Lam Dang

CS335: Advance Data Structure

Priority Queue

First Slide:

A Priority Queue is an extension of the Queue Data Structure with added properties such as:

1/ Every Item in the Queue has a priority key attached to the item

2/ The item with the high priority is deque before the items with low priorities.

3/ If two items have the same priority, the deque will be determined by their order in the queue.

Based on different implementations, the priority can be determined by the lowest number or the highest number.

Second Slide:

The priority queue is the data structure that are useful when the data/items is the queue can be categorized by their priority. Therefore, the data structure is useful for:

- Scheduling: Items that are of more importance/urgency usually get done first and have higher priorities.
- Graph/Path Algorithms: Finding the shortest path between two nodes. For example, Dijkstra's shortest path algorithm find the shortest path from source to all vertices
- Queue Application

Third Slide:

There are 5 central priority queue operations for Priority Queue: Insertion, get_first, remove_first, is_empty, len.

Let's say we implement the Priority Queue with an array data structure:

- Insertion takes a priority key and an item, and it will insert those values to the end of the array. This process will take O (1) time.
- Get_First will traverse through the array to find the top priority, based on the priority key and it will return the item that is associated with the highest priority. This process takes O(n) time because the function has to traverse through the array
- will traverse through the array to find the top priority, based on the priority key and it will return the item that is associated with the highest priority, after deleting that item from the array. This process takes O(n) time because the function has to traverse through the array
- Is_Empty: Check if the array's length is equal to 0

- Len: Return the length of the array.

From the implementation above, we can see that get_first and delete_first takes O(n) time to execute. What if there's a way to reduce the time complexity, using a different data structure to implement.

Fourth Slide:

We can use a heap to implement Priority Queue. So, what is a heap:

- A Data Structure based on Complete Binary Tree
- Every node with the same depth in the heap should be filled before traversing down to another depth
- There are 2 types of heap:
- Min-Heap: All parents should be smaller than their descendants
- Max-heap: All parents should be bigger than their children

Fifth Slide:

How can you use heap as implementation for Priority Queue?

You can create a hierarchy using the heap where the root node is the highest priority item, and the closer to root, the higher priority it is.

This way, operation such as delete_first and get_first will be done much easier, since we just have to pop/peek at the root node of the heap.

In our implementation, we're going to use min-heap, where the smaller the key is, the higher the priority.

Sixth Slide + 7th slide:
Let's look at insertion into a Priority Queue Heap. There are a few rules to remember while inserting into a heap:
- Fill all the children of a depth from left to right before moving on to another.
- All parents must be smaller than the children
When we insert into a queue, we need to follow these steps:
Step 1: Move to the available position and add the item as a leaf node.
Step 2: Check the inserted item with its parent

- Case 1: If the item's priority is bigger than the parent => Done
- Case 2: If the item's priority is smaller than the parent:
    - Swap value of the node with its parent
    - Recursively check the priority of the parent with the ancestors until Case 1 is reached or the item becomes the root node

What's the time complexity? O(logN) because a heap is a balance binary tree, therefore execution will be done in log (number of nodes)

8th Slide:

Let's look at the case of Inserting 14. First, you go to the position where a node can be placed (right child of 31), place the 14 into the position. Secondly, you check the 14 node with its parent. We can see that the parent node 31 is bigger than the 14 node. Therefore, the two nodes are swapped. We recursively do the same with the new location of the 14 node, because the parent is still bigger. It is only when it reaches the children of the root node will the 14 node stop, because the root node (13) is smaller than 14

9th Slide:

Inserting 4 into the left child of 19. Swap with 19 because 4 is smaller. And we're done.

10th Slide: (Draw on board)

What about deletion? It is also as simple as insertion, we just need to follow these steps:

- If len(queue) = 1, delete the root and return None
- Delete the root node and replace it with the last item in the heap
- Check if the new root is smaller than any of its children:
    - Case 1: If there's no children or the root priority is smaller than its possible children, then stay the same
    - Case 2: If there exist one child that is smaller than the root: Swap with the child and do the checking step with the new position of the root.
    - Case 3: If there exist two child that is smaller than the root: Swap with the left child and do the checking step with the new position of the root.

The time complexity is also O(logN), due to the same reason as insertion.

11th Slide:

Let's look at delete_min from this tree.

First, we replace the value of the root with the one in the last position. Then, we pop the last position node. After that, we can that 9 is greater than 5, therefore, we swap value. We keep recursively do so until 9 is not greater than any of its children.

12th Slide: (Draw on board)

Let's try to delete_min from this tree.

We replace the root value(1) with 100, and delete the last node. Then, since 100 is greater than 2, we swap 2 with 100. We do so again with 17 and 25 until we get the final tree.

13th slide:

The three other operation is very simple and straight-forward:

- get_first: Return the value of the root => O(1)
- is_empty: return root == None => O(1)
- len: Return a tracker that keep track of number of item inserted, deleted => O(1)

14th slide:

A real world example is the Earlham College Room Selection. The system rewards students who has taken a lot of credits to have PRIORITY to choose room for the next semester. In this case, the school can use max-heap to implement the Priority Queue, with the priority key be the number of credits, the value is the student associated with the credits. When registration for room start, the school can just pop the queue and get the students that are prioritized first.

Reference:

- https://visualgo.net/en/heap
- http://lcm.csa.iisc.ernet.in/dsa/node138.html
- https://www.geeksforgeeks.org/priority-queue-set-1-introduction/
- Data Structure and Algorithm in Python