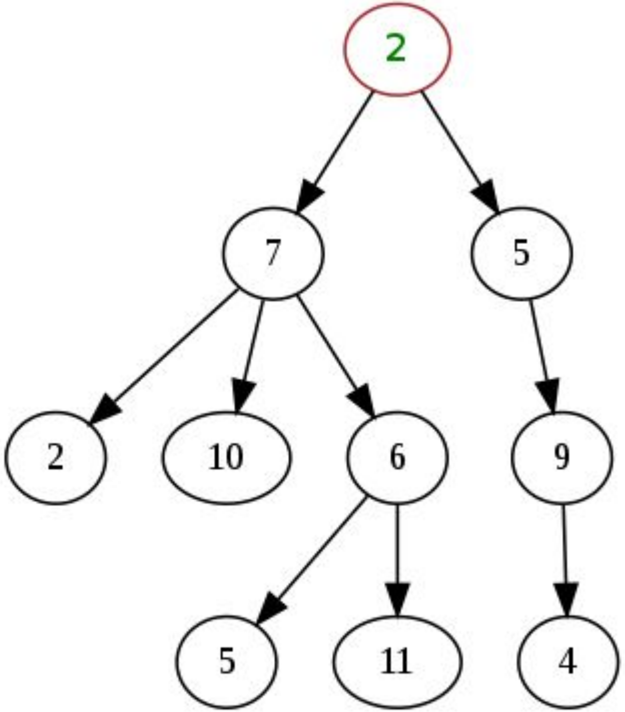
2. Бінарне дерево пошуку. Купа. Heapsort

Дерево

- Дерево (tree) – абстрактний тип даних (ADT), що має колекцію вузлів (nodes), починаючи з кореневого (root) вузла.
- Кожен вузол містить значення (value), а також список посилань на вузли нижчого рівня (вузли-нащадки, child nodes). Цей список посилань:
  - Не може містити дублікатів
  - Жодне з посилань не може вказувати на корінь дерева
- Вузли, що не мають нащадків, називаються листами (leaf nodes)



Довільне дерево

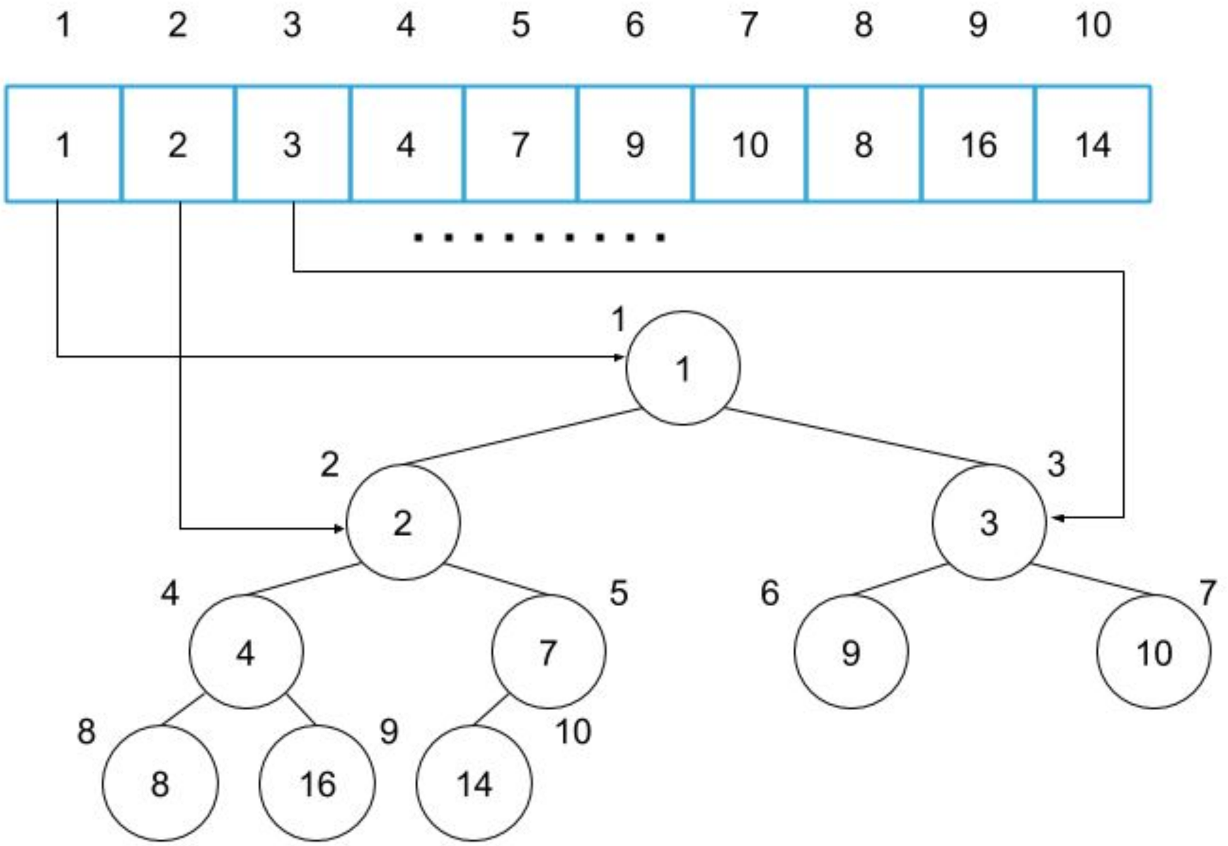
Купа

- Купа (heap) – структура даних, заснована на дереві, яке відповідає властивості купи
- **Властивість купи:** для будь-якого вузла С, якщо Р – це його вузол-предок (parent node), то:
    - Значення у вузлі **Р менше або рівне** значенню у вузлі **С** (мінімальна купа, **min heap**)
    - Значення у вузлі **Р більше або рівне** значенню у вузлі **С** (максимальна купа, **max heap**)
- Купа зазвичай **реалізується** за допомогою **неявної структури, у основі якої лежить масив** (динамічний або фіксованого розміру), кожен елемент якого представляє вузол у дереві, а зв'язок parent-child визначається неявно по індексу

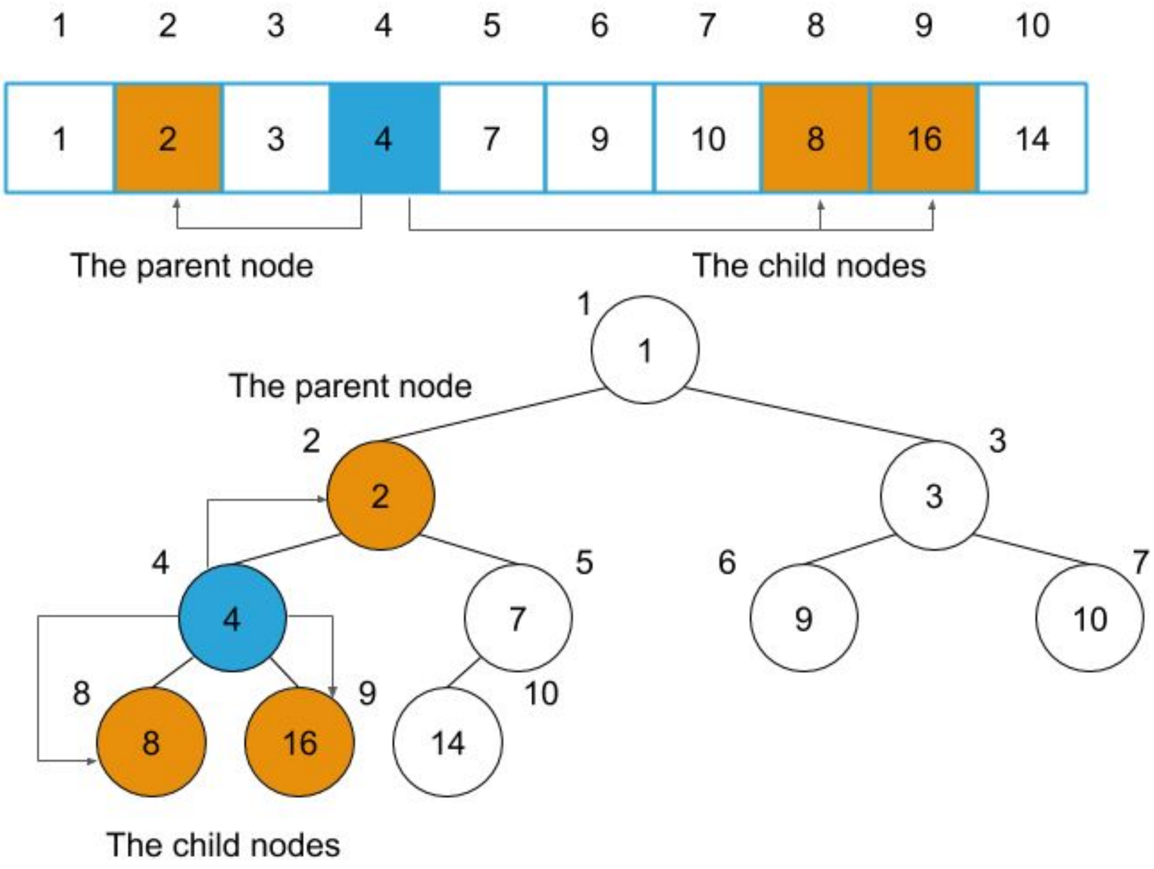
Представлення min heap у вигляді масиву

Індексування (для масивів з індексами, що починаються з 1):

- Кореневий вузол: `i = 1` (перший елемент масиву)
- Вузол-предок: `parent(i) = i / 2`
- Лівий нащадок: `left(i) = 2i`
- Правий нащадок: `right(i) = 2i + 1`



Представлення min heap у вигляді масиву



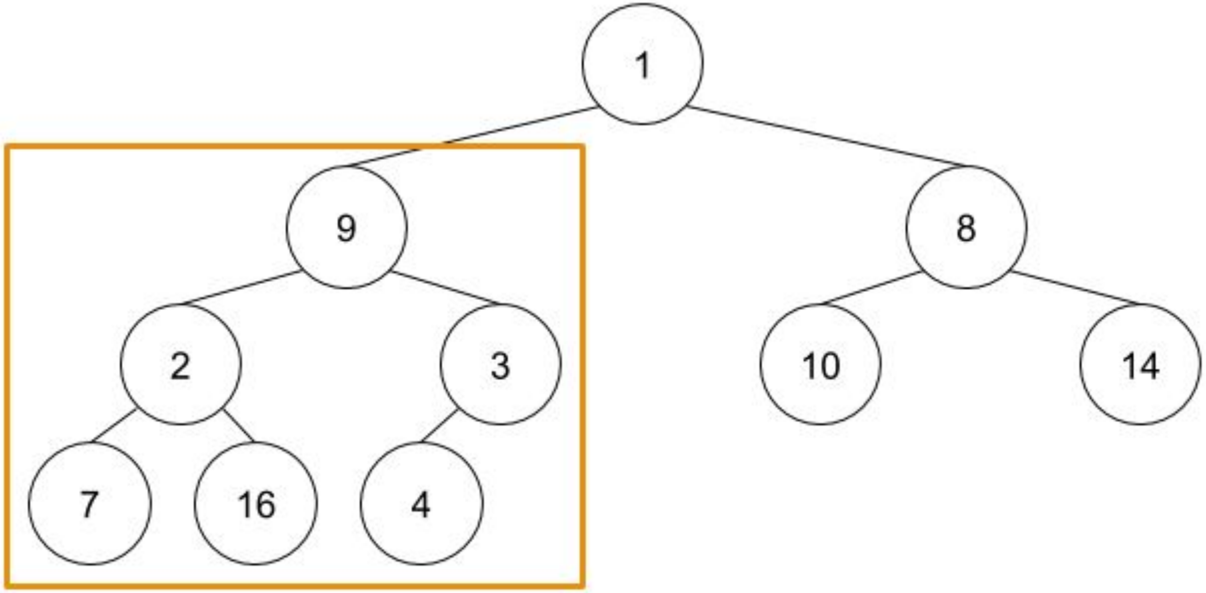
Реалізація купи

- Створюємо клас, що визначає купу
- Конструктор приймає масив `arr` як аргумент. У цьому масиві зберігатимуться елементи купи
- Вхідний масив `arr` ще не впорядкований. Результатом виконання методу `build_min_heap` буде впорядкований масив `arr`, що задовольняє властивості купи
- Реалізували методи `parent`, `left`, `right`, які оперують індексами і повертають індекс відповідного вузла (вузла-предка, лівого нащадка, правого нащадка відповідно)

```python
class Heap:
    def __init__(self, arr):
        self.arr = arr
        self.build_min_heap()

    @staticmethod
    def parent(i):
        return i // 2

    @staticmethod
    def left(i):
        return i * 2 + 1

    @staticmethod
    def right(i):
        return i * 2 + 2

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

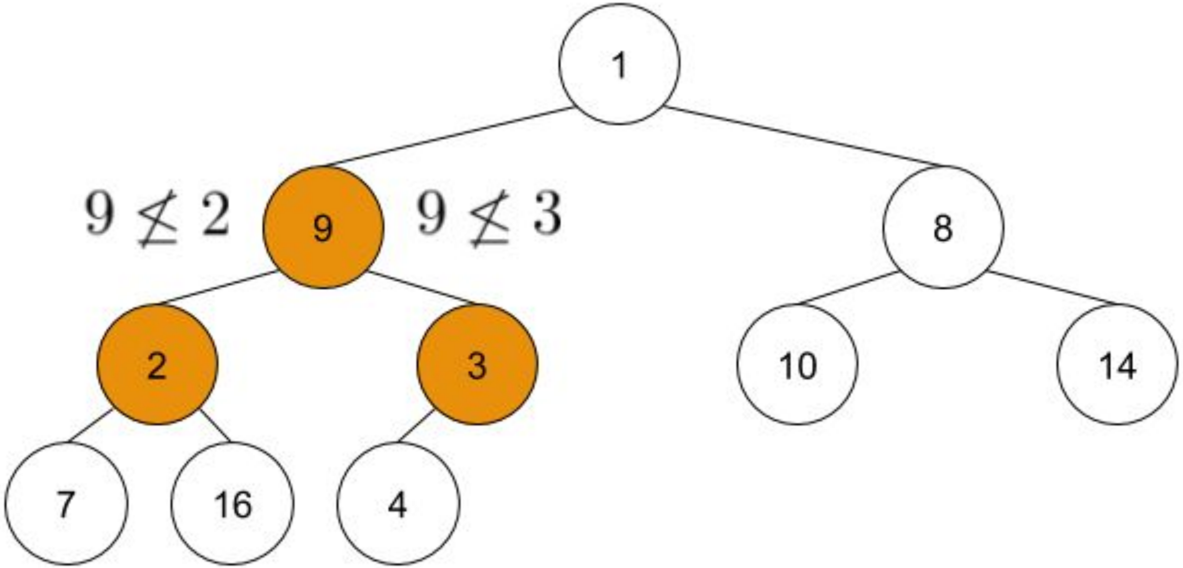
1. Побудова купи

Розглянемо таке невпорядковане дерево, і його виділене помаранчевим піддерево:



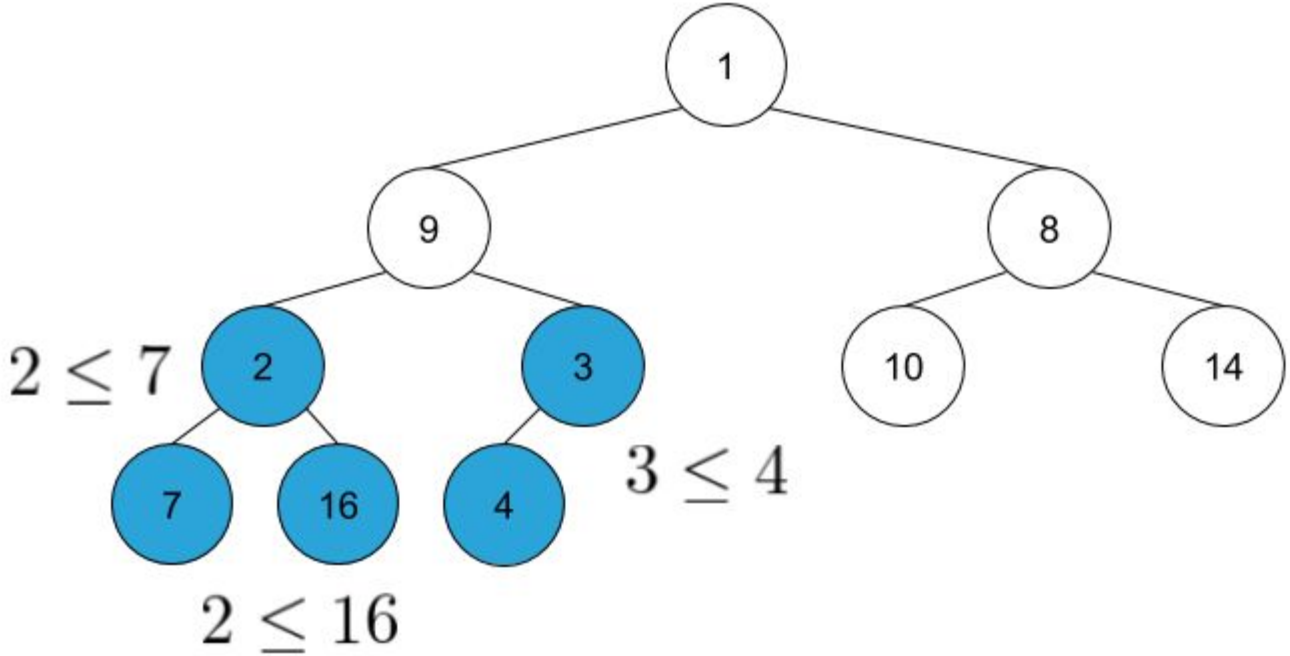
2. Побудова купи

Три виділених вузли не задовольняють властивості min heap:



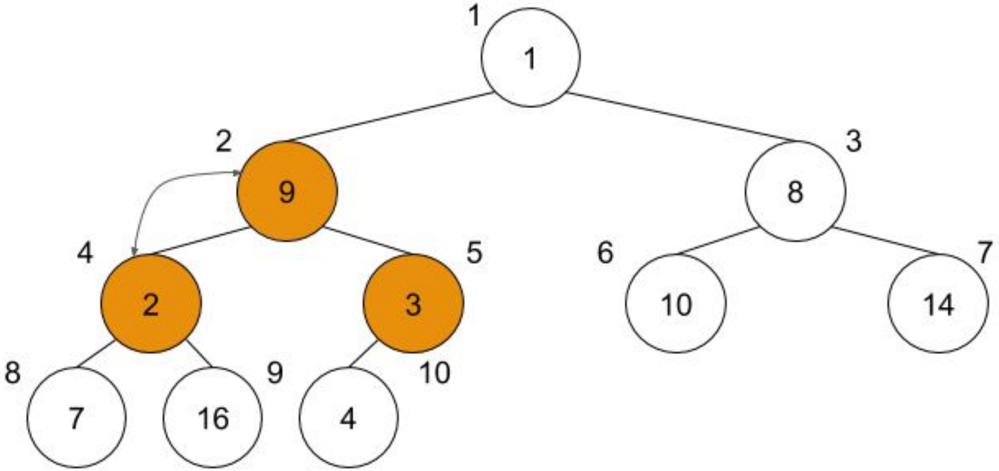
3. Побудова купи

При цьому нащадки задовольняють властивості min heap:



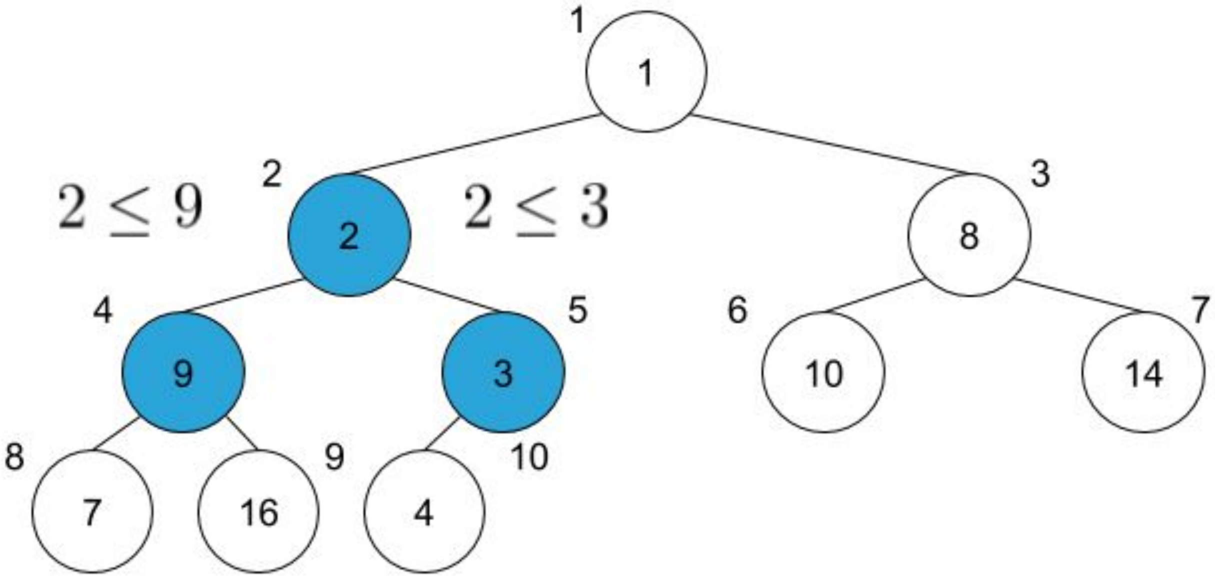
4. Побудова купи – операція sift_down

Потрібно визначити операцію (назвемо її `sift_down`), яка внесе зміни в дерево, переносячи певні вузли вниз, так, щоб відновити властивість купи. Ця операція приймає індекс вузла як аргумент. Нижче показано хід виконання `sift_down(2)`:



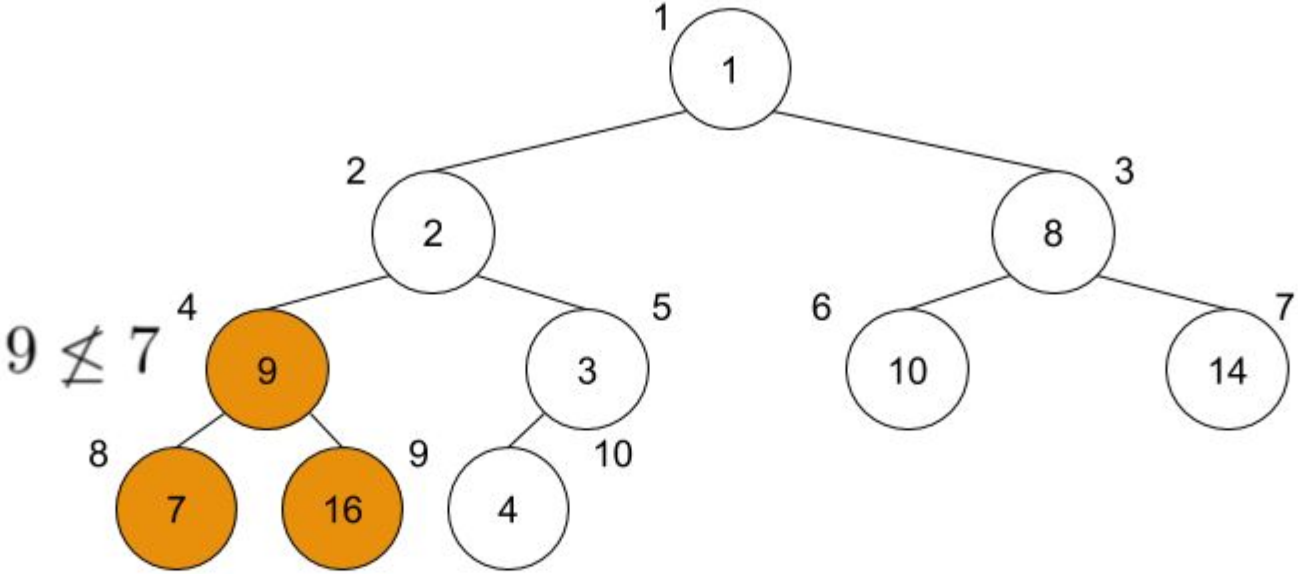
5. Побудова купи – операція sift_down

Результатом виконання `sift_down(2)` є зміна місцями вузла-предка і вузла-нащадка із найменшим значення. Тепер властивість купи виконується у виділеному піддереві:



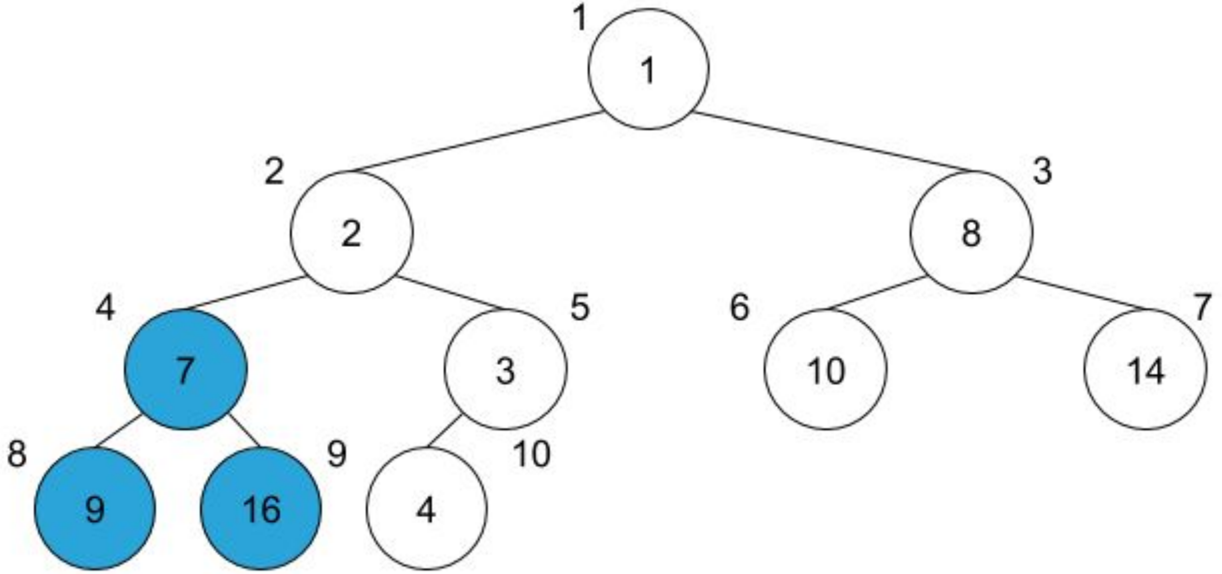
6. Побудова купи – операція sift_down

Після виконання `sift_down(2)` залишається ще одне піддерево, яке не задовольняє властивості купи:



7. Побудова купи – операція sift_down

Це виправиться після виконання `sift_down(4)` (застосування `sift_down` рекурсивно до вузла, на місце якого було переміщено вузол із верхнього рівня). Результуюче дерево є min heap:



8. Побудова купи – операція sift_down

- Таким чином, операція `sift_down(X)`: змінює місцями певний вузол X із його одним із його нащадків, якщо значення у вузлі X більше, ніж значення у вузлі-нащадку
- При цьому для обміну вибирається нащадок із мінімальним значенням з-поміж двох
- Для виконання `sift_down` необхідно, щоб вузли-нащадки та їхні піддерева задовольняли властивості купи

9. Побудова купи – реалізація операції sift_down

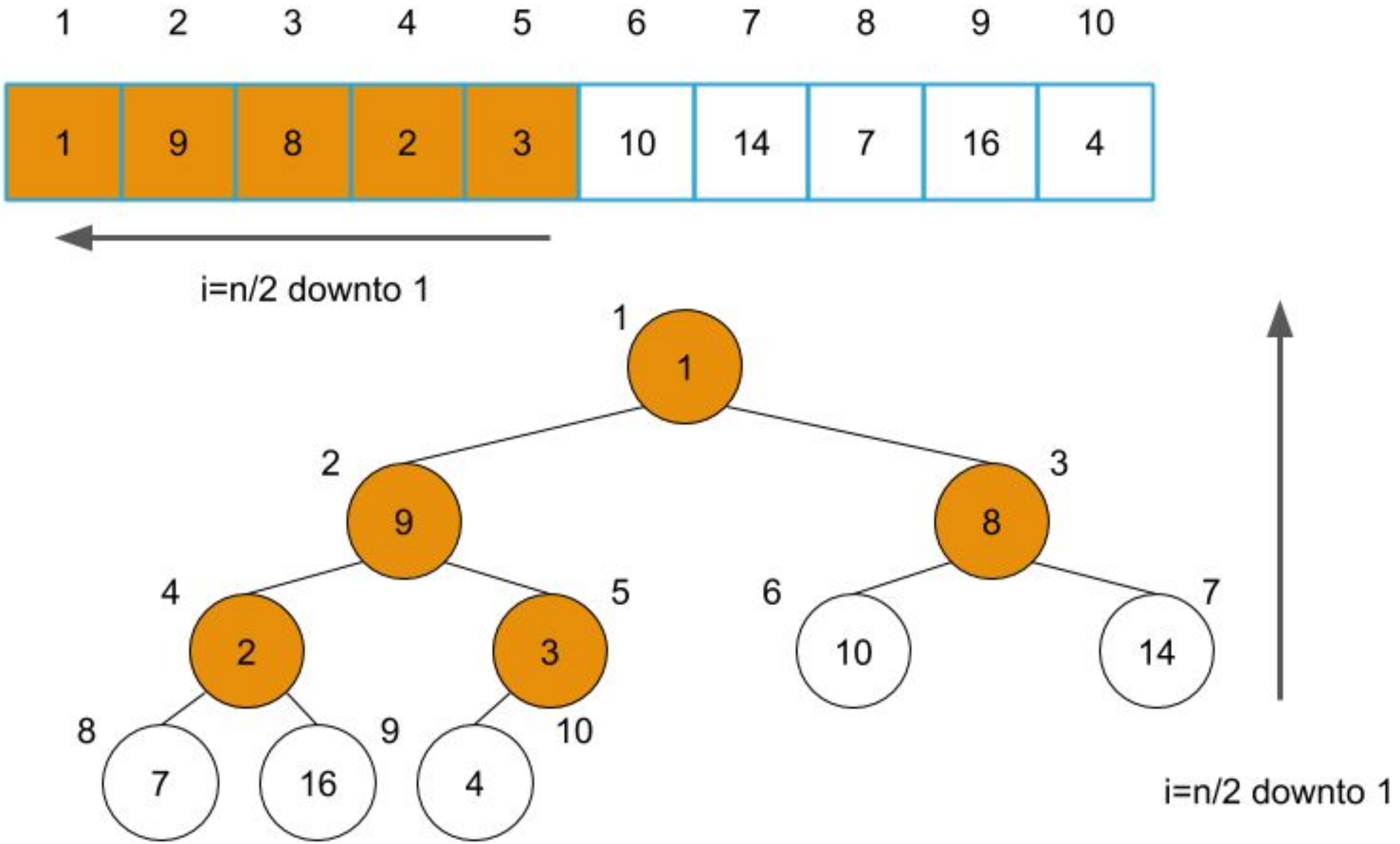
```python
def min_child(self, i):
   left = Heap.left(i)
   right = Heap.right(i)
   length = self.size()
   smallest = i
   if left < length and self.arr[i] > self.arr[left]:
       smallest = left
   if right < length and self.arr[smallest] > self.arr[right]:
       smallest = right
   return smallest

def sift_down(self, i):
   while True:
       min_child = self.min_child(i)
       if i == min_child:
           break
       self.arr[i], self.arr[min_child] = self.arr[min_child], self.arr[i]
       i = min_child
```

10. Побудова купи – операція build_min_heap

- Для побудови min heap потрібно проітеруватись по всіх вузлах, окрім листів, починаючи з нижнього рівня, і виконати sift_down
- Псевдокод:

```
build_min_heap(array)

   for i=n/2 downto 1

       do min_heapify(array, i)
```

11. Побудова купи – операція build_min_heap



12. Побудова купи – реалізація операції build_min_heap

```
def build_min_heap(self):
    for i in reversed(range(len(self.arr) // 2)):
        self.sift_down(i)
```

Операція sift_up. Додавання в купу

- Операція `sift_up` аналогічна до `sift_down`, але переміщує певний вузол вгору, допоки не буде відновлено властивість купи:

```
def sift_up(self, i):
   while i > 0:
       parent = Heap.parent(i)
       if self.arr[i] < self.arr[parent]:
           self.arr[parent], self.arr[i] = self.arr[i], self.arr[parent]
       i = parent
```

- Ця операція використовується для відновлення властивості купи при додаванні нового елемента (`insert`)
- Відповідно для реалізації insert потрібно додати елемент у кінець внутрішнього масиву (`arr`), і викликати `sift_up` для цього нового вузла

```
def insert(self, e):
   self.arr.append(e)
   self.sift_up(self.size() - 1)
```

Операція delete_min (видалення мінімального)

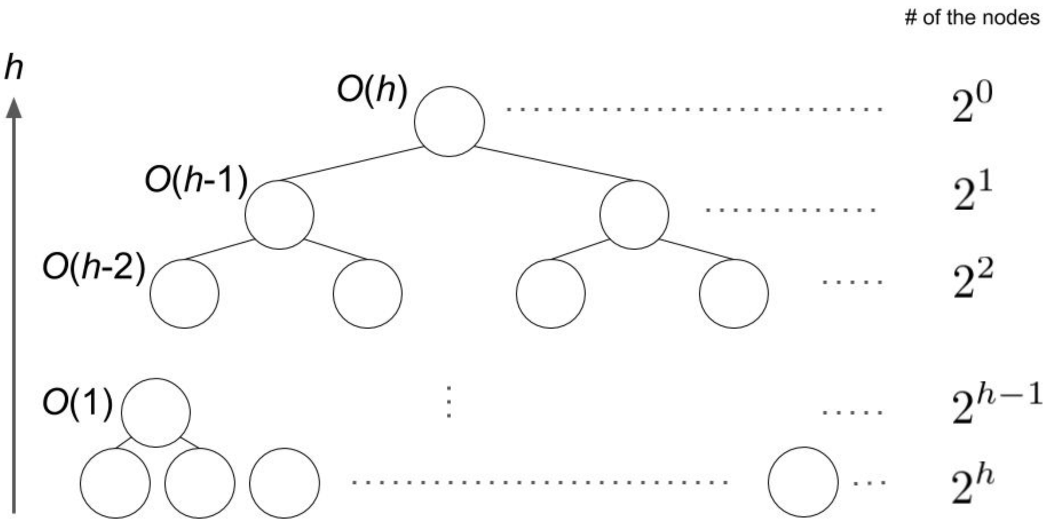
- Операція `sift_down` також використовується для відновлення властивості купи після видалення мінімального елементу (який знаходиться на початку масиву, тобто в корені дерева)
- Реалізація видалення мінімального елементу:

```
def delete_min(self):
   if self.size() == 0:
       return None
   self.arr[0], self.arr[-1] = self.arr[-1], self.arr[0]
   head = self.arr.pop()
   if self.size() > 0:
       self.sift_down(0)
   return head
```

- Цей код: змінює місцями перший (мінімальний елемент) і останній елементи масиву `arr`, видаляє останній елемент масиву (на якому розміщено мінімальний елемент), а потім виконує `sift_down` з кореня

Складність операції sift_down

- На малюнку показано складність операції `sift_down`, залежно від рівня дерева (до вузла на якому застосовується `sift_down`), де h – висота дерева:



Складність операції build_min_heap

- Складність `build_min_heap` – сума складностей для внутрішніх вузлів (виключаючи листи):

$$O(2^0 \times h + 2^1 \times (h-1) + \cdots + 2^{h-1} \times 1) = O(2^h)$$

- Просумувавши кількості вузлів на кожному рівні отримаємо загальну кількість елементів у купі (n):

$$2^0 + 2^1 + \cdots + 2^{h-1} + 2^h = n$$

- Перетворивши геометричну прогресію, отримаємо висоту дерева:

$$1 + 2 + 2 \cdot 2^1 + \cdots + 2 \cdot 2^{h-1} = 1 + 2(2^h - 1) = n$$

$$\therefore h = \log(n+1) - 1$$

Складності операцій

- Таким чином, складність **`build_min_heap`** – **`O(n)`**, **`sift_down`** – **`O(log(n))`**
- Складність **`sift_up`** – теж **`O(log(n))`**, це можна довести аналогічно
- Операція `sift_down` використовується при побудові купи та при видаленні елемента, `sift_up` – при додаванні елемента
- Відповідно і insert, і delete_min мають логарифмічну складність **`O(log(n))`**
- Реалізація `build_min_heap`, що використовує операцію `sift_down`, називається **алгоритмом Флойда**
- Така реалізація дозволяє ефективно будувати купу з `n` елементів (за `O(n)`), бо створення порожньої купи та додавання кожного з `n` елементів через метод `insert` (за `O(log(n))`) потребувало би `O(n*log(n))` операцій

Підсумок. Складності операцій

Операція	Складність
`find_min`	`O(1)`
`delete_min`	`O(log(n)) (*)`
`insert`	`O(log(n)) (*)`
`build_min_heap`	`O(n)`

(*) – якщо купу реалізовано за допомогою динамічного масиву, наведена складність є **амортизованою**. Додавання або видалення елементу з динамічного масиву може призвести до зміни його розміру (зазвичай, подвоєння або зменшення вдвічі), що має лінійну (від кількості елементів) складність, що обумовлена необхідністю копіювання у масив нового розміру.

heapsort

- Маючи відповідну купу (min heap – для сортування за зростанням, max heap – для сортування за спаданням), можна реалізувати алгоритм сортування heapsort:

```
def heap_sort(arr):
   heap = Heap(arr)
   sorted_arr = []
   while heap.size() > 0:
       sorted_arr.append(heap.delete_min())
   return sorted_arr
```

Складність heapsort

- Складність:
  - Створення купи (і відповідно виклик `build_min_heap`) потребує `O(n)` часу
  - Далі, для кожного з n елементів, виконується операція `delete_min`, яка працює за `O(n*log(n))`
  - Відповідно ця реалізація heapsort потребує `O(n + n*log(n)) =` **`O(n*log(n))` часу**, і **`O(n)` додаткової пам'яті** (оскільки ми копіюємо впорядковану послідовність у додатковий масив)

Heapsort (без додаткової пам’яті)

- Також можливо уникнути створення додаткового масиву, використовуючи підхід, схожий до selection sort. Псевдокод:

```
procedure heapsort(a, count) is
    build_min_heap(a, count)
    end ← count - 1
    while end > 0 do
        swap(a[end], a[0])
        end ← end - 1
        siftDown(a, 0, end)
```

Застосування купи

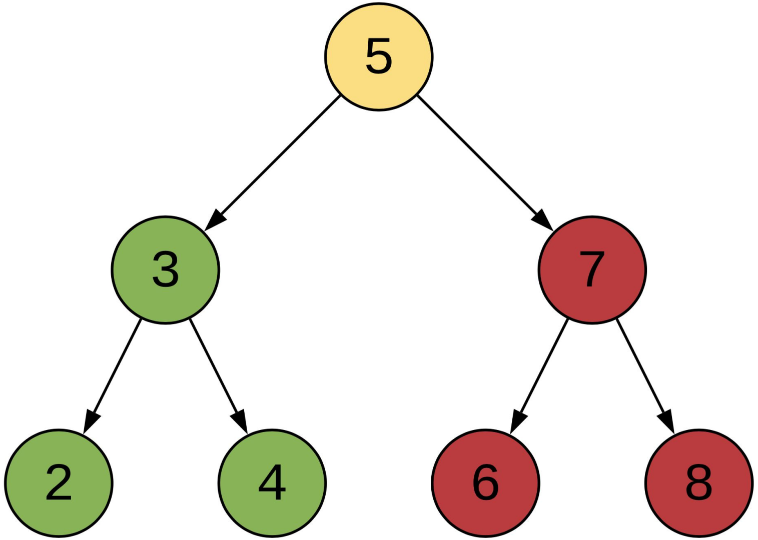
- Сортування – heapsort
- Черга з пріоритетом (priority queue)
- Алгоритми вибірки (selection): пошук мінімального/максимального елементу за константний час, пошук k-того найбільшого/найменшого елементу за суб-лінійний час
- K-way merge: ефективне об’єднання декількох відсортованих потоків даних у один
- Алгоритми на графах

Бінарне дерево пошуку

- Бінарне дерево – дерево, у якому кожен вузол може мати не більше двох нащадків
- Бінарне дерево пошуку (binary search tree – BST) – структура даних, що засновується на бінарному дереві, і зберігає впорядковані елементи
- Бінарне дерево пошуку дозволяє реалізовувати множини (set), і відповідно швидку перевірку на наявність певного ключа в множині, а також пошук значення по ключу
- Пошук у бінарному дереві пошуку засновується на бінарному пошуку: під час обходу дерева на кожному рівні визначається (залежно від значення шуканого ключа та значення у поточному вузлі), у якому піддереві продовжувати пошук – лівому чи правому

Основна властивість BST

- Якщо позначити вузол-предок як P, а вузли-нащадки (лівий/правий відповідно) як C_1 і C_2, а ключ вузла P як `key(P)` (аналогічно для C_1 і C_2), то у BST завжди одночасно виконуються умови:
  - $\mathrm{key}(C_1) < \mathrm{key}(P)$
  - $\mathrm{key}(C_2) > \mathrm{key}(P)$
- На малюнку праворуч жовтим позначено кореневий вузол. Усі елементи, виділені зеленим (ліве піддерево) менше за нього, виділені червоним (праве піддерево) – більше за нього



Реалізація BST

- Цей код визначає основну структуру BST
- Клас Node – це вузол у дереві, що має ключ, значення, а також посилання на ліві та праві вузли
- Сам клас BST містить лише посилання на корінь дерева, екземпляр класу Node

```
class BST:
   class Node:
       def __init__(self, key, value):
           self.key = key
           self.value = value
           self.left = None
           self.right = None

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

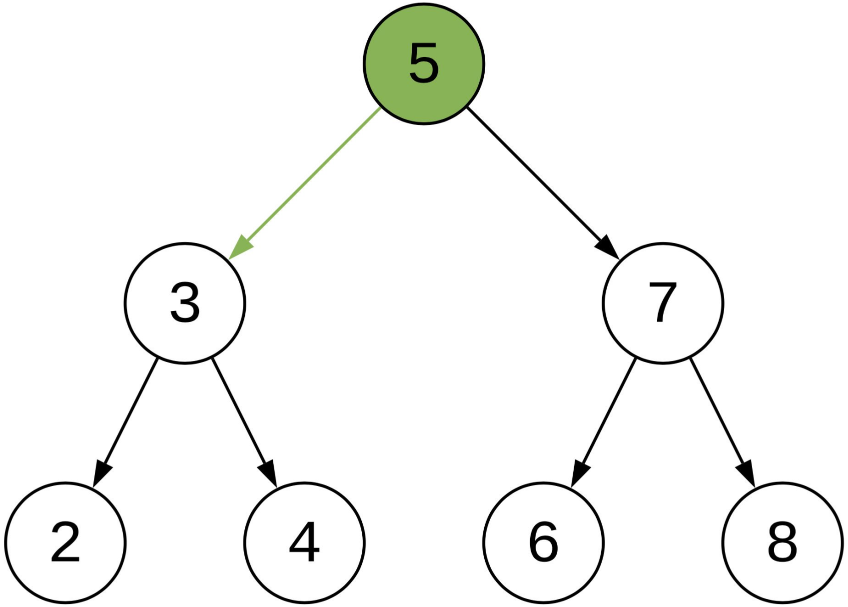
Пошук у бінарному дереві

Алгоритм. Починаючи з кореня дерева:

1. Якщо шуканий ключ дорівнює ключу в поточному вузлі, **повертаємо значення та закінчуємо пошук**
2. Якщо шуканий ключ менше за ключ у поточному вузлі, переходимо до лівого нащадка, знову перевіряємо умови (1)-(3)
3. Якщо шуканий ключ більше за ключ у поточному вузлі, переходимо до правого нащадка, знову перевіряємо умови (1)-(3)
4. Якщо виконалась умова (2) або (3), але відповідного нащадка немає, **завершуємо пошук – шуканого ключа в дереві немає**

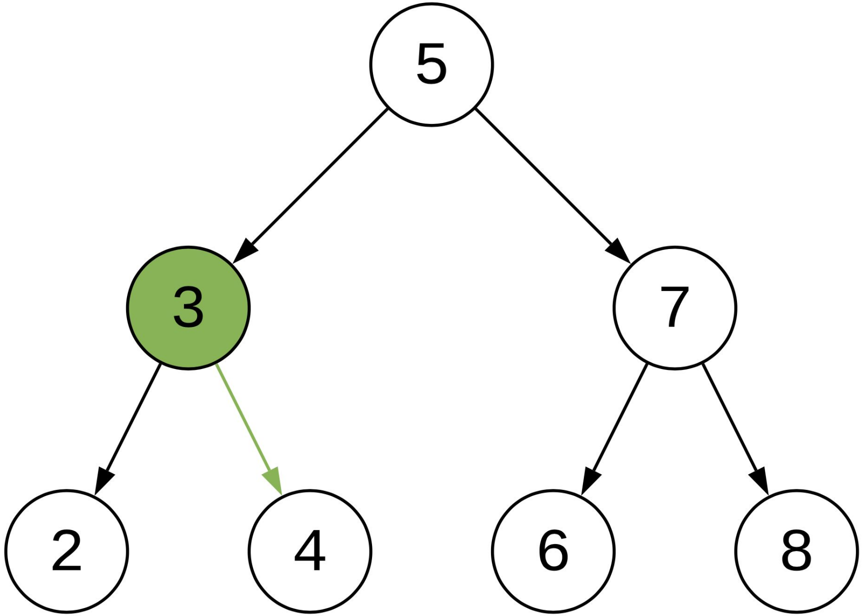
1. Пошук у бінарному дереві. Приклад 1

Шукаємо ключ, що дорівнює **4**. Поточний вузол і напрямок переходу показано зеленим, показано лише ключі (не значення). Шуканий ключ менше за ключ у поточному вузлі, переходимо ліворуч:



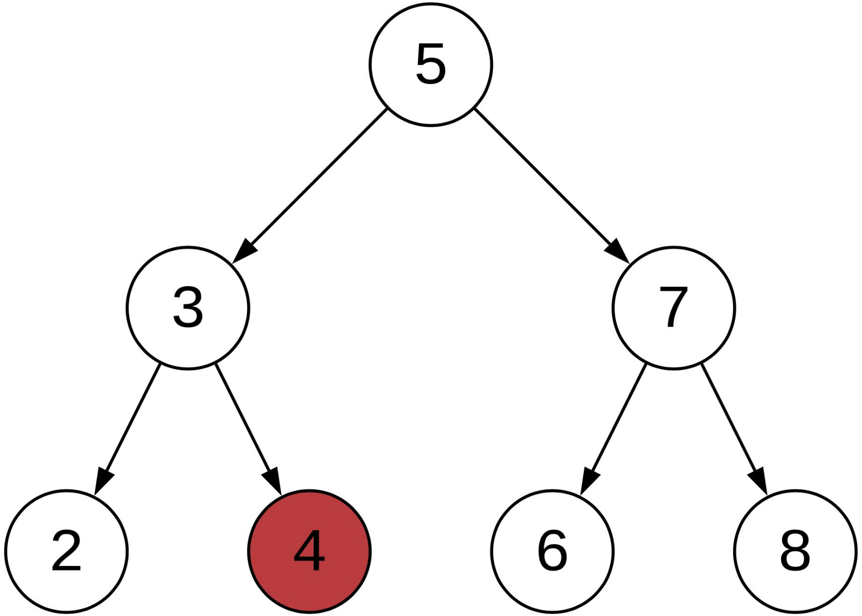
2. Пошук у бінарному дереві. Приклад 1

Шукаємо ключ, що дорівнює **4**. Шуканий ключ більше за поточний, переходимо праворуч:



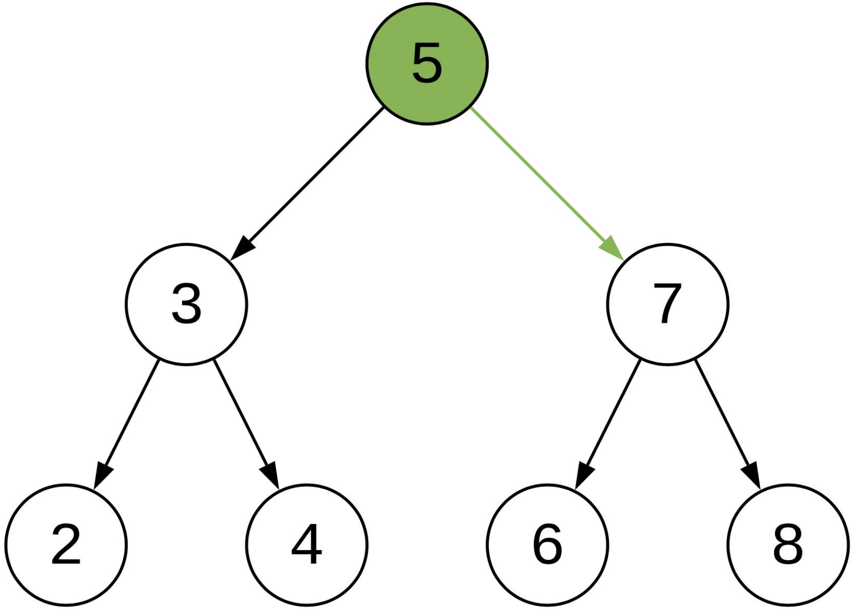
2. Пошук у бінарному дереві. Приклад 1

Шуканий ключ (**4**) знайдено:



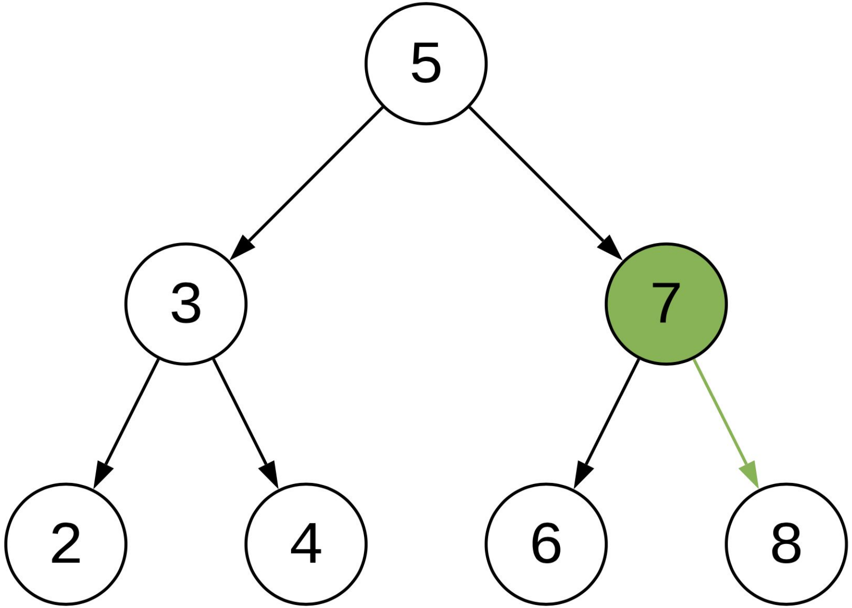
1. Пошук у бінарному дереві. Приклад 2

Шукаємо ключ, що дорівнює **9**. Шуканий ключ більше за поточний, переходимо праворуч:



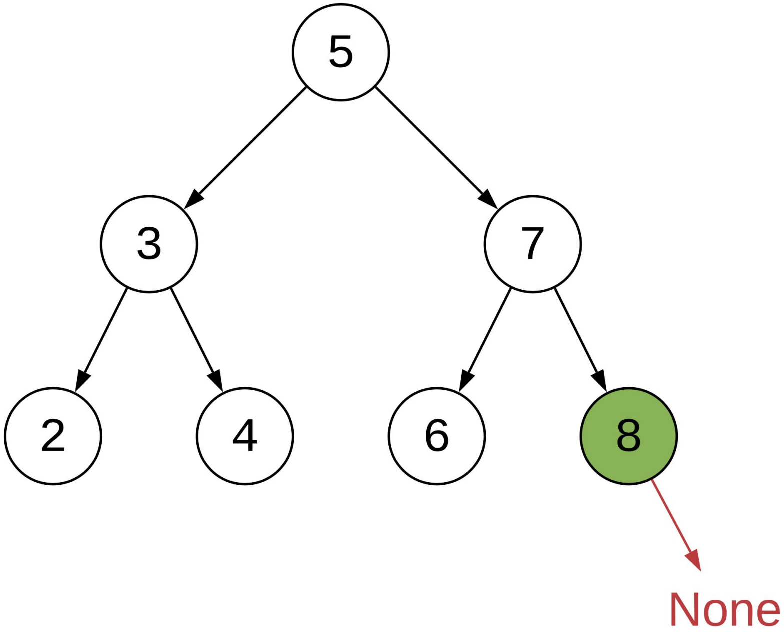
2. Пошук у бінарному дереві. Приклад 2

Шукаємо ключ, що дорівнює **9**. Шуканий ключ більше за поточний, переходимо праворуч:



2. Пошук у бінарному дереві. Приклад 2

Шукаємо ключ, що дорівнює **9**. Шуканий ключ більше за поточний, переходимо праворуч. Правого нащадка немає, отже, шуканого ключа немає у дереві, завершуємо пошук:



Реалізація пошуку у BST

- Відповідно до описаного алгоритму, починаємо від кореня, і виконуємо переходи вліво/вправо залежно від значення шуканого ключа та значення ключа в поточному вузлі
- Якщо поточний вузол має шукане значення, повертаємо значення в цьому вузлі
- Якщо дійшли до вузла, що не має нащадків, і ключ досі не знайдено, повертаємо None

```python
def get(self, key):
    x = self.root
    while x is not None:
        if key < x.key:
            x = x.left
        elif key > x.key:
            x = x.right
        else:
            return x.value
    return None
```

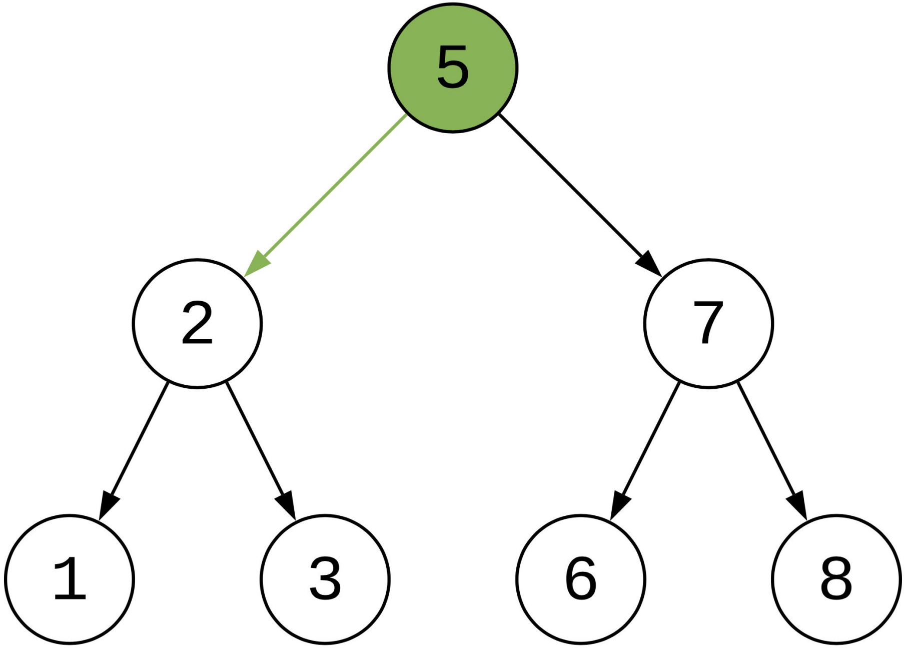
Додавання пари “ключ-значення” у BST

Алгоритм.

- Шукаємо у дереві ключ, який потрібно вставити. Два випадки:
  - Якщо такого ключа немає, додаємо новий вузол
  - Якщо ключ знайдено, замінюємо у відповідному вузлі значення на нове

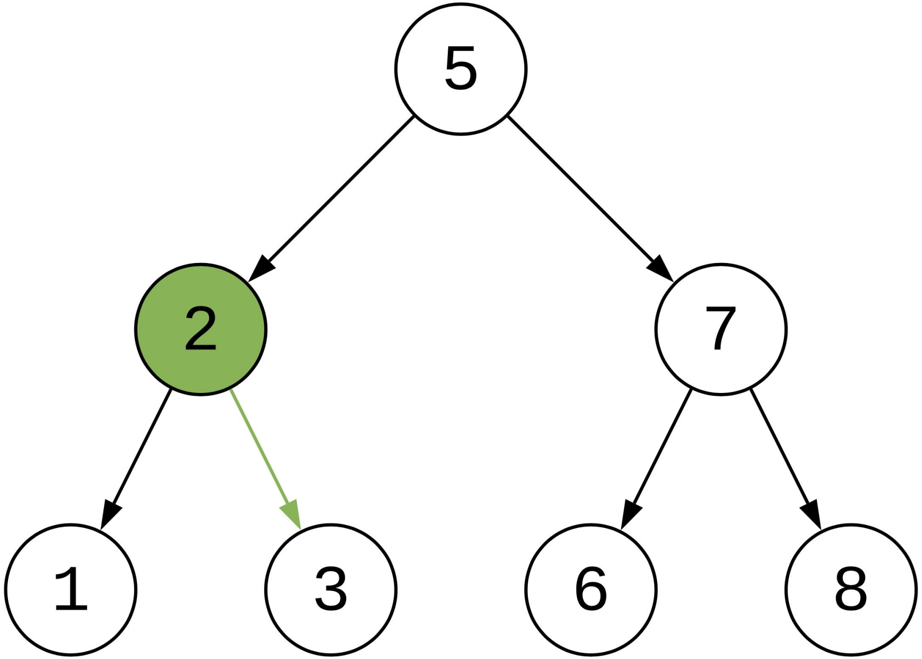
1. Вставка у бінарне дерево. Приклад

Вставимо пару “ключ-значення”, у якій ключ дорівнює **4**. Спочатку виконуємо пошук:



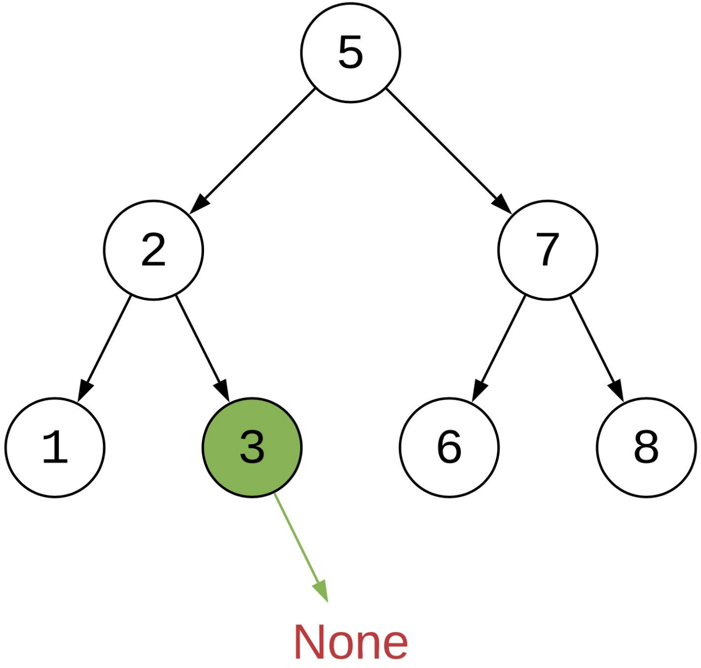
2. Вставка у бінарне дерево. Приклад

Вставляємо пару “ключ-значення”, у якій ключ дорівнює **4**. Продовжуємо пошук:



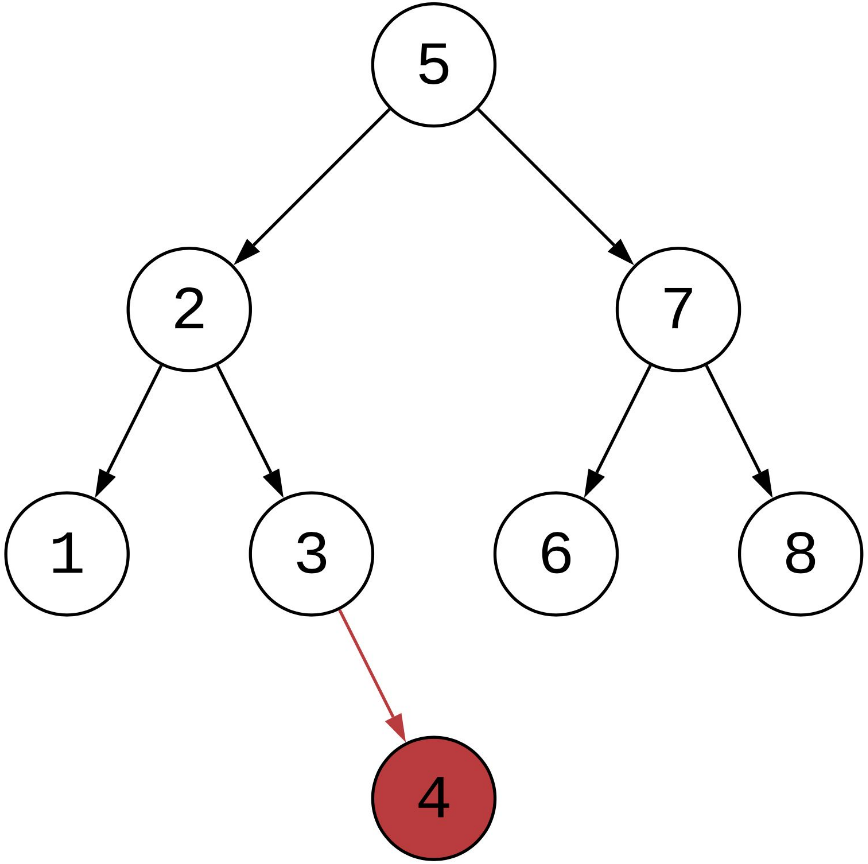
3. Вставка у бінарне дерево. Приклад

Вставляємо пару “ключ-значення”, у якій ключ дорівнює **4**. Дійшли до листа дерева: ключ, що вставляється, більше за ключ у поточному вузлі, але правого нащадка немає, відповідно, цього ключа ще немає в дереві:



4. Вставка у бінарне дерево. Приклад

Вставляємо пару “ключ-значення”, у якій ключ дорівнює **4**. Створюємо новий вузол із цією парою “ключ-значення”, і додаємо посилання на цей вузол (до вузла, що містить ключ, що дорівнює 3). Нові вузол і посилання виділені червоним:



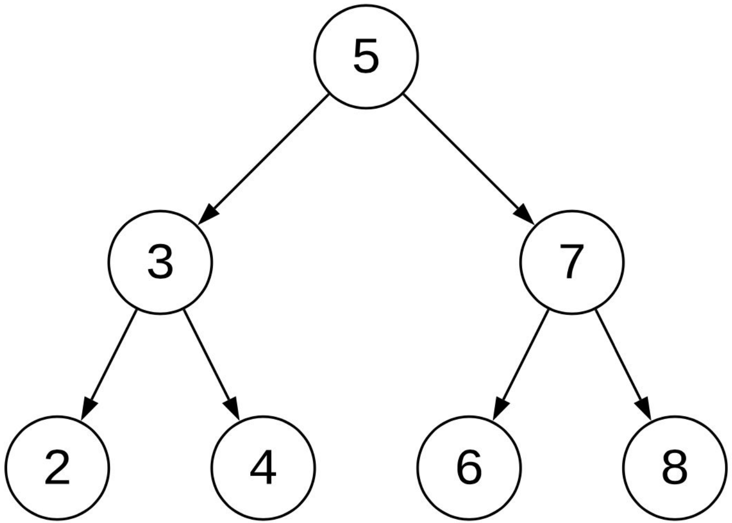
Реалізація вставки у BST

- Рекурсивна реалізація
- Якщо переданий у `__put` вузол порожній, створюємо новий із заданою парою “ключ-значення”
- При розгортанні рекурсії у зворотному боці виставляється посилання на вузол-нащадок у вузлі-предку

```
def put(self, key, value):
   self.root = self.__put(self.root, key, value)

def __put(self, x, key, value):
   if x is None:
       return BST.Node(key, value)

   if key < x.key:
       x.left = self.__put(x.left, key, value)
   elif key > x.key:
       x.right = self.__put(x.right, key, value)
   else:
       x.value = value
   return x
```

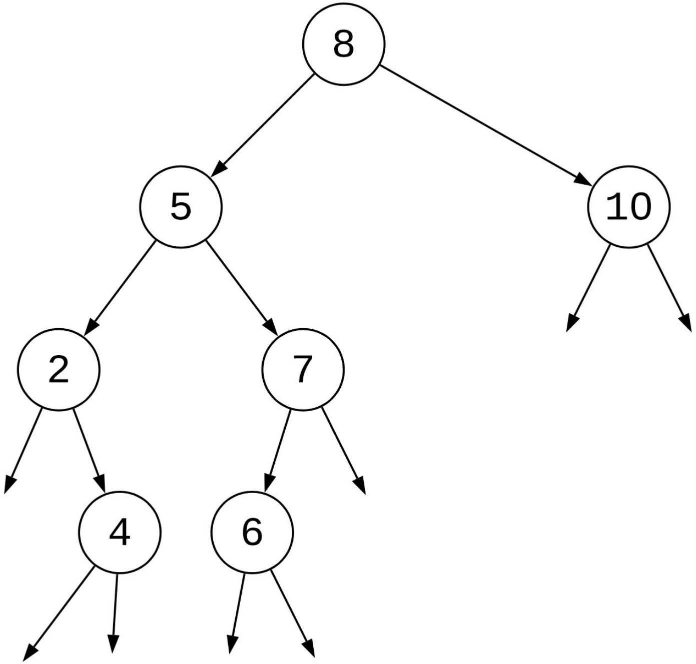
Форма дерева

- Одному й тому ж набору ключів може відповідати декілька BST
- Кількість порівнянь, які необхідно виконати для пошуку/вставки, дорівнює `1 + <глибина вузла>`
- Форма дерева залежить від порядку вставки

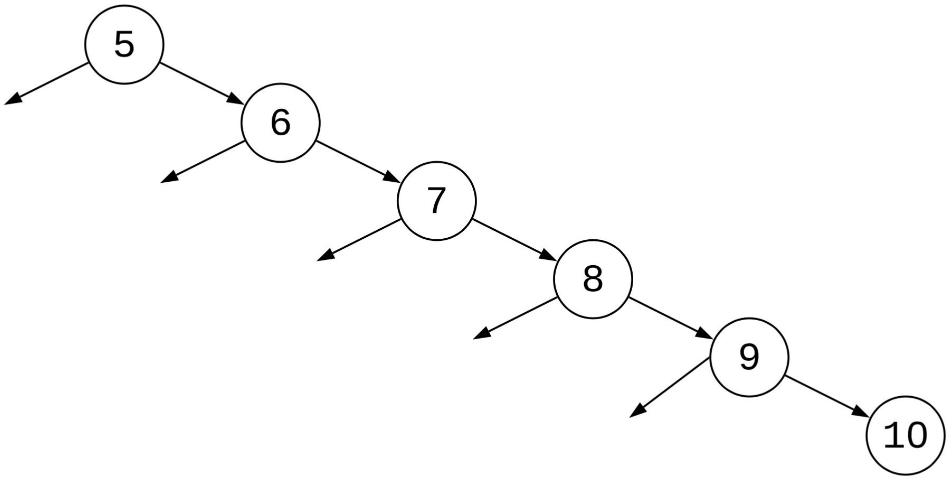


Найкращий випадок

Форма дерева. Середній випадок



Середній випадок



Найгірший випадок

Складність операцій

- Таким чином, **складність операцій вставки** та **пошуку** – **`O(h)`**, де **h** – висота дерева. У найкращому та середньому випадках **`h ~ log(n)`**
- Висота дерева, пропорційна логарифму від кількості елементів, досягається при вставці елементів у випадковому порядку
- **У найгіршому випадку складність цих операцій зростає до `O(n)`**
- Цьому завадити можна:
  - Випадковим чином перемішавши елементи перед вставкою у дерево. Цей метод рідко використовується, оскільки потрібно знати весь набір ключів наперед і не змінювати його
  - Використовуючи самозбалансовані дерева
- **Балансування дерева** – реорганізація дерева таким чином, щоб його висота була близькою до оптимальної – `O(log(n))`

Видалення у BST

- Не розглянуто, буде на наступній лекції про самозбалансовані дерева
- Реалізація є досить складною, до того ж, видалення у звичайному (не самозбалансованому) BST призводить до погіршення балансування, а відповідно, зниження швидкості вставок/пошуку
- Відомо, що в результаті виконання достатньо довгої змішаної випадкової послідовності вставок і видалень, висота дерева наближається до **$O(\sqrt{n})$**

Порівняння хеш-таблиці та BST

Структура даних	Найгірший випадок			Середній випадок		
	Пошук	Вставка	Видалення	Пошук	Вставка	Видалення
Хеш-таблиця	`O(N)`	`O(N)`	`O(N)`	`O(1)`	`O(1)`	`O(1)`
Бінарне дерево пошуку	`O(N)`	`O(N)`	`O(N)`	`O(log(N))`	`O(log(N))`	`O(sqrt(N))`

Переваги хеш-таблиць	Переваги BST
Швидші операції у середньому випадку	Елементи зберігаються впорядковано
Швидше на практиці (не лише з точки зору аналізу) через розміщення елементів в неперервному блоці пам’яті	Передбачуваний логарифмічний час операцій (для збалансованих дерев)
Простіша реалізація	Ефективніше використовується пам’ять

Посилання на код

1. Heap: https://gist.github.com/DmitriyTkachenko/853e096e3be7793ef2e56445dcc0c754
2. BST: https://gist.github.com/DmitriyTkachenko/9735fa5c4c6d917c7267fe36e6244140

Додаткові джерела

1. https://towardsdatascience.com/data-structure-heap-23d4c78a6962
2. https://runestone.academy/runestone/books/published/pythonds/Trees/BinaryHeapImplementation.html