This lecture discusses the concept and implementation of priority queues, which are data structures that allow for the efficient retrieval of the highest (or lowest) priority element. Priority queues are useful when we need to repeatedly access the largest or smallest item from a set while maintaining the order of the elements. The lecture begins by explaining the default priority queue, which uses the standard template library's priority queue and is typically implemented using a max heap. In a max heap, the parent node is always greater than or equal to its child nodes, making the top element the largest. Accessing the top element is an $O(1)$ operation, while reordering the queue after removal is an $O(\log n)$ operation.

The lecture then covers the reversed priority queue, which retrieves elements from smallest to largest. This is achieved by defining the priority queue with a vector and specifying a greater-than relationship between the items. The process of inserting elements and retrieving them in ascending order is demonstrated, showing that the priority queue can be adapted to different sorting requirements.

Next, the lecture explores the use of more complex objects in priority queues, such as pairs. Pairs consist of two elements and can be sorted by their first element, and if the first elements are equal, by their second element. This allows for the storage and retrieval of objects that carry multiple pieces of information, such as a priority value and an associated data item.

The lecture also addresses the insertion of custom objects into priority queues. Using a class A with a public integer x, the operator less-than is defined to enable comparison of A objects. This allows objects to be pushed into the priority queue and retrieved in the correct order. However, the lecture notes the issue of object copying when inserting objects into containers, which can be problematic for complex objects.

Finally, the lecture discusses the insertion of pointers to objects into priority queues to avoid copying. This involves setting up a priority queue with a custom comparator that compares the x values of the objects pointed to by the pointers. The comparator ensures that the priority queue sorts the objects based on their values rather than their memory addresses. The lecture concludes by demonstrating the creation and use of a priority queue with dynamically allocated objects, ensuring that the sorting is based on the actual object values.