Самобалансирующиеся деревья

Хайрулин Сергей Сергеевич
s.khayrulin@gmail.com

Overview

1. Двоичная куча и основные операции над ними
    - вставка
    - удаление
    - поиска элемента
2. Самобалансирующиеся деревья
3. Красно-черные деревья
4. Свойства красно-черных деревьев
5. Основные операции
    - Вставка
    - Поиск
    - Удаление
    - Повороты

Литература и др. источники

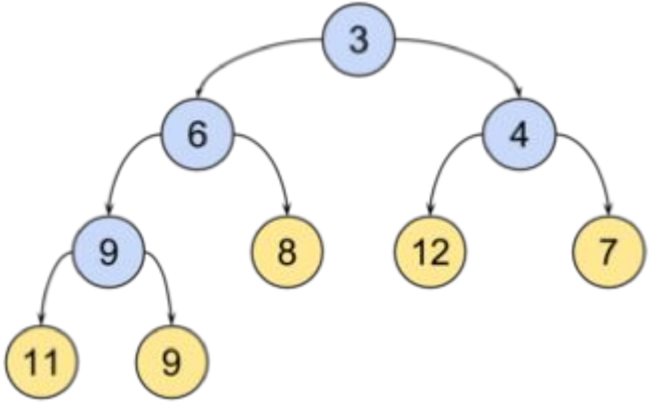
1. Дональд Эрвин Кнут. Искусство программирования (Том 1, 2, 3) // Вильямс 2015.
2. Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман. Структуры данных и алгоритмы // Вильямс 2000.
3. Емеличев В. А., Мельников О. И., Сарванов В. И., Тышкевич Р. И. Лекции по теории графов // М.: Наука, 1990.
4. Харари Ф. Теория графов // М.: Мир, 1973.
5. Косточка А. В. Дискретная математика. Часть 2 //Новосибирск: НГУ, 2001.
6. Котов В. Е., Сабельфельд В. К. Теория схем программ // Наука 1991.
7. http://algolist.manual.ru
8. ….

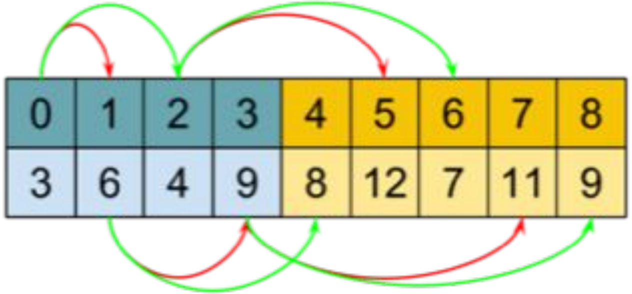
Двоичная куча

Двоичная куча – такое бинарное дерево, для которого выполняются следующие условия:

- Значение в любой вершине не меньше (если куча для максимума), чем значения её потомков.
- Глубина всех листьев (расстояние до корня) отличается не более чем на 1 слой.
- Последний слой заполняется слева направо без «дырок».

Двоичная куча





Двоичная куча

Если в куче изменяется один из элементов, то она может перестать удовлетворять свойству упорядоченности.

```
function siftDown(i : int):
    while 2 * i + 1 < a.heapSize     // heapSize — количество элементов в куче
        left = 2 * i + 1             // left — левый сын
        right = 2 * i + 2            // right — правый сын
        j = left
        if right < a.heapSize and a[right] < a[left]
            j = right
        if a[i] ⩽ a[j]
            break
        swap(a[i], a[j])
        i = j
```

Двоичная куча

```
function siftUp(i : int):
    while a[i] < a[(i - 1) / 2]     // i == 0 — мы в корне
        swap(a[i], a[(i - 1) / 2])
        i = (i - 1) / 2
```

Двоичная куча

	В среднем
Расход памяти	O(n)
Восстановление свойств	O(logn)
Вставка	O(logn)
Извлечение минимального	O(logn)

Самобалансирующиеся деревья

- Красно-черное дерево
- AVL-дерево
- B-дерево
- Splay-дерево

Красно-черные деревья

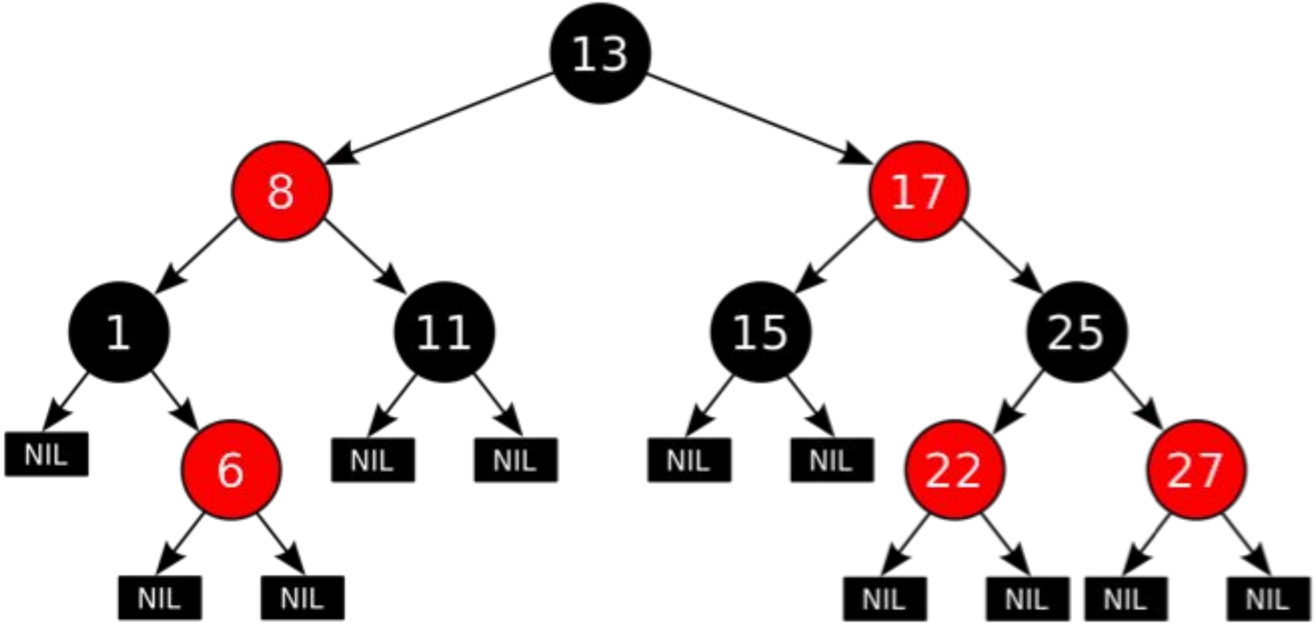
Красно-черное дерево - это бинарное дерево с следующими свойствами:

1. Каждый узел окрашен либо в черный, либо в красный цвет.
2. Листьями объявляются NIL-узлы (т.е. "виртуальные" узлы, наследники узлов, которые обычно называют листьями; на них "указывают" NULL указатели). Листья покрашены в черный цвет.
3. Если узел красный, то оба его потомка черны.
4. На всех ветвях дерева, ведущих от его корня к листьям, число черных узлов одинаково.

Rudolf Bayer, 1972 г. (symmetric binary B-tree)

Leonidas J. Guibas и Robert Sedgewick, 1978 г

Красно-черные деревья



Красно-черные деревья

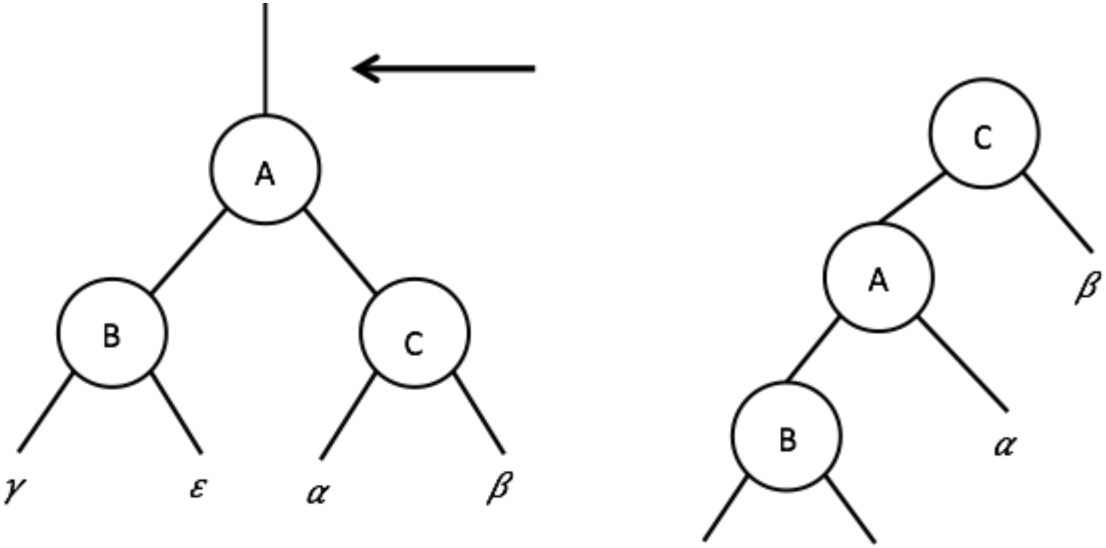
Будем называть черной **высотой дерева** (black-height) **вершины** *x* число черных вершин на пути из *x* в лист, не учитывая саму вершину *x*.

Теорема

Красно-чёрное дерево с N ключами имеет высоту $h=O(\log N)$.

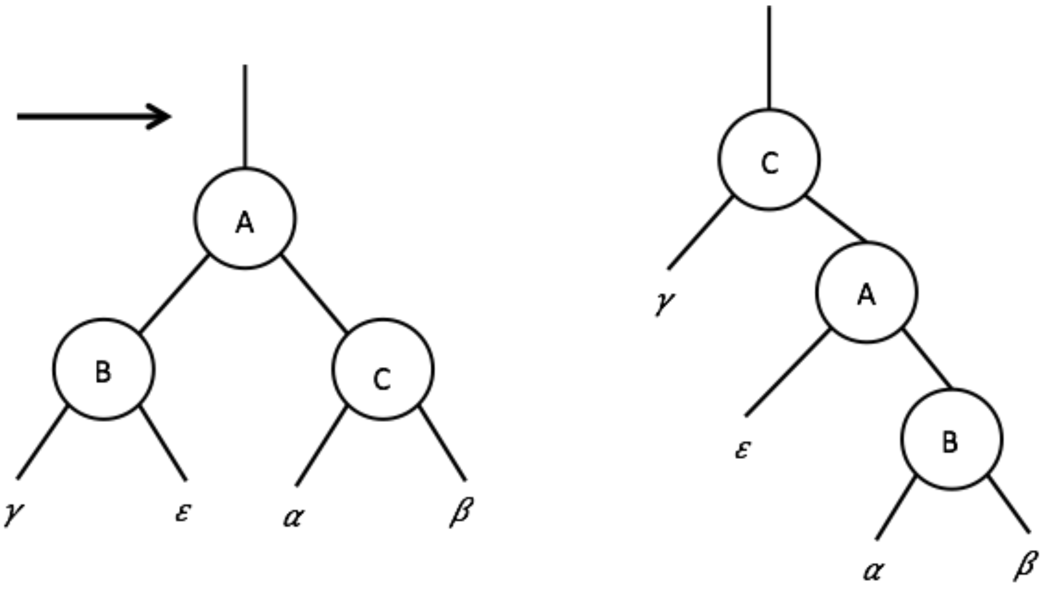
Красно-черные деревья: балансировка

Левый поворот



Красно-черные деревья: балансировка

Правый поворот

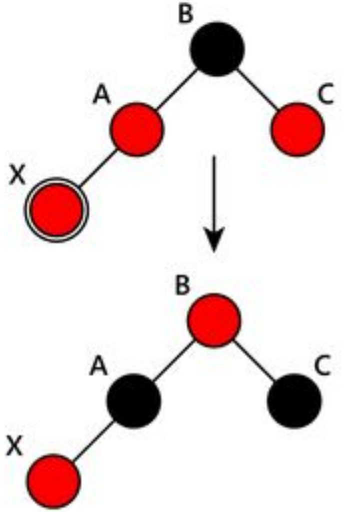


Красно-черные деревья: вставка

Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня до тех пор, пока указатель на следующего сына не станет **NIL** (то есть этот сын — лист). Вставляем вместо него новый элемент с **NIL-потомками** и красным цветом.

Красно-черные деревья: вставка

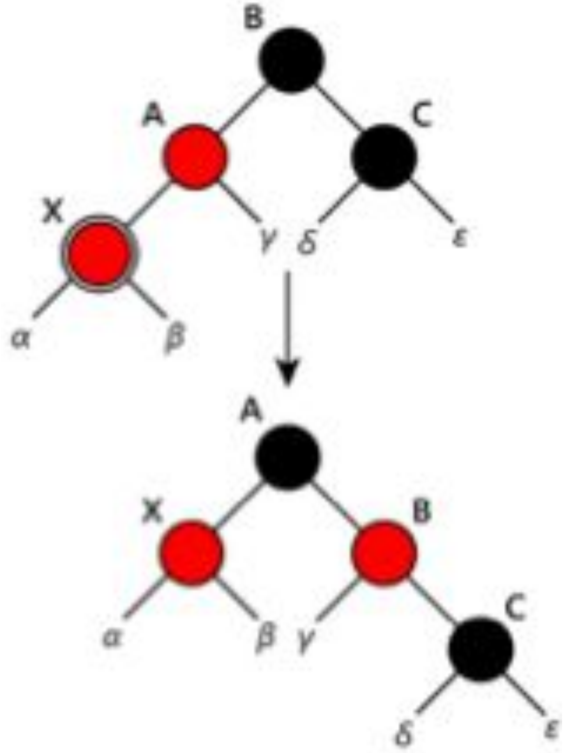
1. "Дядя" этого узла тоже красный. Тогда, чтобы сохранить свойства 3 и 4. просто перекрашиваем "отца" и "дядю" в чёрный цвет, а "деда" — в красный.



Красно-черные деревья: вставка

2."Дядя" черный. Если выполнить только перекрашивание, то может нарушиться постоянство чёрной высоты дерева по всем ветвям. Поэтому выполняем поворот. Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком.

Красно-черные деревья: вставка



Красно-черные деревья: удаление

При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

1. Если у вершины нет детей, то изменяем указатель на неё у родителя на **NIL**.

Красно-черные деревья: удаление

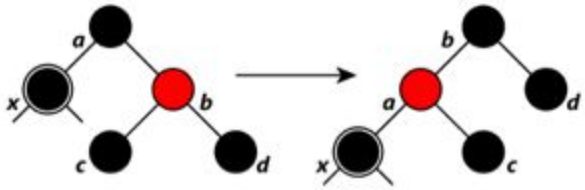
2. Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.

Красно-черные деревья: удаление

3. Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка (так как такая вершина находится в правом поддереве исходной вершины и она самая левая в нем, иначе бы мы взяли ее левого ребенка. Иными словами сначала мы переходим в правое поддерево, а после спускаемся вниз в левое до тех пор, пока у вершины есть левый ребенок). Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

Красно-черные деревья, удаление.

1. Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева.



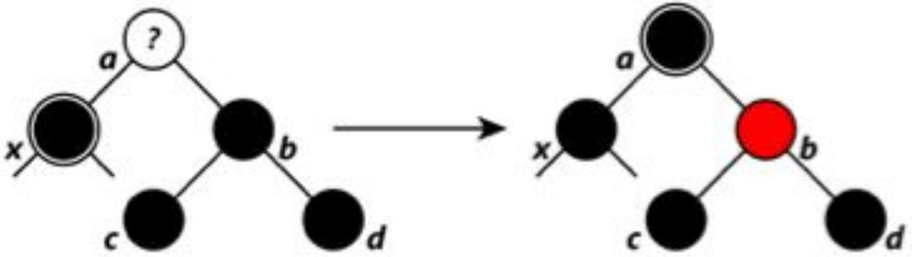
Красно-черные деревья, удаление.

2. Если брат текущей вершины был черным, то получаем три случая:

Красно-черные деревья, удаление.

- Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным, это не повлияет на количество черных узлов на путях, проходящих через b, но добавит один к числу черных узлов на путях, проходящих через x, восстанавливая тем самым влияние удаленного черного узла. Таким образом, после удаления вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.

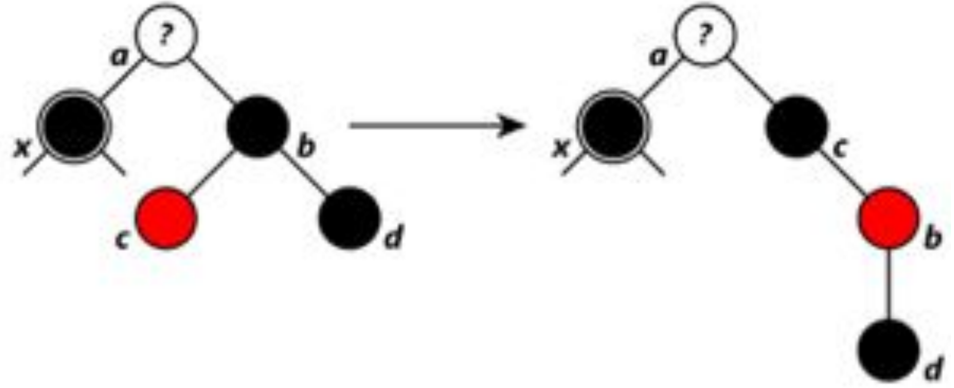
Красно-черные деревья, удаление.



Красно-черные деревья, удаление.

- Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у x есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни x, ни его отец не влияют на эту трансформацию.

Красно-черные деревья, удаление.



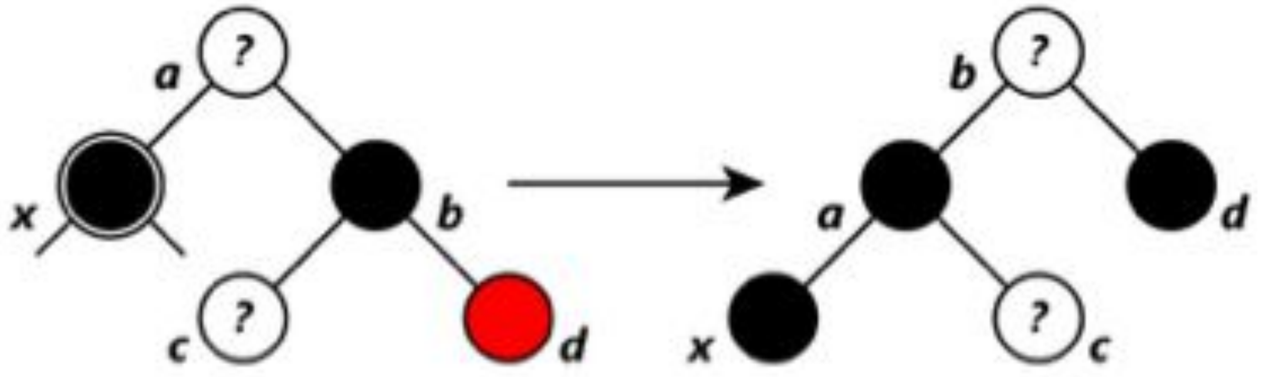
Красно-черные деревья, удаление.

- Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца - в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство 3 и 4 не нарушаются. Но у x теперь появился дополнительный чёрный предок: либо a стал чёрным, или он и был чёрным и b был добавлен в качестве чёрного дедушки. Таким образом, проходящие через x пути проходят через один дополнительный чёрный узел. Выходим из алгоритма.

Красно-черные деревья, удаление.

- Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца - в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство 3 и 4 не нарушаются. Но у x теперь появился дополнительный чёрный предок: либо a стал чёрным, или он и был чёрным и b был добавлен в качестве чёрного дедушки. Таким образом, проходящие через x пути проходят через один дополнительный чёрный узел. Выходим из алгоритма.

Красно-черные деревья, удаление.



Красно-черные деревья, удаление.

	В среднем	**В худшем**
Расход памяти	O(n)	O(n)
Восстановление свойств	O(logn)	O(logn)
Вставка	O(logn)	O(logn)
Извлечение минимального	O(logn)	O(logn)

Преимущества

1. Самое главное преимущество красно-черных деревьев в том, что при вставке выполняется не более O(1) вращений. Это важно, например, в алгоритме построения динамической выпуклой оболочки. Ещё важно, что примерно половина вставок и удалений произойдут задаром.
2. Процедуру балансировки практически всегда можно выполнять параллельно с процедурами поиска, так как алгоритм поиска не зависит от атрибута цвета узлов.
3. Сбалансированность этих деревьев хуже, чем у АВЛ, но работа по поддержанию сбалансированности в красно-черных деревьях обычно эффективнее. Для балансировки красно-чёрного дерева производится минимальная работа по сравнению с АВЛ-деревьями.

Преимущества

4. Использует всего 1 бит дополнительной памяти для хранения цвета вершины. Но на самом деле в современных вычислительных системах память выделяется кратно байтам, поэтому это не является преимуществом относительно, например, АВЛ-дерева, которое хранит 2 бита. Однако есть реализации красно-чёрного дерева, которые хранят значение цвета в бите. Пример — Boost Multiindex. В этой реализации уменьшается потребление памяти красно-чёрным деревом, так как бит цвета хранится не в отдельной переменной, а в одном из указателей узла дерева.

Преимущества

Красно-чёрные деревья являются наиболее активно используемыми на практике самобалансирующимися деревьями поиска. В частности, ассоциативные контейнеры библиотеки STL(map, set, multiset, multimap) основаны на красно-чёрных деревьях. TreeMap в Java тоже реализован на основе красно-чёрных деревьев.

Python & ADT

8.3. collections — Container datatypes

Source code: Lib/collections/__init__.py

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, `dict`, `list`, `set`, and `tuple`.

`namedtuple()`	factory function for creating tuple subclasses with named fields
`deque`	list-like container with fast appends and pops on either end
`ChainMap`	dict-like class for creating a single view of multiple mappings
`Counter`	dict subclass for counting hashable objects
`OrderedDict`	dict subclass that remembers the order entries were added
`defaultdict`	dict subclass that calls a factory function to supply missing values
`UserDict`	wrapper around dictionary objects for easier dict subclassing
`UserList`	wrapper around list objects for easier list subclassing
`UserString`	wrapper around string objects for easier string subclassing

Changed in version 3.3: Moved Collections Abstract Base Classes to the `collections.abc` module. For backwards compatibility, they continue to be visible in this module as well.

https://docs.python.org/3.6/library/collections.html

Указания для задач

Для замера работы функции нужно использовать метод now() класса datetime модуля datetime

Указания для задач

```
array = [0] * N
array.insert(N,0)
```

```
import datetime


def main():
    t1 = datetime.datetime.now()
    #You'r code here
    ...
    print(datetime.datetime.now() - t1)

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

Указания для задач

```python
import numpy as np


...
# Generate numpy Array with N random numbers
array = np.random.rand(N)
#Sort Array by quick sort
sorted(array)
...
```

Указания для задач

jupyter MyTests Last Checkpoint: a minute ago (unsaved changes)

File Edit View Insert Cell Kernel Help — Python 2

CellToolbar

In [6]:
```
import datetime
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
```

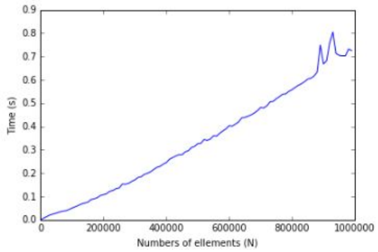
In [7]:
```
times = []
for N in xrange(1000, 1000000, 10000):
    array = np.random.rand(N)
    t1 = datetime.datetime.now()
    sorted(array)
    t_res = datetime.datetime.now() - t1
    times.append([N,t_res])
y = [t[1].total_seconds() for t in times]
x = [t[0] for t in times]
```

In [8]:
```
%matplotlib inline
```

In [9]:
```
plt.plot(x, y)

plt.xlabel('Numbers of ellements (N)')
plt.ylabel('Time (s)')

plt.savefig("test.png")
plt.show()
```



Задачи

- Реализовать алгоритм перемножения квадратных матриц. Матрицы могут задаваться как список списков. Считывать можно из файла потока ввода, или задавать случайным образом (используя функцию np.random.rand(N)). Оценить временную и ассимптотическую сложность алгоритма, построить график.
- Найти все пифагоровы тройки ($c^2 = a^2 + b^2$) для заданного интервала. Интервал задается парой чисел через пробел считанных из входного потока (например: 10 100) помните, что верхняя грань отрезка должна быть больше нижней. Если задано одно число, то считаем, что ограничение снизу равно по умолчанию 1. Оценить временную и ассимптотическую сложность алгоритма, построить график.
- Реализовать алгоритм факторизации числа (разложение числа как произведение двух других чисел). Оценить временную и ассимптотическую сложность алгоритма, построить график.
- Реализовать алгоритм рассчитывающий сочетания и размещения.
- Факториал довольно емкостная функция, при расчете которого для больших значений может случится переполнение (т.е. полученное число будет больше чем максимально возможное число в вашей системе). Подумайте как преодолеть эту проблему.

Задачи

Написать оболочку для работы с графами:

- создавать графы
- Выводить граф (в виде таблицы смежности)
- Удалять ребра
- Ищет путь в графе для заданных вершин
    - Флойда-Уоршела
    - Форда-Беллмана
    - Дейкстра

Задачи

1. Скачать файл https://goo.gl/z7H7DU
2. Файл содержит карту препятствия обозначены символом ‘%’ клетки, по которым можно передвигаться обозначены ‘-’, при этом каждая клетка по которой можно двигаться имеет вес 1.
3. Робот начинает движение в клетке обозначенной буквой ‘P’ и движется в клетку обозначенной буквой ‘T’.
4. Нужно рассчитать оптимальную траекторию пути робота с помощью алгоритма A*.
5. Выведите траекторию в отдельный файл.

Задачи

1. Реализуйте функцию DFS
2. С помощью вашей функции реализуйте алгоритм разбиение графа на компоненты связности.
3. Реализуйте алгоритм проверки орграфа на цикличность
4. Реализуйте алгоритм Крускала/Прима для поиска минимального остовного дерева взвешенного графа.

Задачи

1. Как можно объединить два списка. Напишите программу делающую это
2. Реализуйте очередь через два стека.

Спасибо за внимание!