Assignment 3

Benjamin Jakubowski

June 16, 2016

1. Illustrating heapsort

We 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 `max-heapify`

Call	**A**
Initial state	A = [19, 2, 11, 14, 7, 17, 4, 3, 5, 15]
After heapification	A = [19, 15, 17, 14, 7, 11, 4, 3, 5, 2]
After `max-heapify` call 1	A = [17, 15, 11, 14, 7, 2, 4, 3, 5, 19]
After `max-heapify` call 2	A = [15, 14, 11, 5, 7, 2, 4, 3, 17, 19]
After `max-heapify` call 3	A = [14, 7, 11, 5, 3, 2, 4, 15, 17, 19]
After `max-heapify` call 4	A = [11, 7, 4, 5, 3, 2, 14, 15, 17, 19]
After `max-heapify` call 5	A = [7, 5, 4, 2, 3, 11, 14, 15, 17, 19]
After `max-heapify` call 6	A = [5, 3, 4, 2, 7, 11, 14, 15, 17, 19]
After `max-heapify` call 7	A = [4, 3, 2, 5, 7, 11, 14, 15, 17, 19]
After `max-heapify` call 8	A = [3, 2, 4, 5, 7, 11, 14, 15, 17, 19]
After `max-heapify` call 9	A = [2, 3, 4, 5, 7, 11, 14, 15, 17, 19]

2. Illustrating counting sort

We 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 step in the three loops internal to counting sort.

Time	**State**
After counting A[0]	C = [0, 0, 0, 0, 1, 0, 0]
After counting A[1]	C = [0, 0, 0, 0, 1, 0, 1]
After counting A[2]	C = [0, 0, 0, 1, 1, 0, 1]
After counting A[3]	C = [0, 0, 0, 1, 1, 1, 1]
After counting A[4]	C = [1, 0, 0, 1, 1, 1, 1]
After counting A[5]	C = [1, 0, 0, 1, 1, 2, 1]
After counting A[6]	C = [1, 1, 0, 1, 1, 2, 1]
After counting A[7]	C = [1, 1, 0, 2, 1, 2, 1]
After counting A[8]	C = [1, 1, 0, 2, 1, 3, 1]
After counting A[9]	C = [1, 1, 0, 2, 1, 4, 1]
After counting elements $\leq$ C[1]	C = [1, 2, 0, 2, 1, 4, 1]
After counting elements $\leq$ C[2]	C = [1, 2, 2, 2, 1, 4, 1]
After counting elements $\leq$ C[3]	C = [1, 2, 2, 4, 1, 4, 1]
After counting elements $\leq$ C[4]	C = [1, 2, 2, 4, 5, 4, 1]
After counting elements $\leq$ C[5]	C = [1, 2, 2, 4, 5, 9, 1]
After counting elements $\leq$ C[6]	C = [1, 2, 2, 4, 5, 9, 10]
After placing element A[9] in B	C = [1, 2, 2, 4, 5, 8, 10]
After placing element A[8] in B	C = [1, 2, 2, 4, 5, 7, 10]
After placing element A[7] in B	C = [1, 2, 2, 3, 5, 7, 10]
After placing element A[6] in B	C = [1, 1, 2, 3, 5, 7, 10]
After placing element A[5] in B	C = [1, 1, 2, 3, 5, 6, 10]
After placing element A[4] in B	C = [0, 1, 2, 3, 5, 6, 10]
After placing element A[3] in B	C = [0, 1, 2, 3, 5, 5, 10]
After placing element A[2] in B	C = [0, 1, 2, 2, 5, 5, 10]
After placing element A[1] in B	C = [0, 1, 2, 2, 5, 5, 9]
After placing element A[0] in B	C = [0, 1, 2, 2, 4, 5, 9]
Final sorted array B	B = [0, 1, 3, 3, 4, 5, 5, 5, 5, 6]

3. Illustrating radix sort

We 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.

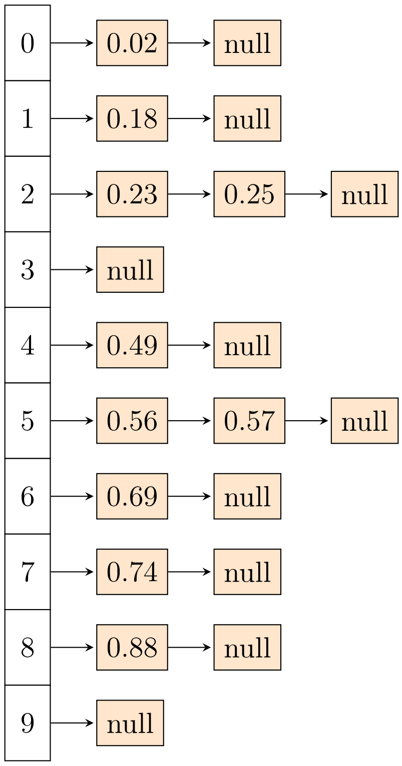
Time	**State**
After sorting in the 10^0s place	[250, 600, 931, 392, 912, 364, 185, 726, 517, 299]
After sorting in the 10^1s place	[600, 912, 517, 726, 931, 250, 364, 185, 392, 299]
After sorting in the 10^2s place	[185, 250, 299, 364, 392, 517, 600, 726, 912, 931]

4. Illustrating bucket sort

We 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 (before concatenation).[1]



5. d-ary heaps

Consider a d-ary heap in which all but one node have d children. Our objective is to design a method to store this heap as an array. Note we present the problem in the assigned, but **the proof of A actually uses the result of part B**. Similarly **the proof of C actually uses the result of part D**. Additionally, note we derive expressions using 0-indexing.

a. Parent of the i-th node

By (B), we know the j-th child of the i-th node in the d-ary heap is

$$child_j(i) = d \cdot i + j$$

[1] LaTeX TikZ code from `http://tex.stackexchange.com/questions/86766/array-of-linked-lists-like-in-data-structure`

Now we show the parent of the i-th node in the d-ary heap is

$$Parent(i) = \left\lfloor \frac{i-1}{d} \right\rfloor$$

Consider the k^{th} node, where $k > 0$. Then there exist unique $i, j \in \mathbb{N}$ such that

$$k = d \cdot i + j$$

where again $1 \leq j \leq d$.
Then, by (B) k must be the j^{th} child of i. Now let's find an expression for i. Note

$$\begin{aligned} & k = d \cdot i + j \\ \implies \quad & (k-1) = d \cdot i + (j-1) \\ \implies \quad & \frac{(k-1)}{d} = i + \frac{(j-1)}{d} \end{aligned}$$

But

$$0 \leq \frac{(1-1)}{d} \leq \frac{(j-1)}{d} \leq \frac{(d-1)}{d} < 1$$

So

$$\left\lfloor \frac{(k-1)}{d} \right\rfloor = \left\lfloor i + \frac{(j-1)}{d} \right\rfloor = i$$

B. j-TH CHILD OF THE i-TH NODE

We use induction to prove that the j-th child of the i-th node in the d-ary heap is

$$child_j(i) = d \cdot i + j$$

where $1 \leq j \leq d$.
Base case: Consider $i = 0$. Then the j^{th} child of the i^{th} node is clearly (for $1 \leq j \leq d$)

$$child_j(0) = j = d \cdot 0 + j = d \cdot i + j$$

Now the inductive step:
Assume $child_j(i) = d \cdot i + j$ for all$i \leq n$. now consider $i = n + 1$.
Then note

$$child_{j=d}(n) = d \cdot n + d$$

by the inductive hypothesis. Thus, the first child of the $(n+1)^{st}$ node must be

$$\begin{aligned} child_{j=0}(n+1) &= (d \cdot n + d) + 1 \\ &= d(n+1) + 1 \end{aligned}$$

Then the j^{th} child (for $1 \leq j \leq d$) is

$$child_j(n+1) = d(n+1) + j$$

C. Maximum number of nodes at height h

First, note the maximum number of nodes at depth x is clearly d^x (this should be obvious). Next, note that the height of a level in the tree is given by

$$\begin{aligned} & h = (\max h) - x \\ \implies \quad & x = (\max h) - h \end{aligned}$$

Then, substituting in the expression for the maximum number of nodes at height h found in part (D) yields

$$x = \lfloor \log_d(n(d-1)+1) - 1 \rfloor - h$$

So the maximum number of nodes at height h is

$$\max \# \text{ nodes} = d^{\lfloor \log_d(n(d-1)+1)-1 \rfloor - h}$$

D. Maximum height h

First, a quick note to disambiguate notation: in part (C), we let h be a particular height. Here, we will let h be the maximum height of an n-element d-ary heap.
Now consider an n-element 1-ary heap. The maximum height h is trivially $n-1$. Going forward, assume $d > 1$. We show the maximum height h is

$$h = \lfloor \log_d(n(d-1)+1) - 1 \rfloor$$

First, note the maximum number of nodes in a d-ary heap of height h is

$$\sum_{i=0}^{h} d^i = \max \text{number of nodes}$$

Additionally, the minimium number of nodes in a d-ary heap of height h is

$$\left(\sum_{i=0}^{h-1} d^i \right) + 1 = \min \text{number of nodes}$$

So, given a heap has n nodes the maximum height h is h such that

$$\sum_{i=0}^{h-1} d^i < n \leq \sum_{i=0}^{h} d^i$$

But recall the

$$\sum_{i=0}^{k} x^i = \frac{x^{k+1}-1}{x-1}$$

Thus

$$
\begin{aligned}
\sum_{i=0}^{h-1} d^i < n \leq \sum_{i=0}^{h} d^i \\
\implies \quad \frac{d^h - 1}{d-1} < n \leq \frac{d^{h+1}-1}{d-1} \\
\implies \quad d^h - 1 < n(d-1) \leq d^{h+1} - 1 \\
\implies \quad d^h < n(d-1) + 1 \leq d^{h+1} \\
\implies \quad h < \log_d(n(d-1)+1) \leq h+1 \\
\implies \quad h - 1 < \log_d(n(d-1)+1) - 1 \leq h \\
\implies \lceil \log_d(n(d-1)+1) - 1 \rceil = h
\end{aligned}
$$

So the maximum height of an n-element d-ary heap is $\lceil \log_d(n(d-1)+1) - 1 \rceil$.

6. Min-priority queue

In this section we consider a min-priority queue representing a set of integers and supporting the following operations:

- `insert(k)`: insert an element with value `k`
- `get-min()`: return the minimum element
- `extract-min()`: remove and return the minimum element

We give pseudocode and worst-case running time for each operation, given the priority queue is implemented with different data structures. Note we assume arrays are dynamic (and support insertions and deletions)

a. Unordered array

Let $A = [a_1, \cdots, a_n]$ be an unordered array. Then
The worst case running times are as follows- note we are assuming space has been pre-allocated for insertion into the array (and as such are ignoring the cost of copying the array into a larger memory block).

- `Insert`: Under our previously stated assumption, this operations is O(1).
- `Get-min`: This operation is O(n) (since we loop through the entire array once).
- `Delete-min`: This operation is O(n). We loop through the entire array once (an O(n) operation), and deleting is constant time (since we are not concerned with array ordering, we can simply replace A[min_index] with A[n], then delete A[n]).

Algorithm 1 Unordered array operations

```
function INSERT(A, k)
    A[n+1] = k
function GET-MIN(A)
    min = A[1]
    for i in 2 to n do
        if A[i] < min then
            min = A[i]
    return min
function DELETE-MIN(A)
    min = A[1]
    min_index = 1
    for i in 2 to n do
        if A[i] < min then
            min = A[i]
            min_index = i
    A[min_index] = A[n]
    delete A[n]
    return min
```

B. Ordered array

Let $A = [a_1, \cdots, a_n]$ be an ordered array. Since our objective is to implement a min-priority queue, assume it is in reverse order (with the least element at $A[n]$). Then

Algorithm 2 Ordered array operations

```
function INSERT(A, k)
    Use binary search to find where to insert k
    Shift all elements in A[j : n] by one index position.
    Insert k at array index j (i.e. A[j] = k)
function GET-MIN(A)
    return A[n]
function EXTRACT-MIN(A)
    temp = A[n]
    delete A[n]
    return temp
```

The worst case running times are as follows- again we are assuming space has been pre-allocated for insertion into the array (and as such are ignoring the cost of copying the array into a larger memory block).

- `Insert`: Finding where to insert k is O(log n), shifting the elements is O(n), and inserting (after the shift) is O(1). Hence the final running time is

$$\text{O(n)} + \text{O(log n)} + \text{O(1)} = \text{O(n)}$$

- `Get-min`: This operation is O(1) (since we no longer need to search through the array once it's ordered).
- `Delete-min`: This operation is O(1) since again we no longer need to loop through the entire array.

c. Unordered linked list

Now let A be an unordered linked list

Algorithm 3 Unordered linked list operations

```
function INSERT(A, k)
    new_node = new Node()
    new_node.value = k
    new_node.next = A.head.next
    A.head.next = new_node
    return A
function GET-MIN(A)
    min = A.head.next.value
    node = A.head.next
    while node not Null do
        node = node.next
        if node.value < min then
            min = node.value
    return min
function EXTRACT-MIN(A)
    points_to_min_node = A.head
    last_seen_node = A.head
    current_node = A.head.next
    min_node = A.head.next
    while current_node not Null do
        last_seen_node = current_node
        current_node = current_node.next
        if current_node.value < min_node.value then
            min_node = current_node
            points_to_min_node = last_seen_node
    points_to_min_node.next = min_node.next
    min = min_node.value
    delete min_node
    return min
```

The worst case running times are as follows:

- `Insert`: Inserting into an unordered linked list is O(1), all operations are constant time.
- `Get-min`: This operation is O(n), since again we need to loop through every node in the linked list.

- `Delete-min`: This operation is also O(n), since again we are looping through the entire linked list.

D. Ordered linked list

Now let A be an ordered linked list

Algorithm 4 Ordered linked list operations

```
function INSERT(A, k)
    last_seen_node = A.head
    current_node = A.head.next
    if current_node not Null and current_node.value < k then
        last_seen_node = current_node
        current_node = current_node.next
    new_node = new Node()
    new_node.value = k
    new_node.next = current_node
    last_seen_node.next = new_node
    return A
function GET-MIN(A)
    return A.head.next.value
function EXTRACT-MIN(A)
    min_node = A.head.next
    min = min_node.value
    A.head.next = min_node.next
    delete min_node
    return min
```

The worst case running times are as follows:

- `Insert`: In the worst case, this operation is O(n), since we may need to search through the entire linked list to find the correct position to insert the node.
- `Get-min`: This operation is O(1), since we need only retrieve the first node in the list (as it is sorted).
- `Delete-min`: This operation is also O(1), since after getting the first node, all we need to do is redirect pointers (from the head of the list to the second node).

E. Min-heap

Now let A be a min-heap.
The worst case running times are as follows:

- `Insert`: In the worst case, this operation is O(log n), since the while loop may require exchanging elements all the way from the initial index (n+1) at a leaf node, up to the root. The length of this path is O(log n).

Algorithm 5 Min-heap operations

```
function INSERT(A, k)
    A.heap_size ++
    k_index = n + 1
    A[k_index] = k
    while k_index not 1 and A[k_index] < A[parent(k_index)] do
        exchange A[k_index] and A[parent(k_index)
        set k_index to parent(k_index)
    return A
function GET-MIN(A)
    return A[1]
function EXTRACT-MIN(A)
    min = A[1]
    A[1] = A[n]
    A.heap_size = A.heap_size - 1
    min-heapify(A, 1)
    return min
```

- `Get-min`: This operation is O(1), since we need only retrieve the root of our min-heap.
- `Delete-min`: This operation is also O(log n). After setting A[1] = A[n], min-heapify will run in O(log n) time, since it will make at most O(log n) exchanges (from the root to a leaf of our binary tree).