Очередь по приоритету, пирамидальная сортировка

Артамонов Ю.Н.

Международный университет
природы, общества и человека "Дубна"
филиал Котельники

24 апреля 2017 г.

Определение очереди по приоритету

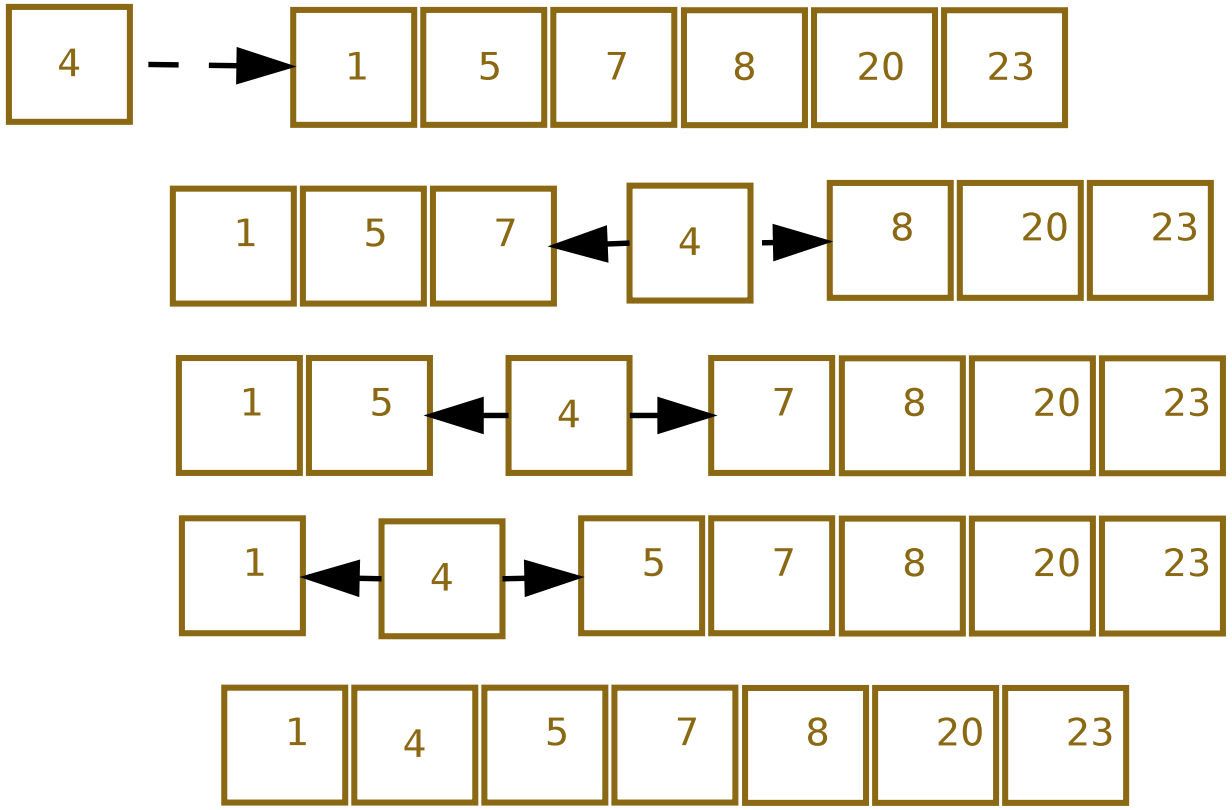
В обычной очереди определены две операции: добавление элемента (в конец очереди) и удаление элемента (из начала очереди). Для очереди с приоритетом операция удаления осуществляется не из начала очереди, а по принципу: «удаление самого большого элемента», или «удаление самого малого элемента». Например, если элементы очереди - конкурирующие процессы, требуется выполнить процесс с наивысшим приоритетом.

Первая простая реализация очереди по приоритету

- Можно использовать простой список, в конец или начало которого добавлять элементы. Каждый элемент очереди имеет специальный атрибут приоритета. Тогда операция добавления имеет оценку $O(1)$.
- Однако, операция удаления элемента будет сопряжена с операцией поиска элемента с наивысшим приоритетом внутри списка, что займет $O(n)$ времени, где n - длина списка.
- При больших n такая структура оказывается неэффективной.

Вторая простая реализация очереди по приоритету

- Можно использовать также простой список, но поддерживать его в отсортированном порядке по приоритету, используя например дихотомический поиск:

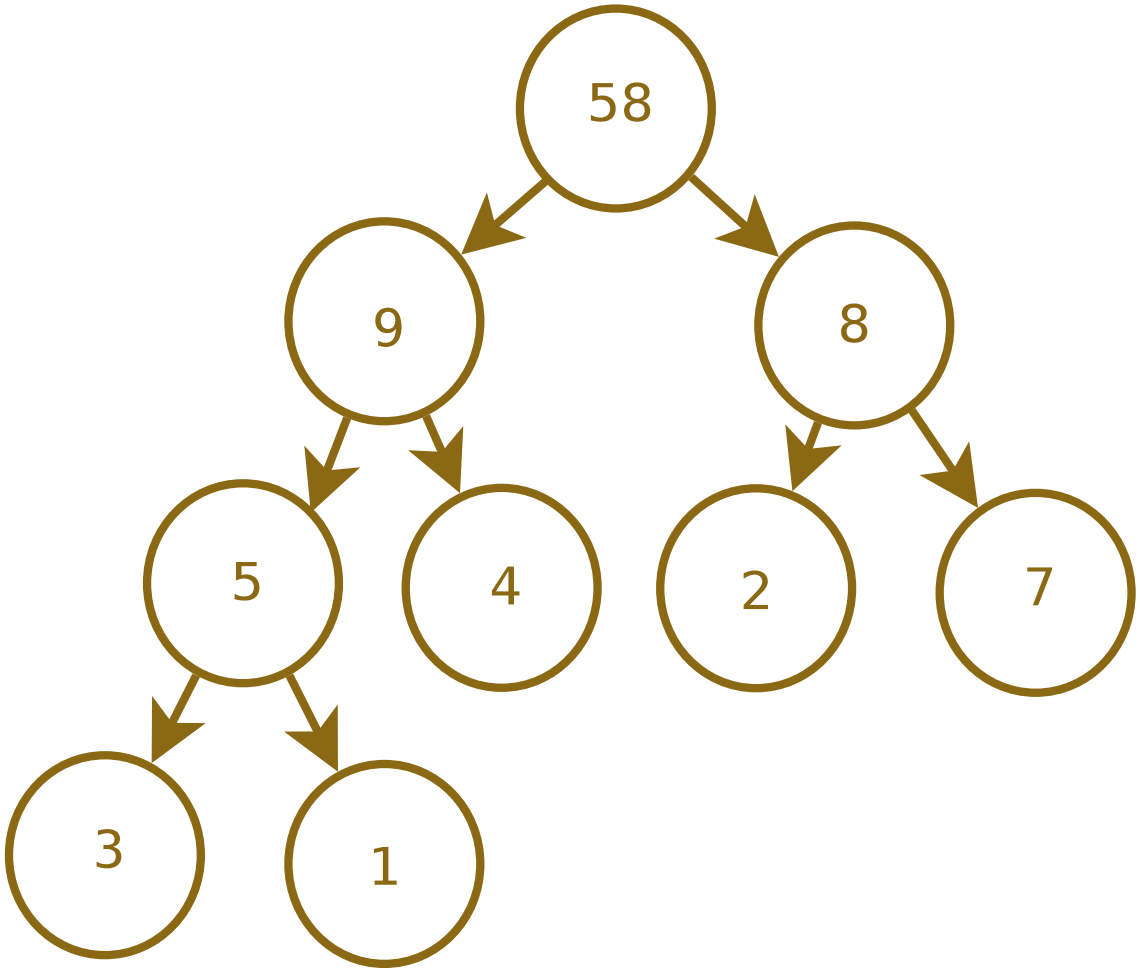


- Операции добавления составит $O(log(n))$, удаление $O(log(n))$. Это много лучше!

Сортирующее дерево (куча - heap)

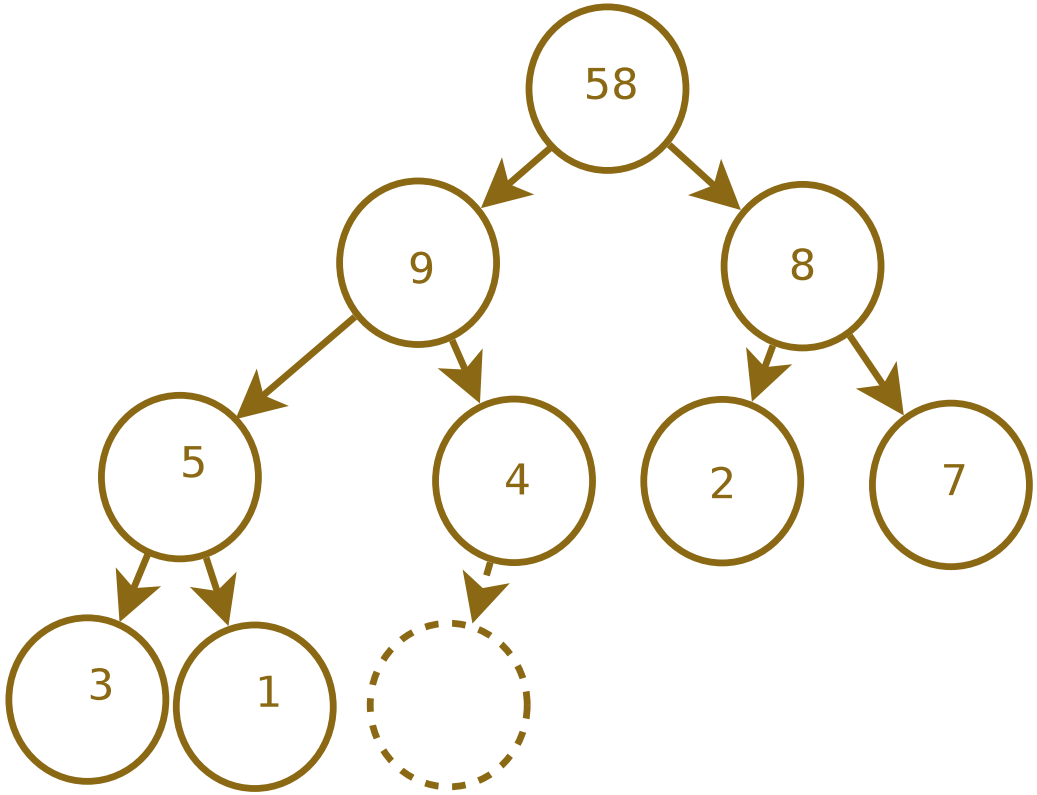
- Классическая структура данных, используемая для создания очереди по приоритету, - сортирующее дерево (другое название - куча, heap).
- Сортирующее дерево является частично упорядоченным полным двоичным деревом.
- Каждый узел сортирующего дерева обладает *пирамидальным свойством* - узел (его приоритет) больше своего левого и правого узлов. Для дочерних узлов нет никаких ограничений: левый может быть больше правого, или наоборот.
- Сортирующее дерево является полным: все уровни, за исключением, может быть, последнего, заполнены. В последнем уровне все узлы размещаются максимально сдвинутыми влево. Такое дерево является максимально сбалансированным.

Пример сортирующего дерева



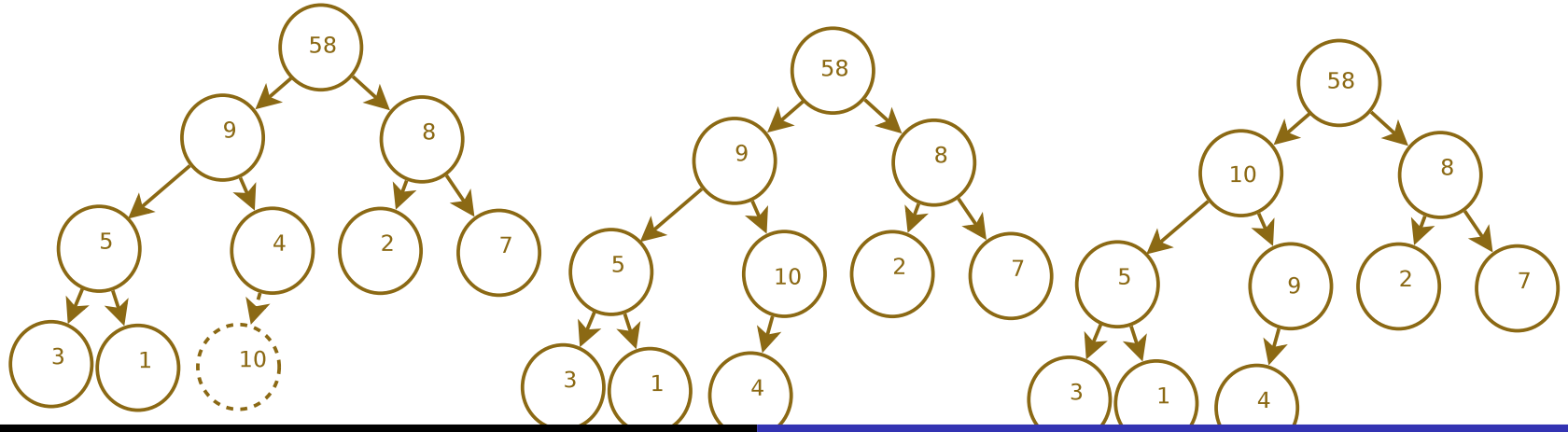
Вставка в сортирующее дерево

- Данный алгоритм известен как алгоритм пузырькового подъема (bubble up).
- Новый элемент добавляется в конец дерева, в единственную позицию, которая соответствует требованию полноты:



Вставка в сортирующее дерево (продолжение)

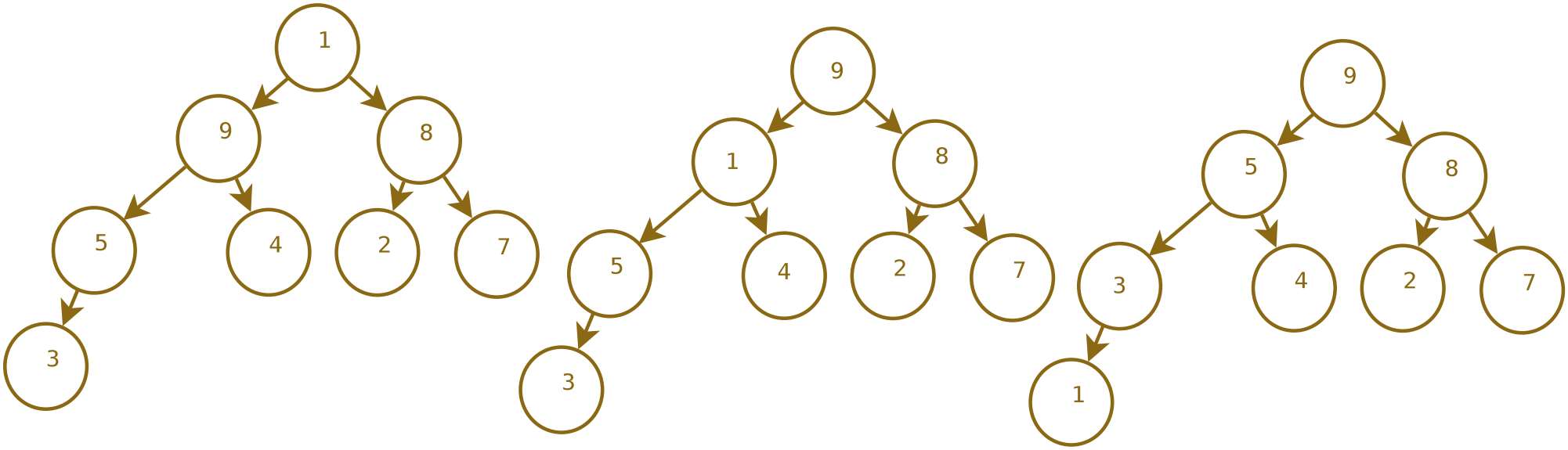
- Для восстановления пирамидального свойства используем следующий алгоритм:
  1. Если новый дочерний узел больше своего родительского узла, меняем его местами с родительским узлом. В своей новой позиции новый узел может быть все еще больше своего нового родительского узла, поэтому их нужно снова помянять местами в этом случае.
  2. Действия п.1 следует повторять, пока не будет достигнута точка, в которой новый узел не больше родительского узла или пока не достигнут корень дерева.



Удаление из сортирующего дерева

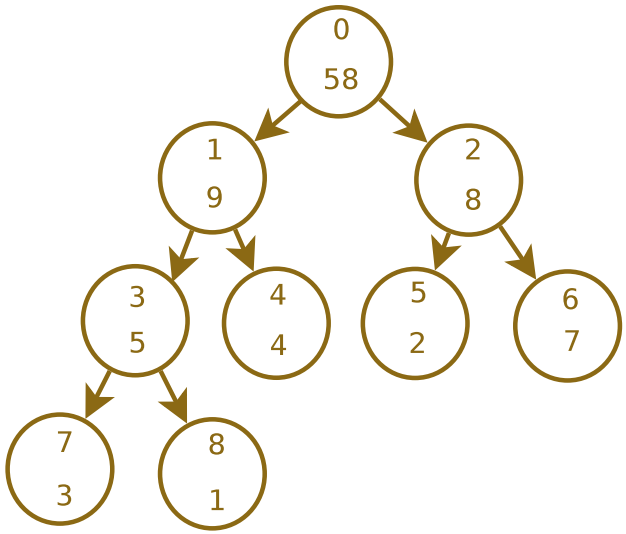
- Удаление осуществляется из корня дерева. Но на самом деле последний узел переписывается вместо корня дерева и сам удаляется
- Для восстановления пирамидального свойства используется алгоритм пузырькового просачивания (bubble down): если корень дерева меньше какого-либо из своих дочерних узлов, то меняем их местами. На новом месте бывший корень дерева может быть также меньше какого-либо из своих новых дочерних узлов, поэтому продолжаем менять его с дочерним узлом, которого он меньше, пока не достигнем ситуации, что все его дочерние узлы меньше, или дочерние узлы отсутствуют (стал листом).

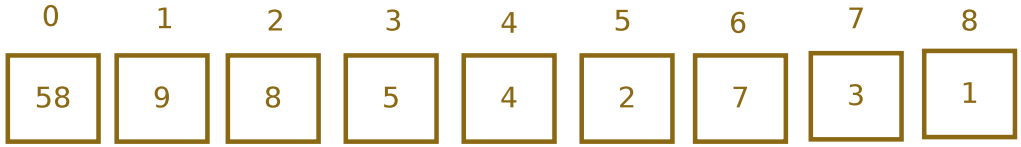
Удаление из сортирующего дерева (продолжение)



Идея пирамидальной сортировки

- В корне любого сортирующего дерева стоит самый максимальный элемент. Поэтому, если удалять из дерева узлы, пока оно не опустеет, все элементы будут удаляться в порядке убывания.
- Однако вместо удаления для экономии памяти можно завернуть дерево в массив:





- Узел под номером n (нумерация с нуля) имеет родителя под номером $\lfloor \frac{n-1}{2} \rfloor$, левый дочерний узел под номером $2n+1$, правый дочерний узел под номером $2n+2$.

Реализация пирамидальной сортировки

```
class heap():
__body = []

def __init__(self):
    self.__body=[]

def add(self,n):
    self.__body.append(n)
    self.bubble_up(len(self.__body)-1)
```

Реализация пирамидальной сортировки (продолжение)

```
    def get_left_child(self,number):
        try:
            return self.__body[2*number+1]
        except:
            return None

    def get_right_child(self,number):
        try:
            return self.__body[2*number+2]
        except:
            return None
```

Реализация пирамидальной сортировки (продолжение)

```
def get_parent(self,number):
    if number-1>=0:
        return self.__body[(number-1)//2]
    else:
      return self.__body[0]

def look(self):
  return self.__body

def add_list(self,L):
    for i in L:
        self.add(i)
```

Реализация пирамидальной сортировки (продолжение)

```
def bubble_up(self,number):
        if number == 0:
            return True
        if (self.__body[number]<=self.get_parent(number)):
            return True
        else:
            a=self.__body[number]
            self.__body[number] = self.get_parent(number)
            self.__body[(number-1)//2]=a
            self.bubble_up((number-1)//2)
```

Реализация пирамидальной сортировки (продолжение)

```
    def sort(self):
        n=len(self.__body)-1
        while n >= 0:
            self.__body[0] ,self.__body[n] = self.__body[n] ,self.
__body[0]
            self.bubble_down(0,n)
            n -=1
```

Реализация пирамидальной сортировки (продолжение)

```
def bubble_down(self,number,length):
    if number == length:
        return True
    if 2*number+1>=length:
        return True
    else:
        left=self.get_left_child(number)
    if 2*number+2>=length:
        right=None
    else:
        right=self.get_right_child(number)
```

Реализация пирамидальной сортировки (продолжение)

```
        if (right == None):
            right = left-1
        if left>right:
            if self.__body[number]<left:
                self.__body[2*number+1] = self.__body[number]
                self.__body[number] = left
                self.bubble_down(2*number+1,length)
        else:
            if self.__body[number]<right:
                self.__body[2*number+2] = self.__body[number]
                self.__body[number] = right
                self.bubble_down(2*number+2,length)
```