[Overall Information]

In today’s class, besides having the second quiz, we studied heap data structure in detail.

With the help of heap, we also analyzed the running time of Dijkstra's shortest path algorithm.

[During the Lecture]

1. We had our second quiz today. The quiz, which lasted 20 minutes, includes questions of BFS, DFS and Topological Sort as well as Big-O notation. The grading result will be posted on course web site later.
After today’s lecture began, we recalled the four operations (insert, update, find-min and delete-min) the shortest path algorithm uses. To carry out these four operations, if an unsorted array is used, the running times are $\Theta(1)$, $\Theta(1)$, $\Theta(n)$, $\Theta(n)$ respectively. If we use sorted array, the running times become $\Theta(n)$, $\Theta(n)$, $\Theta(1)$, $\Theta(1)$. As we can see, for each of those two data structures, there are some operations that are comparatively slow. On the other hand, it seems wasteful to maintain a completely sorted array if all we need is to find the smallest element efficiently. Some “partial sorting” should be sufficient.

A data structure achieving this “partial sorting” is a **Heap** (also called **Priority Queue**). The definition of a heap is that it is a balanced binary tree, in which each node contains a data item with a numerical value. The value at any parent node must be no larger than the values at both children. There is no requirement on the relative ordering of the two children’s values. To illustrate the definition, we saw two examples, one of them a legal heap, and one that is not.

One convenient way to implement a heap is to use an array. The root is at position a[1], and for any node at array position i, its children are at positions a[2i] and a[2i+1]. This allows us to easily extend or shrink the heap (by adding or subtracting 1 to the array size), and access elements.
4. Using heap data structure, finding the minimum element of the array will only take constant time (it is located at the root of the heap, a[1]).

5. Deleting the minimum element is a little tricky, as something needs to take the root position. First, the root is replaced by the last element a[k] of the array and k is decreased. The last element is pretty big, so we need to bring it back to a position where it does not violate the heap property. This is done by the procedure “heapify-down”, called at the root (this procedure is discussed in detail later).
6. To insert a new element, we increase k (adding a new node at the end of the array), put the new element there, and then call “heapify-up” at the new node in order to move the element into its correct position.

7. To update the value d[v], find heap position j where a[j]=v, change the value, and call “heapify-up” on that position j.
8. What shall we do when performing heapify-up(a, j)? During the procedure call, the node j is compared with its immediate parent located at a[j/2] and their positions are swapped if necessary. After that, heapify-up(a, [j/2]) is called if swapping takes place in the previous step. In other words, the procedure is recursive.

9. The algorithm of heapify-down is similar. The only difference is that, this time, we compare node a[j] with its two immediate children a[2j] and a[2j+1] and swap it with the one having smaller value if necessary in order to reinstate the heap property.
10. Both heapify-up and heapify-down will fix the “problem” in the specific sub-tree without affecting the heap property of other sub-trees. This can be proved using induction although we are not going into details in the class.

11. To analyze the running time of heapify-up and heapify-down, we notice that if $n$ is the number of nodes, the heap structure contains $\log(n)$ levels. Dealing with each level takes $\theta(1)$ time. So the total running time of heapify-up and heapify-down are both $\theta(\log(n))$.

Because heapify-up and heapify-down are the most time-consuming steps, all the operations invoked when exploring a node or following an edge take $\theta(\log(n))$ time.
Let’s come back to the running time analysis of Dijkstra's shortest path algorithm. Now we have the running time of all the four operations, the total running time of the algorithm is \( \Theta(n) + \sum \Theta(\text{degree}(v) \cdot \log(n)) \)

\[ = \Theta(n + m \log(n)) = \Theta(m \log(n)). \]

For comparison purpose, the running time of shortest path using unsorted array is \( \Theta(mn) \) while the running time using sorted array is \( \Theta(n^2) \).
13. Given that we already have all those useful procedures for heaps, we can use them to design another sorting algorithm. First insert all elements into a heap one by one, then extract them one by one (always the smallest one next) and write them in an array. As each insertion or extraction takes only $\theta (\log n)$ steps, the running time of this sorting algorithm (which is known as HeapSort) is $\theta(n \log(n))$, which is faster than the algorithms we have seen so far.

14. Finally, it should be noticed that while Heaps let us implement Dijkstra's algorithm much faster than arrays, it can be done even faster. This uses a data structure called “Fibonacci Heaps” (see Chapters 19-20 in the textbook). In Fibonacci Heaps, any one insert or extract operation may still take $\log(n)$ steps. However, they have the property that by careful rearranging, it is impossible that many operations in a row take that number of steps. A more difficult analysis shows that on average,
most operations only take a constant amount of time. (This type of analysis is called “amortized analysis”.) Specifically, using Fibonacci Heaps, a sequence of \( m \) operations on \( n \) nodes takes at most \( O(m + n \log n) \) steps. Hence, the running time of Dijkstra's algorithm using Fibonacci Heaps is \( O(m + n \log n) \).