сортування списку nn елементів розділиться на сортування списків розміром 0 і n−1n−1 елементів. Далі сортування списку довжиною n−1n−1 знову дасть підписки з 0 та n−2n−2 елементів, тощо. Результат: O(n2)O(n2) з усіма накладними витратами, потрібними для рекурсії.

Раніше ми згадували, що є кілька способів вибору опорного значення. будуть 54, 77 та 20. Тепер визначимо з них медіану – 54 у даному випадку – і візьмемо її в якості опори (природно, це було опорне значення, яке ми використовували спочатку). частковому сортуванню. Ми залишаємо реалізацію такого вибору опорного значення як вправу для вас.

Лекція 10. Структура даних «Дерева»

Вузол

Вузол – це основна частина дерева. Він може мати назву, яку ми називатимемо “ключом”. Також вузол може містити додаткову інформацію, яку ми називатимемо “корисним навантаженням”. Хоча в багатьох алгоритмах для дерев їй не приділяється достатньо уваги, для додатків, які використовують цю структуру даних, вона часто виявляється критичним фактором.

Гілка

Гілка – інша фундаментальна частина дерева. Воно з'єднує два вузли разом, показуючи наявність з-поміж них певних відносин. Кожен вузол (крім кореня) має рівно одну вхідну гілку. При цьому він може мати декілька вихідних гілок.

Корінь

Корінь дерева - єдиний вузол, що не має гілок, що входять. на*малюнку 2*/- Корінь дерева.

Шлях

Шлях – це впорядкований перелік вузлів, з'єднаних гілками. Наприклад, Mammal →→ Carnivora →→ Felidae →→ Felis →→ Domestica - це шлях.

Діти (нащадки)

Набір вузлів cc, що мають гілки від одного вузла, називаються його дітьми. на*малюнку 2*вузли log/, spool/ та yp/ - нащадки вузла var/.

Батько (предок)

Вузол є батьком усіх вузлів, із якими пов'язаний вихідними гілками. на*малюнку 2*вузол var/ є батьком вузлів log/, spool/ та yp/.

Брати

Вузли дерева, які є дітьми одного з батьків, називають братами. Прикладом можуть бути etc/ і usr/ в дереві файлової системи.

Піддерево

Поддерево - це набір вузлів і гілок, що складається з батька та всіх його нащадків.

Аркуш

Аркуш - це вузол, який не має дітей. Наприклад, Human і Chimpanzee - листя на*малюнку 1*.

Рівень

Рівень вузла nn - це число гілок у дорозі від кореня до nn. Наприклад, рівень Felis на*малюнку 1*дорівнює п'яти. За визначенням, рівень кореня – нульовий.

Висота

Висота дерева дорівнює максимальному рівню будь-якого його вузла. Наприклад, висота дерева на*малюнку 2*дорівнює двом.

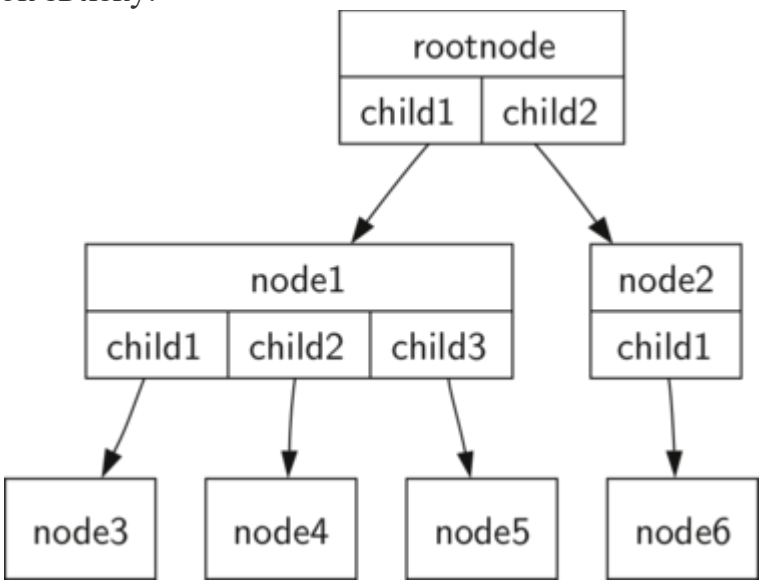
А тепер, визначившись із основною термінологією, дамо дереву формальне визначення. Фактично, ми дамо два формулювання: перше включатиме вузли та гілки, а друге (чию корисність ми доведемо на практиці) буде рекурсивним.

*Визначення 1:*Дерево складається з набору вузлів та набору гілок, що з'єднують пари вузлів. Воно має такі властивості:

- Один із вузлів дерева визначений, як його корінь.

- Кожен вузол nn (крім кореневого) з'єднується гілкою з єдиним іншим вузлом pp, де pp – батько nn.
- Кожен вузол з'єднаний з коренем єдиним можливим шляхом.
- Якщо кожен із вузлів дерева має максимум двох нащадків, то така структура називається двійковим деревом.

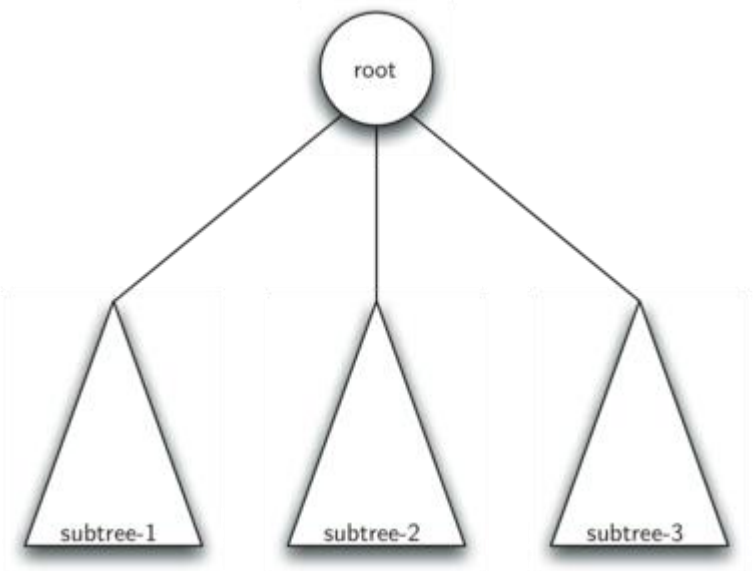
на*малюнку 3*зображено дерево, що задовольняє визначення 1. Стрілки на гілках показують напрямок зв'язку.



Малюнок 3: Дерево, що містить набір вузлів та гілок.

*Визначення 2:*Дерево або пусте, або містить корінь і нуль або більше піддерев, кожне з яких теж є деревом. Корінь кожного піддерева з'єднаний гілкою з батьківським деревом.

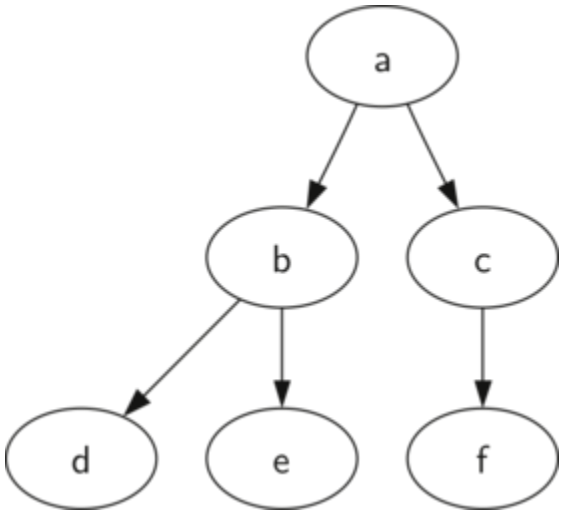
*Малюнок 4*ілюструє це визначення. Використовуючи його, можна сказати, що зображена структура має як мінімум чотири вузли, оскільки кожен із трикутників, що представляють піддерева, повинен мати корінь. У цьому дереві може бути набагато більше вузлів, але сказати точніше не можна доти, доки ми не просунемося по ньому глибше.



Малюнок 4: Рекурсивне визначення дерева

Подання дерева у вигляді списку списків

Втілювати дерево у вигляді списку списків ми почнемо зі структури даних Python “список” і напишемо для неї функції, визначені вище. Також ми створимо інтерфейс – набір операцій над списком, трохи відмінний від тих АТД, які вже були нами реалізовані. Це буде цікаво та надасть у наше розпорядження просту рекурсивну структуру даних, яку потім можна буде вивчати та тестувати. У дереві, представленому як список списків, на першій позиції ми зберігатимемо значення кореневого вузла. Другий елемент сам буде списком і представить ліве піддерево. Третій елемент стане правим піддеревом. Щоб проілюструвати таку техніку зберігання, розглянемо приклад.*Малюнок 1*демонструє просте дерево та пов'язану з ним спискову реалізацію.



Малюнок 1: Маленьке дерево

```
myTree = ['a', #root
      ['b', #left subtree
       ['d'[], []],
       ['e' [], []] ],
      ['c', #right subtree
       ['f' [], []],
       [] ]
     ]
```

Зверніть увагу, що ми маємо доступ до кожного з піддерев'їв з використанням стандартної спискової індексації. Корінь деревамyTree[0], ліве піддерево -myTree[1], праве -myTree[2].*ActiveCode 1*демонструє створення простого дерева за допомогою списку. Після того, як дерево буде готове, ми зможемо отримати доступ до його кореня, правого та лівого піддерев'я. Одна з приємних властивостей підходу зі списком списків полягає в тому, що структура списку, що представляє піддерево, твердо дотримується визначення дерева - рекурсивна сама по собі! У піддерева є корінь і два порожні списки як листя. Інша позитивна якість списку полягає в тому,

що він легко розширюється до дерева, що має багато піддерев. Тобто. у випадку, коли дерево не є двійковим, нове піддерево - це лише новий підсписок.

```
1
myTree= ['a', ['b', ['d',[],[]], ['e',[],[]] ], ['c', ['f',[],[]], []] ]
2
print(myTree)
3
print('left subtree = ', myTree[1])
4
print('root=', myTree[0])
5
print('right subtree = ', myTree[2])
6
```

Використання індексації для доступу до піддерев'я. (tree_list1)

Давайте формалізуємо це визначення за допомогою деяких функцій, які зроблять простіше використання списків як дерева. Зверніть увагу, ми не збираємось визначати новий клас для двійкового дерева. Функції, які будуть написані, лише допоможуть маніпулювати стандартним списком, з яким ми працюємо, як з деревом.

```
def BinaryTree(r):
   return[r, [], []]
```

ФункціяBinaryTreeпросто створює список з кореневого вузла та двох порожніх підсписків як його нащадків. Щоб додати до кореня ліве піддерево, нам потрібно вставити новий список на другу позицію. Тут слід бути уважними. Якщо на другій позиції вже є, то цей факт потрібно відстежити і зрушити елемент вниз по дереву, як лівого нащадка списку, що додається.

Лістинг 1

```
def insertLeft(root,newBranch):
   t=root.pop(1)
   if len(t) > 1:
      root.insert(1,[newBranch,t,[]])
   else:
      root.insert(1,[newBranch,[], []])
   returnroot
```

Зверніть увагу, що перш ніж вставляти будь-що, ми отримуємо (можливо порожній) список, пов'язаний з поточним лівим нащадком. Коли ми вставляємо нове ліве піддерево, старе робимо його лівим нащадком. Завдяки цьому ми можемо вбудовувати новий вузол на будь-яку позицію дерева. Код дляinsertRightаналогічнийinsertLeftі показаний у*лістингу 2*.

Лістинг 2

```
def insertRight(root,newBranch):
   t=root.pop(2)
```

```
    if len(t) > 1:
        root.insert(2,[NewBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    returnroot
```

Щоб закінчити з набором для дерева (див.*лістинг 3*), давайте напишемо кілька функцій доступу для встановлення та отримання значень у корені і правого та лівого піддерев'я.

Лістинг 3

```
def getRootVal(root):
    returnroot[0]

def setRootVal(Root, NewVal):
    root[0] =newVal

def getLeftChild(root):
    returnroot[1]

def getRightChild(root):
    returnroot[2]
```

*ActiveCode 2*використовує щойно написані функції для дерева. Спробуйте попрацювати із ними самостійно. Одна з вправ попросить вас намалювати структуру дерева, яка стане результатом такого набору дзвінків:

```
def BinaryTree(r):
    return [r, [], []]

def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root

def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root
```

```
def getRootVal(root):
    return root[0]

def setRootVal(root,newVal):
    root[0] = newVal

def getLeftChild(root):
    return root[1]

def getRightChild(root):
    return root[2]

r = BinaryTree(3)
insertLeft(r,4)
insertLeft(r,5)
insertRight(r,6)
insertRight(r,7)
l = getLeftChild(r)
print(l)

setRootVal(l,9)
print(r)
insertLeft(l,11)
print(r)
print(getRightChild(getRightChild(r)))
```

Наш другий спосіб представлення дерев буде використовувати вузли та посилання. Для цього випадку ми визначимо клас, чиїми атрибутами стануть кореневе значення і ліве та праве піддерев'я.
Використовуючи вузли та посилання, про дерево можна думати, як про структуру, приклад якої наведено*малюнку 2*.

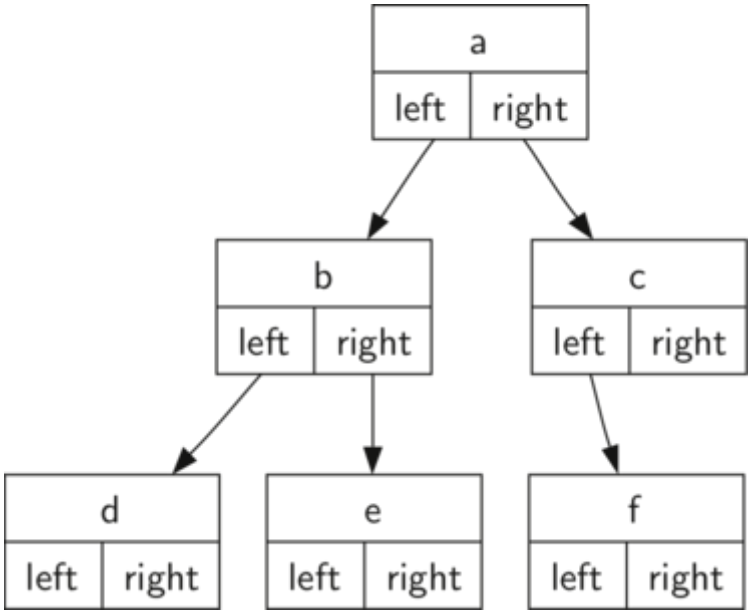


Figure 2: Варіант простого дерева з використанням вузлів та посилань

Почнемо з простого визначення класу для варіанта з вузлами та посиланнями (*лістинг 4*). Важливо пам'ятати, що у цій уявленні атрибутиleftirightє посиланнями на інші сутності класуBinaryTree. Наприклад, коли ми вставляємо нового лівого нащадка у дерево, ми створюємо інший об'єктBinaryTreeі змінюємоself.leftChildкореня, щоб цей атрибут посилався на нове дерево.

Лістинг 4

```
клас BinaryTree:
   def __init__(self,rootObj):
      self.key=rootObj
      self.leftChild= None
      self.rightChild= None
```

Зверніть увагу, що в*лістингу 4*конструктор очікує отримати об'єкт якогось виду, щоб зберегти його докорінно. Подібно до того, як ви можете зберігати в списку будь-який об'єкт, коренем дерева може бути будь-яке посилання. У наших ранніх прикладах як кореневе значення зберігалося ім'я вузла. Використовуючи вузли та посилання для представлення дерева (як це показано на*малюнку 2*), нам потрібно створити шість сутностей класу``BinaryTree``.

Далі розглянемо функцію, яку потрібно написати для будівництва дерева за межі кореневого значення. Щоб додати лівого нащадка в дерево, ми створимо новий об'єкт двійкового дерева і помістимо його атрибут кореня.leftпосилання на новий об'єкт. Код дляinsertLeftпоказаний у*лістингу 5*.

Лістинг 5

```
1
2  def insertLeft(self,NewNode):
3     if self.leftChild== None:
4        self.leftChild=BinaryTree(newNode)
5     else:
6        t=BinaryTree(newNode)
```

```
7           t.leftChild= self.leftChild
            self.leftChild=t
```

Нам необхідно розглянути два випадки вставки. Перший - для вузла, який не має лівого нащадка. У цьому варіанті вузол просто вставляється у дерево. Другий варіант характеризується вузлом, що має лівого нащадка. Тоді нам треба вставити новий вузол і спустити нащадка на один рівень нижче. Цей випадок керується операторомelseу рядку 4*лістингу 5*.

Код дляinsertRightмає містити симетричний набір випадків. Тут також може бути відсутній правий нащадок, або існувати необхідність вставити вузол між коренем і правим нащадком. Код операції вставки показано в*лістингу 6*.

Лістинг 6

```
def insertRight(self,NewNode):
    if self.rightChild== None:
        self.rightChild=BinaryTree(newNode)
    else:
        t=BinaryTree(newNode)
        t.rightChild= self.rightChild
        self.rightChild=t
```

Завершуючи наше визначення простого двійкового дерева, напишемо методи доступу до кореня, правого та лівого нащадків (див.*лістинг 7*).

Лістинг 7

```
def getRightChild(self):
    return self.rightChild

def getLeftChild(self):
    return self.leftChild

def setRootVal(self, obj):
    self.key=obj

def getRootVal(self):
    return self.key
```

Тепер у нас є все необхідне для створення та маніпулювання двійковим деревом. Давайте використовуємо його, щоб дослідити структуру трохи глибше. Створимо просте дерево із вузломаяк корінь і вузлибіяк нащадки.*ActiveCode 4*конструює таке дерево і дивиться, які значення збереглися вkey,leftiright. Зверніть увагу, що і лівий, і правий нащадки кореня – різні сутності класуBinaryTree. Як ми вже говорили у нашому оригінальному рекурсивному визначенні дерева, це дозволяє нам працювати з будь-яким нащадком двійкового дерева, як із самим деревом.

```
class BinaryTree:
    def __init__(self,rootObj):
        self.key = rootObj
```

```
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self,newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self,newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t


    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self,obj):
        self.key = obj

    def getRootVal(self):
        return self.key


r = BinaryTree('a')
print(r.getRootVal())
print(r.getLeftChild())
r.insertLeft('b')
print(r.getLeftChild())
print(r.getLeftChild().getRootVal())
r.insertRight('c')
print(r.getRightChild())
print(r.getRightChild().getRootVal())
r.getRightChild().setRootVal('hello')
print(r.getRightChild().getRootVal())
```

Список використаних джерел

1. Introduction to Algoritms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Third Edition. The MIT Press Cambridge, Massachusetts London, England. 2009. 1292p.
2. Креневич О.П. Алгоритми та структури даних. Підручник. - К.: ВПЦ "Київський Університет", 2021. - 200 с.
3. Данильченко О.М., Данильченко А.О., Россінський Ю.М. Алгоритми та структури даних.ЖІТІ, 2009.- 296 с.
4. Алгоритми та структури даних: навчальний посібник / Н. Б. Шаховська; Р.О. Голощук; за заг. ред. Пасічника В.В. – Львів : Магнолія 2006, 2011. – 215 с.
5. Алгоритми та структури даних: конспект лекцій. Частина 2. Алгоритми пошуку, стиснення даних, внутрішнього та зовнішнього сортування, алгоритми на графах / упоряд.: О. Д. Воробйов, Л. В. Глазунов. Одеса : ОНАЗ ім. О.С. Попова, 2017. 52 с.
6. Багач І. В., Довгалець С. М., Дубовий В, М. Алгоритми розв'язання задач з програмування. Вінниця: ВНТУ, 2017. 119 с.
7. Дудзяний І. М. Програмування на мові С++. Частина 1: Парадигма процедурного програмування: навчальний посібник. Львів: ЛНУ імені Івана Франка, 2013. 468 с.
8. Клакович Л. С., Левицька С. М., Костів О, В. Теорія алгоритмів Львів : Видавничий центр ЛНУ імені Івана Франка, 2008. 138 с.
9. Коротєєва Т. О. Алгоритми та структури даних : навчальний посібник. Львів: Видавництво Львівської політехніки, 2014. 80 с.
10. Мелешко Є. В., Якименко М. С., Поліщук Л. І. Алгоритми та структури даних: Навчальний посібник для студентів технічних спеціальностей денної та заочної форми навчання.
11. Махровська Н.А., Погромська Г. З. Алгоритми та структури даних: навчально-методичний посібник. Миколаїв: МНУ ім. В.О. Сухомлинського, 2019. 279 с.
12. Онищенко В. В., Конік Р. С. Алгоритми та структури даних.
13. Прийом С.М. Теорія алгоритмів: навчальний посібник. Мелітополь: ФОП Однорог Т. Ст, 2018. 116 с.
14. Ришковець Ю. В., Висоцька В. О. Алгоритмізація та програмування. Частина 2: навчальний посібник. Львів: Видавництво «Новий Світ-2000», 2020. 320 с.
15. Сергієнко О. М., Марченка О. І. Конспект лекцій з курсу “Алгоритми та структури даних” для напряму підготовки 123 Комп'ютерна інженерія. : Національний технічний Університет України «Київський Політехнічний Інститут імені Ігоря Сікорського», 2017. 74 с.
16. Ткачук В. М. Алгоритми та структура даних: навчальний посібник. Івано-Франківськ.

17. Трофіменко О. Г., Прокоп Ю. В., Логінова Н. І., Задерейко О. В. С++. Алгоритмізація та програмування: підручник. Одеса: Фенікс, 2019. 477 с.
18. Шаховська Н. Б., Голощук Р. О. Алгоритми та структури даних.
19. Шевчук І.Б. Конспект лекцій з навчальної дисципліни “Алгоритмізація та програмування”. Львів : Львівський національний університет ім. Івана Франка, 2018. 30с.

Курс лекцій

Укладачі:
МОГИЛЬНИЙ Геннадій Анатолійович
СЕМЕНОВ Микола Анатолійович
СМАГІНА Ольга Олександрівна
ПЕРЕЯСЛАВСЬКАСвітлана Олександрівна

АЛГОРИТМІ І СТРУКТУРИ ДАНИХ

Курс лекцій до вивчення освітнього компонента для здобувачів освіти спеціальності
122 – „Комп'ютерні науки”