Data Structures & Algorithms (CS-212)

Week 8: Queues

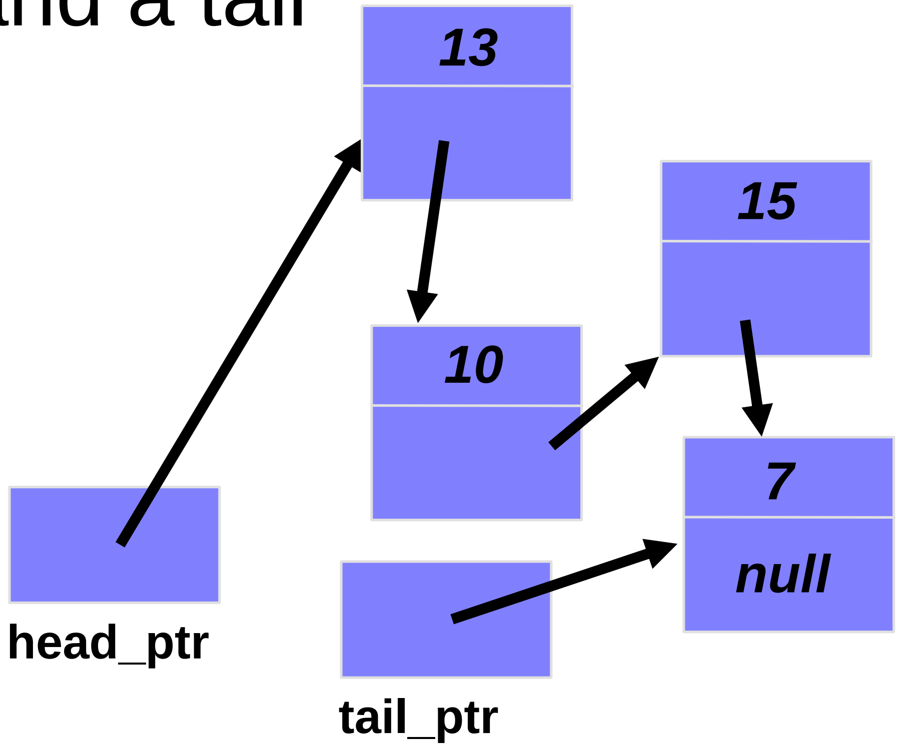
Implementing Queue ADT: Array Queue

- Keep track of the number of elements in the queue, `size`.
- Enqueue at the back of the array (`size`).
- Dequeue at the front of the array (index 0)

QUEUE USING LINKED LIST

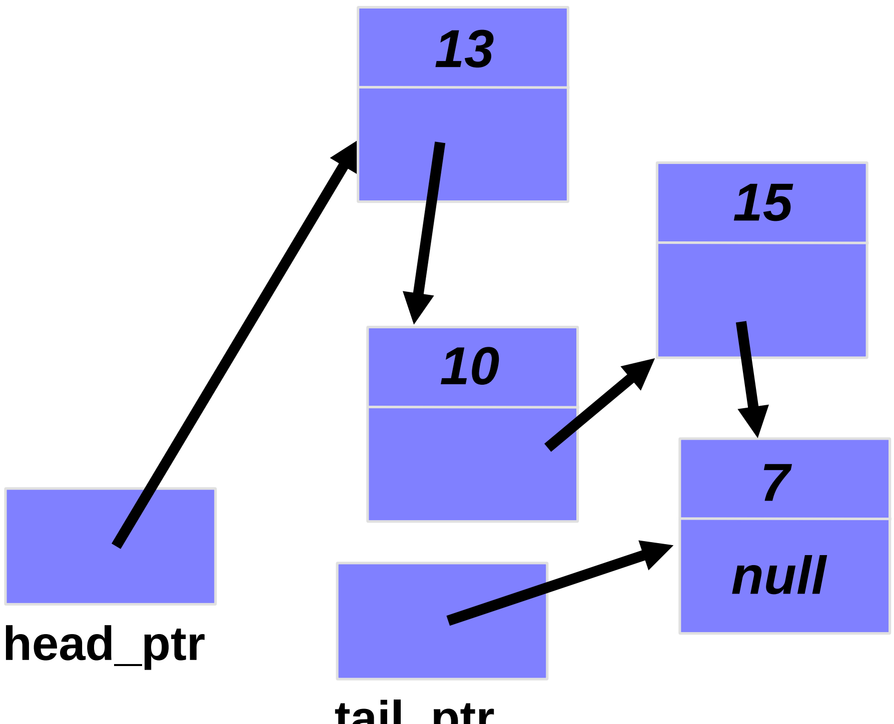
Linked List Implementation

- A queue can also be implemented with a linked list with both a head and a tail pointer.



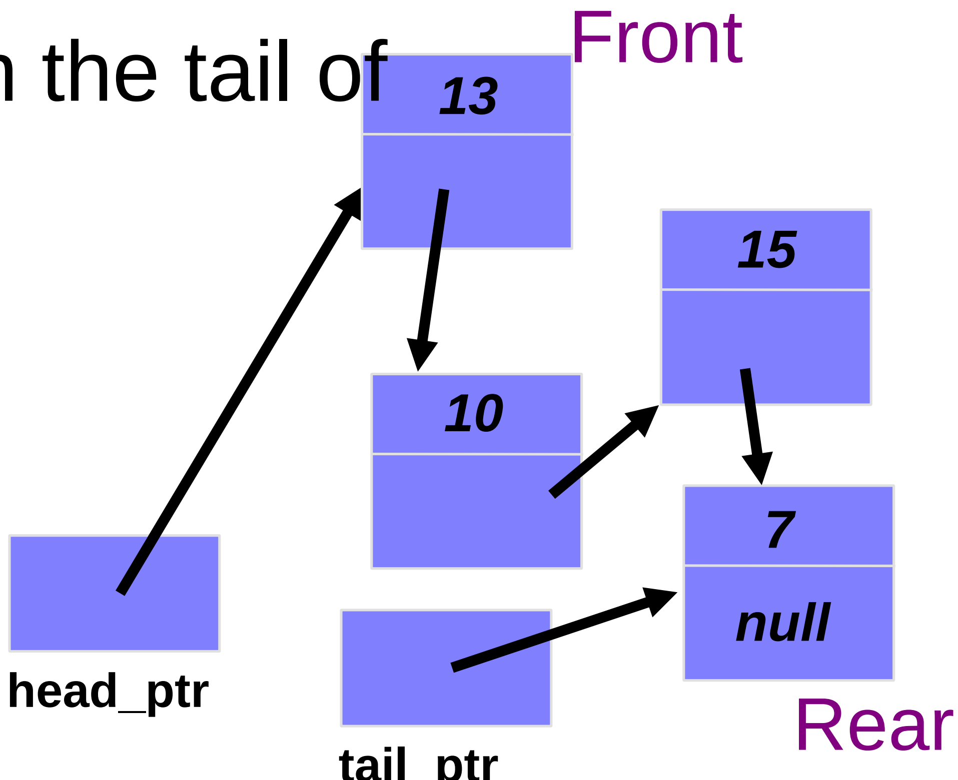
Linked List Implementation

- Which end do you think is the front of the queue? Why?



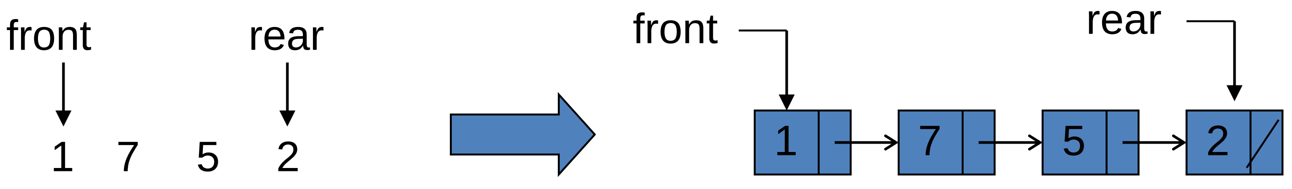
Linked List Implementation

- The head_ptr points to the front of the list.
- Because it is harder to remove items from the tail of the list.



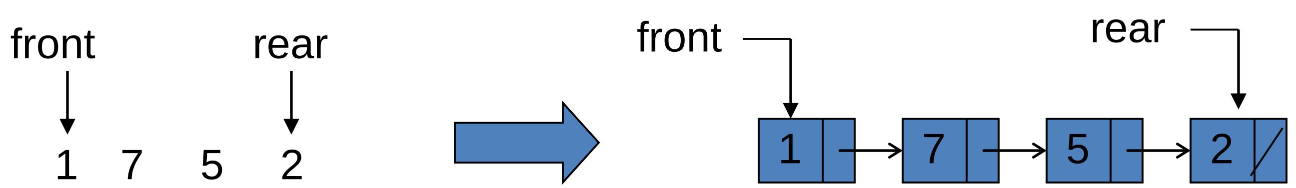
Implementing Queue

- Using linked List:

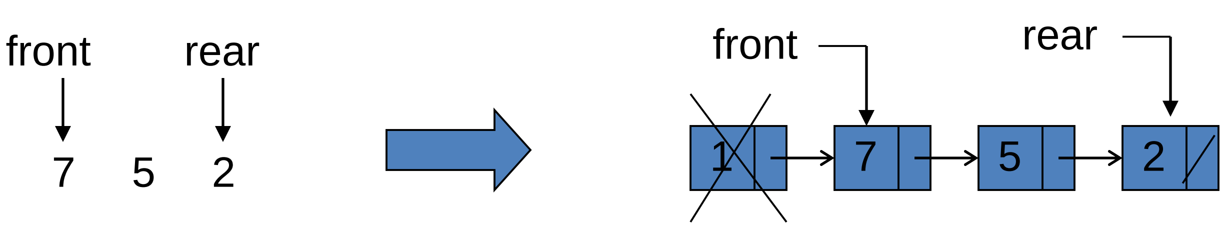


Implementing Queue

- Using linked List:

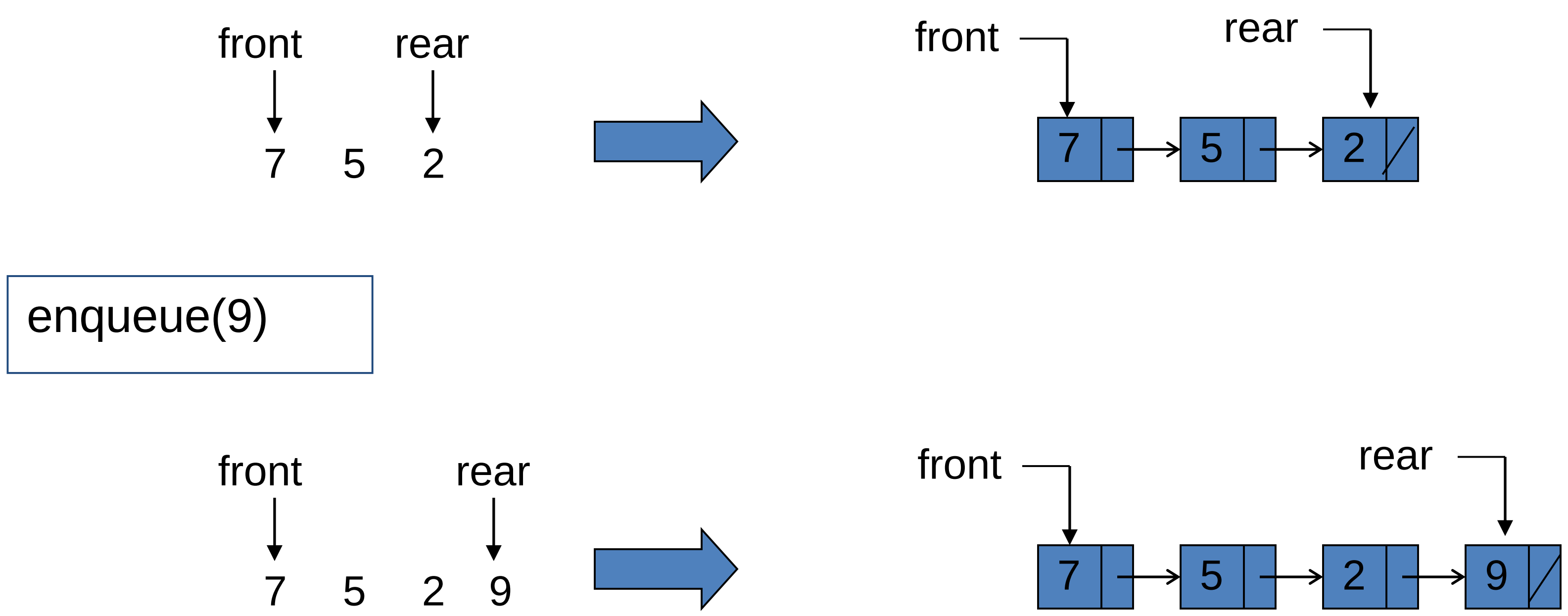


dequeue()



Implementing Queue

- Using linked List:



IMPLEMENTATION

Queue Implementation Using Linked List

- We use two classes:
  - **`Node`**
  - **`Queue`**
- Declare `Node` class for the nodes
  - `data`: `int`-type data in this example
  - `next`: a pointer to the next node in the list

```
class Node {
public:
        int    data;          // data
        Node*        next;          // pointer to next

};
```

Queue Implementation Using Linked List

- Declare `Queue`, which contains
  - `front`: a pointer to the front of the queue
  - `rear` : a pointer to the rear of the queue
  - Operations on Queue

```
class Queue{
public:
        Queue(void) // constructor
        {
                front= NULL;
                rear = NULL
        }
        ~Queue(void); // destructor
private:
        Node* front;
        Node* rear;

};
```

Queue Operations

- Operations of `Queue`
  - `IsEmpty`: determine whether or not the Queue is empty i.e. front== NULL
  - `enqueue`: insert a new item in queue at rear
  - `dequeue`: delete an item from queue from front

Queue Implementation Using Linked List

- Declare `Queue`, which contains
  - `front`: a pointer to the front of the queue
  - `rear` : a pointer to the rear of the queue
  - Operations on Queue

```
class Queue{
public:
        Queue(void) // constructor
        {
                front= NULL;
                rear = NULL
        }
        ~Queue(void); // destructor
        int isEmpty();
        int dequeue();
        void enqueue();
        int front();
private:
        Node* front;
        Node* rear;

};
```

Implementing Queue

```
int queue::dequeue()
{
    int x = front->data;
    Node* p = front;
    front = front->next;
    delete p;
    return x;
}
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->x = x;
    newNode->next = NULL;
    rear->next = newNode;
    rear = newNode;
}
```

Implementing Queue

```
int Queue:: front()
{
    return front->data;
}


int Queue::isEmpty()
{
    return ( front == NULL );
}
```

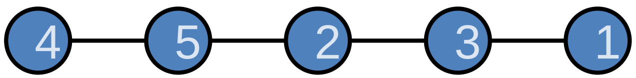
PRIORITY QUEUE

Priority Queue

- Sometimes it is not enough just do FIFO ordering
  - may want to give some items a higher priority than other items
    - these should be serviced before lower priority even if they arrived later
- Two major ways to implement a priority queue
  - insert items in a sorted order
    - always remove the head
  - insert in unordered order and search list on remove
    - always add to the tail
  - either way, time is O(n)
    - either adding data takes time and removing is quick, or
    - adding data is quick and removing takes time

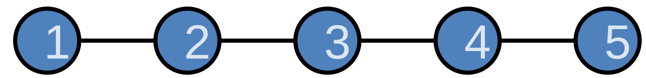
List-based Priority Queue

- <u>Unsorted list implementation</u>
  - Store the items of the priority queue in a list-based sequence, in arbitrary order



- Performance:
  - Enqueue takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - Dequeue take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- <u>sorted list implementation</u>
  - Store the items of the priority queue in a sequence, sorted by key



- Performance:
  - enqueue takes $O(n)$ time since we have to find the place where to insert the item
  - dequeue take $O(1)$ time since the smallest key is at the beginning of the sequence

Priority Queue

```
#define PQMAX 30

class PriorityQueue {
public:
   PriorityQueue()
  {
         noElements =0;
         rear = -1;
   }
   ~PriorityQueue() {}
    int isfull(void);
    int enqueue(int p);
    int dequeue();
    int length();
Private:
   int nodes[PQMAX];
   int rear;
   int noElements;

};
```

Check Priority Queue is Full

```
int PriorityQueue::isfull(void)
  {
        return (noElements ==
  PQMAX ) ? 1 : 0;
  }
```

Priority Queue

```
int PriorityQueue::dequeue()
{
    if( noElements> 0 )
    {
          int e = nodes[0];
           for(int j=0; j < noElements -2; j++ )
                  nodes[j] = nodes[j+1];

           noElements = noElements - 1;
           rear=rear-1;
           if(noElements == 0 )
                  rear = -1;

           return e;
    }
    cout << "queue is empty." << endl;

    return -1;
}
```

Priority Queue

```
int PriorityQueue::enqueue(int e)
{
    if( !isfull() ) {
            rear = rear+1;
            nodes[rear] = e;
            noElements = noElements + 1;
            sortElements(); // in descending order
            return 1;
    }
    cout << "insert queue is full." << endl;
    return 0;
}

int PriorityQueue ::length() { return noElements; }
```

Lecture content adapted from Michael T. Goodrich textbook, chapters 5.