Lab 11: Heaps and Treaps Design Document

Lab 11: Heaps and Treaps Group 14: Ju Ho Kim, Sangmin Kim Date: May, 11th, 2025

Part 1: Design

A. Binary Min-Heap

Purpose A Binary Min-Heap is a complete binary tree where each parent node has a value less than or equal to its children. This property ensures the minimum element is always at the root, making it efficient for priority queue operations.

Input/Output

- **Input**: Comparable elements (integers, strings, etc.)
- **Output**: Elements in min-heap order, with the smallest element always accessible at the root

Pseudocode **Insert Operation:** function insert(heap, item): append item to end of heap array current_index = heap.length - 1 while current_index > 0: parent_index = (current_index - 1) // 2 if heap[current_index] < heap[parent_index]: swap heap[current_index] and heap[parent_index] current_index = parent_index else: break

Remove Min Operation: function remove_min(heap): if heap is empty: return None min_value = heap[0] heap[0] = heap[last_index] remove last element heapify_down from index 0 return min_value

Peek Operation: function peek_min(heap): if heap is empty: return None return heap[0]

Data Representation

- Array-based implementation using Python list
- Parent of index i: (i-1)//2
- Left child of index i: 2*i + 1
- Right child of index i: 2*i + 2

B. Treap

Purpose A Treap is a randomized binary search tree that combines properties of both BST (for keys) and heap (for priorities). Each node has a key and a randomly assigned priority, maintaining BST property for keys and heap property for priorities.

Input/Output

- **Input**: Key-value pairs where keys are comparable and priorities are randomly assigned
- **Output**: A balanced BST structure that supports efficient search, insert, and delete operations

Pseudocode **Insert Operation:** function insert(root, key): if root is None: return new TreapNode(key) if key < root.key: root.left = insert(root.left, key) if root.left.priority > root.priority: root = rotate_right(root) else: root.right = insert(root.right, key) if root.right.priority > root.priority: root = rotate_left(root)

return root

Rotation Operations: function rotate_left(root): new_root = root.right root.right = new_root.left new_root.left = root return new_root function rotate_right(root): new_root = root.left root.left = new_root.right new_root.right = root return new_root

Search Operation: function search(root, key): if root is None: return False if key == root.key: return True if key < root.key: return search(root.left, key) else: return search(root.right, key)

Data Representation

- Node-based implementation with TreapNode class
- Each node contains:
  - key: comparable value
  - priority: randomly assigned integer
  - left: reference to left child
  - right: reference to right child