Queues
Queues

• A queue is a special type of structure that can be used to maintain data in an organized way.

• This data structure is commonly implemented in one of two ways: as an array or as a linked list.

• In either case, the important rule is that when data is added to the queue, it is tacked onto the end, and so if an element needs to be removed, the element at the front is the only element that can legally be removed.
  • First in, first out (FIFO)
Queues

• There are only two operations that may legally be performed on a queue.

  • *Enqueue*: Add a new element to the end of the queue.

  • *Dequeue*: Remove the oldest element from the front of the queue.
Queue-based implementation

typedef struct _queue {
    VALUE array[CAPACITY];
    int front;
    int size;
} queue;
Queues

• Array-based implementation

typedef struct _queue
{
    VALUE array[CAPACITY];
    int front;
    int size;
}
queue;
Queues

• Array-based implementation

```c
typedef struct _queue
{
    VALUE array[CAPACITY];
    int front;
    int size;
}
queue;
```
Queues

• Array-based implementation

```
typedef struct _queue
{
    VALUE array[CAPACITY];
    int front;
    int size;
}
queue;
```
Queues

• Array-based implementation

```c
typedef struct _queue
{
    VALUE array[CAPACITY];
    int front;
    int size;
}
queue;
```
Queues

• Array-based implementation

queue q;
Queues

- Array-based implementation

```java
queue q;
```
Queues

• Array-based implementation

```c
queue q;
q.front = 0;
q.size = 0;
```
Queues

- Array-based implementation

```c
queue q;
q.front = 0;
q.size = 0;
```
Queues

• Array-based implementation
  • **Enqueue**: Add a new element to the end of the queue.

In the general case, `enqueue()` needs to:
  • Accept a pointer to the queue.
  • Accept data of type `VALUE` to be added to the queue.
  • Add that data to the queue at the end of the queue.
  • Change the size of the queue.
Queues

- Array-based implementation

```c
void enqueue(queue* q, VALUE data);
```

![Diagram of a queue with elements 0 and 0]
Queues

- Array-based implementation

enqueue(&q, 28);
Queues

• Array-based implementation

enqueue(&q, 28);
Queues

• Array-based implementation

enqueue(&q, 28);
Queues

• Array-based implementation

enqueue(&q, 33);
Queues

- Array-based implementation

enqueue(&q, 33);
Queues

- Array-based implementation

```c
enqueue(&q, 33);
```
Queues

- Array-based implementation

enqueue(&q, 19);
Queues

• Array-based implementation

enqueue(&q, 19);
Queues

- Array-based implementation

```c
enqueue(&q, 19);
```
Queues

• Array-based implementation
  • Dequeue: Remove the most recent element from the front of the queue.

In the general case, dequeue() needs to:
  • Accept a pointer to the queue.
  • Change the location of the front of the queue.
  • Decrease the size of the queue.
  • Return the value that was removed from the queue.
Queues

- Array-based implementation

VALUE dequeue(queue* q);
Queues

- Array-based implementation

```c
int x = dequeue(&q);
```
Queues

- Array-based implementation

```c
int x = dequeue(&q);
```
Queues

- Array-based implementation

```c
int x = dequeue(&q);
```
Queues

• Array-based implementation

```c
int x = dequeue(&q);
```

![Queues diagram with an array-based implementation](image)
Queues

• Array-based implementation

```c
int x = dequeue(&q);
```
Queues

• Array-based implementation

```c
int x = dequeue(&q);
```

![Queue implementation diagram]

- Arrays are used to implement queues.
- The dequeue operation removes elements from the queue.
Queues

• Array-based implementation

```c
int x = dequeue(&q);
```

![Queue Array-based Implementation Diagram](image-url)
Queues

- Array-based implementation

```c
enqueue(&q, 40);
```

![Queue diagram with elements 28, 33, 19, 2, and 1]
Queues

- Array-based implementation

```c
enqueue(&q, 40);
```

![Queue diagram with elements 28, 33, 19, 40, and 2, 1]
Queues

• Array-based implementation

enqueue(&q, 40);
Queues

- Linked list-based implementation

```c
typedef struct _queue
{
    VALUE val;
    struct _queue *prev;
    struct _queue *next;
} queue;
```
Queues

• Just make sure to always maintain pointers to the head *and* tail of the linked list! (probably global)

• To **enqueue**:
  • Dynamically allocate a new node;
  • Set its next pointer to NULL, set its prev pointer to the tail;
  • Set the tail’s next pointer to the new node;
  • Move the tail pointer to the newly-created node.
Queues

enqueue(tail, 10);
Queues

enqueue(tail, 10);
Queues

enqueue(tail, 10);

head

12

15

9

13

tail

new

10
Queues

enqueue(tail, 10);
Queues

enqueue(tail, 10);
Queues

enqueue(tail, 10);
Queues

• **To dequeue:**
  • Traverse the linked list to its second element (if it exists);
  • Free the head of the list;
  • Move the head pointer to the (former) second element;
  • Make that node’s `prev` pointer point to NULL.
Queues
dequeue(head);
Queues

dequeue(head);

head

12

15

9

13

10
tail

trav
Queues

depqueue(head);

head

12

15

9

13

10
tail

trav
Queues

dequeue(head);

head

current

tail

dequeue(head);
Queues

dequeue(head);

head

15

9

13

10
tail

dequeue(head);

trav
Queues

dequeue(head);

head

trav

15

9

13

10
tail