Assignment 3

Haohu Shen UCID: 30063099

CPSC 331 - Data Structures, Algorithms, and Their Analysis

Jun 15, 2019

Question 1

NOTE The file *ArrayUtils.java* has been uploaded to D2L dropbox.

Question 2

CITATION

Please note before grading, in order to providing a completed, clear and professional proof, solution or implementation of an algorithm,

- The implementation of the new ***bubbleUp*** refers to the contents from page 28 to page 34 in L17_binary_heaps.pdf[1].
- The implementation of the new ***bubbleDown*** refers to the contents from page 43 to page 51 in L17_binary_heaps.pdf[1].
- The format of the proof of a loop invariant acts in a similar manner to the proof provided from page 9, 10, 14, 15 in L03_iterative_correctness.pdf[4] .
- The format of the proof of a **bound function** for a *while* loop refers to the proof provided from page 38 to page 39 in L03_iterative_correctness.pdf[4].
- The format of the proof of the partial correctness of an algorithm refers to the proof provided from page 29 to page 33 in L03_iterative_correctness.pdf[4].
- The format of the proof of the termination of an algorithm refers to the proof provided from page 42 to page 43 in L03_iterative_correctness.pdf[4].
- The format of the answer for the upperbound of steps taken by the algorithm in the worst case refers to the contents from page 7 to page 10 in L05_efficiency.pdf[5].
- The format of the proof of the upperbound using the definition of Big-O notation refers to the proof provided in page 9 in L06_asymptotic_notation.pdf[6].

Solution I selected the **heapsort** base on the implementation given in slides of Lecture #17[1] and Lecture #18[2], then turned the original recursive function ***bubbleUp*** and the original recursive function ***bubbleDown***

from the slides into iterative ones by elimination of tail calls.

The reason why I chose this algorithm is that, we can prove that the new algorithm of ***bubbleUp*** still correctly solves the **Maxheap Restoration After Insertion** computational problem and the new algorithm of ***bubbleDown*** still correctly solves the **Maxheap Restoration After Deletion** computational problem without changing their upperbound of running time in the worst case but they reduce the amount of additional storage into a constant number by elimination of recursive calls. Since we do not change any precondition or postcondition of any computational problem involved, the modified algorithm of **heapsort** can still be proved to correctly solve the **Sorting in Place** described in the slide of Lecture #15[3] and it still uses $O(n \log n)$ operations in the worst case where n is the length of the array being sorted, but now it only uses a **constant** amount of additional storage since the algorithm does not contain any recursive calls inside.

Thus in this report, we firstly try to prove the new ***bubbleUp*** correctly solves the **Maxheap Restoration After Insertion** computational problem using $O(\log n)$ steps in the worst case where n is the length of the array, and using a **constant** amount of additional storage.

- The pseudocode of the new ***bubbleUp*** and the utility methods ***isRoot(x)*** and ***parent(x)*** are listed below, by inspection we can see an execution of ***isRoot(x)*** or ***parent(x)*** each takes 2 steps in the worst case of the running time when the **Uniform Cost Criterion** is defined and used. Notice the precondition and the postcondition of the problem are not changed, thus node x is in a binary tree H.

```
        boolean isRoot(int x) {
   1        if (x == 0) {
   2            return true
            } else {
   3            return false
            }
        }

        int parent(int x) {
   4        if (x == 0) {
   5            throw a NoSuchElementException
            } else {
   6            return floor((x - 1) / 2)
            }
        }

        void bubbleUp(int x) {
   7        while (!isRoot(x)) {
   8            if (A[x] > A[parent(x)]) {
   9                T temp = A[x]
  10                A[x] = A[parent(x)]
  11                A[parent(x)] = temp
  12                x = parent(x)
                } else {
  13                break
                }
            }
        }
```

- We prove the following assertions is a loop invariant of the *while* loop

  1. A is an array with length $A.length \geq 1$, storing values of some ordered type T and is given as input.
  2. x is a node in the binary tree H, which represents as an index integer in A.
  3. The multiset S represented by A is unchanged.

  Proof We try to show all assertions list above satisfy all 3 properties included in the **Loop Theorem #1**

  - By inspection we can see the loop test at line 7 has no side-effects since it is a simple boolean test without changing the value of x.
  - Since there are no statements before the *while* loop at line 7 after the algorithm is executed, the assertions are satisfied when the *while* loop is reached during an execution of the algorithm with the precondition being satisfied.

    - If all assertions are satisfied at the beginning of any execution of the *while* loop body during an execution of the algorithm with the precondition being satisfied, since the steps from line 8 to line 12 do not change the length of A and the entries in A have been exchanged or are not changed otherwise, all assertions are still satisfied when the execution of the loop body ends.

  Thus, we can conclude that the loop invariant for the *while* loop is correct by the **Loop Theorem #1**.

- We prove the partial correctness of the algorithm ***bubbleUp***.

  Proof Consider an execution of the algorithm with the precondition of the **Maxheap Restoration After Insertion** computational problem being satisfied, thus x is an input node in a binary tree H that represents an integer of index in A. Notice that A is documented in the precondition and is accessed and modified as global data. We prove the partial correctness of the algorithm by showing that

  (1) The execution of the algorithm ends eventually with the postcondition being satisfied and no undocumented inputs or global data accessed or modified.

  or

  (2) The execution of the algorithm never ends.

    - If x is the root of H, then the loop test at line 7 is checked and failed, thus the algorithm terminates after that. Since x is the root of H, the position of node x does not need to be adjusted and H is a Maxheap already, all nodes in H are not changed as well, thus the postcondition is satisfied. Also, by inspection, we can see no undocumented data are accessed or modified.

    - If x is not the root of H, then the loop test at line 7 is checked and passed,

        - If the value stored at x is equal to or smaller than the value stored at the parent of x, then the test at line 8 is checked and failed, thus the execution continues at step 13 and terminates after that. Since the value stored at the parent of x is equal to or greater than the value stored at x in this case, H is a Maxheap by the definition and all nodes in H are not changed, thus the postcondition is satisfied. Also, by inspection, we can see no undocumented data are accessed or modified.

        - If the value stored at x is greater than the value stored at the parent of x,

            - If the execution of the algorithm never ends, then it is sufficient to establish the partial correctness of the algorithm.

            - If the execution of the algorithm ends, then it follows the loop invariant which is proved to be true above after the execution of the loop, thus x is still a node in H, since it must be true that the loop test at line 7 was checked and failed, thus x is the root of H and H is a

Maxheap in this case. Since all nodes in H are either not changed or exchanged, the multiset S represented by H is not changed, thus the postcondition is satisfied. Also, by inspection, we can see no undocumented data are accessed or modified.

- Since all cases of x that satisfied the problem's precondition discussed above satisfies part **(1)** or part **(2)** in the definition of **partial correctness**, we can conclude that the algorithm is partially correct.

- We prove $f(n) = n$ where n is the level of the node x is a **bound function** for the *while* loop in the algorithm ***bubbleUp***.

  Proof We show that the function $f(n) = n$ where n is the level of the node x satisfies all 3 properties included in the definition of a **bound function** for a *while* loop.

  - By inspection, we can see that n is an integer, thus $f(n) = n$ is an integer-valued function of n.
  - Since the level of the node x is decreased by 1 at step 12 after the execution of the loop body of *while*, the value of $f(n) = n$ is decreased by at least 1.
  - If the value of $f(n) = n \leq 0$, since x is a node in H from the precondition, $n \geq 0$, thus $n = 0$, that is, x is the root of H, in this case the loop test at step 7 is checked and failed.

  Therefore, since all properties of a definition of a **bound function** for a *while* loop are satisfied, $f(n) = n$ is a bound function for the *while* loop in this algorithm where n is the level of the node x.

- We prove the termination of the algorithm ***bubbleUp***.

  Proof We prove the termination by showing all properties included in the **Loop Theorem #2** are satisfied. Consider an execution of the algorithm with the precondition of the **Maxheap Restoration After Insertion** computational problem being satisfied.

  - By inspection we can see the loop test at line 7 has no side-effects since it is a simple boolean test without changing the value of x, thus every execution of the loop test will halt.
  - Since there are no statements after the execution of the algorithm and before the loop test at line 7, the execution of the algorithm includes a beginning of the *while* loop, thus
    - If x is the root of H, then the test at line 7 is checked and failed, and the algorithm terminates after it.
    - If x is not the root of H, then the test at line 7 is checked and passed and the execution continues at line 8,
      - If the value stored at x is equal to or smaller than the value stored at the parent of x, then

the test at line 8 is checked and failed, thus the execution of the loop body that continues at step 13 will terminate after that.

      - If the value stored at x is greater than the value stored at the parent of x, then the test at line 8 is checked and passed, since the steps from line 9 to line 12 are all simple statements of assignments, the execution of the loop body certainly terminates after line 12.

    - Since we have proved that $f(n) = n$ is a bound function for the *while* loop in the algorithm where n is the level of the node x above, the *while* loop has a bound function.

  Therefore, by **Loop Theorem #2**, we can conclude that if the algorithm is executed when the precondition for the **Maxheap Restoration After Insertion** problem is satisfied, and the *while* loop is reached and executed, then the execution of the loop eventually ends.

- Since the algorithm ***bubbleUp*** is both partially correct and terminates, we can conclude that this algorithm is correct for solving the **Maxheap Restoration After Insertion** problem.

- We show that the running time for the algorithm ***bubbleUp*** is $O(\log n)$ in the worst case where n is the length of A.

  Proof Suppose an execution of the algorithm with the precondition of the **Maxheap Restoration After Insertion** computational problem being satisfied, thus x is an input node in a binary tree H that represents an integer of index in A. Suppose the node of x has level $k \geq 0$ and $T_{bubbleUp}(k)$ is the number of steps executed by the algorithm in the worst case.

  - If $k = 0$, the level of x is zero and x is the root of H in this case, thus the loop test at line 7 is checked and failed, since the steps taken by **isRoot(x)** is at most 2, we have

$$T_{bubbleUp}(k) = T_{bubbleUp}(0) = 2 + 1 = 3 \leq 3$$

  - If $k \geq 1$, x is not the root of H, thus the loop test at line 7 is checked and passed, notice the steps taken by **parent(x)** is at most 2.

    - If the value stored at x is equal to or smaller than the value stored at the parent of x, then the test at line 8 is checked and failed, thus the execution of the loop body that continues at step 13 will terminate after that, since there are totally 8 steps including the steps taken in the subroutines,

$$T_{bubbleUp}(k) = 3 + 2 + 3 = 8 \leq 8$$

    - If the value stored at x is equal to or smaller than the value stored at the parent of x, then the test at line 8 is checked and passed, since $f(k) = k$ is a bound function for the *while* loop and its initial value is k, the loop body is executed at most k times. Since the loop body includes 13 steps in this case, and the loop test is at most 1 more step executed after that, the steps of the execution in this case is at most

$$T_{bubbleUp}(k) \leq \sum_{i=1}^{k}(5+8) + \sum_{i=1}^{k+1}(1+2) = 13k + 3(k+1) = 16k+3$$

Since when $k \geq 1$, $16k+3 \geq 16+3 = 19 \geq 8$, thus

$$T_{bubbleUp}(k) \leq \max(8, 16k+3) = 16k+3$$

Therefore, when the algorithm is executed with the precondition being satisfied, the upperbound of $T_{bubbleUp}(k)$ where k is the level of node x as input of the algorithm such that $k \geq 0$, can be written as

$$T_{bubbleUp}(k) \leq 16k+3$$

Suppose after the execution of the algorithm, h is the height of the Maxheap H such that $h \geq 0$, we also suppose n is the size of H, which is the length of A as well, then according to the relationship between h and n which is proven to be true in the slide of Lecture #17[1], we have

$$2^h \leq n \leq 2^{h+1} - 1$$

Thus

$$2^h \leq n \leq 2^{h+1}$$

$$\log_2 2^h \leq \log_2 n \leq \log_2 2^{h+1}$$

$$h \leq \log_2 n \leq h+1$$

Since $k \leq h$, we have

$$T_{bubbleUp}(k) \leq 16k+3 \leq 16h+3 \leq 16\log_2 n + 3$$

Now we show that $T_{bubbleUp}(k) \in O(\log_2 n)$ by the definition.

By the definition of $O(\log_2 n)$, it is sufficient to show that there exist constants $c > 0$ and $N_0 \geq 0$ such that $T_{bubbleUp}(k) \leq c\log_2 n$ for all $n \in \mathbb{Z}_{\geq 0}$ such that $n \geq N_0$.

Let $c = 17$ and $N_0 = 8$, let n be arbitrarily chosen from $\mathbb{Z}_{\geq 0}$ such that $n \geq N_0 = 8$, thus

$$\begin{aligned} T_{bubbleUp}(k) \leq 16\log_2 n + 3 &= 16\log_2 n + \log_2 8 \\ &= 16\log_2 n + \log_2 N_0 \\ &\leq 16\log_2 n + \log_2 n \\ &= 17\log_2 n \\ &= c\log_2 n \end{aligned}$$

Since n is arbitrarily chosen from $\mathbb{Z}_{\geq 0}$ such that $T_{bubbleUp}(k) \leq c \log_2 n$ for all $\mathbb{Z}_{\geq 0}$ such that $n \geq N_0 = 8$ and $c = 17 > 0$, $N_0 = 8 \geq 0$ are both constants, $T_{bubbleUp}(k) \in O(log_2 n)$ by the definition.

Since the constant base of logarithm is irrelevant to big-O classification of logarithmic time, we can conclude that

$$T_{bubbleUp}(k) \in O(\log n)$$

as required.

- By inspection of the pseudocode of the algorithm ***bubbleUp***, we can see that it only use a **constant** amount of additional storage, including the size of call stacks for ***parent(x)***, ***isRoot(x)*** and the temporary variable ***temp*** inside the *while* loop.

- Therefore, we can conclude that, the new ***bubbleUp*** correctly solves the **Maxheap Restoration After Insertion** computational problem using $O(\log n)$ steps in the worst case where n is the length of the array, and using a **constant** amount of additional storage.

Secondly, we try to prove the new ***bubbleDown*** correctly solves the **Maxheap Restoration After Deletion** computational problem using $O(\log n)$ steps in the worst case where n is the length of the array, and using a **constant** amount of additional storage.

- The pseudocode of the new ***bubbleDown***, the utility methods ***hasLeft(x)***, ***left(x)***, ***hasRight(x)*** and ***right(x)*** are listed below, by inspection we can see that, when the **Uniform Cost Criterion** is defined and used, an execution of ***hasLeft(x)*** or ***hasRight(x)*** each takes 2 steps in the worst case of the running time, thus an execution of ***left(x)*** or ***right(x)*** each takes 4 steps in the worst case of the running time. Notice that ***heapSize*** is the size of the heap represented using the array A as a documented global data. Also note the precondition and the postcondition of the problem are not changed, thus node x is in a binary tree H.

```
       boolean hasLeft(int x) {
    1      if (2 * x + 1 < heapSize) {
    2          return true
           } else {
    3          return false
           }
       }

       int left(int x) {
    4      if (hasLeft(x)) {
    5          return 2 * x + 1
           } else {
```

```
   6            throw a NoSuchElementException
            }
        }

        boolean hasRight(int x) {
   7        if (2 * x + 2 < heapSize) {
   8            return true
            } else {
   9            return false
            }
        }

        int right(int x) {
  10        if (hasRight(x)) {
  11            return 2 * x + 2
            } else {
  12            throw a NoSuchElementException
            }
        }

        void bubbleDown(int x) {
  13        while (hasLeft(x)) {
  14            if (hasRight(x)) {
  15                if (A[left(x)] >= A[right(x)]) {
  16                    if (A[left(x)] > A[x]) {
  17                        T temp = A[left(x)]
  18                        A[left(x)] = A[x]
  19                        A[x] = temp
  20                        x = left(x)
                        } else {
  21                        break
                        }
  22                } else if (A[right(x)] > A[x]) {
  23                    T temp = A[right(x)]
  24                    A[right(x)] = A[x]
  25                    A[x] = temp
  26                    x = right(x)
                    } else {
  27                    break
                    }
                } else {
  28                if (A[left(x)] > A[x]) {
  29                    T temp = A[left(x)]
  30                    A[left(x)] = A[x]
  31                    A[x] = temp
```

```
32                  x = left(x)
                } else {
33                  break
                }
            }
        }
    }
```

- We prove the following assertions is a loop invariant of the *while* loop

  1. A is an array with length $A.length \geq 1$, storing values of some ordered type T and is given as input.
  2. x is a node in the binary tree H, which represents as an index integer in A.
  3. The multiset S represented by A is unchanged.

  Proof We try to show all assertions list above satisfy all 3 properties included in the **Loop Theorem #1**

  - By inspection we can see the loop test at line 13 has no side-effects since it is a simple boolean test without changing the value of x.
  - Since there are no statements before the *while* loop at line 13 after the algorithm is executed, the assertions are satisfied when the *while* loop is reached during an execution of the algorithm with the precondition being satisfied.
  - If all assertions are satisfied at the beginning of any execution of the *while* loop body during an execution of the algorithm with the precondition being satisfied, since the steps from line 14 to line 33 do not change the length of A and the entries in A have been exchanged or are not changed otherwise, all assertions are still satisfied when the execution of the loop body ends.

  Thus, we can conclude that the loop invariant for the *while* loop is correct by the **Loop Theorem #1**.

- We prove the partial correctness of the algorithm ***bubbleDown***.

  Proof Consider an execution of the algorithm with the precondition of the **Maxheap Restoration After Deletion** computational problem being satisfied, thus x is an input node in a binary tree H that represents an integer of index in A. Notice that A and ***heapSize*** are documented in the precondition and is accessed and modified as global data. We prove the partial correctness of the algorithm by showing that

  (1) The execution of the algorithm ends eventually with the postcondition being satisfied and no undocumented inputs or global data accessed or modified.

  $$or$$

  (2) The execution of the algorithm never ends.

- If x is a leaf of H, then it has no children, thus the loop test at line 13 is checked and failed and the algorithm terminates after that. Since x is a leaf of H, the position of node x does not need to be adjusted and H is already a Maxheap, all nodes in H are not changed as well, thus the postcondition is satisfied. By inspection, we can see no undocumented data are accessed or modified.

- If x is not a leaf of H, then the loop test at line 13 is checked and passed, and the *while* loop is reached and executed.

  - If the execution of the algorithm never ends, then it is sufficient to establish the partial correctness of the algorithm.

  - If the execution of the algorithm ends, then it follows the loop invariant which is proved to be true above after the execution of the loop, thus x is still a node in the heap, since it must be true that the loop test at line 13 was checked and failed, thus x is a leaf of H, and H is a Maxheap in this case and we do not need to change the position node x anymore. Since all nodes in H are either not changed or exchanged, the multiset S represented by H is not changed, thus the postcondition is satisfied. Also, by inspection, we can see no undocumented data are accessed or modified.

- Since all cases of x that satisfied the problem's precondition discussed above satisfies part **(1)** or part **(2)** in the definition of **partial correctness**, we can conclude that the algorithm is partially correct.

* We prove $f(n) = n$ where n is the height of the node x is a **bound function** for the *while* loop in the algorithm ***bubbleDown***.

  Proof We show that the function $f(n) = n$ where n is the height of the node x satisfies all 3 properties included in the definition of a **bound function** for a *while* loop.

  - By inspection, we can see that n is an integer, thus $f(n) = n$ is an integer-valued function of n.

  - Since the height of the node x is decreased by 1 at step 20, 26 or 32 after the execution of the loop body of *while*, the value of $f(n) = n$ is decreased by at least 1.

  - If the value of $f(n) = n \leq 0$, since x is a node in H from the precondition, $n \geq 0$, thus $n = 0$ in this case, that is, x has no children and the loop test at step 13 is checked and failed.

  Therefore, since all properties of a definition of a **bound function** for a *while* loop are satisfied, $f(n) = n$ is a bound function for the *while* loop in this algorithm where n is the height of the node x.

* We prove the termination of the algorithm ***bubbleDown***.

  Proof We prove the termination by showing all properties included in the **Loop Theorem #2** are satisfied. Consider an execution of the algorithm with the precondition of the **Maxheap Restoration After Deletion** computational problem being satisfied.

- By inspection we can see the loop test at line 13 has no side-effects since it is a simple boolean test without changing the value of x, thus every execution of the loop test will halt.
- Since there are no statements after the execution of the algorithm and before the loop test at line 13, the execution of the algorithm includes a beginning of the *while* loop, thus
    - If x is a leaf of H, then x has no children and the test at line 13 is checked and failed, and the algorithm terminates after it.
    - If x is not a leaf of H, then the test at line 13 is checked and passed, thus the execution continues at line 14,
        - If x has two children, then the test at line 14 is checked and passed, so the execution continues at line 15,
            - If the value stored at the left child of x is greater than or equal to the right child of x, then the test at line 15 is checked and passed, so the execution continues at line 16,
                - If the value stored at the left child of x is greater than the value stored at x as well, then the test at line 16 is checked and passed, so the execution continues at line 17, since the steps from line 17 to line 20 are all simple statements of assignments, the execution of the loop body certainly terminates after line 20.
                - Otherwise, the test at line 16 is checked and failed, thus the execution continues at line 21, where a 'break' statement will cause the termination of the *while* loop.
            - If the value stored at the left child of x is less than the value stored at the right child of x, then the test at line 15 is checked and failed, so the execution continues at line 22,
                - If the value stored at the right child of x is greater than the value stored at x, then the test at line 22 is checked and passed, so the execution continues at line 23, since the steps from line 23 to line 26 are all simple statements of assignments, the execution of the loop body certainly terminates after line 26.
                - Otherwise, the test at line 22 is checked and failed, thus the execution continues at line 27, where a 'break' statement will cause the termination of the *while* loop.
        - If x has only one child, then the test at line 14 is checked and failed, so the execution continues at line 28,

            - If the value stored at the left child of x is greater than the value stored at x, then the test at line 28 is checked and passed, so the execution continues at line 29, since the steps from line 29 to line 32 are all simple statements of assignments, the execution of the loop body certainly terminates after line 32.
            - Otherwise, the test at line 28 is checked and failed, thus the execution continues at line 33, where a 'break' statement will cause the termination of the *while* loop.

        Therefore, the execution of the loop body terminates in every branch inside the *while* loop if the problem's precondition is satisfied and the loop begins.

    - Since we have proved that $f(n) = n$ is a bound function for the *while* loop in the algorithm where n is the height of the node x above, the *while* loop has a bound function.

    Therefore, by **Loop Theorem #2**, we can conclude that if the algorithm is executed when the precondition for the **Maxheap Restoration After Deletion** problem is satisfied, and the *while* loop is reached and executed, then the execution of the loop eventually ends.

- Since the algorithm ***bubbleDown*** is both partially correct and terminates, we can conclude that this algorithm is correct for solving the **Maxheap Restoration After Deletion** problem.
- We show that the running time for the algorithm ***bubbleDown*** is $O(\log n)$ in the worst case where n is the length of A.

    Proof Suppose an execution of the algorithm with the precondition of the **Maxheap Restoration After Deletion** computational problem being satisfied, thus x is an input node in a binary tree H that represents an integer of index in A. Suppose the node of x has height $k \geq 0$ and $T_{bubbleDown}(k)$ is the number of steps executed by the algorithm in the worst case.

    - If $k = 0$, the height of x is zero, thus x is a leaf of H in this case and it has no children, thus the loop test at line 13 is checked and failed, since the steps taken by **hasLeft(x)** is at most 2, we have

    $$T_{bubbleDown}(k) = T_{bubbleDown}(0) = 2 + 1 = 3 \leq 3$$

    - If $k \geq 1$, x is not a leaf of H in this case, thus the loop test at line 13 is checked and passed,
        - If x is a leaf of H, then x has no children and the test at line 13 is checked and failed, and the algorithm terminates after it.
        - If x is not a leaf of H, then the test at line 13 is checked and passed, thus the execution continues at line 14,
            - If x has two children, then the test at line 14 is checked and passed, so the execution

continues at line 15,

- If the value stored at the left child of x is greater than or equal to the right child of x, then the test at line 15 is checked and passed, so the execution continues at line 16,

    - If the value stored at the left child of x is greater than the value stored at x as well, then the test at line 16 is checked and passed, so the execution continues at line 17, since the steps from line 17 to line 20 are all simple statements of assignments and the execution of the loop body certainly terminates after line 20, since $f(k) = k$ is a bound function for the *while* loop and its initial value is k, the loop body is executed at most k times and the loop test is at most 1 more step executed after that, thus in this case if the steps taken in the subroutines are included,

$$T_{bubbleDown}(k) \leq \sum_{i=1}^{k}(7+2+6\cdot 4) + \sum_{i=1}^{k+1}(1+2) = 33k + 3(k+1) = 36k+3$$

    - Otherwise, then the test at line 16 is checked and failed, thus the execution continues at line 21, where a 'break' statement will cause the termination of the *while* loop, there are totally 21 steps including the steps taken in the subroutines, thus in this case,

$$T_{bubbleDown}(k) = 5 + 2\cdot 2 + 4\cdot 3 = 21 \leq 21$$

- If the value stored at the left child of x is less than the value stored at the right child of x, then the test at line 15 is checked and failed, so the execution continues at line 22,

    - If the value stored at the right child of x is greater than the value stored at x, then the test at line 22 is checked and passed, so the execution continues at line 23, since the steps from line 23 to line 26 are all simple statements of assignments, the execution of the loop body terminates after line 26, since $f(k) = k$ is a bound function for the *while* loop and its initial value is k, the loop body is executed at most k times and the loop test is at most 1 more step executed after that, thus in this case if the steps taken in the subroutines are included,

$$T_{bubbleDown}(k) \leq \sum_{i=1}^{k}(7+2+6\cdot 4) + \sum_{i=1}^{k+1}(1+2) = 33k + 3(k+1) = 36k+3$$

    - Otherwise, the test at line 22 is checked and failed, thus the execution continues at line 27, where a 'break' statement will cause the termination of the *while* loop, there are totally 21 steps including the steps taken in the subroutines, thus in this case,

$$T_{bubbleDown}(k) = 5 + 2 \cdot 2 + 4 \cdot 3 = 21 \leq 21$$

- If x has only one child, then the test at line 14 is checked and failed, so the execution continues at line 28,
  - If the value stored at the left child of x is greater than the value stored at x, then the test at line 28 is checked and passed, so the execution continues at line 29, since the steps from line 29 to line 32 are all simple statements of assignments, the execution of the loop body terminates after line 32, since $f(k) = k$ is a bound function for the *while* loop and its initial value is k, the loop body is executed at most k times and the loop test is at most 1 more step executed after that, thus in this case if the steps taken in the subroutines are included,

$$T_{bubbleDown}(k) \leq \sum_{i=1}^{k}(6 + 2 + 4 \cdot 4) + \sum_{i=1}^{k+1}(1 + 2) = 24k + 3(k + 1) = 27k + 3$$

  - Otherwise, the test at line 28 is checked and failed, thus the execution continues at line 33, where a 'break' statement will cause the termination of the *while* loop, there are totally 12 steps including the steps taken in the subroutines, thus in this case,

$$T_{bubbleDown}(k) = 4 + 2 \cdot 2 + 4 \cdot 1 = 12 \leq 12$$

Since $k \geq 1$,

$$T_{bubbleDown}(k) \leq \max(36k + 3, 21, 27k + 3, 12) = 36k + 3$$

Therefore, when the algorithm is executed with the precondition being satisfied, the upperbound of $T_{bubbleDown}(k)$ where k is the height of node x as input of the algorithm such that $k \geq 0$, can be written as

$$T_{bubbleDown}(k) \leq 36k + 3$$

Suppose after the execution of the algorithm, h is the height of the Maxheap H such that $h \geq 0$, we also suppose n is the size of H, which is the length of A as well, then according to the relationship between h and n which is proven to be true in the slide of Lecture #17[1], we have

$$2^h \leq n \leq 2^{h+1} - 1$$

Thus

$$2^h \leq n \leq 2^{h+1}$$

$$\log_2 2^h \leq \log_2 n \leq \log_2 2^{h+1}$$

$$h \leq \log_2 n \leq h + 1$$

Since $k \leq h$, we have

$$T_{bubbleDown}(k) \leq 36k + 3 \leq 36h + 3 \leq 36 \log_2 n + 3$$

Now we show that $T_{bubbleDown}(k) \in O(\log_2 n)$ by the definition.

By the definition of $O(\log_2 n)$, it is sufficient to show that there exist constants $c > 0$ and $N_0 \geq 0$ such that $T_{bubbleDown}(k) \leq c \log_2 n$ for all $n \in \mathbb{Z}_{\geq 0}$ such that $n \geq N_0$.

Let $c = 37$ and $N_0 = 8$, let n be arbitrarily chosen from $\mathbb{Z}_{\geq 0}$ such that $n \geq N_0 = 8$, thus

$$\begin{aligned} T_{bubbleDown}(k) \leq 36 \log_2 n + 3 &= 36 \log_2 n + \log_2 8 \\ &= 36 \log_2 n + \log_2 N_0 \\ &\leq 36 \log_2 n + \log_2 n \\ &= 37 \log_2 n \\ &= c \log_2 n \end{aligned}$$

Since n is arbitrarily chosen from $\mathbb{Z}_{\geq 0}$ such that $T_{bubbleDown}(k) \leq c \log_2 n$ for all $\mathbb{Z}_{\geq 0}$ such that $n \geq N_0 = 8$ and $c = 37 > 0$, $N_0 = 8 \geq 0$ are both constants, $T_{bubbleDown}(k) \in O(log_2 n)$ by the definition.

Since the constant base of logarithm is irrelevant to big-O classification of logarithmic time, we can conclude that

$$T_{bubbleDown}(k) \in O(\log n)$$

as required.

  - By inspection of the pseudocode of the algorithm ***bubbleDown***, we can see that it only use a **constant** amount of additional storage, including the size of call stacks for ***hasLeft(x)***, ***left(x)***, ***hasRight(x)***, ***right(x)*** and the temporary variable ***temp*** inside the *while* loop.

- Therefore, we can conclude that, the new ***bubbleDown*** correctly solves the **Maxheap Restoration After Deletion** computational problem using $O(\log n)$ steps in the worst case where n is the length of the array, and using a **constant** amount of additional storage.

Conclusion Therefore, the modified ***bubbleUp*** and the modified ***bubbleDown*** are proved to be correct again without changing their upperbound of running time in the worst case respectively but both optimizing their cost of additional storage to a **constant** amount.

Since we know the call stacks in the old algorithms ***bubbleUp*** and ***bubbleDown*** require $O(\log n)$ auxiliary storage space for recursion and it is the **only** reason why the implementation of **heapsort** given in the slide of Lecture #18[2] cost $O(\log n)$ additional storage space, we can now conclude that, the new implementation of **heapsort** that uses the new implementations of **bubbleUp** and **bubbleDown** correctly solves the **Sorting in Place** computational problem, with $O(n \log n)$ operations in the worst case and a **constant** amount of additional storage.

References

[1] Wayne Eberly, 2019, CPSC 331: Data Structures, Algorithms, and Their Analysis: Spring, 2019, Searching and Sorting, Lecture #17: Binary Heaps. Retrieved from http://www.cpsc.ucalgary.ca/~eberly/Courses/CPSC331/2019a/6_Sorting/L17/L17_binary_heaps.pdf

[2] Wayne Eberly, 2019, CPSC 331: Data Structures, Algorithms, and Their Analysis: Spring, 2019, Searching and Sorting, Lecture #18: Applications of Binary Heaps. Retrieved from http://www.cpsc.ucalgary.ca/~eberly/Courses/CPSC331/2019a/6_Sorting/L18/L18_heap_applications.pdf

[3] Wayne Eberly, 2019, CPSC 331: Data Structures, Algorithms, and Their Analysis: Spring, 2019, Searching and Sorting, Lecture #15: Basic Algorithms for Searching and Sorting. Retrieved from http://www.cpsc.ucalgary.ca/~eberly/Courses/CPSC331/2019a/6_Sorting/L15/L15_basic.pdf

[4] Wayne Eberly, 2019, CPSC 331: Data Structures, Algorithms, and Their Analysis: Spring, 2019, Introduction to the Analysis of Algorithms, Lecture #3: Introduction to the Correctness of Algorithms II — Correctness of Simple Algorithms with a while Loop. Retrieved from http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC331/2019a/2_Analysis/L03/L03_iterative_correctness.pdf

[5] Wayne Eberly, 2019, CPSC 331: Data Structures, Algorithms, and Their Analysis: Spring, 2019, Introduction to the Analysis of Algorithms, Lecture #5: Analyzing the Running Times of Algorithms. Retrieved from http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC331/2019a/2_Analysis/L05/L05_efficiency.pdf

[6] Wayne Eberly, 2019, CPSC 331: Data Structures, Algorithms, and Their Analysis: Spring, 2019, Introduction to the Analysis of Algorithms, Lecture #6: Asymptotic Notation. Retrieved from http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC331/2019a/2_Analysis/L06/L06_asymptotic_notation.pdf