CS261 Data Structures

Assignment 5

Spring 2024

MinHeap Implementation

Contents

General Instructions

1. Programs in this assignment must be written in Python 3 and submitted to Gradescope before the due date specified on Canvas and in the Course Schedule. You may resubmit your code as many times as necessary before the due date (this is encouraged). Gradescope allows you to choose which submission will be graded. **We will grade the currently activated submission.**

2. In Gradescope, your code will run through several tests. Any failed tests will provide a brief explanation of testing conditions to help you with troubleshooting. You earn points as you pass tests. There are no hidden Gradescope tests and you will know your maximum possible score with each submission.

3. We encourage you to create your own test cases (such as with Python's **unittest** unit testing framework, which is more powerful and efficient than "print statement debugging"), even though this work doesn't have to be submitted and won't be graded. Gradescope tests are limited in scope and may not cover all edge cases. We reserve the right to evaluate your submission with additional methods beyond Gradescope.

4. Unless indicated otherwise, we will test your implementation with different types of objects, not just integers. We guarantee that all such objects will have the correct implementation of:

    a. "rich comparison" methods:

        i. `__eq__()`

        ii. `__lt__()`

        iii. `__gt__()`

        iv. `__ge__()`

        v. `__le__()`

    b. `__str__()`

5. **Your code must have an appropriate level of comments.** At minimum, each method must have a descriptive docstring. Additionally, write comments throughout your code to make it easy to follow and understand any non-obvious code. However, be mindful of the amount of comments in your code. Cluttering comments negatively impact readability and are discouraged. If your code includes an excessive amount of comments, consider refactoring. Please refer to "Styling Your Code" from the "Coding Guides and Tips - Style and Debugging" module for specifics.

6. You will be provided with a starter "skeleton" code, on which you will build your implementation. Methods defined in the skeleton code must retain their names and input/output parameters. Variables defined in the skeleton code must also retain their names. We will only test your solution by making calls to methods defined in

the skeleton code, and by checking values of variables defined in the skeleton code. You can add more helper methods and variables, as needed.**The MinHeap skeleton code file includes a suggested helper method.**

You are allowed to:

- add more helper methods and variables, as needed
- add optional default parameters to method definitions
- modify or add to the basic testing section within the scope of:

```
if __name__ == "__main__":
```

However, certain classes and methods cannot be changed in any way. Please see the comments in the skeleton code for guidance. The content of any methods pre-written for you as part of the skeleton code must not be changed.

All points will be deducted from build_heap() for the incorrect time complexity. 25% of points will be deducted from all other methods with the incorrect time complexity.

7. The skeleton code and code examples provided in this document are part of the assignment requirements. They have been carefully selected to demonstrate requirements for each method. Refer to them for detailed descriptions of expected method behavior, input/output parameters, and handling of edge cases. Code examples may include assignment requirements not explicitly stated elsewhere.

8. **For each method, you are required to use an iterative solution.** Recursion is not permitted.

9. You may not use any imports beyond the ones included in the assignment source code.

Summary and Specific Instructions

1. Implement the MinHeap class by completing the provided skeleton code in the file `min_heap.py`. Once completed, your implementation will include the following methods:

```
add()
is_empty()
get_min()
remove_min()
build_heap()
size()
clear()
heapsort()
```

2. The MinHeap must be implemented with a DynamicArray as per the skeleton code. You are to use your existing DynamicArray for the implementation.

3. You may wish to augment your existing DynamicArray to assist you in this assignment. For instance, a method similar to Python's `pop()` function for lists, that removes the last item in your DynamicArray, may be helpful. Alternatively, you may implement this functionality inline in your heap implementation, if you prefer.

4. The number of objects stored in the MinHeap will be between 0 and 1,000,000 inclusive.

5. **RESTRICTIONS:** You are NOT allowed to use ANY built-in Python data structures and/or their methods. Your solutions should not call double underscore ("dunder") methods.

   You are NOT allowed to directly access any variables of the DynamicArray class. All work must be done only by using class methods.

6. You do not need to write any getter or setter methods for the MinHeap class.

7. **Ensure that your methods meet the specified runtime requirements.**

add(self, node: object) -> None:

This method adds a new object to the MinHeap while maintaining heap property.

The runtime complexity of this implementation must be **amortized O(log N)**.

Example #1:
```
h = MinHeap()
print(h, h.is_empty())
for value in range(300, 200, -15):
    h.add(value)
    print(h)
```

Output:
```
HEAP [] True
HEAP [300]
HEAP [285, 300]
HEAP [270, 300, 285]
HEAP [255, 270, 285, 300]
HEAP [240, 255, 285, 300, 270]
HEAP [225, 255, 240, 300, 270, 285]
HEAP [210, 255, 225, 300, 270, 285, 240]
```

Example #2:
```
h = MinHeap(['fish', 'bird'])
print(h)
for value in ['monkey', 'zebra', 'elephant', 'horse', 'bear']:
    h.add(value)
    print(h)
```

Output:
```
HEAP ['bird', 'fish']
HEAP ['bird', 'fish', 'monkey']
HEAP ['bird', 'fish', 'monkey', 'zebra']
HEAP ['bird', 'elephant', 'monkey', 'zebra', 'fish']
HEAP ['bird', 'elephant', 'horse', 'zebra', 'fish', 'monkey']
HEAP ['bear', 'elephant', 'bird', 'zebra', 'fish', 'monkey', 'horse']
```

is_empty(self) -> bool:

This method returns True if the heap is empty; otherwise, it returns False.

The runtime complexity of this implementation must be **O(1)**.

Example #1:

```
h = MinHeap([2, 4, 12, 56, 8, 34, 67])
print(h.is_empty())
```

Output:

```
False
```

Example #2:

```
h = MinHeap()
print(h.is_empty())
```

Output:

```
True
```

get_min(self) -> object:

This method returns an object with the minimum key, without removing it from the heap. If the heap is empty, the method raises a MinHeapException.

The runtime complexity of this implementation must be **O(1)**.

Example #1:
```
h = MinHeap(['fish', 'bird'])
print(h)
print(h.get_min(), h.get_min())
```

Output:
```
HEAP ['bird', 'fish']
bird bird
```

remove_min(self) -> object:

This method returns an object with the minimum key, and removes it from the heap. If the heap is empty, the method raises a MinHeapException.

For the downward percolation of the replacement node: if both children of the node have the same value (and are both smaller than the node), swap with the left child.

The runtime complexity of this implementation must be **amortized O(log N)**.

Example #1:
```
h = MinHeap([1, 10, 2, 9, 3, 8, 4, 7, 5, 6])
while not h.is_empty() and h.is_empty() is not None:
    print(h, end=' ')
    print(h.remove_min())
```

Output:
```
HEAP [1, 3, 2, 5, 6, 8, 4, 10, 7, 9] 1
HEAP [2, 3, 4, 5, 6, 8, 9, 10, 7] 2
HEAP [3, 5, 4, 7, 6, 8, 9, 10] 3
HEAP [4, 5, 8, 7, 6, 10, 9] 4
HEAP [5, 6, 8, 7, 9, 10] 5
HEAP [6, 7, 8, 10, 9] 6
HEAP [7, 9, 8, 10] 7
HEAP [8, 9, 10] 8
HEAP [9, 10] 9
HEAP [10] 10
```

build_heap(self, da: DynamicArray) -> None:

This method receives a DynamicArray with objects in any order, and builds a proper MinHeap from them. The current content of the MinHeap is overwritten.

The runtime complexity of this implementation must be **amortized O(N)**. *If the runtime complexity is amortized O(N log N), you will not receive any points for this portion of the assignment, even if your method passes Gradescope.*

Example #1:

```
da = DynamicArray([100, 20, 6, 200, 90, 150, 300])
h = MinHeap(['zebra', 'apple'])
print(h)
h.build_heap(da)
print(h)

print("Inserting 500 into input DA:")
da[0] = 500
print(da)

print("Your MinHeap:")
print(h)
if h.get_min() == 500:
    print("Error: input array and heap's underlying DA reference same object
    in memory")
```

Output:

```
HEAP ['apple', 'zebra']
HEAP [6, 20, 100, 200, 90, 150, 300]

Inserting 500 into input DA:
DYN_ARR Size/Cap: 7/8 [500, 20, 6, 200, 90, 150, 300]
Your MinHeap:
HEAP [6, 20, 100, 200, 90, 150, 300]
```

size(self) -> int:

This method returns the number of items currently stored in the heap.

The runtime complexity of this implementation must be **O(1)**.

Example #1:
```
h = MinHeap([100, 20, 6, 200, 90, 150, 300])
print(h.size())
```

Output:
```
7
```

Example #2:
```
h = MinHeap([])
print(h.size())
```

Output:
```
0
```

clear(self) -> None:

This method clears the contents of the heap.

The runtime complexity of this implementation must be **O(1)**.

Example #1:
```
h = MinHeap(['monkey', 'zebra', 'elephant', 'horse', 'bear'])
print(h)
print(h.clear())
print(h)
```

Output:
```
HEAP ['bear', 'elephant', 'monkey', 'zebra', 'horse']
None
HEAP []
```

heapsort(arr: DynamicArray) -> None:

Write a function that receives a DynamicArray and sorts its content in non-ascending order, using the Heapsort algorithm. You must sort the array in place, **without** creating any data structures. This function does not return anything.

You may assume that the input array will contain one or more homogeneous elements (either all numbers, or strings, or custom objects, but never a mix of these). You do not need to write checks for these conditions.

The runtime complexity of this implementation must be **O(N log N)**. *If the sort uses an algorithm other than Heapsort, you will not receive any points for this portion of the assignment, even if your function passes Gradescope.*

Example #1:

```
da = DynamicArray([100, 20, 6, 200, 90, 150, 300])
print(f"Before: {da}")
heapsort(da)
print(f"After:  {da}")
```

Output:

```
Before: DYN_ARR Size/Cap: 7/8 [100, 20, 6, 200, 90, 150, 300]
After:  DYN_ARR Size/Cap: 7/8 [300, 200, 150, 100, 90, 20, 6]
```

Example #2:

```
da = DynamicArray(['monkey', 'zebra', 'elephant', 'horse', 'bear'])
print(f"Before: {da}")
heapsort(da)
print(f"After:  {da}")
```

Output:

```
Before: DYN_ARR Size/Cap: 5/8 [monkey, zebra, elephant, horse, bear]
After:  DYN_ARR Size/Cap: 5/8 [zebra, monkey, horse, elephant, bear]
```