Solution to Homework 3

Problem 1. (1 point) Illustrate the operation of heapsort on the array:
\[ A = (19, 2, 11, 14, 7, 17, 4, 3, 5, 15) \]
By showing the values in array \( A \) after initial heapification and after each call to \( \text{max-heapify} \).

Solution:
\[
\begin{align*}
A &= [19, 15, 17, 14, 7, 11, 4, 3, 5, 2] & \text{# After initial heapification} \\
A &= [17, 15, 11, 14, 7, 2, 4, 3, 5, 19] & \text{# After extracting 19} \\
A &= [14, 7, 11, 5, 3, 2, 4, 15, 17, 19] & \text{# After extracting 17} \\
A &= [11, 7, 4, 5, 3, 2, 14, 15, 17, 19] & \text{# After extracting 15} \\
A &= [7, 5, 4, 2, 3, 11, 14, 15, 17, 19] & \text{# After extracting 14} \\
A &= [5, 3, 4, 2, 7, 11, 14, 15, 17, 19] & \text{# After extracting 11} \\
A &= [4, 3, 2, 5, 7, 11, 14, 15, 17, 19] & \text{# After extracting 7} \\
A &= [3, 2, 4, 5, 7, 11, 14, 15, 17, 19] & \text{# After extracting 5} \\
A &= [2, 3, 4, 5, 7, 11, 14, 15, 17, 19] & \text{# After extracting 3}
\end{align*}
\]

Problem 2. (1 point) Illustrate the operation of counting sort on the array:
\[ A = (4, 6, 3, 5, 0, 5, 1, 3, 5, 5) \]
By showing the values in array \( C \) after each loop, and the final array \( B \).

Solution:
\[
\begin{align*}
&0 1 2 3 4 5 6 \\
C &= [0, 0, 0, 0, 0, 0, 0] & \text{# After initialization} \\
C &= [1, 1, 0, 2, 1, 4, 1] & \text{# After counting elements} \\
C &= [1, 2, 2, 4, 5, 9, 10] & \text{# After computing running sum} \\
C &= [0, 1, 2, 2, 4, 5, 9] & \text{# After producing sorted } B \\
B &= [0, 1, 3, 3, 4, 5, 5, 5, 5, 6] & \text{# Sorted } B
\end{align*}
\]

Problem 3. (1 point) Illustrate the operation of radix sort on the array:
\[ A = (392, 517, 364, 931, 726, 912, 299, 250, 600, 185) \]
By showing the values in array \( A \) after each intermediate sort.

Solution:
\[
\begin{align*}
A &= [392, 517, 364, 931, 726, 912, 299, 250, 600, 185]
\end{align*}
\]
Problem 4. (1 point) Illustrate the operation of bucket sort on the array:
\[ A = (0.88, 0.23, 0.25, 0.74, 0.18, 0.02, 0.69, 0.56, 0.57, 0.49) \]
By showing the final array \( B \) of sorted buckets.

Solution:
\[
B = \{0: [0.02],
1: [0.18],
2: [0.23, 0.25],
3: [],
4: [0.49],
5: [0.56, 0.57],
6: [0.69],
7: [0.74],
8: [0.88],
9: []\}
\]

Problem 5. (3 points) Consider a \( d \)-ary heap – a generalization of a binary heap, in which all internal nodes (with at most one exception) have \( d \) children. Design a method to store a \( d \)-ary heap in an array and derive the expressions for:

(a) Parent of \( i \)-th node in a \( d \)-ary heap.

(b) \( j \)-th child of \( i \)-th node in a \( d \)-ary heap.

(c) Maximum number of nodes of height \( h \) in any \( n \)-element \( d \)-ary heap.

(d) Maximum height of any \( n \)-element \( d \)-ary heap.

Solution:

(a) Parent of \( i \)-th node in a \( d \)-ary heap.

(b) \( j \)-th child of \( i \)-th node in a \( d \)-ary heap.

Assuming 1-based indexing, we can extend the standard level-wise method of storing binary heaps to \( d \)-ary heaps by taking:

\[
parent(i) = \left\lfloor \frac{i - 1}{d} \right\rfloor,
\]

\[
child(i, j) = d(i - 1) + j + 1.
\]
(c) Maximum number of nodes of height $h$ in any $n$-element $d$-ary heap.

As $parent(i) = \left\lfloor \frac{i-1}{d} \right\rfloor$, the parent of the last node is $parent(n) = \left\lfloor \frac{n-1}{d} \right\rfloor$. Defining $n_h$ as the number of nodes of height $h$, and observing that the parent of the last node is the last non-leaf node, we can express the number of leaf nodes as:

$$n_0 = n - \left\lfloor \frac{n-1}{d} \right\rfloor.$$

Removing all leaf nodes from the heap of size $n$ would result in a heap of size $n'$:

$$n' = n - n_0 = \left\lfloor \frac{n-1}{d} \right\rfloor$$

with $n'_0$ leaves:

$$n'_0 = n' - \left\lfloor \frac{n'-1}{d} \right\rfloor = \left\lfloor \frac{n-1}{d} \right\rfloor - \left\lfloor \frac{n-1}{d} \right\rfloor = \left\lfloor \frac{n-1}{d} \right\rfloor - \left\lfloor \frac{n-1-d}{d^2} \right\rfloor.$$

Removing all leaf nodes from the heap of size $n'$ would result in a heap of size $n''$:

$$n'' = n' - n'_0 = \left\lfloor \frac{n-1-d}{d^2} \right\rfloor$$

with $n''_0$ leaves:

$$n''_0 = n'' - \left\lfloor \frac{n''-1}{d} \right\rfloor = \left\lfloor \frac{n-1-d}{d^2} \right\rfloor - \left\lfloor \frac{n-1-d}{d^2} \right\rfloor = \left\lfloor \frac{n-1-d}{d^2} \right\rfloor - \left\lfloor \frac{n-1-d-d^2}{d^3} \right\rfloor,$$

suggesting a general expression for a number of leaves in a heap obtained from a heap of size $n$ by removing all leaf nodes $k$ times:

$$n^{(k)}_0 = \left\lfloor \frac{n-1-d-\ldots-d^{k-1}}{d^k} \right\rfloor - \left\lfloor \frac{n-1-d-\ldots-d^k}{d^{k+1}} \right\rfloor = \left\lfloor \frac{n-\sum_{i=0}^{k-1} d^i}{d^k} \right\rfloor - \left\lfloor \frac{n-\sum_{i=0}^{k} d^i}{d^{k+1}} \right\rfloor = \left\lfloor \frac{n-d^{k-1}}{d^k} \right\rfloor - \left\lfloor \frac{n-d^{k+1-1}}{d^{k+1}} \right\rfloor \text{ for } d > 1.$$
We now observe that the leaves in heap $H'$, obtained from heap $H$ by removing all leaf nodes, had height 1 in the original heap $H$. The leaves in heap $H''$, obtained from heap $H'$ by removing all leaf nodes, had height 1 in $H'$ and height 2 in $H$. In general, $n_h$ – the number of nodes of height $h$ in an $n$-element $d$-ary heap $H$ – is the number of leaves in $H^{(h)}$, obtained from $H$ by removing all leaf nodes $h$ times:

$$n_h = \left\lfloor \frac{n - \frac{d^h - 1}{d - 1}}{d^h} \right\rfloor - \left\lfloor \frac{n - \frac{d^{h+1} - 1}{d - 1}}{d^{h+1}} \right\rfloor \quad \text{for } d > 1.$$

(d) Maximum height of any $n$-element $d$-ary heap.

We notice that a maximum number of nodes in a $d$-ary heap of height $h$ is observed when the last level is full, and a minimum number of nodes – when the last level has only one node:

$$n_{\max}(h) = d^0 + d^1 + ... + d^{h-1} + d^h = \frac{d^{h+1} - 1}{d - 1} \quad \text{for } d \neq 1,$$

$$n_{\min}(h) = d^0 + d^1 + ... + d^{h-1} + 1 = \frac{d^h - 1}{d - 1} + 1 \quad \text{for } d \neq 1.$$

This allows us to provide a tight bound on the size of a $d$-ary heap of height $h$ (we assume $d \neq 1$ for the rest of this problem):

$$\frac{d^h - 1}{d - 1} + 1 \leq n \leq \frac{d^{h+1} - 1}{d - 1}.$$

With integer $n$ we can get rewrite and then reduce the inequality as follows:

$$\frac{d^h - 1}{d - 1} < n \leq \frac{d^{h+1} - 1}{d - 1},$$

$$d^h - 1 < n(d - 1) \leq d^{h+1} - 1,$$

$$d^h < n(d - 1) + 1 \leq d^{h+1},$$

$$h < \log_d (n(d - 1) + 1) \leq h + 1.$$

For integer $n$ and real $x$:

$$n - 1 < x \leq n \quad \text{iff} \quad n = \lceil x \rceil,$$

and so:

$$h = \left\lfloor \log_d (n(d - 1) + 1) \right\rfloor - 1.$$
(a) Unordered array.
(b) Ordered array.
(c) Unordered linked list.
(d) Ordered linked list.
(e) Min-heap.

Solution:

(a) Unordered array of size $n$ referenced by variable $array$:

- **insert($k$)**: Without the need to maintain order, we can insert in $O(1)$ by adding the new element to the end of the array.

```python
def insert($k$):
    $array$.append($k$)
```

We note that the true worst case is $O(n)$ and is observed when the array needs to be reallocated (though amortized running time is still constant).

- **get-min()**: With no order to rely upon, we have to resort to a $O(n)$ linear scan.

```python
def get-min():
    assert len($array$) > 0
    index = find-min($array$) # O(n)
    return $array$[index]
```

- **extract-min()**: As with get-min(), we have no way around a $O(n)$ linear scan. We don't, however, need to spend another $O(n)$ closing the gap left after removing the element, as with no order to worry about we can just fill the hole with the last element.

```python
def extract-min():
    assert len($array$) > 0
    index = find-min($array$) # O(n)
    value = $array$[index]
    exchange($array$[index], $array$[len($array$) - 1])
    $array$.delete(len($array$) - 1) # O(1)
    return value
```

(b) Ordered array of size $n$ referenced by variable $array$:

- **insert($k$)**: One way to insert the element into a sorted array is to use modified binary search to find correct position in $O(\lg n)$ time, then insert and shift the elements in $O(n)$. An alternative is to "drag" the element into its correct position by successive exchanges (not unlike insertion sort). Both ways are $O(n)$ in the worst case.
def insert(k):
    index = find-insert-position(array, k) # O(lgn)
    array.insert(k, index) # O(n)

• get-min(): It is more convenient to represent a min-heap with an array sorted
in non-increasing order (so we can delete the minimum element in O(1)). The
minimum element is then in the last position and can be accessed in O(1).

def get-min():
    assert len(array) > 0
    return array[len(array) - 1] # O(1)

• extract-min(): Deleting the last element does not require closing any gaps and
can be done in O(1):

def extract-min():
    assert len(array) > 0
    value = array[len(array) - 1] # O(1)
    array.delete(len(array) - 1) # O(1)
    return value

(c) Unordered linked list of size n referenced by variable list:

• insert(k): Without the need to maintain order, we can insert in O(1) by adding
the new element to the head of the list.

def insert(k):
    list.insert(k, list.head())

    We note that the linked list data does not have to be contiguous and we are unaf-
fected by the array’s "true worst case" when the storage needs to be reallocated.

• get-min(): As with the unordered array, we have to resort to a O(n) linear scan.

def get-min():
    return find-min(list) # O(1)

• extract-min(): As with get-min(), we have no way around a O(n) linear scan. There are, however, no gaps to be closed – we can delete from any position in O(1).

def extract-min():
    # O(n) to find, O(1) to remove
    return find-and-remove-min(list)

(d) Ordered linked list of size n referenced by variable list:

• insert(k): Linked lists do not support indexing, so we cannot use binary search to
find the insertion position and have to resort to a linear scan. Once the position is
found, however, we can insert in O(1) (assuming the position is given as a reference,
not an index):
def insert(k):
    ref = find-insert-position(list, k) # O(n)
    list.insert(k, ref) # O(1)

• get-min(): Assuming the list is sorted in a non-decreasing order, the minimum element is the head of the list and can be accessed in O(1):

    def get-min():
        return list.head() # O(1)

• extract-min(): We can delete at any position in O(1):

    def extract-min():
        min = list.head() # O(1)
        remove-head(list) # O(1)
        return min

(e) Min-heap referenced by variable heap:

• insert(k): We can insert an element into the heap in O(lgn) by adding it as the last node and then percolating it up until the min-heap property is no longer broken:

    def insert(k):
        ++heap.size
        i = heap.size
        heap[i] = k
        while i > 0 and heap[parent(i)] > heap[i]: # O(lgn)
            exchange(heap[parent(i)], heap[i])
            i = parent(i)

• get-min(): The minimum element in a min-heap is at the root and can be accessed in O(1):

    def get-min():
        return heap[0] # O(1)

• extract-min(): We can remove the minimum element from the heap by replacing the root element with the last element, excluding the last element from the heap, and rebuilding the heap, all in O(lgn):

    def extract-min():
        min = heap[0]
        heap[0] = heap[heap.size - 1]
        --heap.size
        min-heapify(heap, 0) # O(lgn)
        return min