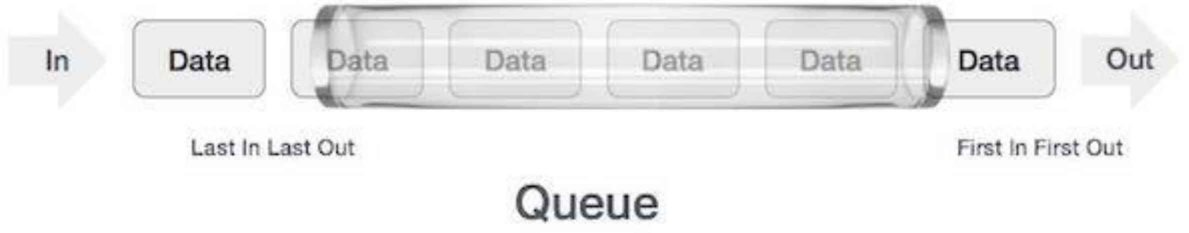
Chapter 3

Queue

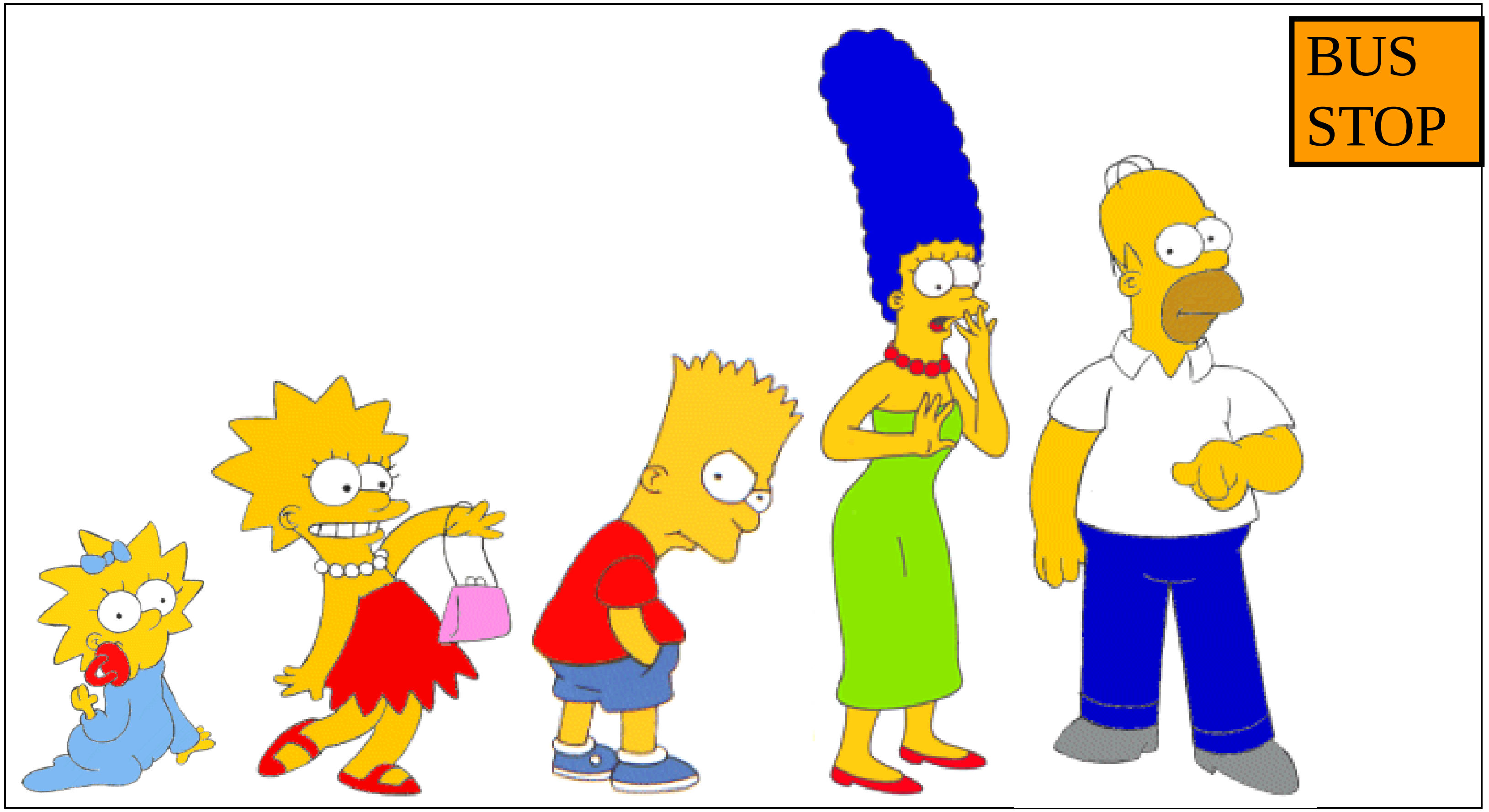
Basic Concepts

- A **queue** an ADT. It is an ordered collection of items from which items may be deleted at one end (called **front** of the queue) and into which items may be inserted at the other end (called the **rear** of the queue).
- Queue is also called a **first-in first-out** (**FIFO**) because the first element inserted into a queue is the first element to be removed.
- For example, print server (maintains a queue of print jobs), disk driver (maintains a queue of disk input/output requests) etc.



Basic Concepts

Fig: Queue of persons



Queue Operations

- Two primary operations of the stack are **enqueue** and **dequeue**. In queue terminology, **enqueue** is the insertion operation and **dequeue** is the removal operation. To use queue efficiently, we also need:
  - **peek** – gets the element at the front of the queue, without removing it.
  - **isfull** – checks if queue is full.
  - **isempty** – checks if queue is empty.
- The result of an illegal attempt to **dequeue** or **peek** an item from an empty queue is called **underflow**. Array implementation may introduce **overflow** if the queue grows larger than the size of the array.

Example: Demerging

- Consider a file of person records, each of which contains a person’s name, gender, date-of-birth, etc. The records are sorted by date-of-birth. We are required to rearrange the records such that females precede males but they remain sorted by date-of-birth within each gender group.

Example: Demerging

- **Demerging algorithm**

  To rearrange a file of person records such that females precede males but their order is otherwise unchanged:

```
1.    Make queues *females* and *males* empty.
2.    Until the input file is empty, repeat:
      2.1.   Let *p* be the next person read in from the file.
      2.2.   If *p* is female, add *p* to the rear of *females*.
      2.3.   If *p* is male, add *p* to the rear of *males*.
3.    Until *females* is empty, repeat:
      3.1.   Write out the person removed from the front of *females*.
4.    Until *males* is empty, repeat:
      4.1.   Write out the person removed from the front of *males*.
5.    Terminate.
```

Queue Implementation

- When elements are added to the queue, **rear** of the queue moves rightward. When elements are removed from the queue, its **front** also moves rightward. Hence, adding an element will increment rear and removing an element will increment front. Such queue is called **linear queue** and we cannot insert items into the queue even if there is space in the queue.
- One solution is to shift whole queue leftward. Since, shift operation takes more time, a much more efficient solution is to store the new element in the leftmost array component, that is, we treat the array as if it were cyclic. Such queue is called **circular queue**.

Linear Queue Implementation

```
#include<stdio.h>
#define MAXSIZE 5
int queue[MAXSIZE];
int front = -1, rear = -1;
int isempty()
{
        if(rear == -1)
                return 1;
        else
                return 0;
}
int isfull()
{
        if(rear == MAXSIZE - 1)
                return 1;
        else
                return 0;
}
```

```
void enqueue(int data)
{
        if(isfull())
                printf("Queue is full.\n");
        else
        {
                if(isempty())
                {
                        rear = front = 0;
                        queue[0] = data;
                }
                else
                        queue[++rear] = data;
        }
}
```

Linear Queue Implementation

```
int dequeue()
{
        int data;
        if(isempty())
                printf("Queue is empty.\n");
        else
        {
                data = queue[front++];
                if(front > rear)
                        front = rear = -1;
                return data;
        }
}
int peek()
{
        if(isempty())
                printf("Queue is empty.\n");
```

Linear Queue Implementation

```
            else
                        return queue[front];
}
int main()
{
            enqueue(3);
            enqueue(5);
            enqueue(9);
            enqueue(1);
            enqueue(12);
            printf("Element at front of the queue: %d\n", peek());
            printf("Removed element: %d\n", dequeue());
            printf("Removed element: %d\n", dequeue());
            enqueue(15);
            printf("Element at front of the queue: %d\n", peek());

}
```

Circular Queue Implementation

```
#include <stdio.h>
#define MAXSIZE 5
int queue[MAXSIZE];
int front = -1;
int rear = -1;
int isempty()
{
        if(rear == -1)
                return 1;
        else
                return 0;
}
int isfull()
{
        if((rear + 1) % MAXSIZE == front)
                return 1;
        else
                return 0;
}
```

Circular Queue Implementation

```
void enqueue(int data)
{
        if(isfull())
                printf("Queue is full.\n");
        else
        {
                if(isempty())
                {
                        rear = front = 0;
                        queue[0] = data;
                }
                else
                {
                        rear = (rear + 1) % MAXSIZE;

                        queue[rear] = data;
                }
        }
}
```

Circular Queue Implementation

```
int dequeue()
{
        int data;
        if(isempty())
                printf("Queue is empty.\n");
        else
        {
                data = queue[front];
        if(front == rear)
                        front = rear = -1;
                else
                        front = (front + 1) % MAXSIZE;

        }
        return data;
}
```

Circular Queue Implementation

```
int peek()
{
        if(isempty())
                printf("Queue is empty.\n");
        else
                return queue[front];
}
int main()
{
        enqueue(3);
        enqueue(5);
        enqueue(9);
        enqueue(1);
        enqueue(12);
        printf("Element at front of the queue: %d\n", peek());
        printf("Removed element: %d\n", dequeue());
        printf("Removed element: %d\n", dequeue());
        enqueue(15);
        printf("Element at front of the queue: %d\n", peek());
}
```

Priority Queue

- Elements in both stack and queue are ordered based on the sequence in which they have been inserted.
- Priority queue is an ADT in which the intrinsic ordering of elements does determine the results of its basic operations.
- There are two types of priority queues: an **ascending priority queue** and a **descending priority queue**.
- An **ascending priority queue** is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed.
- A **descending priority queue** is similar but allows deletion of only the largest item.

Priority Queue

- There are different array implementations of priority queue:
  - Insert item in the queue as before. Under this insertion method, elements of the priority queue are not kept ordered in the array. To remove an item from ascending priority queue, first locate the smallest element in the queue and remove it. This requires accessing every element of the priority queue. When rear of the queue reaches MAXSIZE the array elements are compacted into the front of the array. The same case happens for descending priority queue.
  - The new item is inserted in the first empty position. Insertion involves accessing every array element to find the first empty position.

Priority Queue

- Each deletion can compact the array by shifting all elements by one position.
- Instead of maintaining priority queue as an unsorted array, maintain it as ordered (ascending for ascending priority queue and descending for descending priority queue respectively) circular array. Insertion requires locating the proper position of the new element and shifting the preceding or succeeding elements.

Queue Applications

- Queues turn out to be enormously useful in computing, particularly in system programming. Some common applications are:
  - A print server maintains a queue of print jobs.
  - A disk driver maintains a queue of disk access requests.
  - An operating system's schedular maintains a queue of processes awaiting a slice of processor time.