CS 677 Homework
Assignment 05

Hai Nguyen

October 3, 2018

1. (a) Illustrate the operation of BUILD-MAX-HEAP on the array A = [5, 3, 17, 10, 84, 19, 6, 22, 9]. The operation is shown in Figure 1.

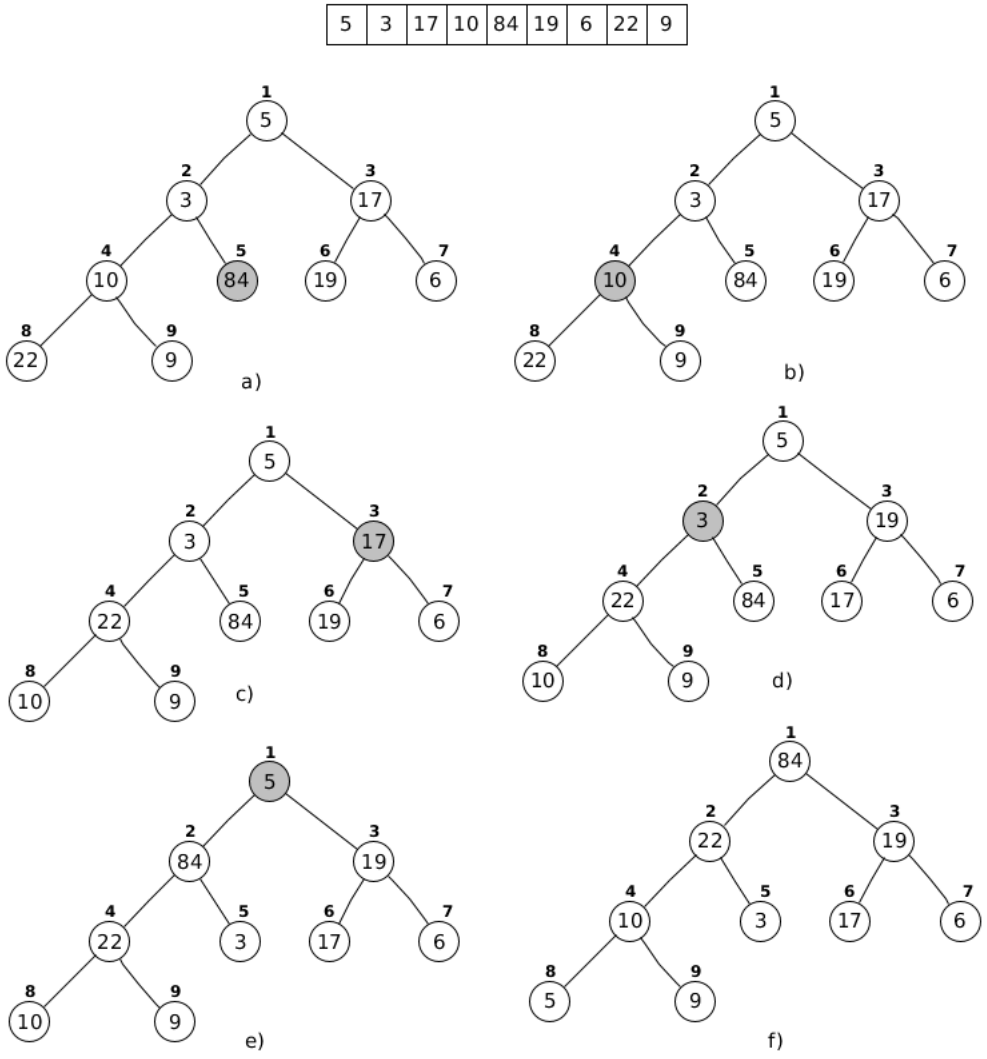

Figure 1: Illustration of BUILD-MAX-HEAP.

(b) Illustrate the operation of COUNTING-SORT on the array A = [6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2] The operation is shown in Figure 2.

2. Suppose that we were to write the **for** loop header in line 10 of the COUNTING-SORT as

   10 **for** j = 1 **to** A.*length*

   Show that the algorithm still works properly. Is the modified algorithm stable?

   for j = A.*length* **downto** 1

   B[C[A[j]]] = A[j]

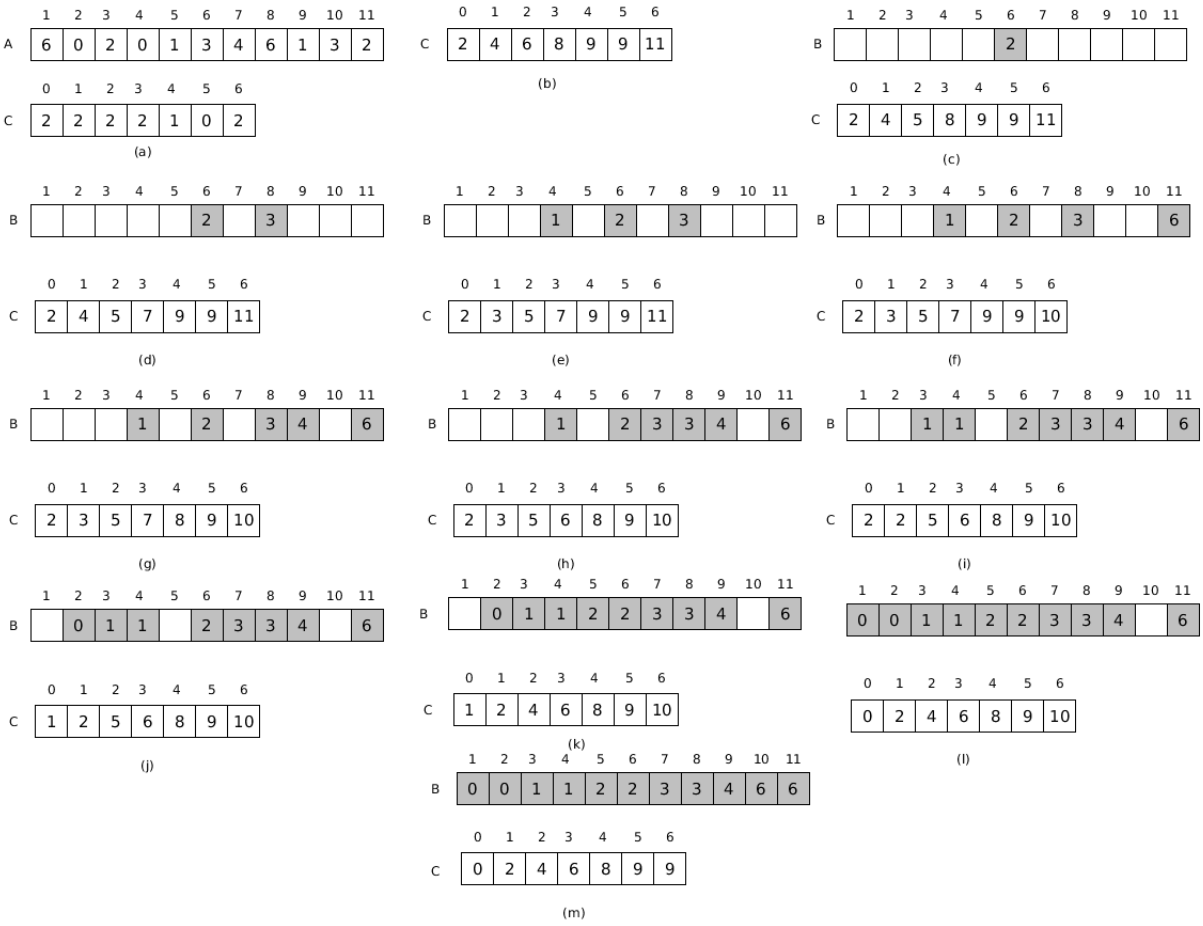

Figure 2: Illustration of COUNTING-SORT.

C[A[j]] = C[A[j]] - 1

The main purpose of the loop is to place each element A[j] into its correct sorted position in the output array B. If n elements are distinct, then for each A[j] the value C[A[j]] is the correct final position of A[j] in the output array, since there are C[A[j]] elements less than or equal to A[j]. Because the elements might not be distinct, we decrement C[A[j]] each time we place a value A[j] into the B array. Decrementing C[A[j]] causes the next input element with a value equal to A[j], if one exists, to go to the position immediately before A[j] in the output array.

Therefore, we see that the correct position of A[j] in the output array is independent on the direction of j in the loop in line 10, which guarantees the correctness of the modified algorithm. When we increase j, the next element with a value equal to a previous A-front (labelled A-back) will still be right before the location of A-front in the output array due to decremented index. However, in the input array, A-front is before A-back. The reversed order makes the algorithm to be unstable.

3. Implement in C/C++ an algorithm that checks if an array of n elements is a heap and determine its running time. The algorithm should print YES, heap or Not a heap, depending on the outcome. Show how your algorithm works on the following arrays A = [16 14 10 8 7 9 3 2 4 1] and B = [10 3 9 7 2 11 5 1 6].

   Source code:

```
#include <stdio.h>

void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
}

void checkHeap(int arr[], int n)
{
    int parent = 0;
    int left_child = 0;
    int right_child = 0;

    bool isHeap = true;

    for (int i = 0; i < n/2; i++)
    {
        parent = arr[i];
        left_child = arr[2 * i];
        right_child = arr[2 * i + 1];

        if ((parent < left_child) || (parent < right_child))
        {
            isHeap = false;
            break;
        }
    }

    printArray(arr, n);
```

```
    if (isHeap)
    {
        printf("YES, heap\n");
    }
    else
    {
        printf("Not a heap\n");
    }

}


int main()
{
    int arr1[] = { 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 };
    int arr2[] = {10, 3, 9, 7, 2, 11, 5, 1, 6};
    checkHeap(arr1, sizeof(arr1) / sizeof(arr1[0]));
    checkHeap(arr2, sizeof(arr2) / sizeof(arr2[0]));
    return 0;
}
```

Outputs:

```
16 14 10 8 7 9 3 2 4 1 YES, heap
10 3 9 7 2 11 5 1 6 Not a heap
```

4. Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor$A.length/2$\rfloor$ to 1 rather than increase from 1 to $\lfloor$A.length/2$\rfloor$.

   BUILD-MAX-HEAP (A)

   A.*heap-size* = A.*length*

   for $i = \lfloor$A.*length/s*$\rfloor$ **downto** 1

   MAX-HEAPIFY(A, i)

   Because it is required for MAX-HEAPIFY to work properly. When MAX-HEAPIFY is called, it assumes that the binary trees rooted at LEFT(i and RIGHT(i) are max-heaps. When we go bottom up, we have the loop invariant that at the start of each iteration of the for loop, each node $i + 1$, $i + 2$, ..., n is the root of a max-heap. This allows MAX-HEAPIFY to work correctly when it is called. However, when we go in the opposite direction, the loop invariant does not hold anymore making the assumption invalid.

5. Describe an algorithm that, given n integers in range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range [a..b] in O(1) time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

   For n given integers in range 0 to k, we preprocess them similarly when using COUNTING-SORT to have the array C[0..k] so that C[i] holds the number of input elements equal to i for each integer i = 0, 1, ..., k. In this pre-processing step, to get C, we need Θ(n + k) time. After we have the array C, there are different cases to answer the query, each takes a constant time:

   - b $\leq$ a: Invalid case

- a < b:
  - a < b ≤ 0: Return 0
  - a ≤ 0 < b ≤ k: Return C[b]
  - a ≤ 0 < k ≤ b: Return n
  - 0 < a < b ≤ k: Return C[b] - C[a-1]
  - 0 < a < k ≤ b: Return C[k] - C[a-1]

6. (a) Find the smallest and the largest number of keys that a heap height h can have.
   From the definition of heap, Figure 3 shows when the heap has minimum and maximum number of keys.
   Minimum number of keys: When the heap is a complete binary tree till height (h-1) and has only one node at level h.

$$m = 2^0 + 2^1 + ... + 2^{h-1} + 1 = 2^h$$

   Maximum number of keys: When the heap is a complete binary tree of height h.

$$M = 2^0 + 2^1 + ... + 2^h = 2^{h+1} - 1$$

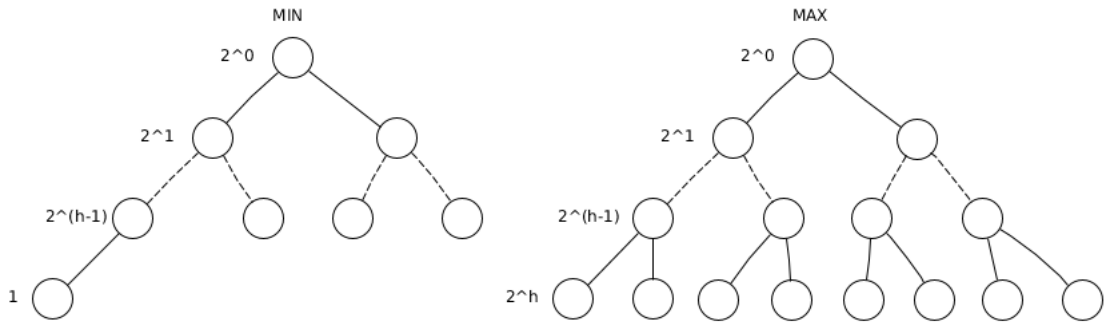

Figure 3: Cases with minimum and maximum number of keys.

   (b) Prove that the height of a heap with n nodes is $\lfloor \lg n \rfloor$.
   From (a) we have:

$$\begin{aligned} m \le n \le M \\ 2^h \le n \le 2^{h+1} - 1 \\ 2^h \le n < 2^{h+1} \\ h \le lgn \le h + 1 \\ \rightarrow h = \lfloor lgn \rfloor \end{aligned}$$