INFO 6205 - Program Structure and Algorithms

Crash Course in Algorithms - Worked Examples

Authors: Vidhi Patel, Nik Bear Brown

NUID: 002762999

Worked Example 1: Heapsort for Embedded

Overview

Embedded systems are devices that have limited resources such as memory and processing power. Heap sort is an algorithm that can help these systems sort and process data efficiently with minimal memory usage and computational complexity.

Heap sort is useful in embedded systems because it can sort large datasets quickly and identify the top-k elements in a dataset efficiently. This is important for real-time processing of data, where quick identification of important data points is critical.

Heap sort has low memory requirements and computational complexity compared to other sorting algorithms, making it a good choice for embedded systems with limited resources.

Overall, heap sort can help embedded systems process and analyze data efficiently, identify important data points quickly, and minimize memory usage and computational complexity.

Problem Statement

Design a sorting algorithm for an embedded system that is processing large amounts of sensor data in real-time. The algorithm must be able to efficiently sort the data and identify the top-k sensors that are indicating a problem. The system has limited memory and processing power, so the algorithm must be designed to minimize memory usage and computational complexity.

Solution and Implementation

Heap sort can help in embedded systems in several ways, including:

1. Sorting large datasets efficiently: Embedded systems often need to process and analyze large datasets, such as sensor data or image data. Heap sort is an efficient algorithm for

sorting large datasets, which can help embedded systems to quickly process and analyze the data.

2. Identifying top-k elements efficiently: Heap sort can also efficiently identify the top-k elements in a dataset, which can be important in embedded systems where real-time processing is required. For example, in a system that is monitoring a process, it may be important to quickly identify the top-k sensors that are indicating a problem.
3. Minimizing memory usage: Heap sort has low memory requirements compared to other sorting algorithms, which can be important in embedded systems where memory is limited. By minimizing memory usage, we can ensure that the system can run efficiently and avoid running out of memory.
4. Minimizing computational complexity: Heap sort has a relatively low computational complexity of O(n log n), which makes it a good choice for embedded systems where processing power is limited. By minimizing computational complexity, we can ensure that the system can respond to events in real-time and avoid delays that could be costly or dangerous.

Overall, heap sort can help embedded systems to efficiently process and analyze large datasets, quickly identify important data points, and minimize memory usage and computational complexity. By using heap sort, we can ensure that embedded systems can operate efficiently and effectively, even with limited resources.

We can use the heap data structure to efficiently find the top-k sensors. The algorithm can be implemented as follows:

1. Initialize an empty heap.
2. Insert the first k elements of the sensor data into the heap.
3. For each remaining element in the sensor data, compare it with the smallest element in the heap.
4. If the new element is larger than the smallest element in the heap, replace the smallest element with the new element and re-heapify the heap.
5. Once we have processed all the elements in the sensor data, the heap contains the top-k sensors.
6. Use the heap sort algorithm to sort the heap in descending order and return the top-k sensors.

```
import heapq

def top_k_sensors(sensor_data, k):
    # initialize an empty heap
    heap = []
    # insert the first k elements into the heap
```

```
    for i in range(k):
        heapq.heappush(heap, sensor_data[i])
    # iterate over the remaining elements of the sensor data
    for i in range(k, len(sensor_data)):
        # if the current element is larger than the smallest element in the heap
        if sensor_data[i] > heap[0]:
            # replace the smallest element in the heap with the current element
            heapq.heapreplace(heap, sensor_data[i])
    # extract the top-k sensors from the heap and sort in descending order
    top_k = [heapq.heappop(heap) for i in range(k)][::-1]
    return top_k


sensor_data = [10, 2, 7, 5, 3, 6, 8, 1, 9, 4]
k = 3
top_k = top_k_sensors(sensor_data, k)
print("Top {} sensors: {}".format(k,top_k))
```

```
Top 3 sensors: [10, 9, 8]
```

Explanation

1. We start by initializing an empty heap using the 'heapq' module in Python. We'll be using a max heap, which means the largest elements will be at the top of the heap.
2. We insert the first k elements of the sensor data into the heap using the 'heappush' function. This initializes the heap with the first k largest elements.
3. We then iterate over the remaining elements of the sensor data, comparing each element with the smallest element in the heap. We use the 'heapreplace' function to replace the smallest element with the new element if the new element is larger than the smallest element.
4. If the new element is larger than the smallest element in the heap, the heap is re-heapified using the 'heapify' function. This ensures that the largest elements are still at the top of the heap.
5. Once we have processed all the elements in the sensor data, the heap contains the top-k sensors.
6. We use the 'heappop' function to extract the top-k sensors from the heap in ascending order, and then sort the list in descending order using the [::-1] slice notation. This returns the top-k sensors in descending order, which is the desired output.

By using the heap data structure and the heap sort algorithm, we can efficiently find the top-k sensors in a given list of sensor readings in an embedded system.

Worked Example 2: Heapsort for Security Systems

Overview

Heap Sort is a sorting algorithm that works by transforming the input array into a binary heap and then repeatedly extracting the maximum element from the heap and placing it at the end of the array. The algorithm has a worst-case time complexity of O(n log n) and is not as commonly used as other sorting algorithms like Quicksort or Mergesort, but it has the advantage of being an in-place sort.

Problem Statement

Suppose we have a large dataset of log entries from a security system, and we want to identify the top 10 most suspicious events. The dataset contains millions of records and is stored in a log file, but we want to perform the analysis offline on a local machine.

Solution and Implementation

- Parse the log file and extract the relevant information, such as the type of event and the severity level.
- Store the event data in a binary heap, where the priority of each element is determined by its severity level. The higher the severity level, the higher the priority.
- Use a modified version of the 'heapify' function to build a heap of the top 'k' events, where 'k' is 10.
- Extract the top 'k' events from the heap by repeatedly extracting the maximum element and adding it to a separate list of the top 'k' suspicious events. This is done by swapping the root node with the last element in the heap, reducing the heap size by one, and then calling 'heapify_top_k' on the new root to restore the heap property.
- Display the top 'k' suspicious events to the user, along with information such as the event type, severity level, and timestamp.

```
import heapq

def extract_top_k_events(log_file_path, k):
    # Parse the log file and extract the relevant information
    log_data = parse_log_file(log_file_path)

    # Build a binary heap of the top k events based on their severity level
    heap = []
    for event in log_data:
        if len(heap) < k:
```

```
            heapq.heappush(heap, event)
        else:
            heapq.heappushpop(heap, event)

    # Extract the top k events from the heap
    top_k_events = []
    while heap:
        top_k_events.append(heapq.heappop(heap))

    # Return the top k events
    return top_k_events

def parse_log_file(log_file_path):
    # Parse the log file and extract the relevant information
    # For this example, let's assume each log entry is a tuple
    # containing the event type, severity level, and timestamp
    log_data = []
    with open(log_file_path, 'r') as log_file:
        for line in log_file:
            entry = line.strip().split(',')
            event_type = entry[0]
            severity = int(entry[1])
            timestamp = entry[2]
            log_data.append((severity, event_type, timestamp))
    return log_data
```

This code uses the 'heapq' module from the Python standard library to build a binary heap of the top 'k' events based on their severity level. The 'parse_log_file' function is used to read in the log file and extract the relevant information for each event, such as the severity level, event type, and timestamp. The 'extract_top_k_events' function then uses a modified version of the 'heapify' function to build the binary heap of the top 'k' events and extract the top 'k' events from the heap.

Explanation

Heap sort is a sorting algorithm that is well-suited for problems where we need to identify the top 'k' elements in a large dataset. In this example, we used heap sort to efficiently identify the top 10 most suspicious events in a large log file generated by a security system.

The solution involved parsing the log file to extract the relevant information about each event, and then building a binary heap of the top 'k' events based on their severity level. By using a heap, we were able to quickly identify the most suspicious events and extract them in 'O(klogk)' time.

The implementation of heap sort involved modifying the standard 'heapify' function to consider the priority of each element based on its severity level, and then using a loop to repeatedly extract the maximum element from the heap. Once we had extracted the top 'k' suspicious

events, we could display them to the user along with additional information such as the event type, severity level, and timestamp.

Overall, heap sort is a powerful algorithm for identifying the top 'k' elements in a dataset, and it can be used in a wide range of applications where this is a common task, including security systems.