Harvard John A. Paulson School of Engineering and Applied Sciences

F. Wermelinger
Office: Pierce 211

Homework 6

BST node deletion, traversal, heaps and priority queues

Issued:	November 8th, 2022
Due:	November 29th, 2022

Note this is a 3 week exercise. *Do not procrastinate the work.*

Problem 1: Homework Submission Requirements (*10 points*)

Requirements:

1. Complete the homework on the designated branch (***5 points***)
2. Create a pull request to merge the designated homework branch into your default branch, e. g., `main` or `master` (***2 points***)
3. Merge the open pull request of the *previous homework* into your default branch **after** you have received feedback from the teaching staff and regrade requests are resolved (***3 points***).

Deliverables: your solutions to the problems below must be submitted in a directory called `submission`.

See https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-hw.

- ***5 points*** if homework has been solved on the required branch.
- ***2 points*** if a pull request has been opened to merge this homework branch into the default branch. No points if the pull request has been merged prematurely (before receiving feedback and regrade requests have been resolved).
- ***3 points*** if the previous branch of the previous homework has been merged *after* homework feedback has been received and all regrade request are resolved.

Problem 2: Binary Search Tree Extensions (*30 points*)

Deliverables:

1. `P2.py`

Important: this problem will be fully *auto-graded*. You are expected to work with the skeleton code provided in `code/p2/P2.py`. Do not change existing code in that skeleton code. Only add your solution code in appropriate places or update values in existing variables. Otherwise you will risk that the auto-grader fails and you lose points. Read carefully the problem statement and ensure your implementation follows the specifications given.
The auto-grader works by running a set of tests on your implementation. It is therefore important that you are throwing exceptions at places in your code where values do not correspond to what is expected.

In this problem you extend the binary search tree (BST) implementation of last homework by adding support for node deletion and finally implement an iterator class to traverse the BST in different ways. The skeleton code in `code/p2/P2.py` contains a partially implemented BST which is the starting point for this problem.

Hint: *Assume that keys are unique (one-to-one mapping) and of the same type.*

a) ***10 points***

In this task you are implementing a private helper method `_remove_min(self, node)` to remove the *minimum node* (the node with the *smallest* key) in a subtree with root node given by the method argument `node`. The method must return a reference to the root node of the altered subtree with the minimum node removed as well as a reference to the removed minimum node that is no longer in the subtree. The method should raise an `AssertionError` if the input `node` is of `None` type. An example application could look like the following:

```
>>> tree = BinarySearchTree()
>>> tree.put(key=0, val='A')
>>> tree.put(key=-1, val='minimum')
>>> tree.put(key=2, val='B')
>>> tree.put(key=5, val='maximum')
>>> print(tree)
Node(key=0, val=A)
left  -> Node(key=-1, val=minimum)
         left  -> None
         right -> None
right -> Node(key=2, val=B)
         left  -> None
         right -> Node(key=5, val=maximum)
                  left  -> None
                  right -> None
```

The minimum node in this tree is given by the key `-1`. To remove this node the helper method you implement in this task can be used like this

```
>>> tree._root, min_node = tree._remove_min(tree._root)
>>> print(tree)
Node(key=0, val=A)
left  -> None
right -> Node(key=2, val=B)
         left  -> None
         right -> Node(key=5, val=maximum)
                  left  -> None
                  right -> None
>>> print("Removed node:", min_node)
Removed node: Node(key=-1, val='minimum')
```

Hint: *Remember to update the size of the subtrees as well.*

Please see `solution/code/p2/P2.py` for the solution code.

b) ***10 points***

In this task you are implementing node removal from the binary tree for a given key. The public `remove` interface is already implemented in the skeleton code `code/p2/P2.py`. It delegates the call to the private `_remove` method which you are asked to implement in this task. The `_remove(self, node, key)` method returns the root node of the new subtree after the node with matching `key` has been removed. The argument `node` is a reference to the root node of the input tree. If no node with `key` exists in the tree, the method must raise a `KeyError` exception. An example application could look like the following:

```
>>> tree = BinarySearchTree()
>>> tree.put(key=0, val='A')
>>> tree.put(key=-1, val='minimum')
>>> tree.put(key=2, val='B')
>>> tree.put(key=5, val='maximum')
>>> tree.remove(0)
>>> print(tree)
Node(key=2, val=B)
left  -> Node(key=-1, val=minimum)
         left  -> None
         right -> None
right -> Node(key=5, val=maximum)
         left  -> None
         right -> None
```

Hint: *Remember to update the size of the subtrees as well.*

Hint: *You should be able to remove nodes with degrees 0, 1 and 2 from the binary tree. For nodes with degree 2, the* `_remove_min` *helper from the previous task may be helpful.*

Hint: *Feel free to implement additional helper functions where you see fit.*

Please see `solution/code/p2/P2.py` for the solution code.

c) ***10 points***

In this task you are going to implement an interface to obtain iterators for tree traversal. The public `get_iterator` method is already implemented in the skeleton code `code/p2/P2.py`. It delegates the call to the private `_preorder`, `_inorder` or `_postorder` methods that you should implement, following the corresponding traversal protocol. These three private methods must be *generator functions* that yield references to tree nodes during iteration. They take an argument `node` which is the root node of the subtree to be traversed. Note that you will need to call these generator functions *recursively*. Because a call to a generator function returns a generator object from which you need to yield during the recursion, one approach to deal with this could be to loop over the generator object returned by the recursive call and yield from it in the loop body. This is not efficient and is one of the reasons in Python 3.3 the "`yield from`" statement has been introduced. One of the motivations for its introduction was the tree traversal you are going to implement in this task. The `yield from` statement is very useful when working with recursive generator functions, where the recursion produces a *chain of generator objects*. The `yield from` statement is used to conveniently yield from such a chain of generator objects. Please have a look at the "Motivation" and "Proposal" sections in `https://peps.python.org/pep-0380/` for more information.

An example application for a preorder tree traversal could look like the following:

```
>>> tree = BinarySearchTree()
>>> for k, v in zip(list([0, -2, -3, -1, 2, 1, 3]), list('ABCDEFG')):
...     tree.put(key=k, val=v)
>>> for node in tree.get_iterator(DFSOrder.PREORDER):
...     print(node)
Node(key=0, val='A')
Node(key=-2, val='B')
Node(key=-3, val='C')
Node(key=-1, val='D')
Node(key=2, val='E')
Node(key=1, val='F')
Node(key=3, val='G')
```

Hint: *The* `DFSOrder` *type is an enumeration for traversal order and is defined at the beginning of the skeleton code. It is similar to the enumeration you have used for the bank account type in homework 3.*

Please see `solution/code/p2/P2.py` for the solution code.

Problem 3: Heaps (*30 points*)

Deliverables:

1. `P3.py`

Important: this problem will be fully *auto-graded*. You are expected to work with the skeleton code provided in `code/p3/P3.py`. Do not change existing code in that skeleton code. Only add your solution code in appropriate places or update values in existing variables. Otherwise you will risk that the auto-grader fails and you lose points. Read carefully the problem statement and ensure your implementation follows the specifications given.
The auto-grader works by running a set of tests on your implementation. It is therefore important that you are throwing exceptions at places in your code where values do not correspond to what is expected.

This problem is about the heap data structure or simply "heap" in short. A heap represents a *balanced binary tree* structure where the heap elements h_i are numbers that are stored compactly in an *array* for efficiency. That is, for heaps we are not working with node types that we chain together which then form the tree. In a heap, the "nodes" are the array elements h_i which maintain a certain order, called the *heap property*, given by

$$\begin{aligned} h_i &\leq h_{2i+1}, \\ h_i &\leq h_{2i+2}, \end{aligned} \tag{1}$$

where the index pattern used here assumes the first element in the heap, h_0, starts at index $i = 0$. The heap property in equation 1 results in a *min-heap* where the least (smallest) element in the heap is h_0. Alternatively, a *max-heap* results if the relation "$\leq$" is changed to "$\geq$" in equation 1. In this case the element h_0 in the heap is the greatest element.

Hint: *Have a look at the heap visualization provided at the following link for an intuition of how a heap works:* `https://www.cs.usfca.edu/~galles/visualization/Heap.html`

Hint: *See* `https://en.wikipedia.org/wiki/Binary_heap` *for some more implementation ideas.*

Hint: *Feel free to add additional helper methods in your implementation if you see fit.*

a) ***15 points***

In this task you are going to implement the `_siftup` and `_siftdown` private methods required to maintain the heap property in a *min-heap* data structure, as shown in equation 1. The `_siftup(self, child)` method takes an argument `child` which is an index in the array representing a child node that needs to sift-up the binary tree such that the heap property is satisfied. Similarly, the `_siftdown(self, parent)` method takes an argument `parent` which is an index in the array representing a parent node that needs to sift-down the tree such that the heap property is satisfied. Example implementations have been discussed in the lecture for a 1-index based implementation. *You must implement the heap based on a 0-indexed array*, that is, the first heap element h_0 starts at index 0 in the `_heap` array, see equation 1. Note that you can implement these methods either in a loop where you compute array indices in the loop body or by recursive

function calls. The loop option is more efficient because it avoids additional overhead due to recursive function calls. You will need a comparison operator for these methods that you should implement in the `_compare(self, a, b)` method such that the heap property corresponds to a *min-heap*. The parameters `a` and `b` are *actual comparable values* and not array indices. The method should return a boolean. Please start with the skeleton code given in `code/p3/P3.py`. An example application for a 0-indexed heap could look like this:

```
>>> h = MinHeap([0])  # initialize some dummy heap
>>> h._heap = [3, 2, 1]  # set an explicit state of the heap
>>> h._siftup(2)  # sift-up last element in array
>>> h._heap
[1, 2, 3]
>>> h._heap = [3, 2, 1]  # reset the starting state
>>> h._siftdown(0)  # sift-down the first element in the array
>>> h._heap
[1, 2, 3]
```

Hint: *The heap property* does not *imply a specific order among the children for a given parent node. It is therefore possible that your output in the examples above may look different, but the first element in the output* must be 1.

Please see `solution/code/p3/P3.py` for the solution code.

b) ***10 points***

In this task you are going to implement the public interface of your heap data structure. A user should be able to create a heap based on a random input array that could look like this

```
>>> h = MinHeap([1, 8, -2, 3])
>>> print(h)
-2 3 1 8
```

The `__init__` method will call a helper method `_build_heap` which will create a heap by enforcing the heap property, starting with a random arrangement of values in the `_heap` attribute. There are different ways to accomplish this, one naive approach was discussed in class. The sift-up and sift-down methods you have implemented in the previous task could be used here. Please implement the `_build_heap` method to create the initial heap.

A user further needs to be able to push (insert) new values to the heap and pop (remove) the top element h_0 in the heap (i. e. the root of the binary tree). The public `push(self, value)` is used to insert new elements. This method does not return anything. The public `pop(self)` method is used to obtain h_0 and remove it from the heap. If the `pop` method is called on an empty heap, an `IndexError` exception must be raised. The `pop` method will return h_0. Both of these operations will destroy the heap property and you must restore it by using the sift-up and sift-down operations you have implemented in the previous task.

After implementation of the `push` and `pop` operations, a continued example application could be:

```
>>> print(h.pop())
-2
>>> print(h)
1 3 8
>>> h.push(2)
>>> print(h)
1 2 8 3
```

Please see `solution/code/p3/P3.py` for the solution code.

c) ***5 points***

In this task you are going to extend your heap library by further providing a `MaxHeap` class for *max-heap* data structures. The max-heap implementation should inherit from the min-heap implementation of the previous two tasks and overwrite only the necessary method(s), such that the heap correctly maintains the max-heap property. A user should be able to create a max-heap based on a random input array that could look like this

```
>>> h = MaxHeap([1, 8, -2, 3])
>>> print(h)
8 3 -2 1
```

Please see `solution/code/p3/P3.py` for the solution code.

Problem 4: Priority Queues (*30 points*)

Deliverables:

1. `P4.py`
2. `P4.png`

Important: this problem will be partially *auto-graded*. You are expected to work with the skeleton code provided in `code/p4/P4.py`. Do not change existing code in that skeleton code. Only add your solution code in appropriate places or update values in existing variables. Otherwise you will risk that the auto-grader fails and you lose points. Read carefully the problem statement and ensure your implementation follows the specifications given.
The auto-grader works by running a set of tests on your implementation. It is therefore important that you are throwing exceptions at places in your code where values do not correspond to what is expected.

In this problem you will test three implementations of priority queues. A priority queue is a queue where its elements are assigned a certain priority. In this problem we will consider elements that compare less of higher priority, that is, a queue element q_i has higher priority over queue element q_j if $q_i < q_j$ resulting in a minimum ordered priority queue. Getting elements from such a priority queue and appending them into a list would result in a sorted list in ascending order.

The priority queue we consider here implements three public methods: `put(value)` is used to put (insert) `value` into the priority queue, `get()` is used to retrieve and remove the element with the highest priority from the queue and `peek()` is used to retrieve the element with the highest priority but keep it in the queue. These methods are inherited from the `PriorityQueue` base class. *You are not supposed to add any new attributes to your implementation.*

Hint: *Assume the elements in your queue are all of the same type.*

a) ***10 points***

In this task you are implementing the `NaivePriorityQueue` using a simple built-in list. You can insert new values by appending to the list and retrieve values by scanning the list for the minimum value. Removing elements may be achieved by using the `remove()` method of built-in Python lists.

The public interface is already implemented in the base class `PriorityQueue` in the skeleton code `code/p4/P4.py`. Your implementation should inherit from this class and implement the `put`, `get` and `peek` methods. The priority queue is further limited by a maximum number of elements that are allowed in the queue. This capacity parameter is given by the `_max_size` attribute and initialized in the `__init__` method of the `PriorityQueue` base class.

The `put(self, value)` method should insert `value` at the end of the `_elements` list and it should raise an `IndexError` if the capacity limit of the priority queue is reached. The `put` method does not return a value. The `get(self)` method should remove the

smallest element from the list and return it. Finally, the `peek(self)` method will return the smallest element in the list without removing it. Both `get` and `peek` should raise an `IndexError` if the queue is empty. An example application could look like this

```
>>> queue = NaivePriorityQueue(2)
>>> queue.put(1)
>>> queue.put(2)
>>> queue.peek()
1
```

Please see `solution/code/p4/P4.py` for the solution code.

b) ***10 points***

A more efficient priority queue implementation would make use of a heap data structure. Similar to task 4a, implement the `HeapPriorityQueue` class using your `MinHeap` implementation of Problem 3. You can use the `push`, `pop` and `top` methods in the heap to implement `put`, `get` and `peek`, respectively. The implementation should raise `IndexError` in situations similar to the ones described in task 4a.

For another reference, implement the `BuiltinHeapPriorityQueue` class using Python's built-in module `heapq` from the standard library.[1] You will need the `heappush` and `heappop` methods from this module. Your implementation should raise `IndexError` in situations similar to the ones described in task 4a.

Please see `solution/code/p4/P4.py` for the solution code.

c) ***10 points***

In this final part you are going to benchmark your priority queue implementations to investigate performance differences. The homework handout provides the benchmark code in `code/p4/merge_lists_benchmark.py`. The benchmark application makes use of priority queues to merge multiple sorted lists into one single sorted list. You can think of it as if you would zip together multiple lists into one single list maintaining the sort order. The following demonstration merges two sorted lists of length 3 into one sorted list of length 6 using the `NaivePriorityQueue` to perform the merge:

```
>>> benchmark(2, NaivePriorityQueue, list_length=3, n_samples=1, debug=True)
Sample list: [-3, -1, 0]
Sample list: [-1, 0, 1]
Merged list: [-3, -1, -1, 0, 0, 1]
```

You can use the benchmark to measure execution time in milliseconds for a number of list counts to be merged together. For example, the benchmark run

```
>>> benchmark([2, 10], NaivePriorityQueue)
[0.1595020294189453, 0.8007049560546875]
```

[1] https://docs.python.org/3/library/heapq.html

merges 2 and 10 lists using the `NaivePriorityQueue` in 0.16 and 0.80 milliseconds, respectively. Create a plot to compare the execution time of the list merge benchmark for the three implementations of the `NaivePriorityQueue`, `HeapPriorityQueue` and `BuiltinHeapPriorityQueue` priority queues using the provided `n_lists` counts. You should create *one plot* with three curves, where the list counts should be on the abscissa and the execution time in milliseconds should be on the ordinate. Save your plot in the file `P4.png`. Do not forget to add a title, a legend and to label the axes appropriately. Use a linear scale for the abscissa and ordinate.

Please see `solution/code/p4/P4.py` for the solution code. The plot should look like figure 1. The built-in heap version performs best because it uses a C implementation

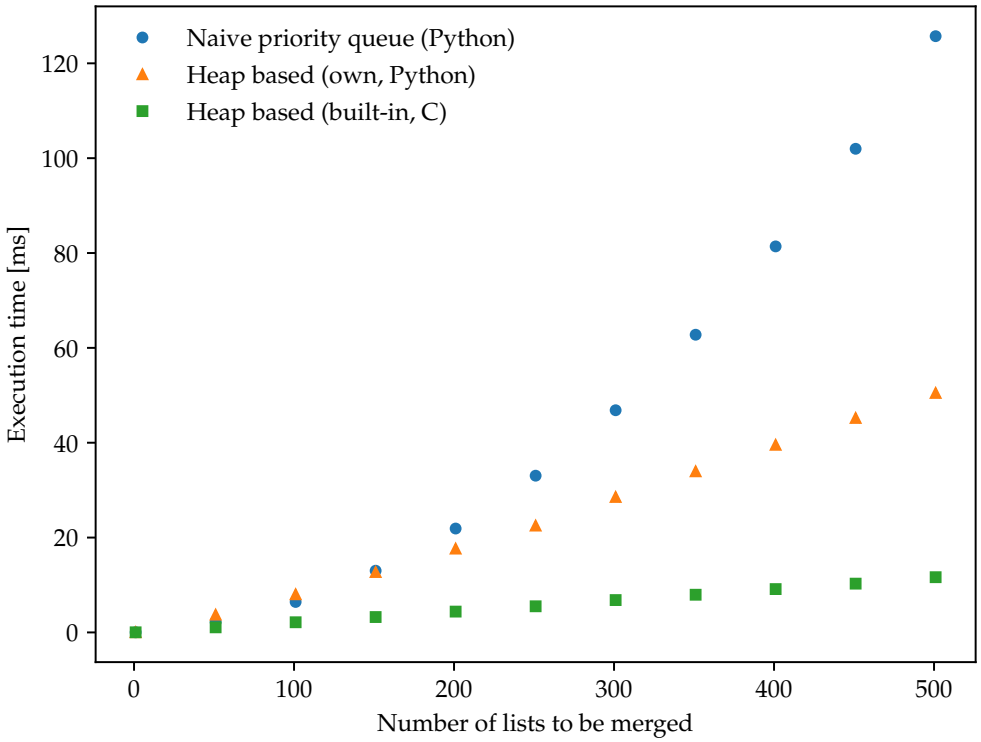

FIGURE 1: Benchmark results for the three priority queue implementations. The built-in heap version from the Python standard library performs best because it uses a C implementation instead of pure Python.

instead of a pure Python implementation. Whenever possible, use code available to you in the standard library.

- ***2 points*** for each of the three benchmark curves (***6 points*** total).
- ***1 points*** for the title.
- ***1 points*** for the legend.
- ***1 points*** for the two axes labels each (***2 points*** total).