CSC4120 Spring 2024 - Written Homework 3

Yohandi 120040025
Andrew Nathanael 120040007

February 28, 2024

Heaps

Problem 1.

Answer the questions below. Indicate the reason.

(a) The array $[50, 45, 46, 28, 29, 37, 41, 17, 16, 13]$ forms a heap. [T/F]

(b) Consider the heap created from the array $[20, 15, 13, 8, 3, 10, 9, 4, 2, 1]$. If the node with value 3 has its value increased to 15, then 1 swap must occur to convert the heap into a max heap. [T/F]

(c) We have a max-heap of n elements. The minimum total time taken if we want to insert m more elements to this heap is $O(m+n)$. Assume that the end result must also be a max heap. [T/F]

(d) The smallest element in a max-heap can be found in time: (a) $O(\log n)$, (b) $O(\log \log n)$, (c) $O(n)$, (d) $O(n^2)$.

(e) Consider the heap that is represented by the array $A = [5, 1, 4, 2, 18, 9, 10, 15, 8, 7]$. Which of the following is true?

(i) It satisfies the max-heap property. [T/F]

(ii) It does not satisfy the max-heap property at node 5. [T/F]

(iii) It does not satisfy the max-heap property at node 1. [T/F]

(iv) It does not satisfy the max-heap property at node 2. [T/F]

(v) After we apply build-max-heap we get the heap $A' = [10, 8, 7, 9, 18, 15, 4, 2, 5, 1]$. [T/F]

(vi) After we apply build-max-heap we get the heap $A' = [18, 15, 10, 8, 7, 9, 4, 2, 5, 1]$. [T/F]

(vii) We run heap-sort on $A' = [18, 15, 10, 8, 7, 9, 4, 2, 5, 1]$. Then just after the first invocation, the unsorted array is $[18, 8, 10, 5, 7, 9, 4, 2, 1]$ and the sorted part is $[15]$. [T/F]

(viii) We run heap-sort on $A' = [18, 15, 10, 8, 7, 9, 4, 2, 5, 1]$. Then just after the first invocation, the unsorted array is $[15, 8, 10, 5, 7, 9, 4, 2, 1]$ and the sorted part is $[18]$. [T/F]

(ix) If we call max-heap-insert$[A, 6]$ after we build the max-heap from A, it will result in the new array $[18, 15, 10, 8, 7, 9, 4, 2, 5, 1, 6]$. [T/F]

(x) If we call max-heap-insert$[A, 6]$ after we build the max-heap from A, it will result in the new array $[10, 8, 7, 9, 18, 15, 4, 2, 5, 1, 6]$. [T/F]

(a) The following visualize the given array as a binary tree, and it is noticed that the property of the (max) heap is preserved; that is, the key of any node is larger than or equal to the keys of its children. Hence, the statement is true.

```
              50
          /        \
      45              46
    /    \          /    \
  28       29     37       41
 /  \    /
17  16  13
```

(b) The following visualize the given array as a binary tree.

```
             20
         /        \
      15              13
    /      \      /      \
  8          3  10         9
 / \       /
4   2     1
```

When the node with value 3 has its value increased to 15, the property of the (max) heap is still preserved. Any swap is not necessary; hence, the statement is false.

(c) The statement is false. To maintain the end result to be a max heap, a heapify procedure will be always called in every insertions. This implies that each insertion will require $\mathcal{O}(\log(m+n))$, making the total time taken to insert m more elements to be $\mathcal{O}(m\log(m+n))$.

(d) As max-heap generally does not support popping the smallest element, it is required to loop over half of the array (leaves in the binary tree). The loop iterates about $n/2$ elements; hence, it takes $\mathcal{O}(n)$ to find the smallest element in a max-heap.

(e) The following visualize the given array as a binary tree.

```
             5
         /       \
       1            4
    /     \     /      \
  2        18  9         10
 / \      /
15   8    7
```

(i) The binary tree does not hold the property of max-heap; that is, the key of any node is not larger than and not equal to the keys of its children. An example can be taken for the node with value 1. Hence, the statement is false.

(ii) As 18 is larger than or equal to 7, node 5 satisfies the max-heap property; hence, the statement is false.

(iii) As 5 is larger than or equal to 1 and 4, node 1 satisfies the max-heap property; hence, the statement is false.

(iv) As 1 is not larger than and not equal to 2 nor 18, node 2 does not satisfy the max-heap property; hence, the statement is true.

(v) As build-max-heap converts an array to a max heap, the result must be a max heap. The following visualize the upper part of A' as a binary tree.

```
          10
        /    \
      8        7
     / \      ...
    9    18
   ...   ...
```

As 8 is not larger than and not equal to 18, node 8 does not satisfy the max-heap property; hence, the statement is false.

(vi) After running the build-max-heap algorithm, the states of the array are as followings:

```
5 1 4 2 18 9 10 15 8 7
5 1 4 15 18 9 10 2 8 7
5 1 10 15 18 9 4 2 8 7
5 18 10 15 7 9 4 2 8 1
18 15 10 8 7 9 4 2 5 1
```

Hence, the statement is true. The code that is used to run the algorithm is attached in the appendix section.

(vii) As heap-sort algorithm's first step is to look up for the maximum element in the heap to be popped out. The sorted part must contain that maximum element. Being said, it should contain [18] instead of [15]. Hence, the statement is false.

As a further convincement, running the invoke algorithm obtains the followings:

```
15 8 10 5 7 9 4 2 1
18
```

The code that is used to run the algorithm is attached in the appendix section.

(viii) As convinced in part (vii), the statement is true.

(ix) The following visualize A after build-max-heap procedure is called.

```
           18
       /        \
     15          10
   /    \      /    \
  8      7    9      4
 / \    / \
2   5  1   [6]
```

The max-heap-insert procedure will insert an element to the last position and compare the current element with its parent (all the way up until root) to check whether the max-heap property is satisfied. As shown in the visualization, the current one already satisfies the max-heap property. Therefore, the result of the new array is [18, 15, 10, 8, 7, 9, 4, 2, 5, 1, 6].

Hence, the statement is true.

(x) Contrary to the part (ix), the statement is false.

To summarize up, the followings are true: (iv), (vi), (viii), and (ix).

Problem 2.

Present an $O(k \log k)$ time algorithm to return the kth minimum element of a min-heap $\mathcal{H}$ of size n (i.e., $\mathcal{H}$ is given to you), where $1 \leq k \leq n$.

In the heap, notice that the k-th minimum element must be the child of one of the previous $k-1$ elements, $\forall k$. With that, we can perform Best First Search algorithm that stores every elements starting from the minimum one and maintain a heap consisting value and index elements to allow traversal possibilities. The algorithm below can be presented:

Step 1: Initialize a min-heap Q and insert the root of $\mathcal{H}$ into Q.

Step 2: For $j = 1$ to k do

a. Get the minimum element from Q, let this element be e, and its index in $\mathcal{H}$ be i.

b. If the left child of e exists (i.e., $2i + 1 < n$), insert the left child into Q.

c. If the right child of e exists (i.e., $2i + 2 < n$), insert the right child into Q.

Step 3: The element e in the k-th iteration is the k-th minimum element. Return e.

Since there are k iterations and each iterations takes $\mathcal{O}(\log k)$ for insertions, the overall time complexity for the algorithm is $\mathcal{O}(k \log k)$.

Problem 3.

You are given m max-heaps, each containing n elements (think of them as data collected from m different experiments and kept separately). Can you find the l largest elements among the set S of all the $m \times n$ elements of the above 'local' heaps in $O(m + l \log l)$ time? Assume that $l < n$.

Discussion: Suppose you build a max-heap $\mathcal{H}$ containing the numbers at the root of every local heap. This takes $O(m)$ time. Then you (i) extract the root of $\mathcal{H}$ and post it as the largest element in S, (ii) you extract it also from the corresponding local heap H_u, to which it was the root, and (iii) you add to $\mathcal{H}$ the new root of H_u. It is clear that second-largest element of S is contained in $\mathcal{H}$, and hence you can repeat the same procedure: post the root of $\mathcal{H}$ as the second-largest element of S, extract it from both $\mathcal{H}$ and the local heap in which it was the root, etc. What is the complexity of this algorithm? Can you improve it by creating a max-heap where you insert at each step only the minimum number of relevant elements from the local heaps in which we are guaranteed to find the next maximum value? For example, if we like to find the first-largest element of S, it is the root of $\mathcal{H}$. If we like to find the second-largest element, then we need to consider the maximum among its children in $\mathcal{H}$ (2 elements) and its children in the corresponding local heap (also 2 elements), a total of 4 elements. How about if we like the find the third-largest element of S? which is now the minimal relevant set of elements? If we keep them in a max-heap, then this might do the job!

Yes. Consider the following algorithm:

Step 1: Initialize an empty list L and a max-heap $\mathcal{H}$ and insert the root of $\mathcal{H}_k$ into $\mathcal{H}$, $\forall k$.

Step 2: For $j = 1$ to l do

a. Get the maximum element from $\mathcal{H}$, let this element be e, and its index in $\mathcal{H}_k$ be i.

b. Store e to L and remove e from $\mathcal{H}$.

c. Identify e's successor elements in $\mathcal{H}_k$.

   – If e is an internal node, its direct children are the immediate successors; hence, insert into $\mathcal{H}$.

   – If e is an external node, continue.

Step 3: Return L.

Step 1 requires $\mathcal{O}(m)$ to build up the heap. Each iteration in step 2 takes $\mathcal{O}(\log(m + l))$ as the number of elements in $\mathcal{H}$ is initially m and has at most 1 additional element from e's successor elements in $\mathcal{H}_k$; being implied, $m+l$ elements at most. The overall time complexity would be $\mathcal{O}(m + l \log(m + l))$. However, consider that:

- If $m \gg l$, the overall time complexity would be $\mathcal{O}(m)$ due to m outgrowing $\log m$.
- If $m \ll l$, the overall time complexity would be $\mathcal{O}(l \log l)$ due to l being much larger than m.

- Otherwise, given that the logarithmic cost is tied to the number of elements being managed for the purpose of finding the l largest elements, and considering that after each extraction we may insert at most one new element into the heap (the successor), the average size of the heap during the critical operations is more closely aligned with l than $m + l$.

With that, we may conclude a tighter bound, that is the algorithm runs in $\mathcal{O}(m+l\log l)$.

Lower bounds

Problem 4.

Answer the questions below. Indicate the reason.

(a) So far we have used the decision tree method to compute lower bounds for sorting problems. Can we use the same method to show to obtain a lower bound for the 1-D peak finding problem (T/F)? The lower bound we obtain is (a) $\log(\log n)$ (T/F), (b) $\log n$ (T/F).

(b) There is a comparison sort whose running time is linear if we restrict our input space to be half of the $n!$ possible inputs of length n? (T/F)

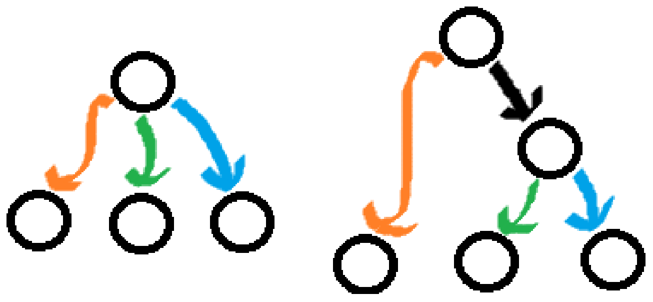
(c) There is a comparison sort whose running time is linear if we restrict our input space to be $1/n$ of the $n!$ possible inputs of length n? (T/F)

(d) There is a comparison sort whose running time is linear if we restrict our input space to be $1/2^n$ of the $n!$ possible inputs of length n? (T/F)

(a) It is true that the decision tree method can be used to obtain a lower bound for the 1D peak finding problem. Instead of having two edges pointing to its children, each node will have three edges with the following conditions:

- The current element is smaller than the left-neighbor element.
- The current element is smaller than the right-neighbor element.
- The current element is a peak.

Though it is possible to reform it to satisfy the binary tree condition illustrated as following:



The obtained lower bound is $\log n$ as the height of the tree will be about $\log_2(n)$ due to the number of nodes is halved (minus one) in each depth. Therefore, the statement (a) is false and the statement (b) is true.

(b) Similar derivation in the lecture can be made as following:

$$\begin{aligned} h &\geq \log(n!/2) \\ &\geq \log(n!) - \log(2) \\ &\geq \log(\sqrt{2\pi n}(\frac{n}{e})^n) - \log(2) \text{ (by Stirling's Approximation)} \\ &\geq \log(\sqrt{2\pi n}) + n\log(n) - n\log(e) - \log(2) \\ &= \Omega(n \log n) \end{aligned}$$

The statement is false as there does not exist a comparison sorting algorithm whose running time is linear even when the space is cut half.

(c) Similar derivation in the lecture can be made as following:

$$\begin{aligned} h &\geq \log(n!/n) \\ &\geq \log((n-1)!) \\ &\geq \log(\sqrt{2\pi(n-1)}(\frac{(n-1)}{e})^{n-1}) \text{ (by Stirling's Approximation)} \\ &\geq \log(\sqrt{2\pi(n-1)}) + (n-1)\log(n-1) - (n-1)\log(e) \\ &= \Omega(n \log n) \end{aligned}$$

The statement is false as there does not exist a comparison sorting algorithm whose running time is linear even when the space is $1/n$ of the original space.

(d) Similar derivation in the lecture can be made as following:

$$\begin{aligned} h &\geq \log(n!/2^n) \\ &\geq \log(n!) - \log(2^n) \\ &\geq \log(\sqrt{2\pi n}(\frac{n}{e})^n) - \log(2^n) \text{ (by Stirling's Approximation)} \\ &\geq \log(\sqrt{2\pi n})) + n\log(n) - n\log(e) - n\log(2) \\ &= \Omega(n \log n) \end{aligned}$$

The statement is false as there does not exist a comparison sorting algorithm whose running time is linear even when the space is $1/2^n$ of the original space.

Appendix

Problem 1e(vi)'s C++ Code

```
/*
    Author: Yohandi, Andrew Nathanael
*/

#include <bits/stdc++.h>
using namespace std;

int main() {
  function<void(vector<int>)> print = [&](vector<int> A) {
    for (auto element : A)
      cerr << element << " ";
    cerr << endl;

    return;
  };

  function<void(vector<int> &, int)> max_heapify = [&](vector<int> &A, int i) {
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    int largest;
    if (l <= (int)A.size() - 1 && A[l] > A[i]) {
      largest = l;
    } else {
      largest = i;
    }
    if (r <= (int)A.size() - 1 && A[r] > A[largest]) {
      largest = r;
    }

    if (largest != i) {
      swap(A[i], A[largest]);
      max_heapify(A, largest);
    }

    return;
  };

  function<void(vector<int> &)> build_max_heap = [&](vector<int> &A) {
    int sz = (int)A.size();
    for (int i = sz / 2 - 1; i >= 0; --i) {
      max_heapify(A, i);
      print(A);
    }

    return;
  };

  vector<int> A = {5, 1, 4, 2, 18, 9, 10, 15, 8, 7};

  build_max_heap(A);

  return 0;
}
```

Problem 1e(vi)'s Console Output:

```
5 1 4 2 18 9 10 15 8 7
5 1 4 15 18 9 10 2 8 7
5 1 10 15 18 9 4 2 8 7
5 18 10 15 7 9 4 2 8 1
18 15 10 8 7 9 4 2 5 1
```

Problem 1e(vii)'s C++ Code

```
/*
    Author: Yohandi, Andrew Nathanael
*/

#include <bits/stdc++.h>
using namespace std;

int main() {
  function<void(vector<int>)> print = [&](vector<int> A) {
    for (auto element : A)
      cerr << element << " ";
    cerr << endl;

    return;
  };

  function<void(vector<int> &, int)> max_heapify = [&](vector<int> &A, int i) {
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    int largest;
    if (l <= (int)A.size() - 1 && A[l] > A[i]) {
      largest = l;
    } else {
      largest = i;
    }
    if (r <= (int)A.size() - 1 && A[r] > A[largest]) {
      largest = r;
    }

    if (largest != i) {
      swap(A[i], A[largest]);
      max_heapify(A, largest);
    }

    return;
  };

  function<void(vector<int> &, vector<int> &)> invoke =
      [&](vector<int> &A, vector<int> &result) {
        swap(A[0], A.back());
        result.push_back(A.back());
        A.pop_back();
        max_heapify(A, 0);
      };

  vector<int> A = {18, 15, 10, 8, 7, 9, 4, 2, 5, 1};
  vector<int> result;

  invoke(A, result);

  print(A);
  print(result);

  return 0;
}
```

Problem 1e(vii)'s Console Output:

```
15 8 10 5 7 9 4 2 1
18
```