CS2040C Semester 1 2018/2019

Data Structures and Algorithms

Tutorial 05 - Trees and Binary Heap

For Week 7 (Week Starting 1 October 2018)

Document is last modified on: September 24, 2018

1 Introduction and Objective

In this tutorial, we will start our discussion on Binary Heap data structure. You may like to review `https://visualgo.net/en/heap` to refresh your memory before the tutorial.

2 Tutorial 05 Questions

Trees

In lecture, you have been introduced to trees. The definition of tree actually results in many interesting and useful properties of trees:

1. All the vertices/nodes are *connected*. This means you can get from one node to another using a sequences of edges, known as a *path*.
2. A tree with N vertices/nodes have **exactly** $N-1$ edges.
3. A tree is acyclic, which means it does not have a cycle.(Does not form a *ring*).

Q1). Prove or disprove: "There is a unique path between any two distinct nodes of a Tree (a connected acyclic undirected graph)".

We can use prove by contradiction.
1. Assume that there are two different paths $p1$ and $p2$ from two distinct vertices u and v in Tree T.
2. Because $p1$ and $p2$ are different, there is at least an edge $e:(x,y)$ in $p1$ but not in $p2$.
3. As subgraph $(p1 \bigcup p2) - e$ remains connected (because we claim both $p1$ and $p2$ connects u to v, path $p1$ can use path $p3$ from x to y from $p2$ even though edge $e:(x,y)$ is removed).
4. But that means path $p3 + e$ is a cycle (between x and y) in an acyclic Tree T, a contradiction.
Conclusion: The path from two distinct vertices u and v in Tree T is unique.

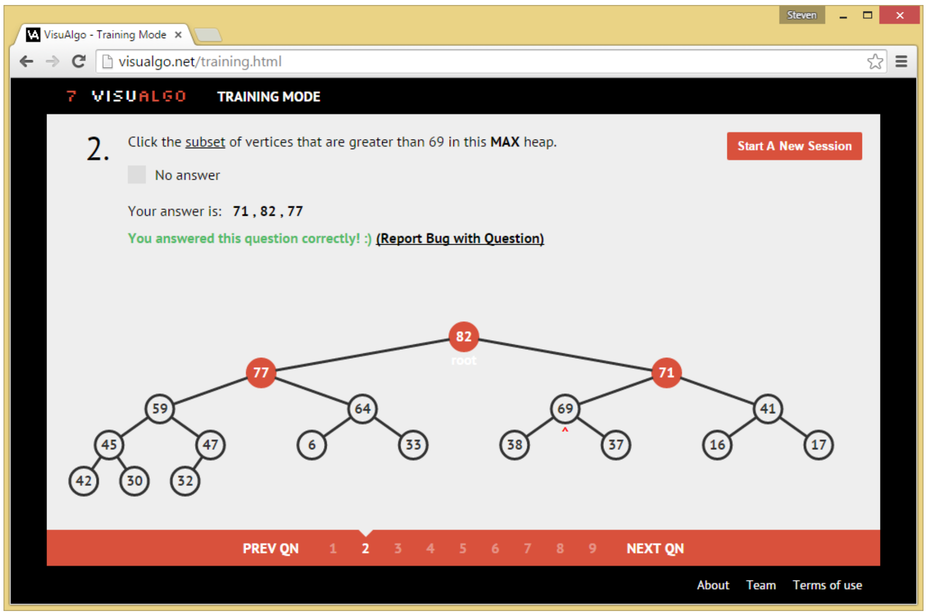
Q2). Prove or disprove: "In a complete binary tree with N nodes ($N > 1$), the number of leaf nodes is strictly less than $N/2$".

Disprove. Long answer:
From `https://visualgo.net/en/heap?slide=7-3`
1. Review the binary heap indexing scheme. The root is index 0 and the two child of node x is indexed $2x + 1$ and $2x + 2$.
2. Assume that the statement is true. This means that in a complete binary tree with N nodes, the number of non-leaf nodes is strictly more than $N/2$.
3. Then, node 0 to node $N/2$ will definitely be non-leaf nodes.
4. For a particular node x to be a non-leaf node, the left child $2x + 1$ must exist. (i.e. $2x + 1 < N$)
5. However, when $x = N/2$, $2x + 1 = N + 1$ which is not less than N.
6. This is impossible and hence the statement must be false. (i.e. In a complete binary tree with N nodes, the number of leaf nodes is **never** strictly less than $N/2$.)

Shorter answer: Just... give a small complete binary tree, e.g. one with $N = 2$ vertices (a root and its left child only). Well, there is there is 1 leaf node and that is not less than $N/2 = 1$.

Binary Heap

Q3). Give an algorithm to find all vertices bigger than some value x in a max heap that runs in $O(k)$ time where k is the number of vertices in the output.
This is a new algorithm analysis type for most of you as the time complexity of the algorithm does not depends on the input size n but rather the output size k :O...
Note that this question has also been integrated in VisuAlgo Online Quiz, so it may appear in future Online Quizzes :).


VisuAlgo - Training Mode

visualgo.net/training.html

2. Click the subset of vertices that are greater than 69 in this **MAX** heap.

No answer

Your answer is: 71, 82, 77

You answered this question correctly! :) (Report Bug with Question)

Figure 1: Also automated :)

Ans: The key insight is that the answers are at the top of the max heap... So we can perform a pre-order traversal (tutor will elaborate the concept of pre-order traversal as this may be new for many students; such graph traversal algorithms will only be discussed later in Week09-10) of the max heap starting from the root. At each vertex, check if vertex key is > x. If yes, output vertex and continue traversal. Otherwise terminate traversal on subtree rooted at current vertex, return to parent and continue traversal.

Algorithm 1 findVerticesBiggerThanX(vertex, x)

```
if (vertex.key > x) then
    output(vertex.key)
    findVerticesBiggerThanX(vertex.left, x)
    findVerticesBiggerThanX(vertex.right, x)
end if
```

Analysis of time bound required:

The traversal terminates when it encounters that a vertex's key $\leq x$. In the worst case, it encounters

$k * 2$ number of such vertices. That is, each of the left and right child of a valid vertex (vertex with key $> x$) are invalid. It will not process any invalid vertex other than those $k * 2$ vertices. It will process all k valid vertices, since there cannot be any valid vertices in the subtrees not traversed (due to the heap property). Thus the traversal encounters $O(k + 2 * k) = O(k)$ number of vertices in order to output the k valid vertices.

Q4). The *second* largest element in a max heap with more than two elements (to simplify this question, you can assume that all elements are unique) is always one of the children of the root. Is this true? If yes, show a simple proof. Otherwise, show a counter example.

Yes it is true. This can be proven easily by proof of contradiction. Suppose the second largest element is not one of the children of the root. Then, it could be the root, but this cannot be since root is the largest element by definition, thus contradiction. Or it has a parent that is not the root :O. There will be a violation to max heap property (there is nothing between largest and 2nd largest element). Contradiction. So, the second element must always be one of the children of the root.

To make things more interesting, tutor can vary the statement, e.g. third largest, smallest, second smallest, etc.

Problem Set 3

Q5). We will end the tutorial with (early) discussion of (seemingly hard but actually easy) PS3. This single PS3 contains two interesting features of Binary Heap data structure that are not available in C++ STL `priority_queue` and Java `PriorityQueue` yet: `UpdateKey(old_v, new_v)` and `DeleteKey(v)` where `v` is not necessarily the max element. You may assume that you are provided with pointer/iterator to the node that currently represents key `v`.

The first one is `UpdateKey(old-v, new-v)` operation. Will UpdateKey affect Complete Binary Tree (compact array) property? (Answer: No) May it violate Max Heap property? (Answer: Yes, then how to fix it; Next answer: Bubble up or down if necessary). What is the time complexity of this? (Answer: O($\log N$))

The second one is `Extract(v)/Delete(v)` Binary Heap element `v` that can be at arbitrary position (not just at the max/root element). This 'delete at arbitrary position' has three known possible ways: Ask last item to replace v at index $i = position[v]$ then call both shiftup+down at index i (only one is true and will be executed), use the previously discussed method: `UpdateKey(position[v], +inf)`, then `ExtractMax()`, or use lazy deletion technique (quite advanced, FYI only).

Q6). Refer to Q5. What if the pointer/iterator to the node that currently represents key `v` is not provided? It is possible to obtain an iterator/pointer to `v` in the Binary Heap quickly? How?

We can use another data structure known as *Hash Table*. This data structure is similar to an array, allowing us to store (key, value) pairs and retrieve them later using just the key in O(1). However, the keys for Hash Table are not restricted to just integer indices from 0 to $N - 1$. For Q6, we store and update (key, value) = (`v`, `pointer` to node) for all keys in the Binary Heap. Using this, we can immediately retrieve the pointer/iterator for any given key `v` in O(1), which will allow us to perform `UpdateKey` and `DeleteKey` efficiently.