DATA STRUCTURES AND ALGORITHMS

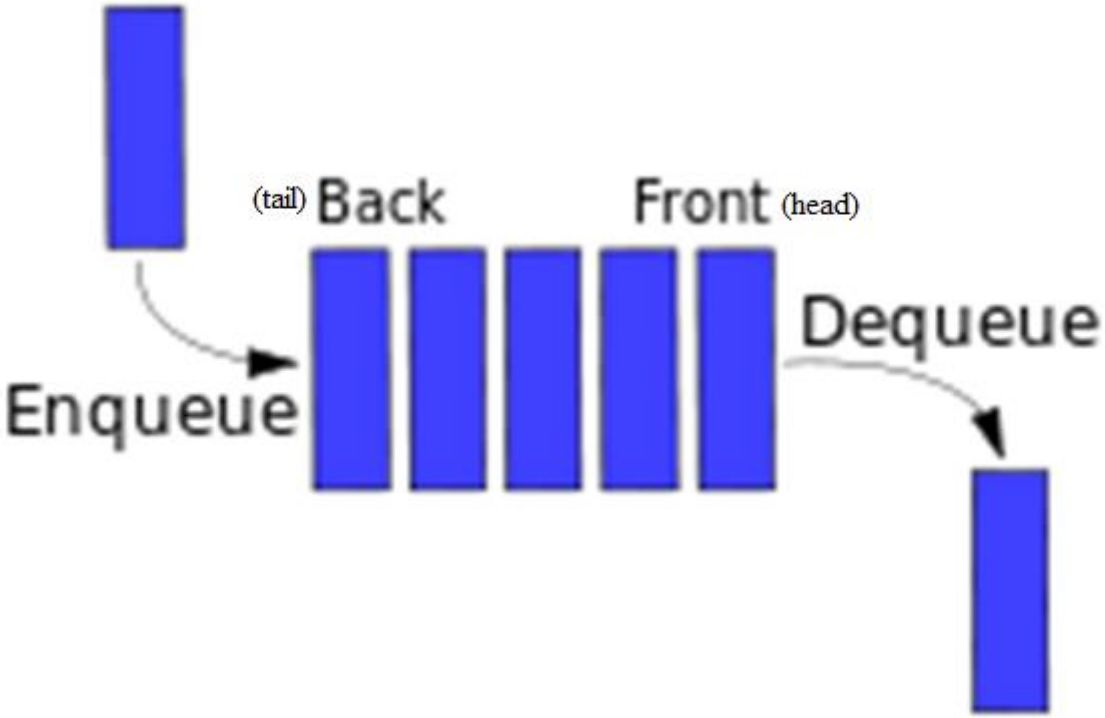
FILS, March 2018

ROADMAP

- Queue
- Queue vs stack
- Applications

QUEUE

- instance of an ADT that formalizes the concept of FIFO (first in, first out)



BASIC OPERATIONS

- Enqueue(x)
  - Adds the element x at the **tail** of the queue
- Dequeue()
  - Removes the element from the **head** of the queue and returns it
  - Returns an error if the stack is empty
- Peek()
  - Returns (but does not remove) the element at the head of the queue
- isEmpty()
  - Returns 1 if the queue is empty and 0 otherwise

VARIANTS

- Deque/dequeue/ double-ended queue:
  - Individual elements can be accessed by their position index.
  - Iteration over the elements can be performed in any order.
  - Elements can be efficiently added and removed from any of its ends (either the beginning or the end of the sequence).

- Priority queue:
  - Each element has an attached priority
  - The basic operations are:
    - enqueue– adds the element with a specified priority to the tail of the queue
    - Dequeue- removes the element having the greatest priority
    - Front - Returns (but does not remove) the element having the greatest priority

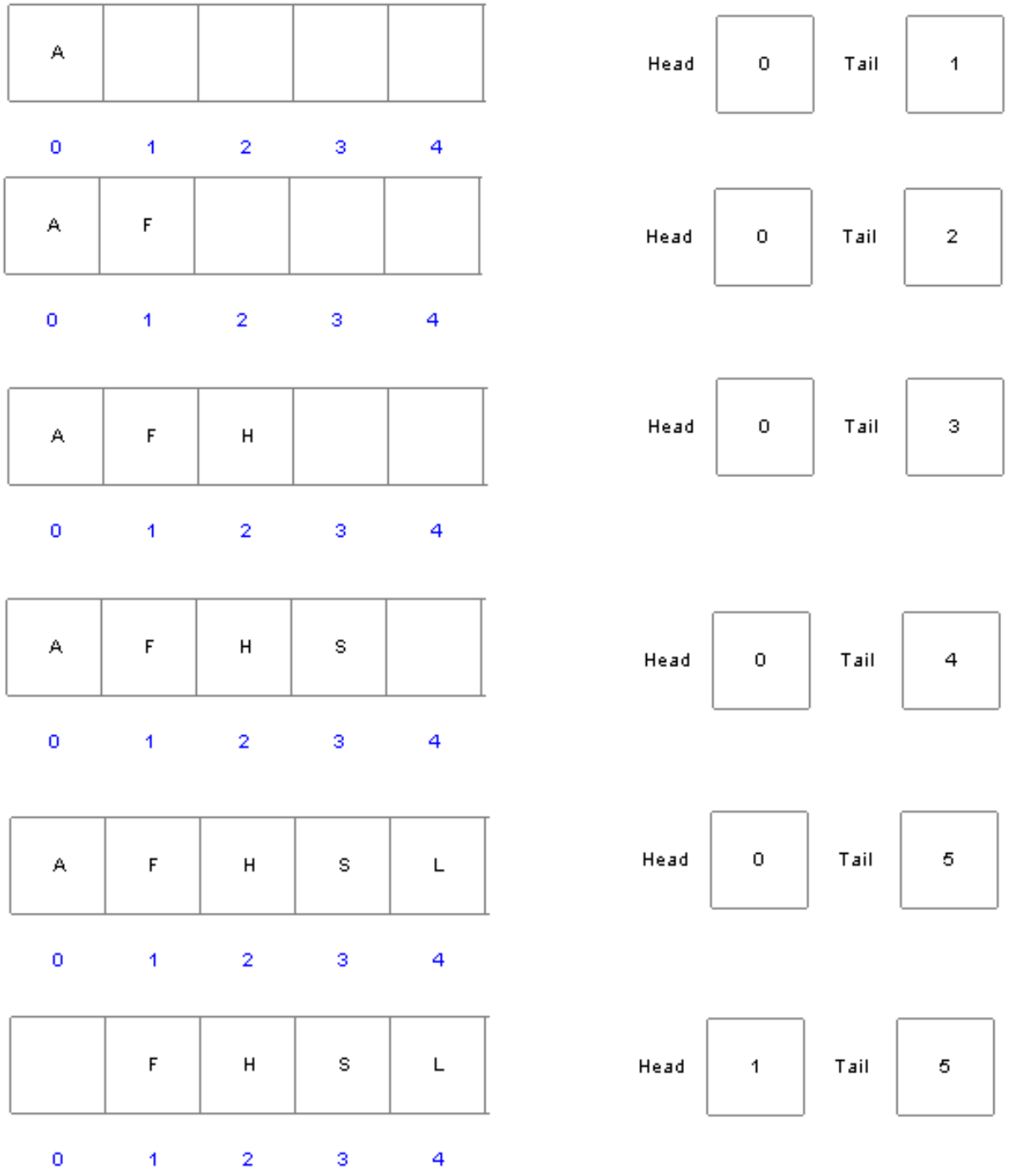
IMPLEMENTATIONS OF QUEUES

- With a static data structure (array, circular array)
- With a dynamic data structure (list)

Queue: Array-Based Implementation

- The queue is stored in an array
- The array indices at which the head and the tail of the queue are currently stored must be maintained
- The head of the queue is not necessary at index 0
- The array can be a circular array: the queue "wraps" round if the last index of the array is reached

EXAMPLE: STORING THE QUEUE IN AN ARRAY OF 5 CHARS



QUEUE1.CPP (1)

```
1  #include <stdio.h>
2  #define NMAX 100
3
4  template<typename T> class Queue {
5      private:
6          T queueArray[NMAX];
7          int head, tail;
8      public:
9          void enqueue(T x) {
10             if (tail >= NMAX) {
11                 fprintf(stderr, "Error 101 - The queue is full!\n");
12                 return;
13             }
14             queueArray[tail] = x;
15             tail++;
16         }
17
18         T dequeue() {
19             if (isEmpty()) {
20                 fprintf(stderr, "Error 102 - The queue is empty!\n");
21                 T x;
22                 return x;
23             }
24             T x = queueArray[head];
25             head++;
26             return x;
27 }
```

QUEUE1.CPP (2)

```
28
29          T peek() {
30              if (isEmpty()) {
31                  fprintf(stderr, "Error 103 - The queue is empty!\n");
32                  T x;
33                  return x;
34              }
35              return queueArray[head];
36          }
37
38          int isEmpty() {
39              return (head == tail);
40          }
41
42      Queue() {
43          head = tail = 0; // the queue is empty in the beginning
44      }
45  };
46
47  int main() {
48
49      Queue<char> q;
50
51      q.enqueue('A'); q.enqueue('F'); q.enqueue('H'); q.enqueue('S');
52      q.enqueue('L'); q.dequeue(); q.dequeue();
53      printf("%c\n", q.peek());
54      return 0;
55  }
56
```

???

- For the previous queue (called q), what will be the results for the following operations:
  - q.dequeue();
  - printf("%c\n", q.peek());
  - printf("head: %d\n", q.head);
  - printf("tail: %d\n", q.tail);

- But the following problem arises in terms of unused space: always space from 0 to head-1 will be useless, and the number of elements that can be stored in the queue will decrease (initially N elements can be store, then after extracting the first element, only N-1 elements can be stored). We want to be able to store always up to N elements.

CIRCULAR ARRAY-BASED QUEUE

- The queue is represented through a template class which has, besides the methods containing the basic operations, the following fields:
  - An array: queryArray
  - A maximum dimension of this array (the second parameter of the template class!!!)
  - The tail
  - The head
  - The size (to know when the queue is full or not)

TEMPLATE CLASS FOR QUEUE EXERCISE 1

```
1  template <typename T, int N>
2  class Queue {
3  private:
4      int head;
5      int tail;
6      int size;
7      T queueArray[N];
8
9  public:
10     // Constructor
11     Queue();
12
13     // Destructor
14     ~Queue();
15
16     void enqueue(T e);
17     T dequeue();
18     T peek();
19     bool isEmpty();
20 };
```

```
21
22 template <typename T, int N>
23 Queue<T, N>::Queue() {
24     //TODO:
25 }
26
27 template <typename T, int N>
28 Queue<T, N>::~Queue() {
29     //TODO:
30 }
31
32 template <typename T, int N>
33 void Queue<T, N>::enqueue(T e) {
34     //TODO:
35 }
36
37 template <typename T, int N>
38 T Queue<T, N>::dequeue() {
39     //TODO:
40 }
41
```

APPLICATIONS OF QUEUES

- Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

- Computer systems must often provide a “holding area” for messages between two processes, two programs, or even two systems. This holding area is usually called a “buffer” and is often implemented as a queue.

EXERCISE 2

- Search in a queue how many increasing sequences of numbers you have.

EXERCISE 3

- Find out if the queue contains a Fibonacci sequence.

HOMEWORK

- **Ex. 1** Implement a queue with 2 stacks. Your stack will be named QueuedStack and it will have: an empty constructor; an empty destructor; a pop method; a push method. ! There are several solutions for this task.

- **Ex. 2** Make a messaging system.
- Messages are received in the order they are sent
- The classes involved are:
  - Message
  - MessageSender
  - MessageReceiver
  - A Message object has a sender, recipient, content string (make an array of chars!!) and a date
  - A Message is placed in a queue by a MessageSender object
  - A Message is removed from a queue by a MessageReceiver object, which can also displays the content of the queue
  - Your queue class can receive any types of objects, including Message Objects