PES UNIVERSITY

Data Structures

Dilip Maripuri
Associate Professor
Department of Computer Applications

Session Outline

PES UNIVERSITY

Queues

Queue ADT : Priority Queues

PES UNIVERSITY

- Each element has a "priority" associated with it.
- In a priority queue, an element with high priority is served before an element with low priority.
- This is an important concept in various areas of computing, like operating system processes scheduling.

Queue ADT : Priority Queues - Concepts

- **Priority Assignment**
  - In a priority queue, each element is associated with a priority.
  - This priority can be assigned based on various criteria depending on the application, such as importance, urgency, or any custom rules defined by the programmer.
- **Order of Elements**
  - Unlike standard queues, where the order of elements is determined by the sequence of their arrival (FIFO - First In First Out), in priority queues, it's the priority that determines the order of service of the elements.
  - The element with the highest priority is served first.

Queue ADT : Priority Queues - Concepts

PES UNIVERSITY

- **Array-Based Implementation**
  - **Unsorted Array**
    - In this implementation, elements are inserted without any specific order.
    - Insertion (enqueue) is very efficient (O(1)) as it only involves adding the element at the end of the array.
    - However, deletion (dequeue) of the highest priority element requires scanning the entire array to find it, making it O(n).

Queue ADT : Priority Queues - Concepts

PES UNIVERSITY

- **Array-Based Implementation**
  - **Sorted Array**
    - The array is kept sorted based on priority.
    - Dequeue operation is efficient (O(1)) as the highest priority element is always at one end of the array.
    - However, enqueue is less efficient (O(n)) as it may require shifting elements to maintain the sorted order.

Queue ADT : Priority Queues - Concepts

PES UNIVERSITY

- **Linked List Implementation**
  - **Unsorted Linked List**
    - Similar to the unsorted array, elements can be added with O(1) complexity, but finding and removing the highest priority element takes O(n) time.
  - **Sorted Linked List**
    - Keeps elements sorted by priority. Dequeue is O(1), but enqueue operation can take up to O(n) time to find the correct insertion point and maintain the sorted order.

Queue ADT : Priority Queues - Classification

- **Terminology**
  - **Element with the highest priority (lowest number)**
  - **Element with the lowest priority (highest number)**

Queue ADT : Priority Queues - Classification

PES UNIVERSITY

- **Min-Priority Queue vs Max-Priority Queue**
  - **Min-Priority Queue**
    - In a min priority queue, an element with the lowest priority value is considered the highest priority.
    - This means that when elements are removed from the queue, the element with the lowest priority value is removed first.
    - It's often used in algorithms like Dijkstra's shortest path algorithm, where the shortest distance (considered as a lower value) has a higher priority.
    - Example: If the elements are {20, 5, 15} with priorities as their values, the removal order will be 5, 15, 20.

Queue ADT : Priority Queues - Classification

- **Min-Priority Queue vs Max-Priority Queue**
  - **Max-Priority Queue**
    - In a max priority queue, an element with the highest priority value is given the highest priority.
    - The element with the highest priority value is removed from the queue first.
    - This type of queue is common in scenarios like CPU scheduling in operating systems, where processes with higher numerical priority are executed first.
    - Example: With the same set of elements {20, 5, 15}, the removal order will be 20, 15, 5.

Queue ADT : Priority Queues - Classification

- **Ascending Priority Queue (Sorted Queue – Ascending Order)**
  - Synonymous with a min priority queue.
  - The priority of elements increases in an ascending order. That is, elements with lower values are given higher priority.
  - As a result, the queue serves elements in ascending order of their priority values.

Queue ADT : Priority Queues - Classification

- **Descending Priority Queue (Sorted Queue – Descending Order)**
  - Similar to a max priority queue.
  - In this queue, the priority decreases as the value of the elements increases.
  - Elements with higher values are given higher priority, so the queue serves elements in descending order of their priority values.

Queue ADT : Ascending Priority Queue – Implementation


```
typedef struct {
    int element, priority;
} Node;

typedef struct PriorityQueueStruct {
    Node array[MAX_SIZE];
    int size;
} *PriorityQueue;
```

```
void InitializeQueue(PriorityQueue queue)
{    queue->size = 0; }

bool IsEmpty(PriorityQueue queue)
{    return queue->size == 0;   }

int Size(PriorityQueue queue)
{    return queue->size;        }
```

Queue ADT : Ascending Priority Queues – Implementation


```
void Add(PriorityQueue queue, int element, int priority) {
    if (queue->size == MAX_SIZE) {
        printf("Error: Queue is full.\n");
        return;
    }

    Node newNode;
    newNode.element = element;
    newNode.priority = priority;

    int i;
    for (i = queue->size - 1; (i >= 0 && queue->array[i].priority > priority); i--)
        queue->array[i + 1] = queue->array[i];

    queue->array[i + 1] = newNode;
    queue->size++;
}
```

Queue ADT : Ascending Priority Queues – Implementation


```
Node Pop(PriorityQueue queue) {
    if (queue->size == 0) {
        printf("Error: Queue is empty.\n");
        return (Node){-1, -1};
    }
    Node popNode = queue->array[0];
    for (int i = 0; i < queue->size - 1; i++)
        queue->array[i] = queue->array[i + 1];
    queue->size--;
    return popNode;
}
```

Queue ADT : Ascending Priority Queues – Implementation

PES UNIVERSITY

```
Node Peek(PriorityQueue queue) {
    if (queue->size == 0) {
        printf("Error: Queue is empty.\n");
        return (Node){-1, -1};
    }
    return queue->array[0];
}
```

Queue ADT : Ascending Priority Queues – Implementation


```c
int main() {
    PriorityQueue queue = malloc(sizeof(struct PriorityQueueStruct));
    InitializeQueue(queue);
    Add(queue, 10, 2);
    printf("Peek: %d\n", Peek(queue).element);
    printf("Pop: %d\n", Pop(queue).element);
    free(queue);
    return 0;
}
```

Queue ADT : Ascending Priority Queues – Implementation

PES UNIVERSITY

Linked List Implementation – Single List

```
typedef struct Node {
    int element;
    int priority;
    struct Node* next;
} *Node;

Node NewNode(int element, int priority) {
    Node temp = (Node)malloc(sizeof(struct Node));
    temp->element = element;
    temp->priority = priority;
    temp->next = NULL;
    return temp;
}

int IsEmpty(Node* head)
{    return (*head) == NULL;    }
```

Queue ADT : Ascending Priority Queues – Implementation

PES UNIVERSITY

Linked List Implementation – Single List

```
void Add(Node* head, int element, int priority) {
    Node temp = NewNode(element, priority);
    Node start = *head;

    if (*head == NULL || (*head)->priority > priority) {
        temp->next = *head;
        *head = temp;
    }
    else
    {
         while (start->next != NULL && start->next->priority <= priority)
            start = start->next;
        temp->next = start->next;
        start->next = temp;
    }
}
```

Queue ADT : Ascending Priority Queues – Implementation

PES UNIVERSITY

Linked List Implementation – Single List

```
int Pop(Node* head) {
    if (*head == NULL) {
        printf("Queue is empty.\n");
        return -1;
    }

    Node temp = *head;
    *head = (*head)->next;
    int popped = temp->element;
    free(temp);
    return popped;
}
```

```
int Peek(Node* head) {
    if (*head == NULL) {
        printf("Queue is empty.\n");
        return -1;
    }
    return (*head)->element;
}
```

Queue ADT : Ascending Priority Queues – Implementation

PES UNIVERSITY

Linked List Implementation – Single List

```
void DestroyQueue(Node* head) {
    while (*head != NULL) {
        Node temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}
```

```
int main() {
    Node pq = NULL;
    Add(&pq, 10, 2);
    printf("Peek: %d\n", Peek(&pq));
    printf("Pop: %d\n", Pop(&pq));
    DestroyQueue(&pq));
    return 0;
}
```

Queue ADT : Ascending Priority Queues – Implementation

PES UNIVERSITY

Linked List Implementation – List of Lists

```
// Define the node structure
typedef struct Node {
    int element;
    struct Node* next;
} *Node;

// Define the header node structure
typedef struct HeaderNode {
    int priority;
    Node head;
    struct HeaderNode* next;
} *HeaderNode;
```

Queue ADT : Ascending Priority Queues – Implementation

PES UNIVERSITY

Linked List Implementation – List of Lists

```
// Function to create a new node
Node Create_Node(int element) {
    Node newNode = (Node)malloc(sizeof(struct Node));
    newNode->element = element;
    newNode->next = NULL;
    return newNode;
}
```

```
// Function to create a new header node
HeaderNode NewHeaderNode(int priority) {
    HeaderNode newHeader = (HeaderNode)malloc(sizeof(struct HeaderNode));
    newHeader->priority = priority;
    newHeader->head = NULL;
    newHeader->next = NULL;
    return newHeader;
}
```

Queue ADT : Ascending Priority Queues – Implementation

Linked List Implementation – List of Lists

```
// Function to add an element with a given priority
void Add(HeaderNode* priorityQueue, int element, int priority) {
    if (*priorityQueue == NULL || (*priorityQueue)->priority > priority) {
        HeaderNode newHeader = NewHeaderNode(priority);
        newHeader->next = *priorityQueue;
        newHeader->head = Create_Node(element);
        *priorityQueue = newHeader;
    } else {
        // Search for the correct position or existing priority
        HeaderNode temp = *priorityQueue;
        while (temp->next != NULL && temp->next->priority < priority) {
            temp = temp->next;
        }
```

Queue ADT : Ascending Priority Queues – Implementation

Linked List Implementation – List of Lists

```
        if (temp->priority == priority) {
            // Priority already exists, add to the existing priority list
            Node newNode = Create_Node(element);
            newNode->next = temp->head;
            temp->head = newNode;
        } else {
            // Create a new priority header
            HeaderNode newHeader = NewHeaderNode(priority);
            newHeader->next = temp->next;
            temp->next = newHeader;
            newHeader->head = Create_Node(element);
        }
    }
}
```

Task

- The logic adds the newest element at the beginning of the respective priority list.
- Modify to insert at end.
- Also Explore the viability of using front and rear in each priority number queue.
- How does it impact the performance?

Queue ADT : Ascending Priority Queues – Implementation

Linked List Implementation – List of Lists

```
// Function to remove and return the element
// with the highest priority (lowest number)
int Pop(HeaderNode* priorityQueue) {
    if (*priorityQueue == NULL) {
        printf("Queue is empty.\n");
        return -1;
    }

    Node tempNode = (*priorityQueue)->head;
    int poppedElement = tempNode->element;
    (*priorityQueue)->head = tempNode->next;
```

Queue ADT : Ascending Priority Queues – Implementation

PES UNIVERSITY

Linked List Implementation – List of Lists

```
    // If no more elements in this priority, remove the header
    if ((*priorityQueue)->head == NULL) {
        HeaderNode tempHeader = *priorityQueue;
        *priorityQueue = (*priorityQueue)->next;
        free(tempHeader);
    }

    free(tempNode);
    return poppedElement;
}
```

Queue ADT : Ascending Priority Queues – Implementation

PES UNIVERSITY

Linked List Implementation – List of Lists

```
// Function to destroy the queue and free allocated memory
void DestroyQueue(HeaderNode* priorityQueue) {
    while (*priorityQueue != NULL) {
        HeaderNode header = *priorityQueue;
        *priorityQueue = (*priorityQueue)->next;
        // Free all nodes in the current header
        Node tempNode = header->head;
        while (tempNode != NULL) {
            Node nextNode = tempNode->next;
            free(tempNode);
            tempNode = nextNode;
        }
        free(header);
    }
}
```

Queue ADT : Ascending Priority Queues – Implementation

Linked List Implementation – List of Lists

```
int IsEmpty(HeaderNode* priorityQueue) {
    return (*priorityQueue) == NULL;
}

int main() {
    HeaderNode pq = NULL;

    Add(&pq, 40, 2);

    while (!IsEmpty(&pq)) {
        printf("Pop: %d\n", Pop(&pq));
    }
    DestroyQueue(&pq);
    return 0;
}
```

Queue ADT : Priority Queues – Comparative Analysis

PES UNIVERSITY

- **Compare the following Approaches**
  - **Implementation using Single List**
  - **Implementation using List of Lists**

Queue ADT : Priority Queues – Comparative Analysis

PES UNIVERSITY

- **Priority Queue Using a Single Linked List**
  - **Space Complexity**
    - More efficient as it only uses a single linked list where each node contains the data and its priority.
  - **Time Complexity**
    - Insertion (Add): O(n) in the worst case, as it may need to traverse the list to find the correct insertion point based on priority.
    - Removal (Remove): O(1), always removes from the head of the list.
  - **General Considerations**
    - Simpler in terms of data structure, as it only maintains one list.
    - However, each node carries an additional priority field, slightly increasing the size of each node.

Queue ADT : Priority Queues – Comparative Analysis

- **Priority Queue Using Linked List of Linked Lists**
  - **Space Complexity**
    - Higher due to the use of an additional HeaderNode structure for each priority level.
    - This increases memory overhead, especially if there are many unique priorities with few elements each.
  - **Time Complexity**
    - Insertion (Add): O(n) in the worst case, as it might need to traverse the entire list of priorities, and in the case of an existing priority, insert the new element at the beginning of its sublist.
    - Removal (Pop): O(1), since it always removes from the head of the highest-priority sublist.

Queue ADT : Priority Queues – Comparative Analysis

PES UNIVERSITY

- **Priority Queue Using Linked List of Linked Lists**
  - **General Considerations**
    - This approach groups elements of the same priority, potentially making some operations faster.
    - However, it adds complexity in managing multiple linked lists and may result in higher memory usage due to additional pointers and nodes.

Queue ADT : Priority Queues – Comparative Analysis

Code 1 : Implementation Using Single List

Code 2 : Implementation Using List of Lists

- **Memory Efficiency**
  - Code 1 is generally more memory-efficient due to its simpler data structure.
- **Insertion Efficiency**
  - Both have similar time complexities for insertion.
  - However, Code 2 might have a slight edge if there are many elements with the same priority, as it doesn't need to traverse the sublist for insertion.

Queue ADT : Priority Queues – Comparative Analysis

Code 1 : Implementation Using Single List

Code 2 : Implementation Using List of Lists

- **Removal Efficiency**
  - Both are efficient in removal operations, offering O(1) time complexity.
- **Complexity**
  - Code 2 is more complex due to handling multiple lists, which might increase the likelihood of bugs or errors.
  - Code 1 is simpler and more straightforward to understand and maintain.

Queue ADT : Priority Queues – Comparative Analysis

Code 1 : Implementation Using Single List

Code 2 : Implementation Using List of Lists

- **Use Case Suitability**
  - Code 2 might be preferred in scenarios where multiple elements of the same priority are frequently added and removed, as it groups these elements together.
  - Code 1 is generally suitable for most use cases due to its simplicity and efficiency.

THANK YOU

Dilip Maripuri
Associate Professor
Department of Computer Applications
dilip.maripuri@pes.edu