CITY MAX HEAP

- Teilnehmende
  - Hermann Siemens
  - Anouk Martinez
  - Fijola Thora Linn Czyborski
  - Daniel Troschin

Team 01

- Algo – P2 Zeitraum: 04. Dez. 2023 - 18. Dez. 2023
- Betreuung:
  - Prof. Dr. Daniel Gaida
  - Tim Yago Nordhoff

INHALTSVERZEICHNIS

INSERT

- Wenn Heap voll
  - List Eigenschaften nutzen zum Erweitern
  - Alternativ: Fehler ausgeben
- Abfrage nach Rekursiv Eigenschaft (Boolean)
- Last Index inkrementieren

```
def insert(self, city):
    if self.check_if_heap_is_full() == True:
        self.heapStorage.append(0)
        self.maximumHeapCapacity = self.maximumHeapCapacity + 1

    self.heapStorage[self.currentHeapLastIndex] = city

    if self.recursive:
        self.heapify_up_recursive(self.currentHeapLastIndex)
    else:
        self.heapify_up_iterative()

    self.currentHeapLastIndex = self.currentHeapLastIndex + 1
```

ITERATIV UP

- Worst Case
  - Schleife bis Root
- Solange Population > Parent Population, Tausche Nodes
- Nach jedem Durchlauf
  - Node = Parent

```
def heapify_up_iterative(self):
    """
    Establish heap conditions for a Max-Heap iterative upwards.
    """
    current_index = self.currentHeapLastIndex
    parent_index = self.get_parent_index(current_index)

    while current_index > 0:
        node_population = self.heapStorage[current_index].population
        parent_population = self.heapStorage[parent_index].population

        if node_population > parent_population:
            self.swap_nodes(current_index, parent_index)
            current_index = parent_index
            parent_index = self.get_parent_index(current_index)
        else:
            return
```

REKURSIV UP

heapify_up_recursive

```
def heapify_up_recursive(self, index):
    """
    Establish heap conditions for a Max-Heap recursive upwards.
    """
    # Exercise
    ...
    if self.has_parent(index) and index >= 0:
        parent_index = self.get_parent_index(index)
        if self.get_city_population(parent_index) < self.get_city_population(index):
            self.swap_nodes(parent_index, index)
        self.heapify_up_recursive(parent_index)
```

HEAPIFY DOWN

- heapify_down_recursive

```python
def heapify_down_recursive(self, index):
    """
    Establish heap conditions for a Max-Heap recursive downwards.
    """
    if index == 0: # Alternativ if self.has_parent(index) oder so
        root_index = index
        root_node = self.heapStorage[root_index]
        last_node_index = self.currentHeapLastIndex
        last_node = self.heapStorage[last_node_index]

        self.swap_nodes(root_index, last_node_index) # Tausche erste und letzte Node
        self.heapStorage[self.currentHeapLastIndex] = 0 # Lösche letzten Eintrag

    left_child_index = None
    right_child_index = None

    if self.has_left_child(index):
        left_child_index = self.get_left_child_index(index)
    else: return
    if self.has_right_child(index):
        right_child_index = self.get_right_child_index(index)
    else: return

    smaller_child_index = None # Instanziiere smallerchild
    if left_child_index.population < right_child_index.population:
        smaller_child_index = right_child_index
    else:
        smaller_child_index = left_child_index

    if root_node.population > smaller_child_index:
        return
    else:
        self.swap_nodes(index, smaller_child_index)
        self.heapify_down_recursive(smaller_child_index)
```

- heapify_down_iterative

```python
def heapify_down_iterative(self):
    """
    Establish heap conditions for a Max-Heap iterative downwards.
    """
    root_node = self.heapStorage[0]
    last_node = self.heapStorage[self.currentHeapLastIndex]

    # Tausche erste und letzte Node
    self.swap_nodes(root_node, last_node)
    self.heapStorage[self.currentHeapLastIndex] = 0

    root_node = last_node # Setze Referenz von root Node auf new root

    while self.has_left_child(root_node) or self.has_right_child(root_node):
        if self.has_left_child(root_node):
            left_child = self.get_left_child_index(root_node)
        if self.has_right_child(root_node):
            right_child = self.get_right_child_index(root_node)

        smaller_child = 0 # Instanziiere smallerchild

        if left_child.population < right_child.population:
            smaller_child = right_child
        else:
            smaller_child = left_child

        if root_node.population > smaller_child:
            return # Heap Invariante ist wieder hergestellt

        root_node = smaller_child
```

FLOYD'S HEAP BUILDING ALGORITHMUS

Floyd

- Gleiches wie
  - Heapify_down_iter
- Ohne check leere Blätter

```
def heapify_floyd(self, index, amount_of_cities):
    """
    Establish heap conditions for a Max-Heap via Floyds Heap Construction Algorithmus.

    """
    # Exercise

    self.heapify_down_iterative()
    '''
    last_leaf_index = index

    while last_leaf_index >= 0 and not (
            self.has_left_child(last_leaf_index) or self.has_right_child(last_leaf_index)):
        last_leaf_index -= 1

    # Start building the heap from the last non-leaf node
    for idx in range(last_leaf_index, -1, -1):
        self.heapify_down_recursive(idx)
```

REMOVE

Remove Stadt
größte Einwohner

- Heap erfüllt
- Swap
- Remove
- Correct Last
- Rebuild heap
  - Rekursive
  - Iterative
- Return removed city

```
def remove(self):
    """
    Remove a City from the Max-Heap
    """
    # Exercise

    my_index = self.currentHeapLastIndex - 1
    self.swap_nodes( fst_node_index: 0, my_index)
    removed_city = self.heapStorage.pop()
    self.currentHeapLastIndex = self.currentHeapLastIndex - 1
    if self.recursive or self.floyd:
        self.heapify_down_recursive(0)
    else:
        self.heapify_down_iterative()
    return removed_city
```

SONSTIGES

HILFSFUNKTIONEN GET INDEX

Get Parent Index

```python
def get_parent_index(self, index):
    """
    Return the index of the parent node.
    """
    if index == 0:
        return None
    else:
        parent_index = (index - 1) / 2
        return math.floor(parent_index)
```

$$\text{parent}(i) = \lfloor \frac{i-1}{2} \rfloor$$

Get Left Child Index

```python
def get_left_child_index(self, index):
    """
    Return the index of the left child.
    """
    if self.has_left_child(index):
        return 2 * index + 1
    return None
```

$$leftChild(i) = 2i + 1$$

Get Right Child Index

```python
def get_right_child_index(self, index):
    """
    Return the index of the right child.
    """
    if self.has_right_child(index):
        return 2 * index + 2
    return None
```

$$rightChild(i) = 2i + 2$$

HILFSFUNKTIONEN HAS

Has Parent

$$parent(i) = \lfloor \frac{i-1}{2} \rfloor$$

```
def has_parent(self, index) -> bool:
    """
    Check if the node has a parent. Return:
        True    = Has parent
        False   = No parent
    """
    return index != 0
```

Has Left Child

$$leftChild(i) = 2i + 1$$

```
def has_left_child(self, index):
    """
    Check if the Node has a left Child. Return:
        True    = Has leftChild
        False   = No leftChild
    """
    left_child_index = 2 * index + 1
    if left_child_index > self.maximumHeapCapacity:
        return False
    elif self.heapStorage[left_child_index] == 0:
        return False

    return True
```

Has Right Child

$$rightChild(i) = 2i + 2$$

```
def has_right_child(self, index):
    """
    Check if the Node has a right Child. Return:
        True    = Has rightChild
        False   = No rightChild
    """
    right_child_index = 2 * index + 1
    if right_child_index > self.maximumHeapCapacity:
        return False
    elif self.heapStorage[right_child_index] == 0:
        return False

    return True
```

HILFSFUNKTIONEN GET POPULATION

Get City Population

```
def get_city_population(self, index):
    """
    Return the Population of a City with the given index
    """
    if(self.heapStorage[index] == 0 or index == None):
        return None
    return self.heapStorage[index].population
```

Get Right Child Population

```
def get_right_child_population(self, index):
    """
    Return of the population of the right child.
    """
    right_child_index = self.get_right_child_index(index)
    self.get_city_population(right_child_index)
```

Get Parent Population

```
def get_parent_population(self, index):
    """
    Returns the population of the parent.
    """
    parent_index = self.get_parent_index(index)
    self.get_city_population(parent_index)
```

Get Left Child Population

```
def get_left_child_population(self, index):
    """
    Return of the population of the left child.
    """
    left_child_index = self.get_left_child_index(index)
    self.get_city_population(left_child_index)
```

SONSTIGE HILFSFUNKTIONEN

Check If Heap Full

```
def check_if_heap_is_full(self):
    """
    Check if the heap has reached its maximum capacity. Return:
        True    = Full
        False   = Not full
    """
    return self.currentHeapLastIndex >= self.maximumHeapCapacity
```

Swap Nodes

```
def swap_nodes(self, fst_node_index, sec_node_index):
    """
    Swap two nodes specified by their index.
    """
    first = self.heapStorage[fst_node_index]
    second = None
    if len(self.heapStorage) > sec_node_index:
        second = self.heapStorage[sec_node_index]

    if second is not None:
        self.heapStorage[sec_node_index] = first
        self.heapStorage[fst_node_index] = second
```

LIVE CODE UND FRAGEN