PRIORITY QUEUES

Explanation: Priority queues are data structures that allow for the efficient retrieval of the highest (or lowest) priority element. They are typically implemented using a max heap, which ensures that the largest element is always at the top, making retrieval an (O(1)) operation and insertion an (O(\log n)) operation.

Key Points:

- Priority queues maintain a set of items sorted by priority.
- The default priority queue uses a max heap where the parent node is greater than or equal to its child nodes.
- Retrieving the top element is an (O(1)) operation, while insertion and reprioritization are (O(\log n)) operations.
- Elements can be inserted in any order, but they will be retrieved from largest to smallest.

REVERSED PRIORITY QUEUES

Explanation: Reversed priority queues are used when the requirement is to retrieve elements from smallest to largest. This is achieved by defining a priority queue with a greater relationship between the items.

Key Points:

- Reversed priority queues retrieve elements from smallest to largest.
- The internal data structure can be defined using a vector.
- The comparison function is set to ensure smaller elements have higher priority.
- Elements are inserted in any order but retrieved from smallest to largest.

PAIRS IN PRIORITY QUEUES

Explanation: Priority queues can handle more complex objects like pairs, which consist of two elements. The sorting of pairs is primarily based on the first element, and if the first elements are equal, the second element is used for sorting.

Key Points:

- Pairs consist of two elements accessed via .first and .second.
- Pairs are sorted by the first element, and if equal, by the second element.
- Useful for storing items with two associated pieces of information.
- Elements are inserted and retrieved based on the priority defined by the pair's elements.

OBJECTS IN PRIORITY QUEUES

Explanation: Objects can be inserted into priority queues, but this requires defining a comparison operator to ensure proper sorting. This can be complex due to the need to avoid copying objects and instead use pointers for efficient memory management.

Key Points:

- Objects require a defined comparison operator for sorting in priority queues.
- Copying objects can be inefficient, so pointers are used to avoid this.
- Pointers must be compared based on the object's values, not memory addresses.

- A custom comparator function is necessary to handle pointer comparisons.
- Dynamic objects created on the heap can be managed within priority queues using pointers.