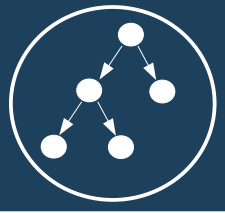
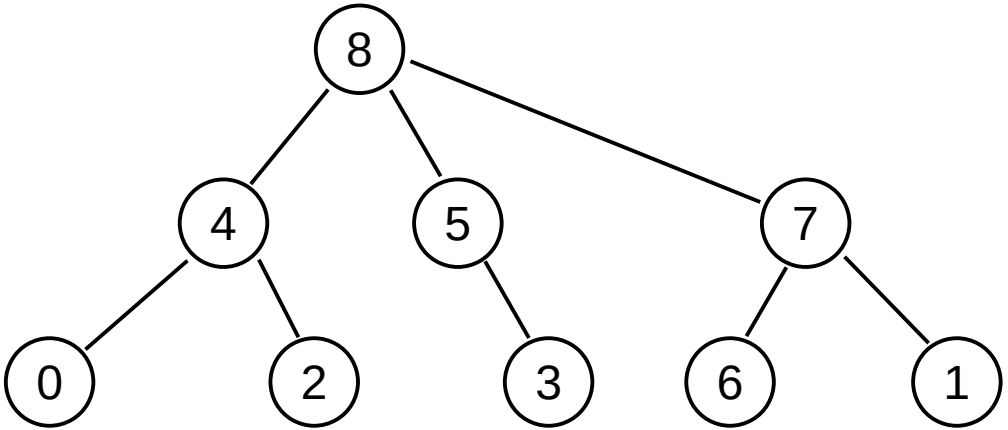


Data Structures and Algorithms

Структуры данных.
Бинарная куча

Куча определение

Куча (heap) — разновидность дерева (и как следствие разновидность графа). Отличительной особенностью является наличие выделенной вершины (узла) — вершина кучи. Упорядочивание которое относит кучу к деревьям следующее - ключ дочернего узла не больше ключа родительского узла. Это приводит к тому, что в вершине кучи всегда расположен элемент с максимальным ключом.



Поддерживаемые операции

Кучи обычно поддерживают следующие операции:

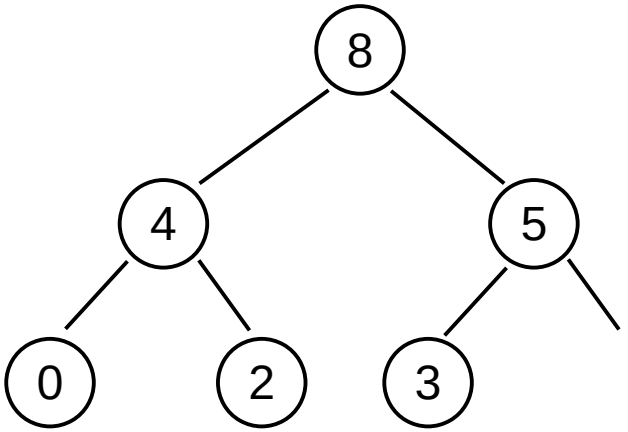
- Добавление нового узла в кучу
- Удаление элемента из кучи
- Найти узел максимальным значением ключа

Бинарная куча

Бинарная, двоичная куча (пирамида) одна из наиболее часто используемых видов кучи. Отличительными особенностями бинарной кучи являются:

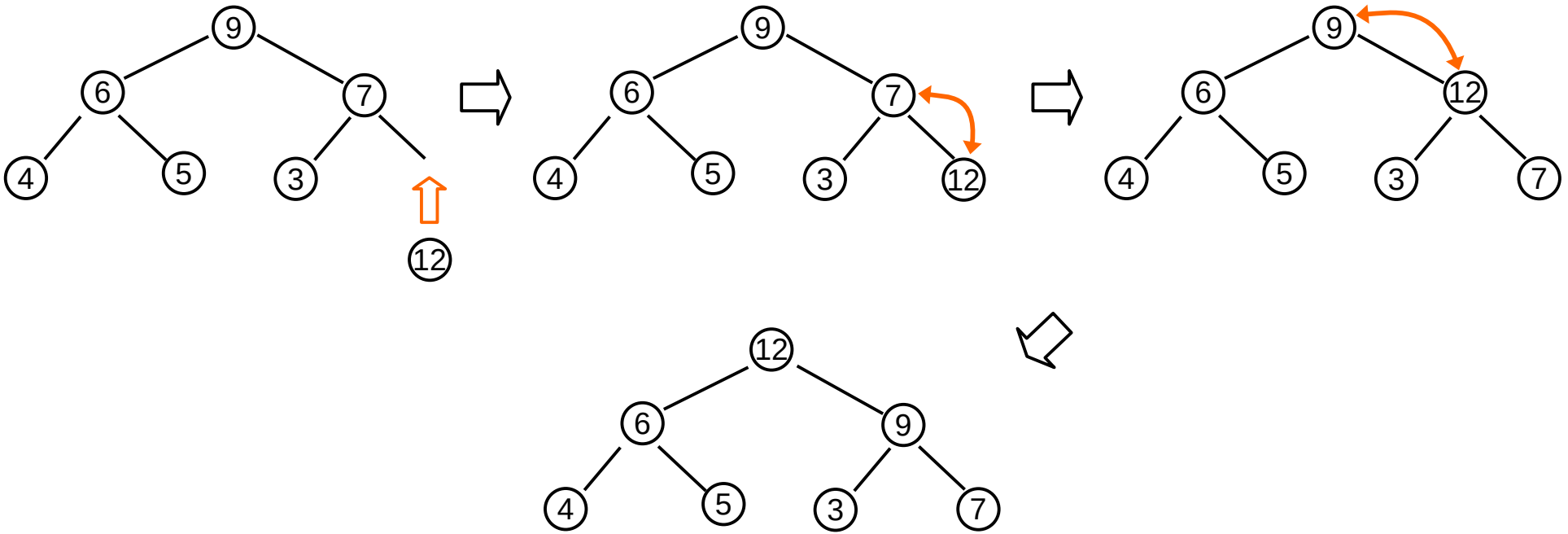
- У каждого узла не более 2-х дочерних узлов
- Все уровни дерева (а куча разновидность дерева) заполнены полностью. Исключение составляет последний уровень он может быть заполнен не полностью, в этом случае заполнение обязательно должно идти слева направо
- Глубина всех листьев отличается максимум на один слой.

Бинарная куча графическое представление



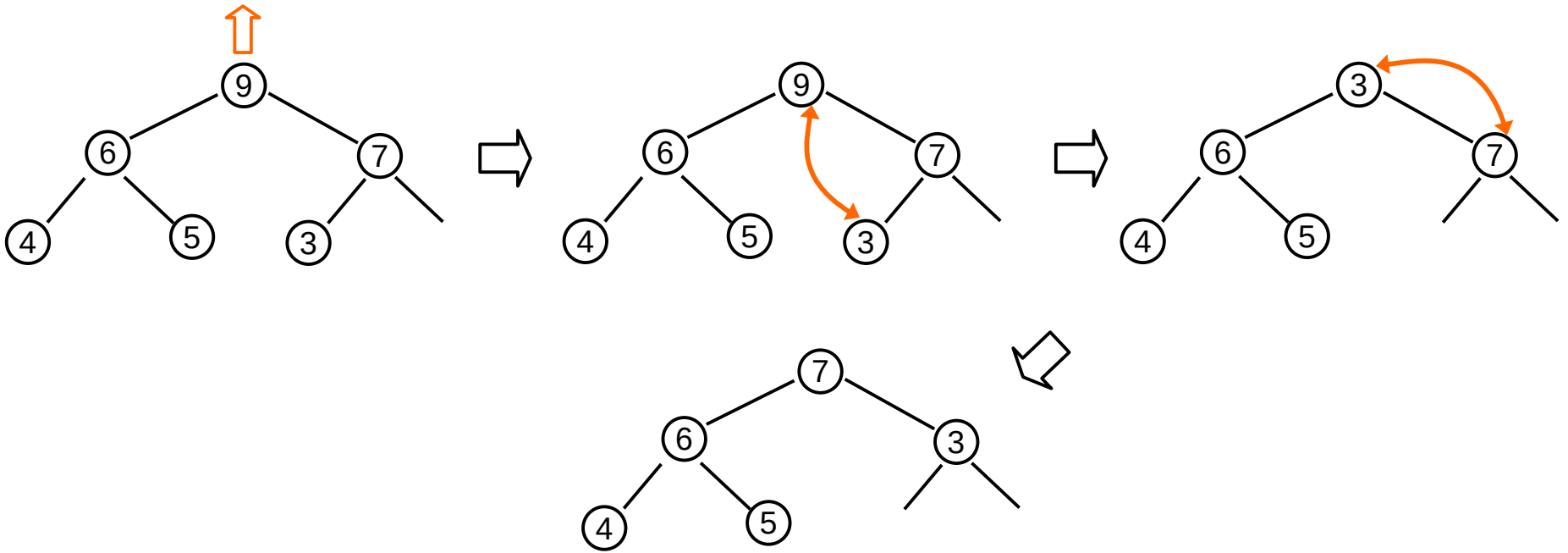
Как можно видеть в данной куче реализованы все особенности бинарной кучи. У каждого узла не более двух дочерних узлов. Значение ключа дочернего узла не больше значения ключа родительского узла. Последний слой заполняется слева направо.

Добавление элемента в кучу



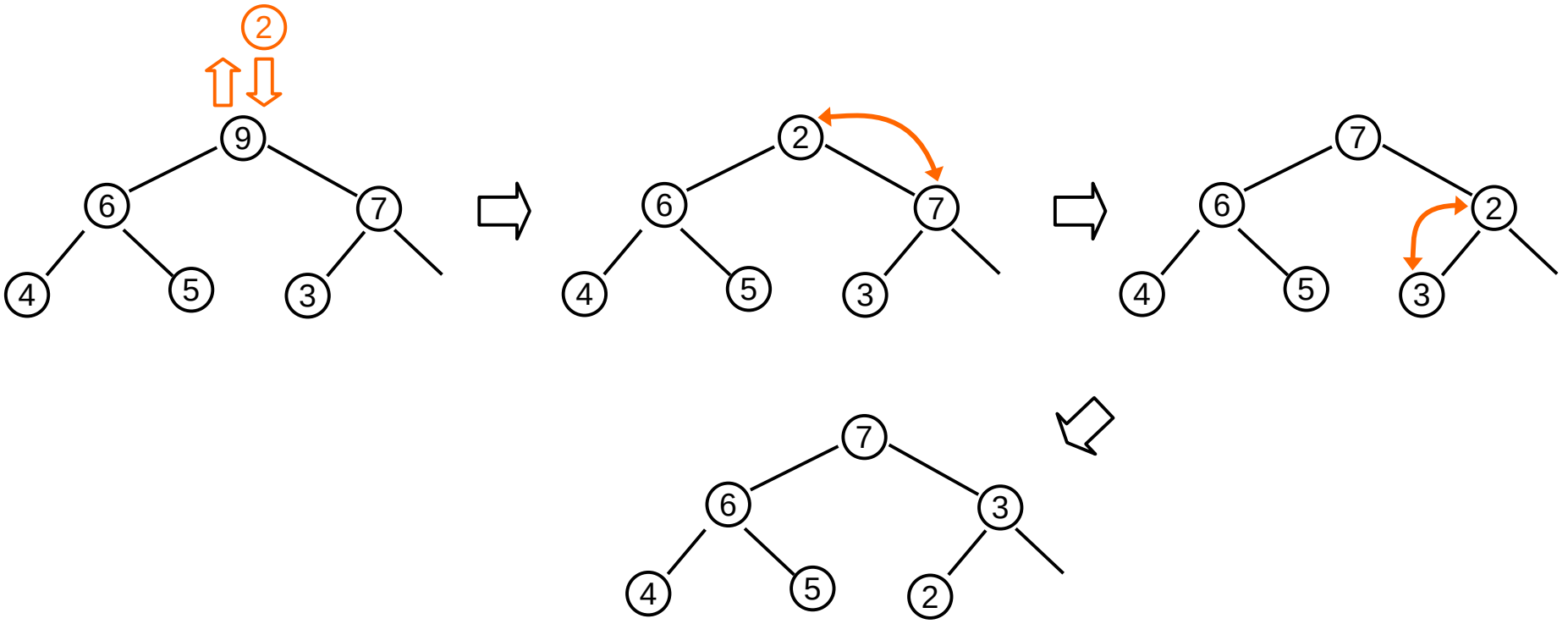
Добавляется элемент всегда на первое незаполненное место слоя. При этом может нарушится свойство бинарной кучи. В таком случае нужно менять значение с родительским узлом до тех пор пока свойство кучи не будет восстановлено. Такой процесс называется **просеиванием вверх**.

Получение максимального элемента с удалением



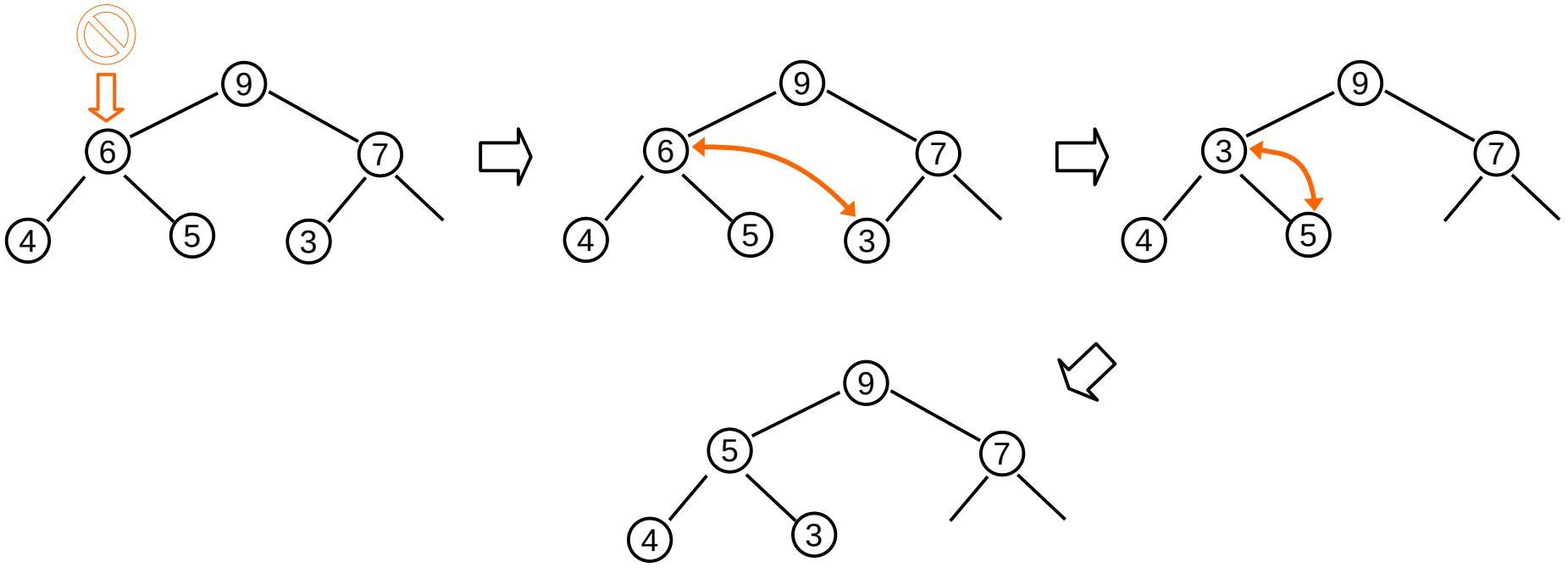
Для удаления максимального элемента, заменяют его последним значением на последнем уровне. Если условие бинарной кучи нарушается, то проводим обмен с максимальным дочерним элементом, до восстановления условий кучи. Такой процесс восстановления свойств кучи называется **просеиванием вниз**.

Одновременная вставка и удаление максимального элемента



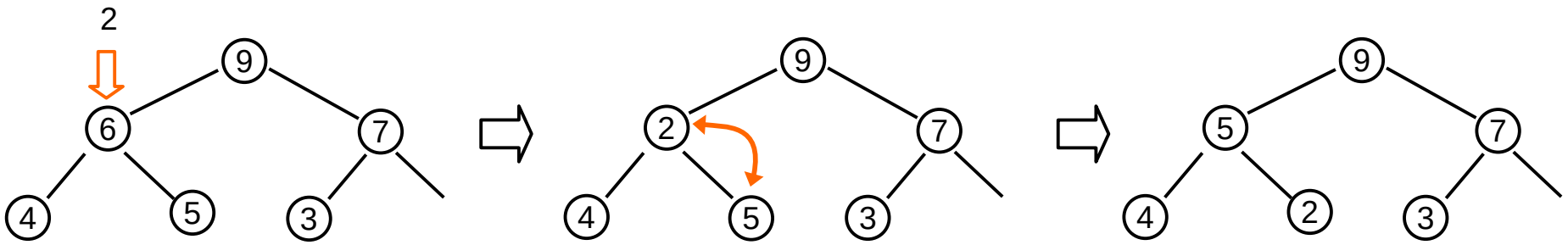
При одновременной вставке и удалении максимального элемента наиболее оптимальным является замена максимального элемента на вставляемый. Если после этого свойства бинарной кучи не соблюдены, то выполняется просеивание вниз.

Удаление произвольного узла



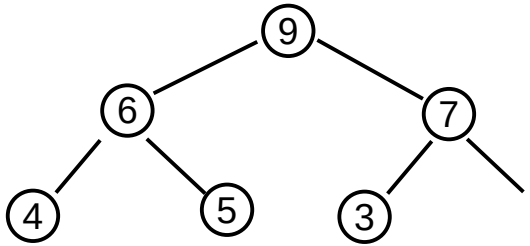
Удаление произвольного узла производится следующим образом — удаляемый узел заменяется на последний узел последнего слоя. Если после замены произошло нарушение свойств бинарной кучи, то применить просеивание вниз.

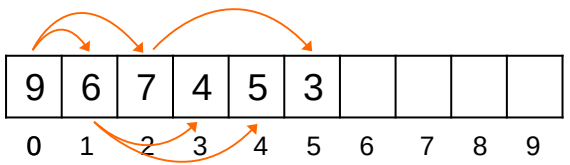
Изменение ключа узла



При замене значения ключа для узла, нужно заменить значение ключа на новое. Если новый ключ больше родительского, то выполнять просеивание вверх. Если новый ключ меньше дочернего узла, то выполнить просеивание вниз.

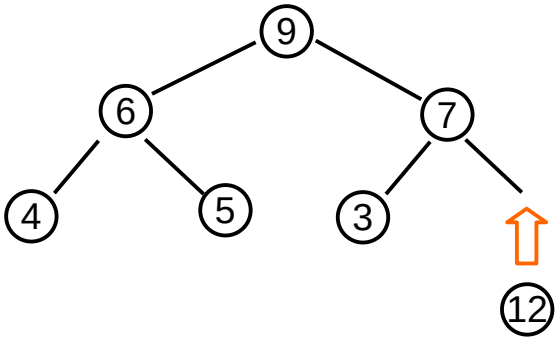
Наиболее простая реализация бинарной кучи

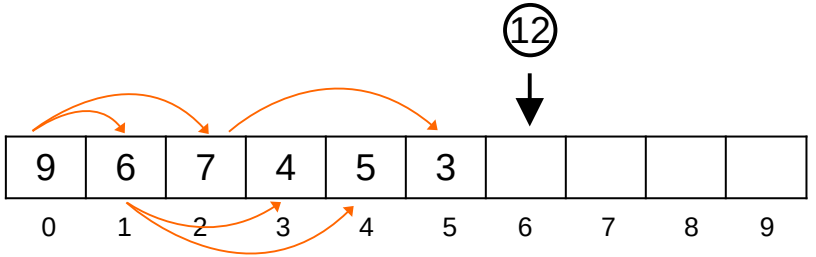


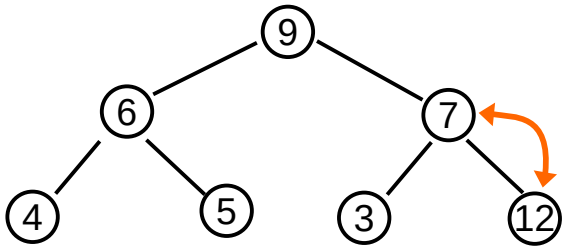


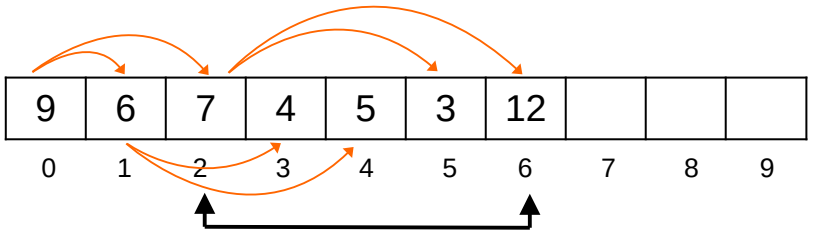
Наиболее простой и популярной реализацией бинарной кучи является реализация на основе массива(списка). Корневой узел всегда располагается на нулевом индексе элементом. Индексы потомков любого элемента рассчитываются как [2i+1] и [2i+2].

Добавление элемента



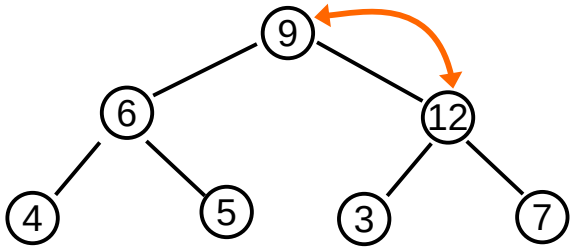


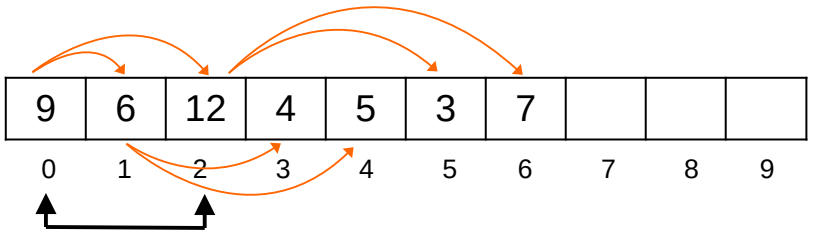


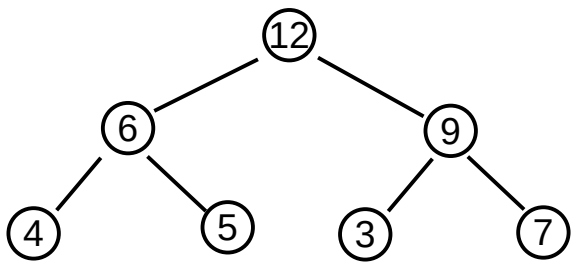


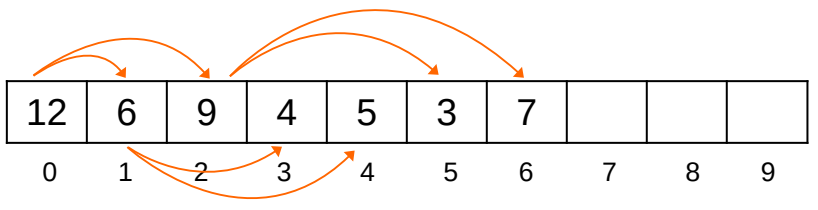
При добавлении элемента стоит хранить указатель на последний добавленный элемент. Этот элемент будет последним элементом последнего слоя. Для вычисления индекса родительского узла использовать выражение [i-1]/2.

Добавление элемента

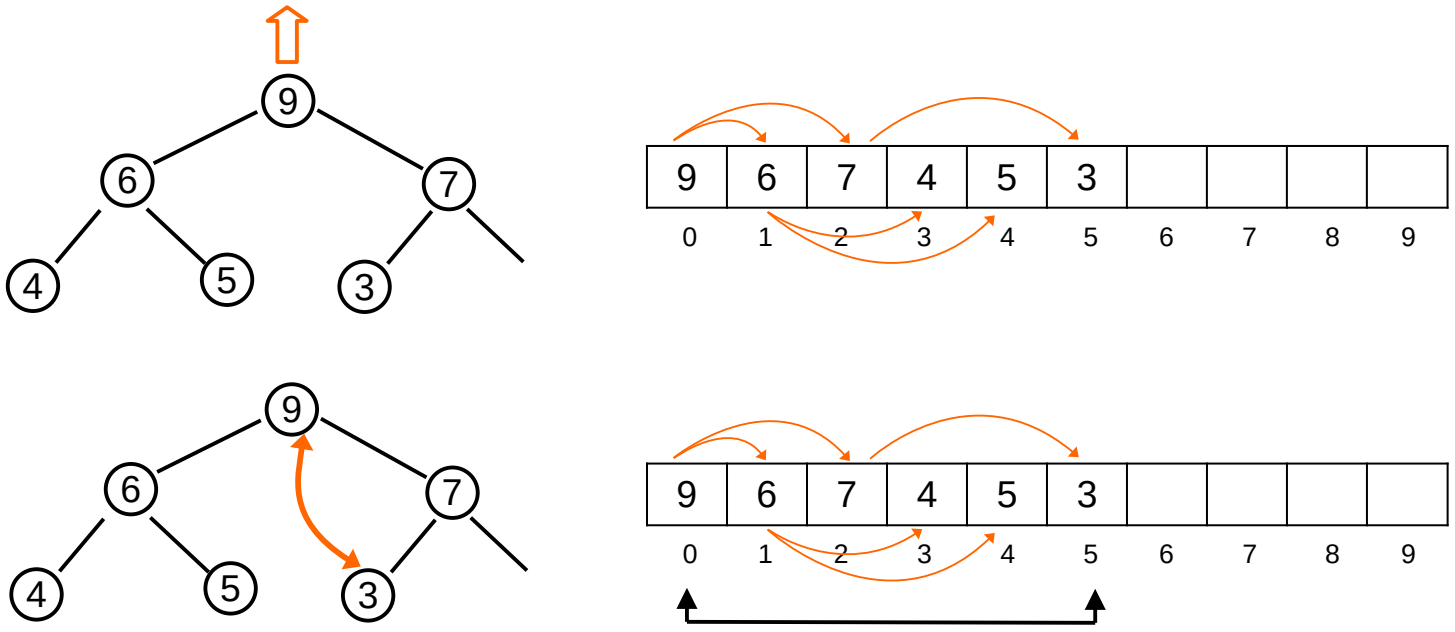






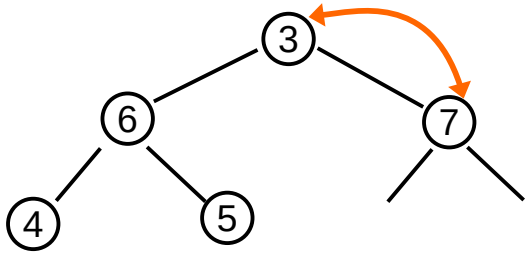


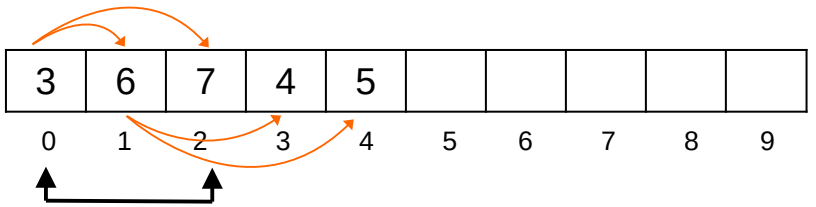
Получение максимального элемента с удалением

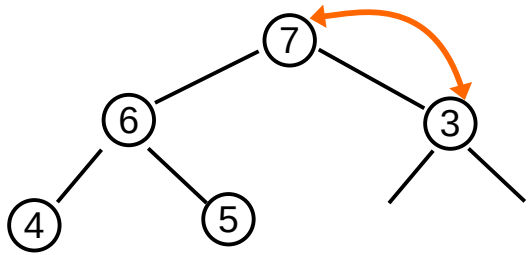


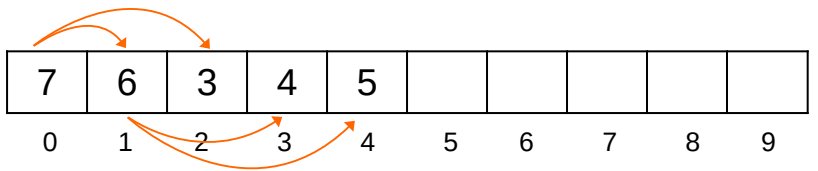
Для удаления максимального элемента, заменяют значение на нулевом индексе значением последнего элемента. После этого выполняют просеивание вниз для этого элемента. Для этого выполняют обмен этого элемента с элементами на индексах [2i+1], [2i+2] (обмен с тем элементом, что больше).

Получение максимального элемента с удалением

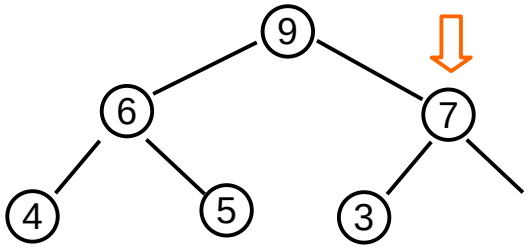


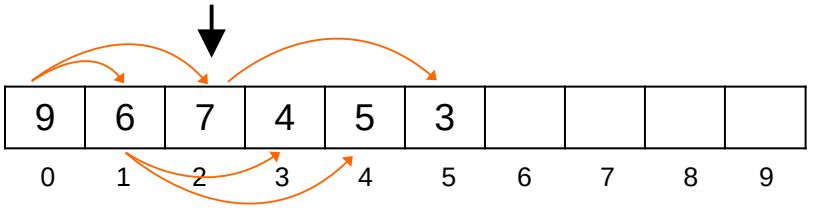






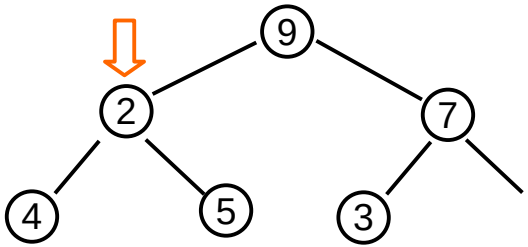
Поиск элемента по ключу

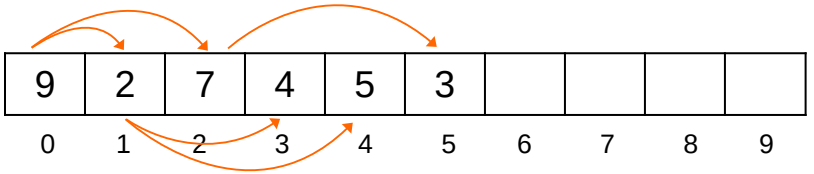


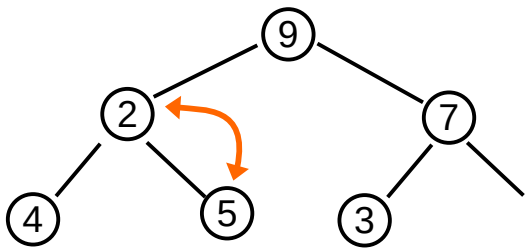


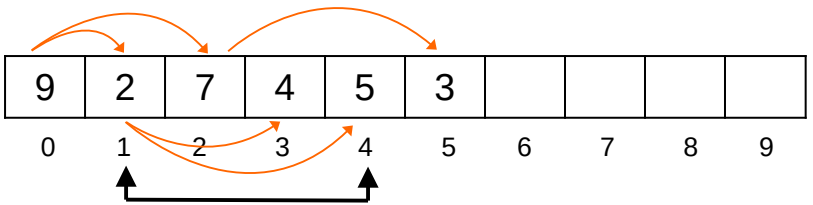
Поиск узла по ключу реализуется с помощью обычного линейного поиска элемента в массиве.

Восстановление свойств кучи





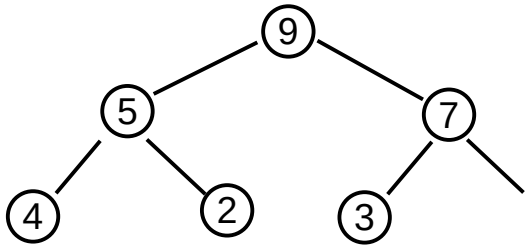


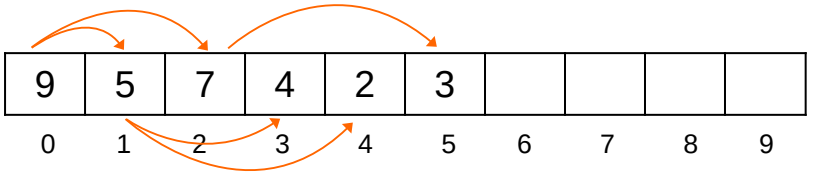


Для восстановления свойств кучи (например узел с ключом 2 стоит не на своем месте) нужно:

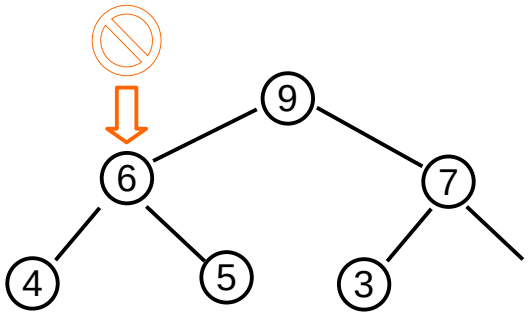
1) Сравнить значение этого узла с родительским. Если значение узла больше родительского, то применяют просеивание вверх (индекс родителя [(i-1)/2]).
2) Сравнить значение этого узла с дочерними. Если значение узла меньше, то применяют просеивание вниз. Обмен с большим из дочерних узлов. Индексы дочерних элементов [2*i + 1], [2*i + 2]

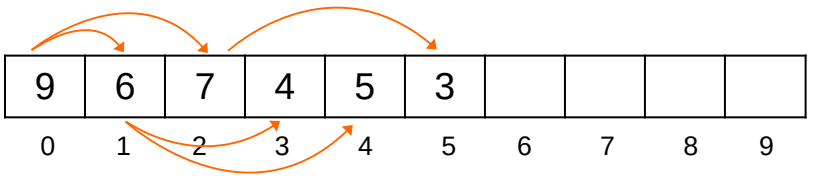
Восстановление свойств кучи

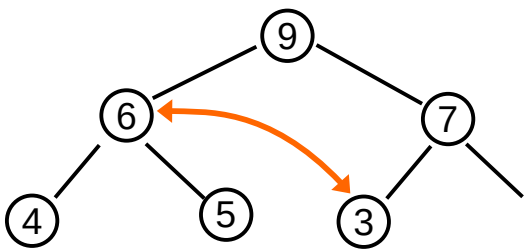


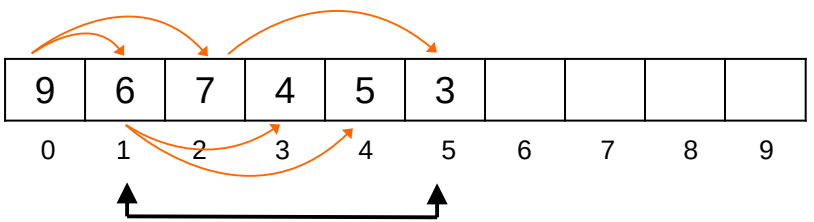


Удаление произвольного узла



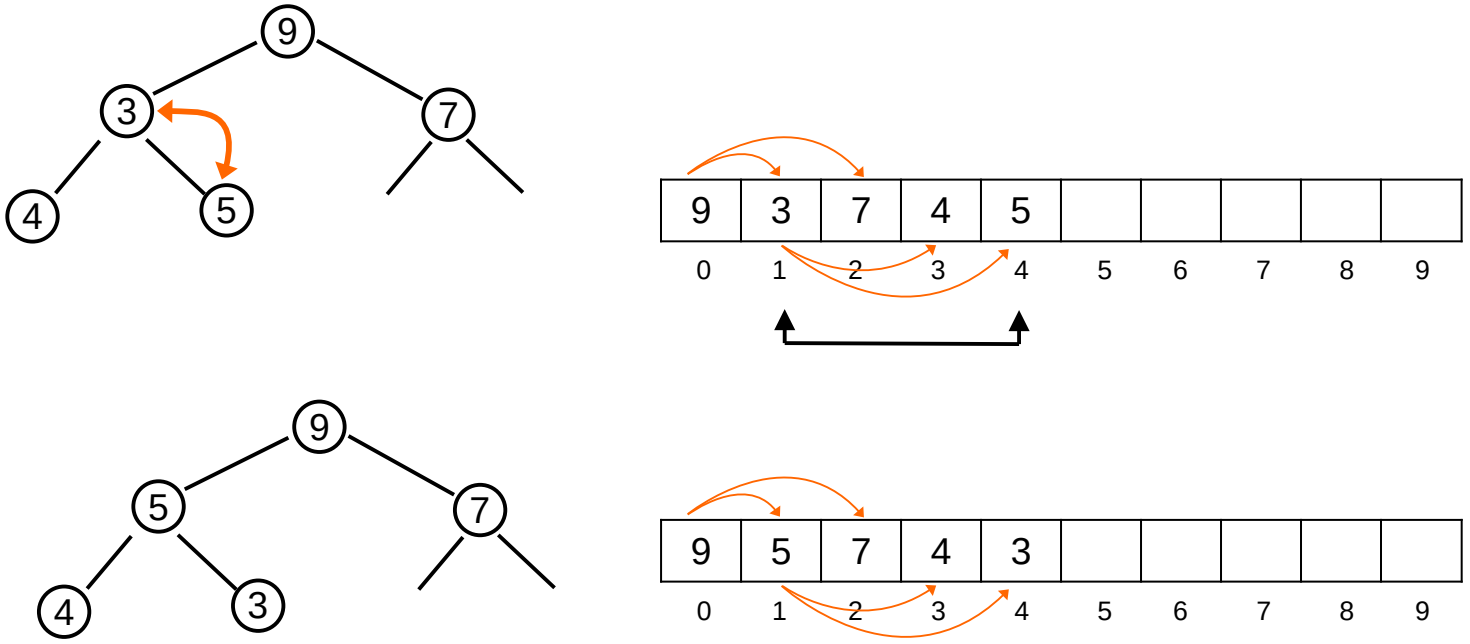






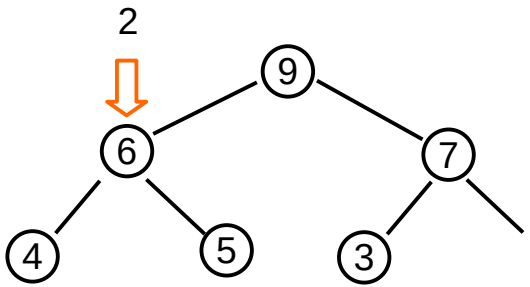
Удаление произвольного узла сводится к замене этого узла на последний. После чего идет восстановление свойств кучи.

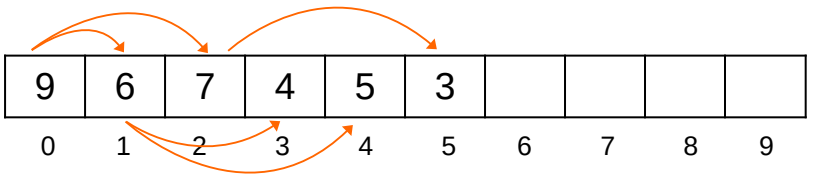
Удаление произвольного узла

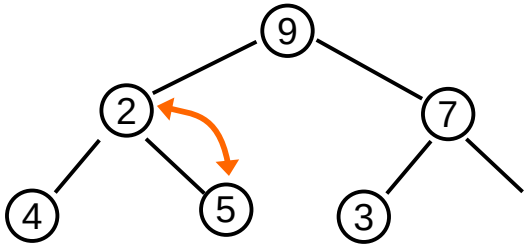


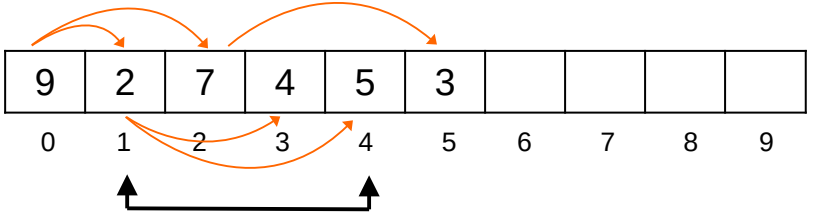
Удаление произвольного узла сводится к замене этого узла на последний. После чего идет восстановление свойств кучи.

Изменение значения ключа



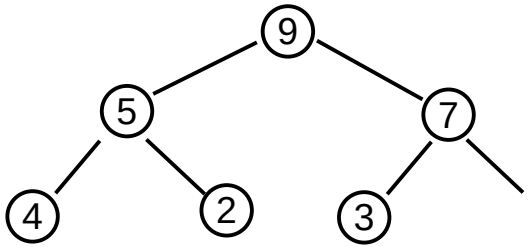


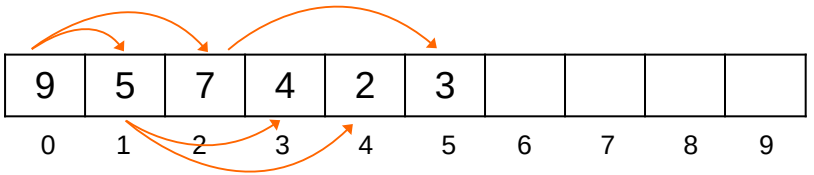




При изменении ключа. Находим старый ключ, заменяем значение на новое. Проводим восстановление свойств кучи при необходимости.

Изменение значения ключа





Области использования бинарной кучи

Бинарная куча является одной из эффективных реализаций очереди с приоритетом.

Очередь с приоритетом (priority queue) - абстрактный тип данных поддерживающий две обязательные операции — добавить элемент и извлечь максимум. Для этого каждому узлу кучи должен соответствовать ключ числового типа (или любого сравнимого между собой типа).

Также бинарная куча используется в алгоритме сортировки — пирамидальная сортировка, сортировка кучей.

Реализация на Python

Python

Описание структуры вершины и бинарной кучи

```
class BinaryHeap:
    class Node:
        def __init__(self, key, data):
            self.key = key
            self.data = data

        def __str__(self):
            return "Node[key = {}, data = {}]".format(self.key, self.data)

    def __init__(self):
        self.nodes = []
```

Методы для просеивания вверх и вниз

```
def sift_up(self, i):
    while i > 0:
        j = (i-1)//2
        if self.nodes[i].key > self.nodes[j].key:
            self.nodes[i], self.nodes[j] = self.nodes[j], self.nodes[i]
        else:
            break
        i = j

def sift_down(self, i):
    n = len(self.nodes)
    while True:
        left_j = 2 * i + 1
        right_j = 2 * i + 2
        largest = i
        if left_j <= n - 1 and self.nodes[left_j].key > self.nodes[largest].key:
            largest = left_j
        if right_j <= n - 1 and self.nodes[right_j].key > self.nodes[largest].key:
            largest = right_j
        if largest != i:
            self.nodes[i], self.nodes[largest] = self.nodes[largest], self.nodes[i]
            i = largest
        else:
            break
```

Python

Метод для восстановления свойств кучи

```
def heap_recovery(self, i):
    n = len(self.nodes)
    if (i-1)//2 >= 0 and self.nodes[i].key > self.nodes[(i-1)//2].key:
        self.sift_up(i)
        return
    self.sift_down(i)
```

Методы добавления, извлечения и извлечения с добавлением

```
def add(self, key, data):
    self.nodes.append(BinaryHeap.Node(key, data))
    i = len(self.nodes) - 1
    self.sift_up(i)

def extract(self):
    if len(self.nodes) == 0:
        return None
    if len(self.nodes) == 1:
        return self.nodes.pop()
    result = self.nodes[0]
    self.nodes[0] = self.nodes.pop()
    self.sift_down(0)
    return result

def insert_and_extract(self, key, data):
    if len(self.nodes == 0):
        self.nodes.append(BinaryHeap.Node(key, data))
        return None
    result = self.nodes[0]
    self.nodes[0] = BinaryHeap.Node(key, data)
    self.sift_down(0)
    return result
```

Методы удаления вершины и изменения ключа вершины

```
def delete_node(self, key):
    i = self.find_key_index(key)
    if i is None:
        return
    if len(self.nodes == 1):
        self.nodes.pop()
        return
    self.nodes[i] = self.nodes.pop()
    self.heap_recovery(i)

def change_key(self, old_key, new_key):
    i = self.find_key_index(old_key)
    if i is None:
        return
    self.nodes[i].key = new_key
    self.heap_recovery(i)
```

Метод поиска индекса узла по ключу

```
def find_key_index(self, key):
    result = None
    for i in range(len(self.nodes)):
        if self.nodes[i].key == key:
            return i
    return result
```

Реализация на Java

Описание представления узла и бинарной кучи

```
class BinaryHeap {
     private class Node {
          int key;
          Object data;

          public Node(int key, Object data) {
               this.key = key;
               this.data = data;
          }

          @Override
          public String toString() {
               return "Node [key=" + key + ", data=" + data + "]";
          }
     }

     private List<Node> nodes = new ArrayList<>();

     public BinaryHeap() {
          super();
     }
```

Методы для просеивания вверх и вниз

```
private void siftUp(int i) {
	for (; i > 0;) {
		int j = (i - 1) / 2;
		if (nodes.get(i).key > nodes.get(j).key) {
			Collections.swap(nodes, i, j);
		} else {
			break;
		}
		i = j;
	}
}

private void siftDown(int i) {
	for (;;) {
		int leftIndex = 2 * i + 1;
		int rightIndex = 2 * i + 2;
		int j = i;
		if (leftIndex <= nodes.size() - 1 && nodes.get(leftIndex).key > nodes.get(j).key) {
			j = leftIndex;
		}
		if (rightIndex <= nodes.size() - 1 && nodes.get(rightIndex).key > nodes.get(j).key) {
			j = rightIndex;
		}
		if (i != j) {
			Collections.swap(nodes, i, j);
			i = j;
		} else {
			break;
		}
	}
}
```

Метод восстановления свойств кучи

```
private void heapRecovery(int i) {
    if (i > 0 && nodes.get(i).key > nodes.get((i - 1) / 2).key) {
        siftUp(i);
        return;
    }
    siftDown(i);
}
```

Метод поиска индекса по ключу узла

```
private int findIndexByKey(int key) {
     for (int i = 0; i < nodes.size(); i++) {
          if (nodes.get(i).key == key) {
               return i;
          }
     }
     return -1;
}
```

Метод для добавления узла

```
public void add(int key, Object data) {
    nodes.add(new Node(key, data));
    siftUp(nodes.size() - 1);
}
```

Методы извлечения из вершины кучи

```
public Object extract() {
     if (nodes.size() == 0) {
          return null;
     }
     if (nodes.size() == 1) {
          return nodes.remove(nodes.size() - 1).data;
     }
     Object result = nodes.get(0).data;
     nodes.set(0, nodes.remove(nodes.size() - 1));
     siftDown(0);
     return result;
}

public Object inserAndExtract(int key, Object data) {
     if (nodes.size() == 0) {
          nodes.add(new Node(key, data));
          return null;
     }
     Object result = null;
     result = nodes.get(0);
     nodes.set(0, new Node(key, data));
     siftDown(0);
     return result;
}
```

Методы удаления узла и изменения ключа узла

```
public void delete(int key) {
     int i = findIndexByKey(key);
     if (i != -1) {
          Node node = nodes.remove(nodes.size() - 1);
          if (nodes.size() == 0) {
               return;
          }
          nodes.set(i, node);
          heapRecovery(i);
     }
}

public void changeKey(int oldKey, int newKey) {
     int i = findIndexByKey(oldKey);
     if (i != -1) {
          nodes.get(i).key = newKey;
          heapRecovery(i);
     }
}
```

Реализация на Fortran

Описание узла и кучи

```
type Node
    integer::key
    character(len = 20)::data_value
end type Node

type BinaryHeap
    type(Node),pointer::nodes(:)
    integer::last_element_index

    contains
    procedure, pass::init
    procedure, pass::destroy
    procedure, pass::resize
    procedure, pass::add
    procedure, pass::sift_up
    procedure, pass::sift_down
    procedure, pass::heap_recovery
    procedure, pass::extract
    procedure, pass::extract_and_insert
    procedure, pass::find_index_by_key
    procedure, pass::delete_by_key
    procedure, pass::change_key
    procedure, pass::show_heap

end type BinaryHeap
```

Процедура инициализации, увеличения размера и очистки памяти

```
subroutine init(this)
    class(BinaryHeap)::this
    allocate(this%nodes(100))
    this%last_element_index = 0
end subroutine init

subroutine destroy(this)
    class(BinaryHeap)::this
    if (associated(this%nodes)) then
        deallocate(this%nodes)
    end if
end subroutine destroy

subroutine resize(this)
    class(BinaryHeap)::this
    type(Node), pointer::new_pointer(:)
    allocate(new_pointer(size(this%nodes) * 2))
    new_pointer(1:this%last_element_index) = this%nodes(1:this%last_element_index)
    deallocate(this%nodes)
    this%nodes => new_pointer
end subroutine resize
```

Процедура добавления узла

```
subroutine add(this, key, data_value)
    class(BinaryHeap)::this
    integer,intent(in)::key
    character(len=*),intent(in)::data_value
    type(Node)::new_node
    if(this%last_element_index == size(this%nodes)) then
        call this%resize()
    end if
    this%last_element_index = this%last_element_index + 1
    new_node%key = key
    new_node%data_value = data_value
    this%nodes(this%last_element_index) = new_node
    call this%sift_up(this%last_element_index)
end subroutine add
```

Процедуры извлечения максимума, и извлечения с добавлением

```
subroutine extract(this, data_value, op_result)
    class(BinaryHeap)::this
    character(len=*),intent(inout)::data_value
    logical, intent(inout)::op_result
    op_result = .true.
    if(this%last_element_index == 0) then
        op_result = .false.
        return
    end if
    data_value = this%nodes(1)%data_value
    this%nodes(1) = this%nodes(this%last_element_index)
    this%last_element_index = this%last_element_index - 1
    call this%sift_down(1)
end subroutine extract

subroutine extract_and_insert (this, data_value_result, key, data_value, op_result)
    class(BinaryHeap)::this
    character(len=*), intent(inout)::data_value_result
    integer,intent(in)::key
    character(len=*),intent(in)::data_value
    logical,intent(inout)::op_result
    op_result = .true.
    if(this%last_element_index == 0) then
        op_result = .false.
    end if
    data_value_result = this%nodes(1)%data_value
    this%nodes(1)%data_value = data_value
    this%nodes(1)%key = key
    call this%sift_down(1)
end subroutine extract_and_insert
```

Процедура поиска индекса узла по ключу

```
subroutine find_index_by_key(this, key, i)
    class(BinaryHeap)::this
    integer,intent(in)::key
    integer,intent(inout)::i
    integer::j
    i = -1
    do j = 1, this%last_element_index
        if(this%nodes(j)%key == key) then
          i = j
          exit
        end if
    end do
end subroutine find_index_by_key
```

Процедуры удаления узла и изменения ключа узла

```
subroutine delete_by_key(this, key)
    class(BinaryHeap)::this
    integer, intent(in)::key
    integer::i
    call this%find_index_by_key(key,i)
    if(i==-1) then
        return
    end if
    this%nodes(i) = this%nodes(this%last_element_index)
    this%last_element_index = this%last_element_index - 1
    call this%heap_recovery(i)
end subroutine delete_by_key

subroutine change_key(this, old_key, new_key)
    class(BinaryHeap)::this
    integer, intent(in)::old_key, new_key
    integer::i
    call this%find_index_by_key(old_key,i)
    if(i==-1) then
        return
    end if
    this%nodes(i)%key = new_key
    call this%heap_recovery(i)
end subroutine change_key
```

Просеивание вверх

```
subroutine sift_up(this,i)
    class(BinaryHeap)::this
    integer,intent(in)::i
    integer::j
    type(Node)::temp_node
    j = i
    do
        if(j < 2) then
            exit
        end if
        if(this%nodes(j)%key > this%nodes(j/2)%key) then
            temp_node = this%nodes(j)
            this%nodes(j) = this%nodes(j/2)
            this%nodes(j/2) = temp_node
            j = j/2
        else
            exit
        end if
    end do
end
```

Просеивание вниз

```
subroutine sift_down(this, i)
    class(BinaryHeap)::this
    integer,intent(in)::i
    integer::j, left_j, right_j, next_j
    type(Node)::temp_node
    j = i
    next_j = i
    do
        left_j = 2 * j
        right_j = 2 * j + 1
        if (left_j <= this%last_element_index) then
            if(this%nodes(left_j)%key > this%nodes(j)%key) then
                next_j = left_j
            end if
        end if

        if (right_j <= this%last_element_index) then
            if(this%nodes(right_j)%key > this%nodes(j)%key) then
                next_j = right_j
            end if
        end if

        if (next_j /= j) then
            temp_node = this%nodes(next_j)
            this%nodes(next_j) = this%nodes(j)
            this%nodes(j) = temp_node
            j = next_j
        else
            exit
        end if
    end do
end subroutine sift_down
```

Процедура нормализация кучи

```
subroutine heap_recovery(this, i)
    class(BinaryHeap)::this
    integer,intent(in)::i
    integer::j, left_j, right_j, next_j
    j = i
    if (j>=2) then
        if(this%nodes(j)%key > this%nodes(j/2)%key) then
            call this%sift_up(j)
            return
        end if
    end if
    call this%sift_down(j)
end subroutine heap_recovery
```

Список литературы

1) Джеймс А. Андерсон «Дискретная математика и комбинаторика». Издательский дом «Вильямс», 2004.
2) Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн // Алгоритмы: построение и анализ 3-е издание. — М.: «Вильямс», 2013. — С. 1328. ISBN 978-5-8459-1794-2
3) Роберт Седжвик, Кевин Уэйн «Алгоритмы на java 4-е издание» Пер. с англ. - М. : ООО "И.Д. Вильямс", 2013. ISBN 978-5-8459-1781-2.