Assignment_5

April 3, 2024

1 Assignment 5

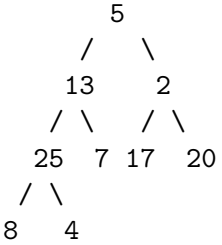
Peder Ørmen Bukaasen, Bård Tollef Pedersen and Eivind Lid Trøen

1.1 Exercise 1

Given the array A = [5, 13, 2, 25, 7, 17, 20, 8, 4], let's illustrate how heapsort operates step by step:

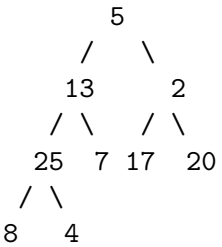
1.1.1 Initial Heap Construction

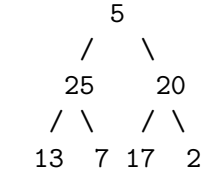
Convert the given array into a max heap.
The initial heap is represented as:

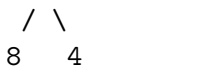


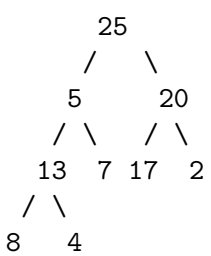
1.1.2 Create Max Heap

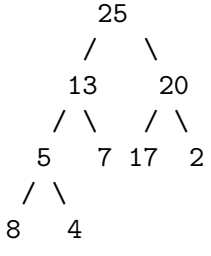
Convert the heap into a max heap. This is done by starting at the last non-leaf node and moving up the tree, swapping the parent with the largest child until the heap property is satisfied.

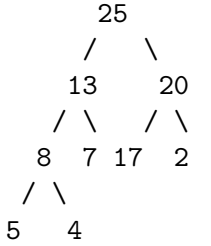






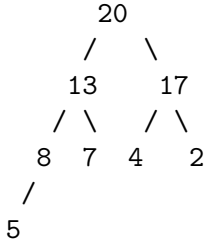




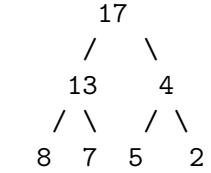


1.1.3 Sorting Phase

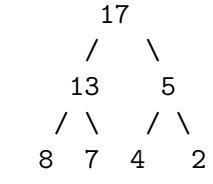
Then we remove the top node and move the highest child of top node to the new top. And then we heapify the heap again.



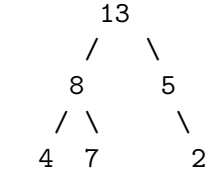
25



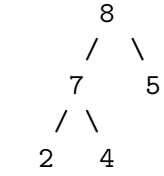
25, 20 Need to heapify the heap again.



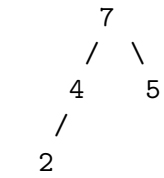
25, 20, 17



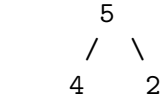
25, 20, 17, 13



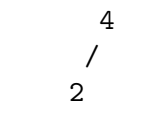
25, 20, 17, 13, 8



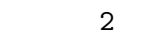
25, 20, 17, 13, 8, 7



25, 20, 17, 13, 8, 7, 5



25, 20, 17, 13, 8, 7, 5, 4



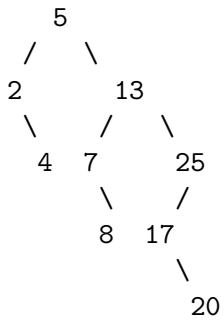
25, 20, 17, 13, 8, 7, 5, 4, 2

1.2 Exercise 2

Given the array A = [5, 13, 2, 25, 7, 17, 20, 8, 4], let's illustrate how tree-sort operates step by step:

1.2.1 Initial Tree Construction

Convert the given array into a binary search tree. The initial tree is represented as:



1.2.2 In-order Traversal

Perform an in-order traversal of the tree to get the sorted array.

Go as far left as possible for the smallest element. 2

Then go to the right child of 2, if any. 4

Then go to the parent of 2. 5

Then go to the right child of 5. And then go as far left as possible. 7

Then go to the right child of 7. 8

Then go to the parent of 7. 13

Then go to the right child of 13. And then go as far left as possible. 17

Then go to the right child of 17. 20

Then go to the parent of 17. 25

The sorted array is [2, 4, 5, 7, 8, 13, 17, 20, 25].

1.3 Exercise 3

Main idea is to have 2 stacks. Stack 1 is used for enqueue operations, and Stack 2 is used for dequeue operations. When an element is dequeued, if Stack 2 is empty, all elements from Stack 1 are popped and pushed to Stack 2.

[ ]:
```
"""
Class Stack retrived from Lecture 11.
"""
class Stack:
    def __init__(self, size):
        self._data = [None] * size
        self._size = size
        self._top = -1

    def empty(self):
```

```
        return self._top < 0

    def push(self, x):
        if self._top == self._size - 1:
            raise RuntimeError('Stack overflow')
        self._top += 1
        self._data[self._top] = x

    def pop(self):
        if self.empty():
            raise RuntimeError('Stack underflow')
        self._top -= 1
        return self._data[self._top + 1]

    def top(self):
        if self.empty():
            raise RuntimeError('Stack underflow')
        return self._data[self._top]


"""
Class Queue is retrived from Lecture 11 and modified to fit the requirements of␣
 ↪the problem.
"""
class Queue:
    def __init__(self, size):
        self.stack_1 = Stack(size)
        self.stack_2 = Stack(size)
        self._size = self.stack_1._size

    def enqueue(self, x):
        self.stack_1.push(x)

    def dequeue(self):
        # Move all elements from stack_1 to stack_2
        while not self.stack_1.empty():
            self.stack_2.push(self.stack_1.pop())

        # Pop the element from stack_2
        x = self.stack_2.pop()

        # Move all elements back to stack_1
        while not self.stack_2.empty():
            self.stack_1.push(self.stack_2.pop())

        # Return the element
        return x
```

[ ]:
```
q = Queue(10)
for i in range(10):
    q.enqueue(i)

for i in range(10):
    print(q.dequeue())
```

```
0
1
2
3
4
5
6
7
8
9
```

[ ]:
```

```