Assignment 4: Priority Queue Implementation and Applications

Ashim Sedhain

University of the Cumberlands

MSCS-532-M20: Algorithms and Data Structures

June 15, 2025

Priority Queue Implementation and Applications

Introduction

Priority queues are fundamental abstract data types that support the efficient retrieval of the highest or lowest priority element. They are widely used in scheduling, graph algorithms (for example, Dijkstra's), operating systems, and simulations. In this project, priority queue has been implemented using a binary heap structure to manage task scheduling operations. The implementation supports insertion, extraction of the minimum or maximum element, priority updates, and basic utility operations.

Design Choices

In this project, priority queue is implemented using a binary heap backed by a dynamic array (Python list). This choice is justified with properties such as:

- Space efficiency: A binary heap does not require additional pointers as in trees, and parent-child relationships can be computed using simple index arithmetic.
- Simplicity: Arrays reduce the overhead of memory management and improve code clarity.
- Performance: Array-backed binary heaps support logarithmic time for insertion and extraction, and constant time access to the minimum or maximum element.

A min-heap variant was chosen to simulate scheduling scenarios where lower numerical priority values represent more urgent tasks. For example, priority 1 is more urgent than priority 5. This choice aligns with common scheduling disciplines like Earliest Deadline First (EDF) or Shortest Job First (SJF).

Core Operations and Their Analysis

insert(task)

This operation appends a new task to the end of the heap and then performs a heapify-up procedure to restore the heap property.

Time complexity: $O(log\ n)$, due to the height of the heap.

extract_min()

This operation removes the task with the highest priority (lowest value) from the root of the heap. The last element in the heap is moved to the root, followed by a heapify-down process.

Time complexity: $O(log\ n)$

decrease_key(task_id, new_priority)

This operation scans the heap for a task by its ID, updates its priority, and then repositions the task in the heap to maintain the min-heap property.

Time complexity: $O(n)$ for the linear search, followed by $O(log\ n)$ for re-heapifying.

is_empty()

Returns whether the priority queue is empty by checking the length of the list.

Time complexity: $O(1)$