Prof. Darrell Long

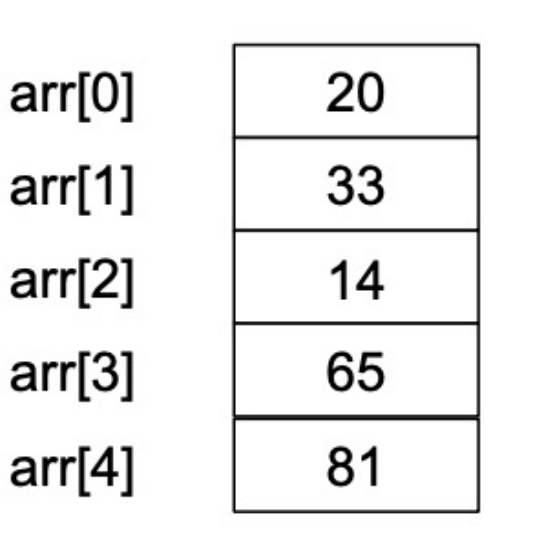
CSE13

Stacks and Queues

Arrays

Arrays allow random access:

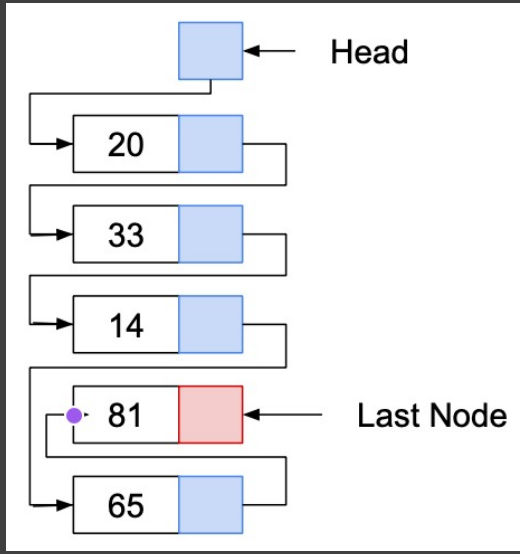
- Each element can be accessed directly in constant time.



Linked Lists

Linked lists allow sequential access:

- Each element can only be accessed in a particular order.



Stacks and Queues

Operate as *containers* for other data types.

- For the most part, data of any type may be placed in a stack or a queue.

Both have:

- Well-defined semantics
- A fixed set of operations
- A distinct ordering to their elements

Stacks

- Objects are added and removed using last-in-first-out (LIFO) order.
- Capacity of a stack is the number of stack elements that will fit.
- This capacity can be increased as number of elements increases through reallocation (dynamic).
- Often implemented with arrays, but other possibilities exist as long as the semantics are preserved.

```c
#ifndef __STACK_H__
#define __STACK_H__

#include <stdbool.h>
#include <stdint.h>

typedef uint32_t item;

typedef struct Stack {
    uint32_t top;
    uint32_t capacity;
    item *items;
} Stack;

Stack *stack_create(void);

void stack_delete(Stack **s);

bool stack_empty(Stack *s);

bool stack_full(Stack *s);

bool stack_push(Stack *s, item i);

bool stack_pop(Stack *s, item *i);
```

stack_create()

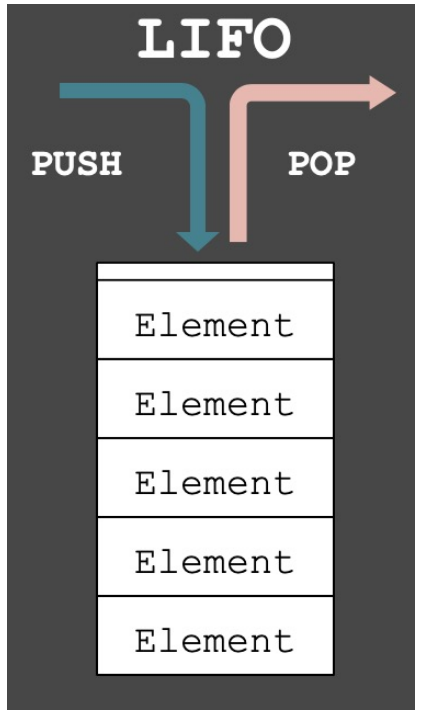
```c
#define MIN_CAPACITY 16

Stack *stack_create(void) {
    Stack *s = (Stack *)malloc(sizeof(Stack));
    if (s != NULL) {
        s->top = 0;
        s->capacity = MIN_CAPACITY;
        s->items = (item *)malloc(s->capacity * sizeof(item));
        if (s->items == NULL) {
            free(s);
            s = NULL;
        }
    }
    return s;
}
```

stack_full()

```
bool stack_full(Stack *s) {
    return s->top == s->capacity;
}
```

stack_empty()

```
bool stack_empty(Stack *s) {
    return s->top == 0;
}
```



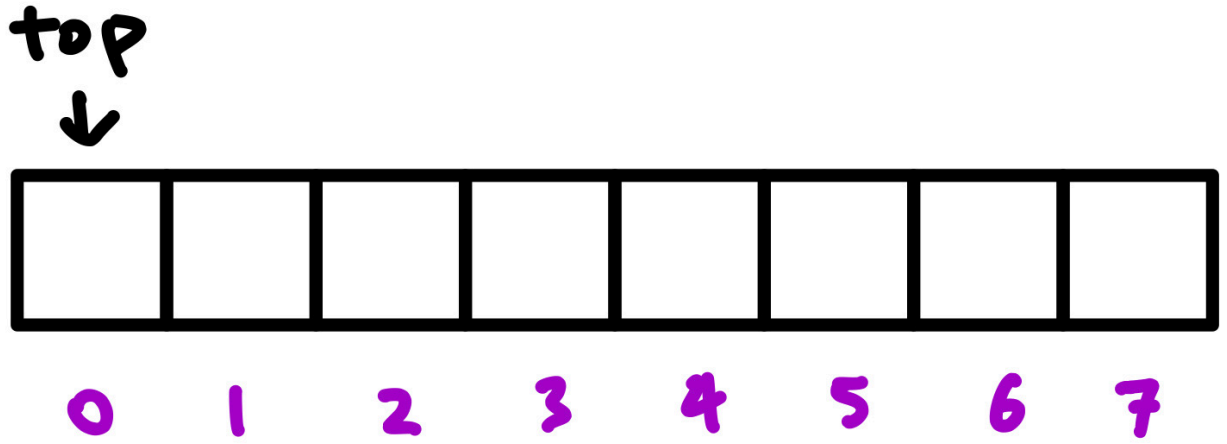

```c
bool stack_push(Stack *s, item i) {
    if (stack_full(s)) {
        s->capacity *= 2;
        s->items = (item *)realloc(s->items, s->capacity * sizeof(item));
    }
    s->items[s->top] = i;
    s->top += 1;
    return true;
}
```

stack_push()

- Adds an element to the top of a stack.
- Example is for a dynamic stack, in which memory is reallocated when the stack is “full”.

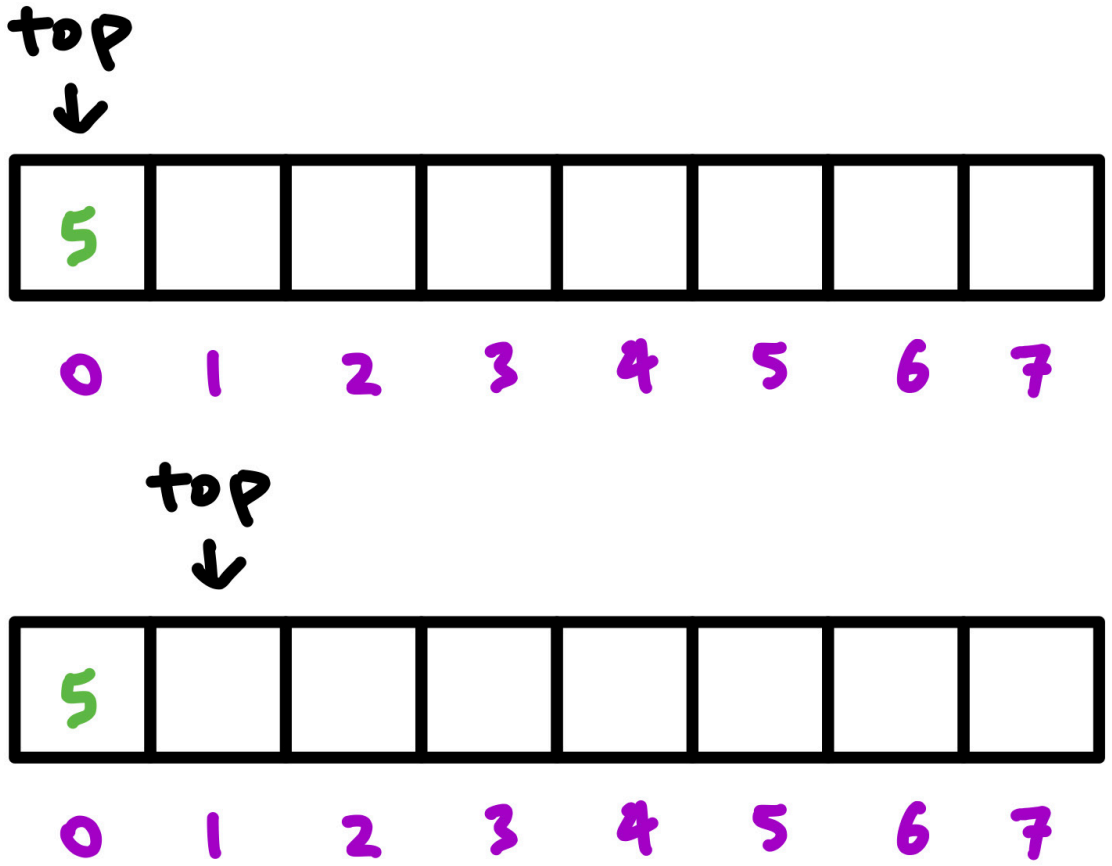
Example: Pushing items onto a stack

- The top of the stack is the index of the next available spot.



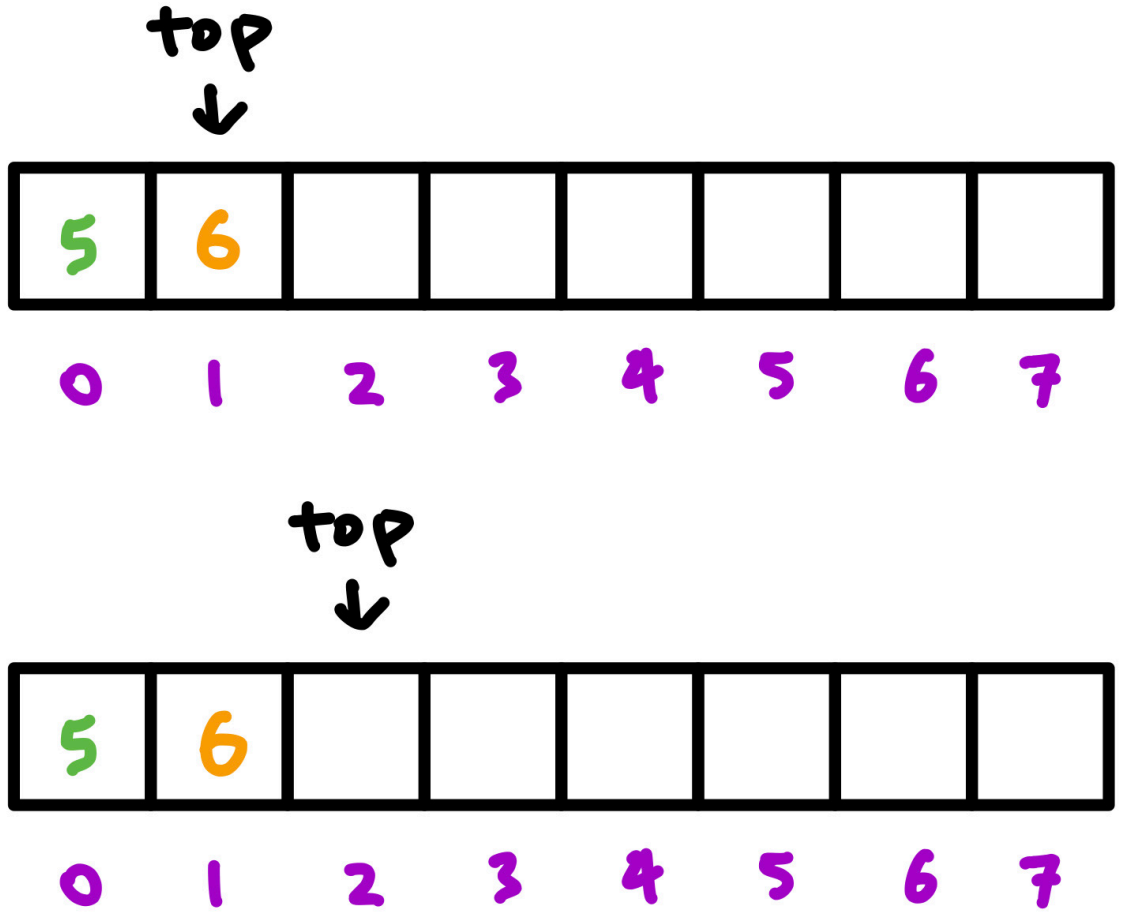
Example: Pushing items onto a stack

- Assign the value, move the top forward.



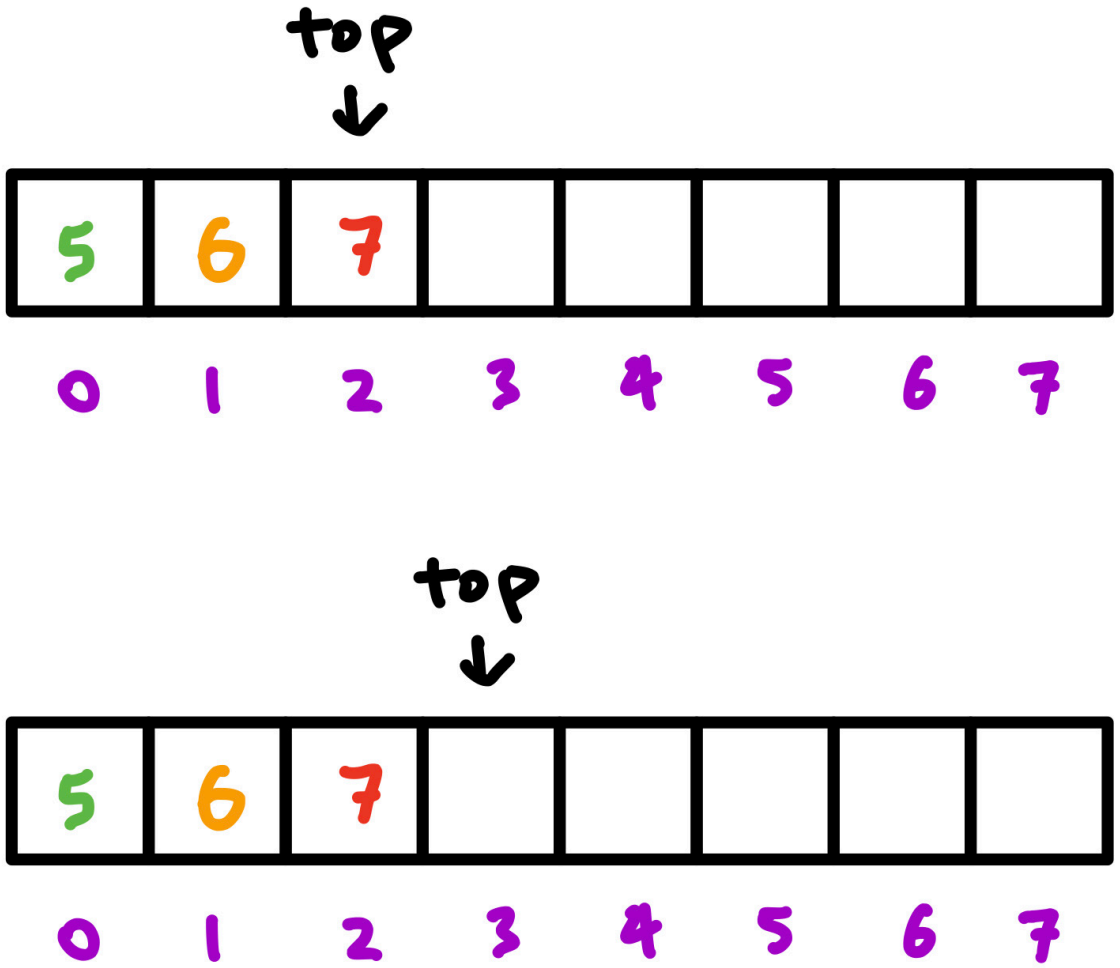
Example: Pushing items onto a stack

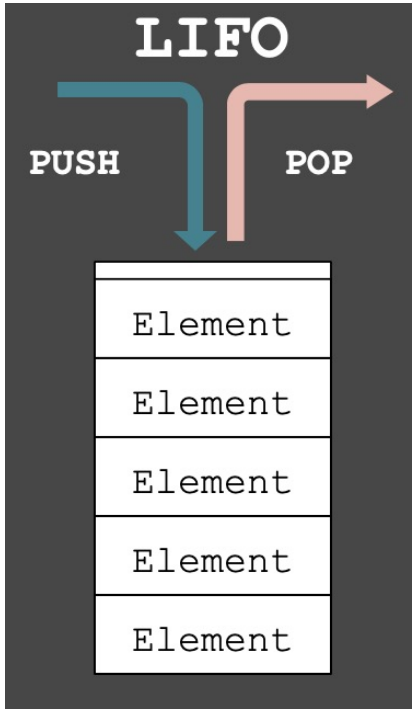
- Assign the value, move the top forward.



Example: Pushing items onto a stack

- Assign the value, move the top forward.





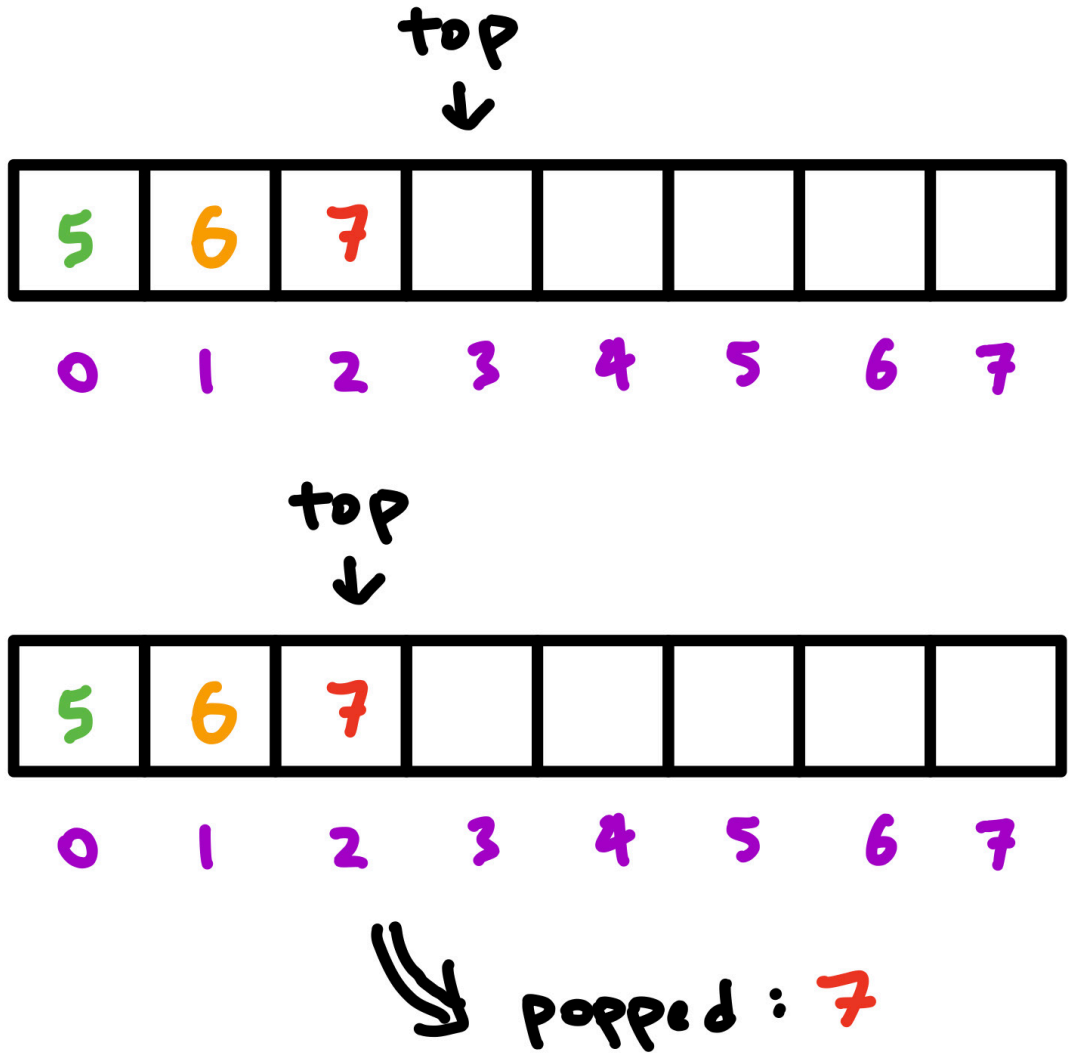

```c
bool stack_pop(Stack *s, item *i) {
    if (stack_empty(s)) {
        return false;
    }
    s->top -= 1;
    *i = s->items[s->top];
    return true;
}
```

stack_pop()

- Removes the element at the top of a stack.
- Returns false if the stack is empty.

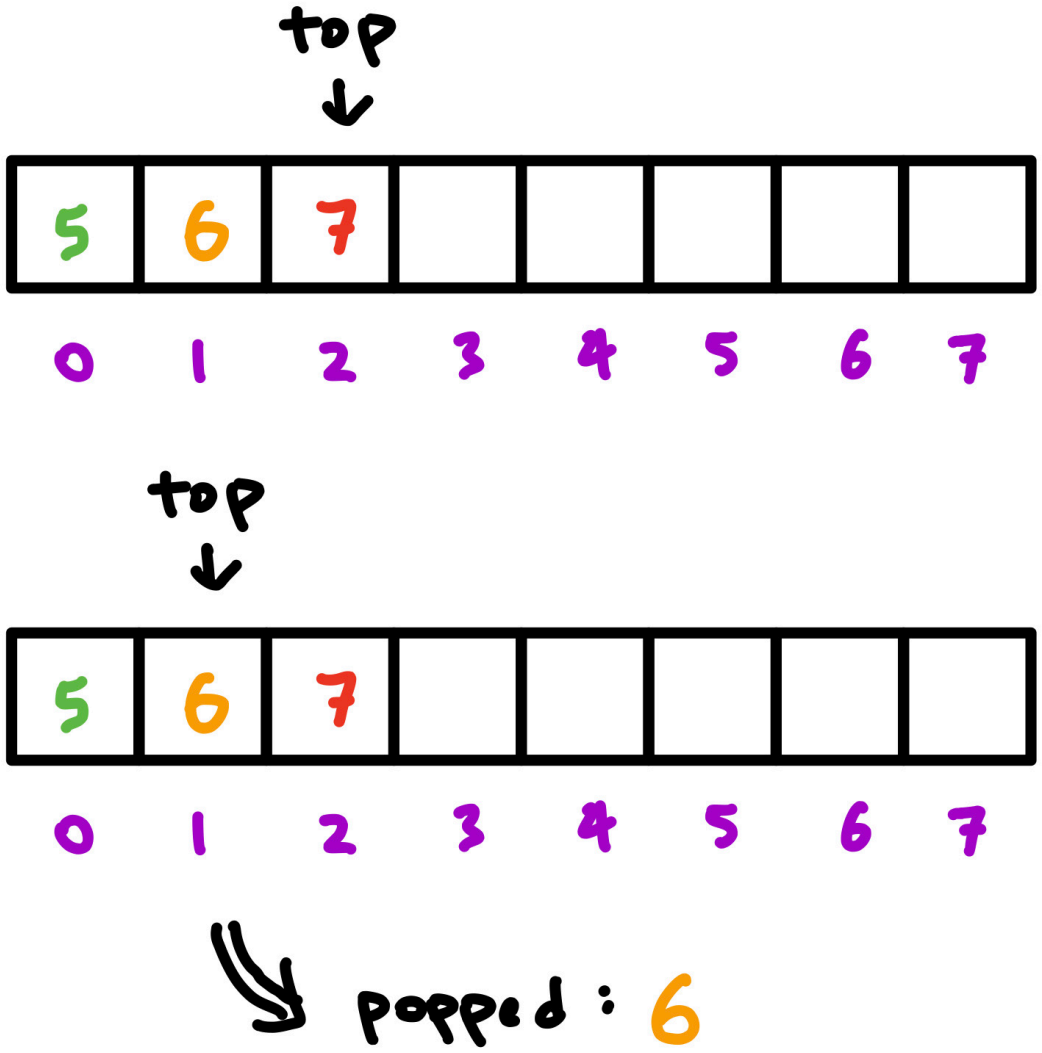
Example: Popping items off a stack

- Move the top back, pass back the value.



Example: Popping items off a stack

- Move the top back, pass back the value.



Stacks using linked lists

- Stacks can also be implemented using linked lists.
- Each element of the stack points to the next.

```c
#ifndef __STACK_H__
#define __STACK_H__

#include <stdbool.h>
#include <stdint.h>

typedef uint32_t item;

typedef struct Node Node;

struct Node {
    Node *next;
    item data;
};

typedef struct Stack {
    Node *top;
} Stack;

Stack *stack_create(void);

void stack_delete(Stack **s);

bool stack_empty(Stack *s);

bool stack_push(Stack *s, item i);

bool stack_pop(Stack *s, item *i);

#endif
```

stack_create()

```c
Stack *stack_create(void) {
    Stack *s = (Stack *)malloc(sizeof(Stack));
    if (s != NULL) {
        s->top = NULL;
    }
    return s;
}
```

stack_push()

```c
static Node *node_create(item i) {
    Node *n = (Node *)malloc(sizeof(Node));
    if (n) {
        n->data = i;
        n->next = NULL;
    }
    return n;
}

bool stack_push(Stack *s, item i) {
    Node *n = node_create(i);
    n->next = s->top;
    s->top = n;
    return true;
}
```

stack_pop()

```
static void node_delete(Node **n) {
    free(*n);
    *n = NULL;
}

bool stack_pop(Stack *s, item *i) {
    if (stack_empty(s)) {
        return false;
    }
    *i = s->top->data;
    Node *n = s->top;
    s->top = s->top->next;
    node_delete(&n);
    return true;
}
```

```c
#ifndef _QUEUE_H
#define _QUEUE_H

#include <stdbool.h>
#include <stdint.h>

typedef uint32_t item;

typedef struct queue {
    uint32_t head;
    uint32_t tail;
    uint32_t size;
    item *Q;
} queue;

queue *newQueue(uint32_t);

void delQueue(queue *);

bool emptyQ(queue *);

bool fullQ(queue *);

bool enqueue(queue *, item *);

bool dequeue(queue *, item *);
#endif
```

queue.h

- A queue is an abstract data type (ADT), so let's define it.
- For our implementation we need:
  - A head, a tail, and a size, and
  - A pointer to an array.
- A constructor
- A destructor
- Empty and full predicates
- Enqueue — inserts into the queue
- Dequeue — removes from the queue

```
queue *newQueue(uint32_t size) {
    queue *q = (queue *) malloc(sizeof(queue));
    if (q) {
        q->head = q->tail = 0;
        q->size = size;
        q->Q = (item *) calloc(size, sizeof(item));
        if (q->Q) {
            return q;
        }
        free(q);
    }
    return (queue *) 0;
}
```

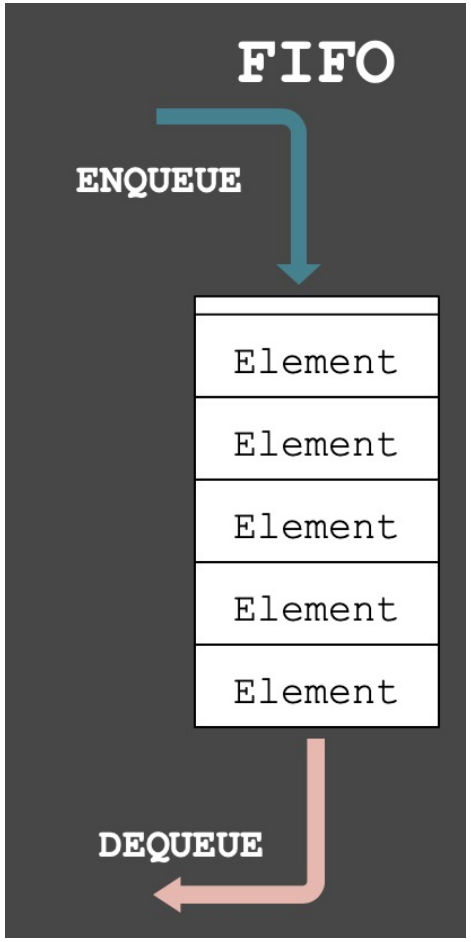
Create a Queue

Delete a Queue

```
void delQueue(queue *q) {
    if (q) {
        free(q->Q);
        free(q);
    }
}
```

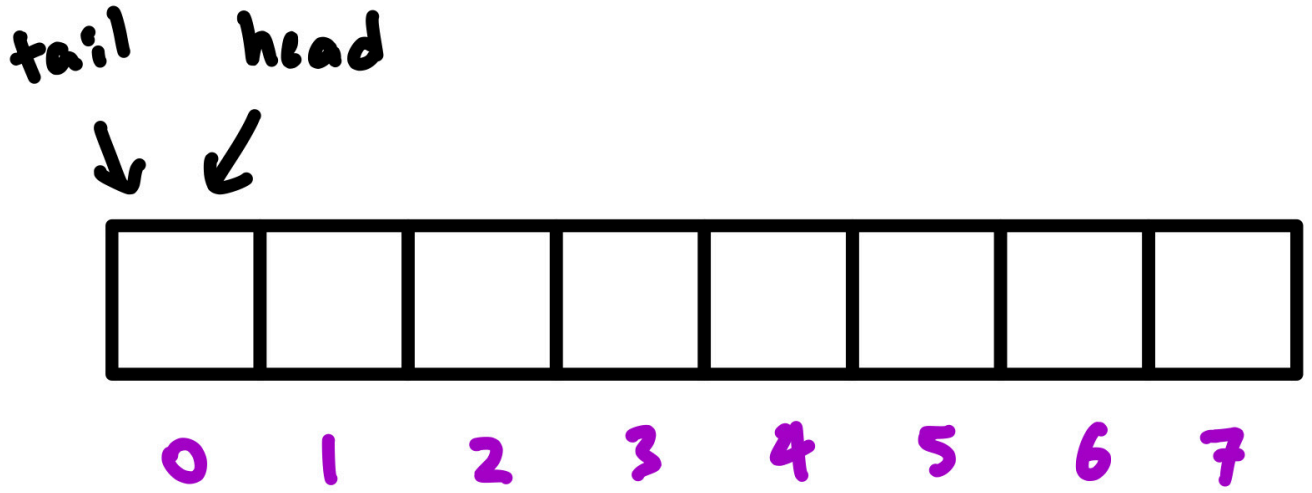
Enqueue an Item

```
bool enqueue(queue *q, item *i) {
    if (q) {
        if (fullQ(q)) {
            return false;
        }
        q->Q[q->head] = *i;
        q->head = succ(q, q->head);
        return true;
    } else {
        return false;
    }
}
```



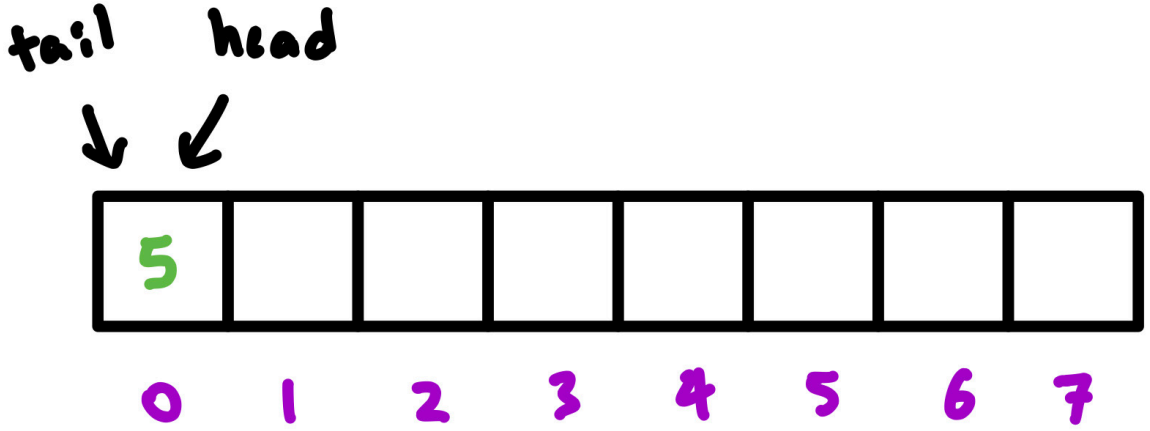
Example: Enqueuing items

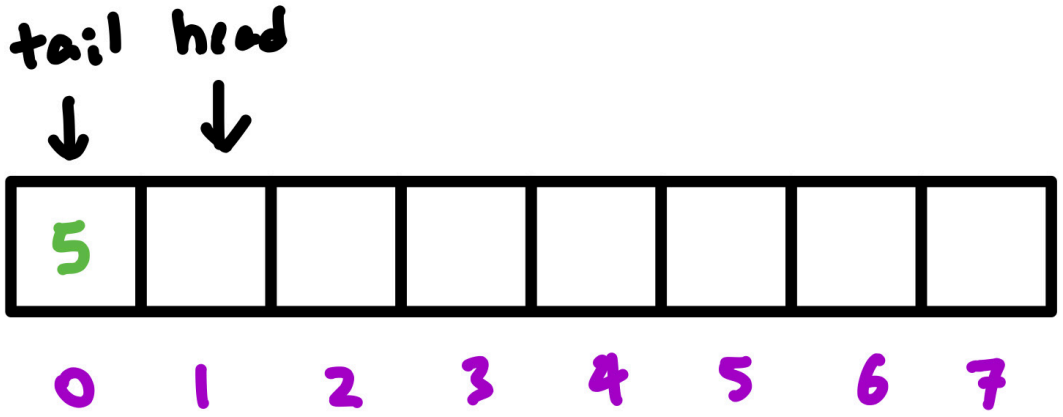
- The head is the index of the next available spot.
- The tail is the index of the item to dequeue.
- You can change these around.



Example: Enqueuing items

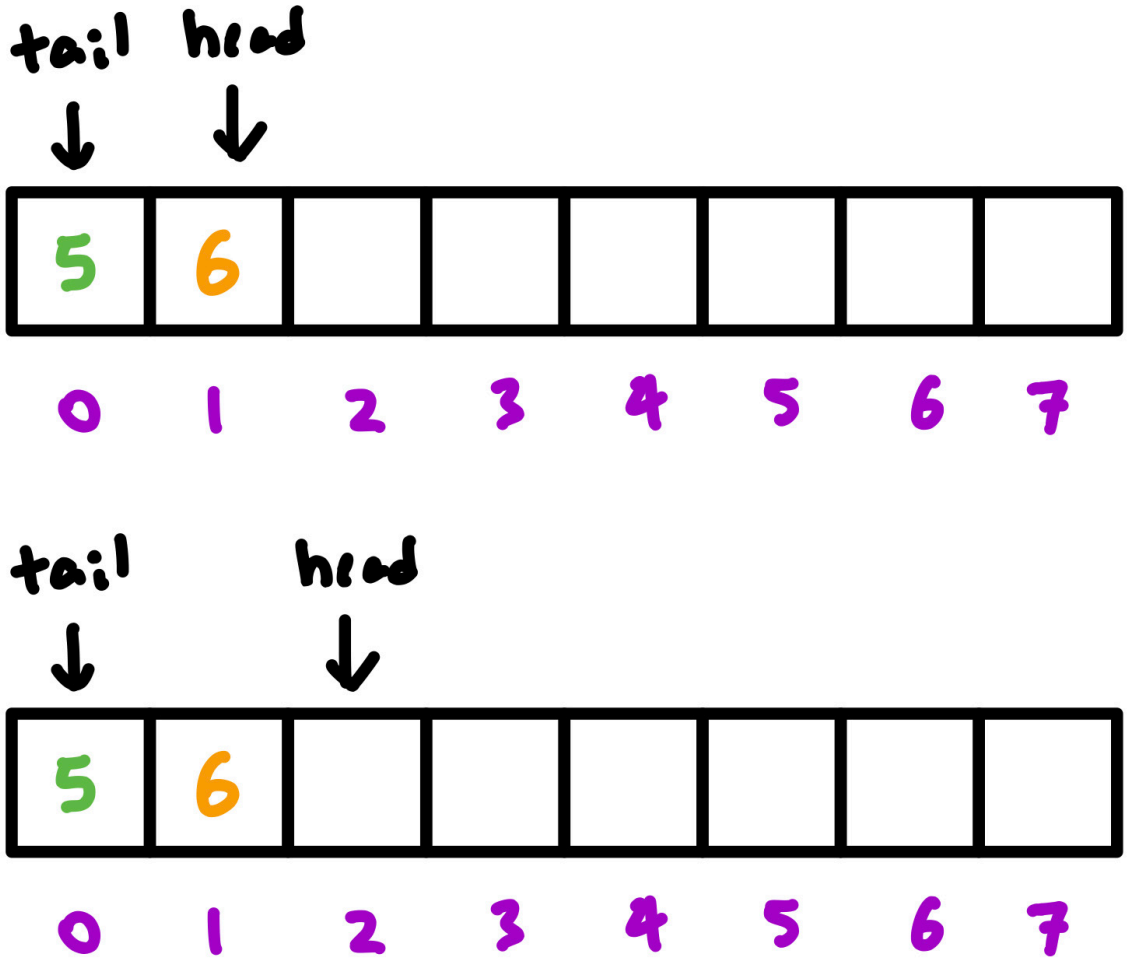
- Insert at the head, move the head forward.





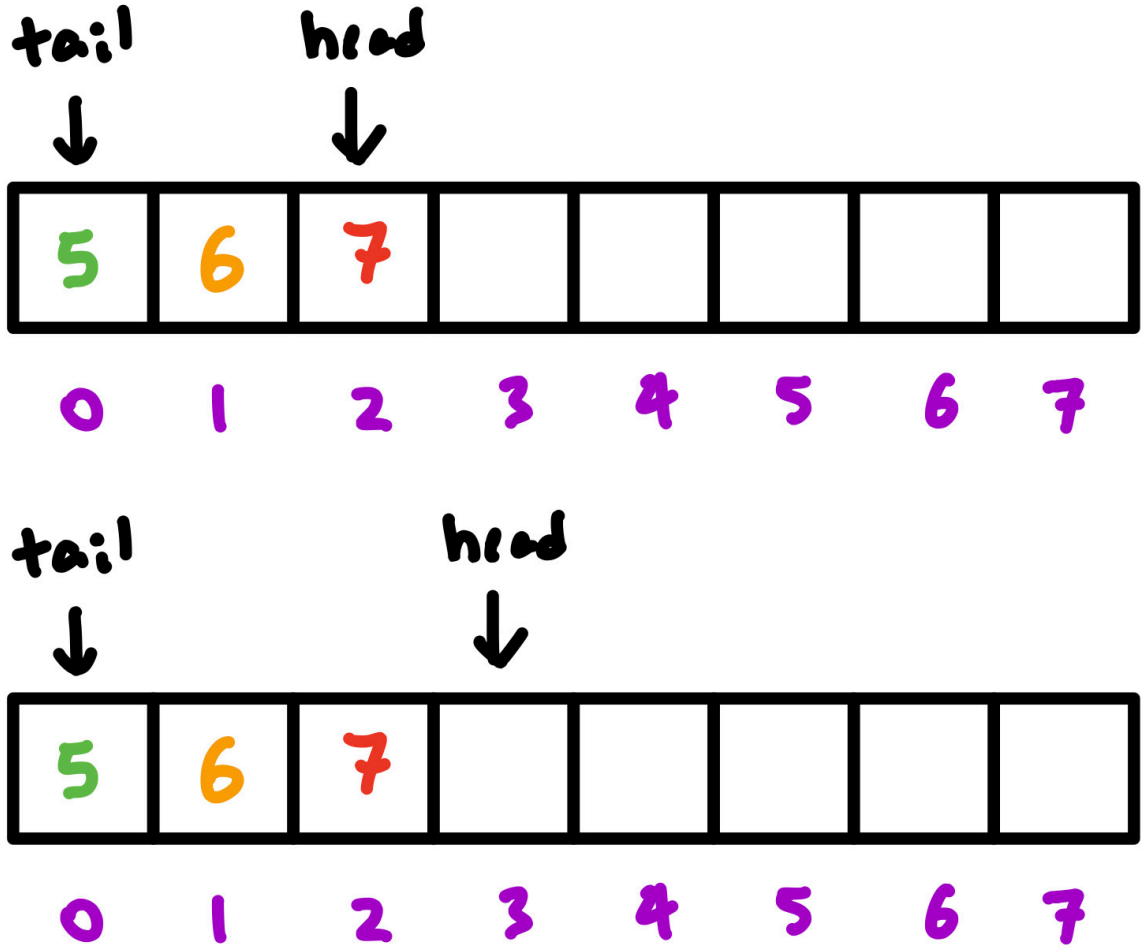
Example: Enqueuing items

- Insert at the head, move the head forward.



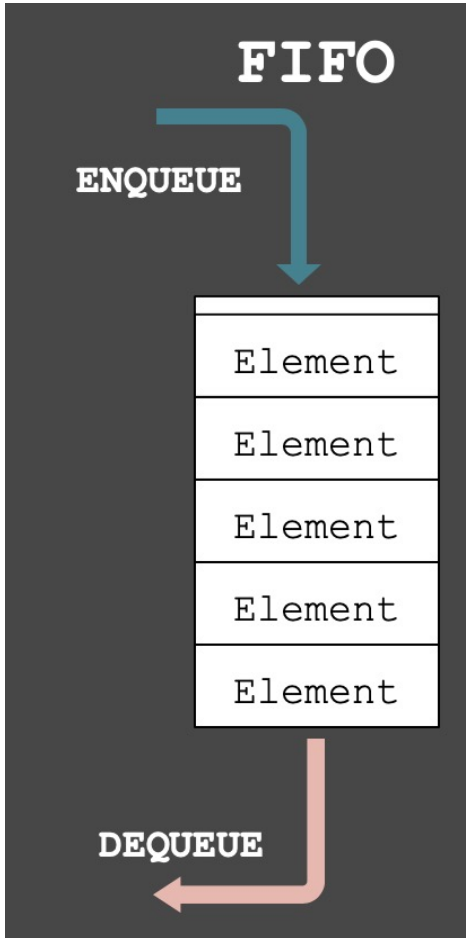
Example: Enqueuing items

- Insert at the head, move the head forward.



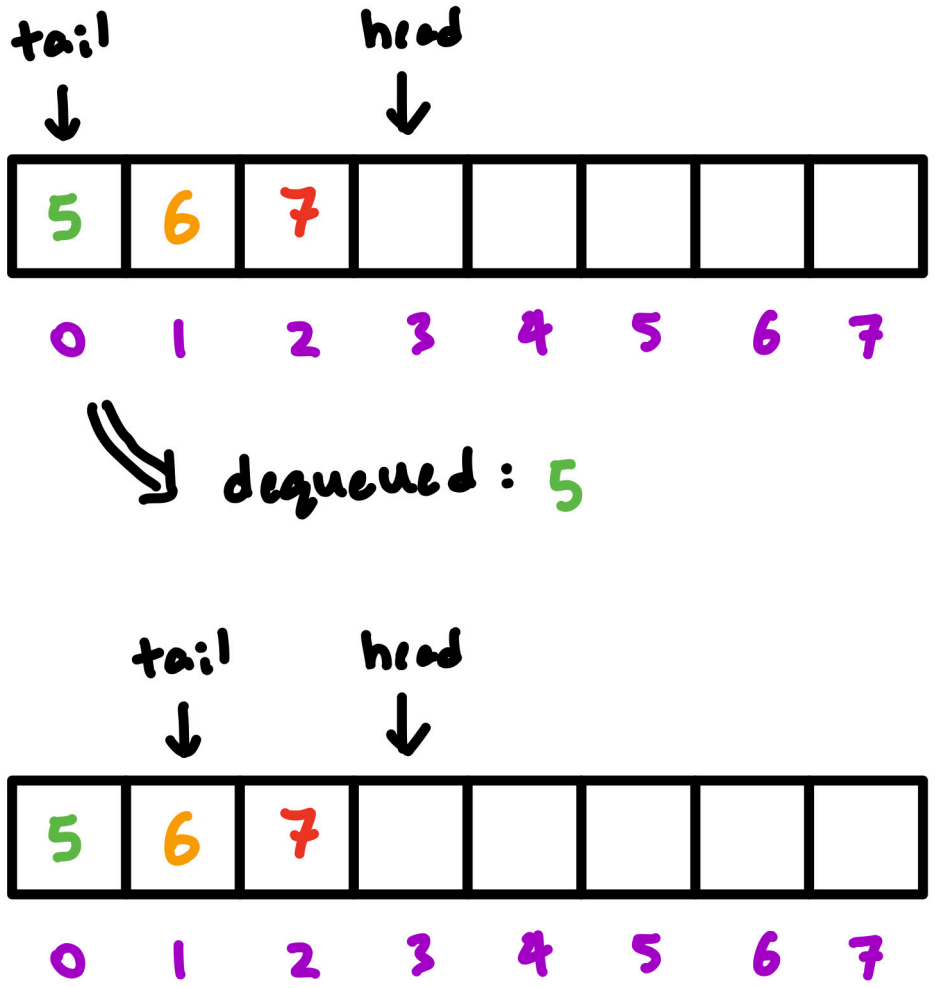
Dequeue an Item

```
bool dequeue(queue *q, item *i) {
    if (q) {
        if (emptyQ(q)) {
            return false;
        }
        *i = q->Q[q->tail];
        q->tail = succ(q, q->tail);
        return true;
    } else {
        return false;
    }
}
```



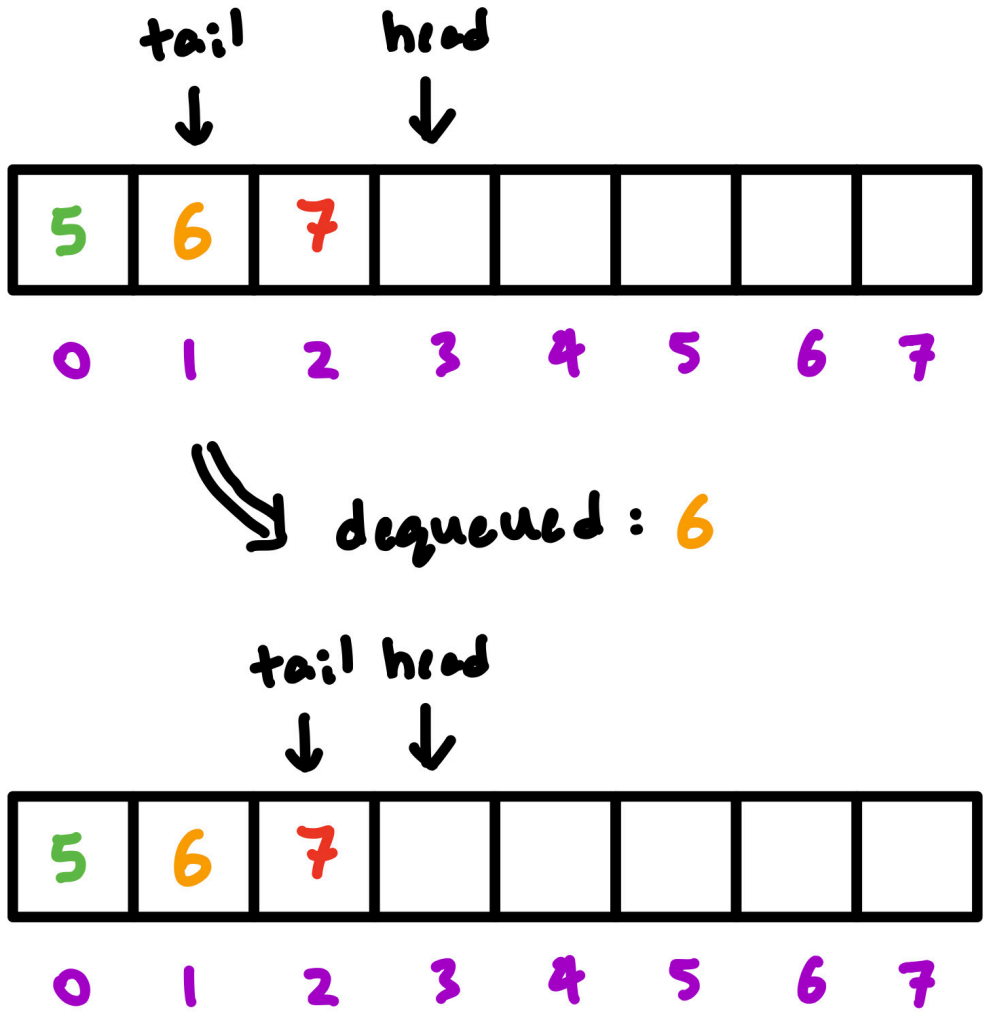
Example: Dequeuing items

- Pass back the item at the tail, move the tail forward.



Example: Dequeuing items

- Pass back the item at the tail, move the tail forward.



Empty? Full?

```
bool emptyQ(queue *q) {
    if (q) {
        return q->head == q->tail;
    }
    return true;
}

bool fullQ(queue *q) {
    if (q) {
        return succ(q, q->head) == q->tail;
    }
    return true;
}
```

- We are making a choice here:
  - We define an empty queue to have the head and tail indicating the same slot.
- How do we define full?
  - We sacrifice one queue slot, and say it is full if:
  - $head + 1 = tail \pmod{n}$

Unbounded Queues

- Unbounded queues can be implemented with a linked list.
- The unbounded queue itself is a wrapper for a linked list.
- The wrapper keeps track of the head and tail.

```c
#ifndef __QUEUE_H__
#define __QUEUE_H__

#include <stdint.h>
#include <stdbool.h>

typedef uint32_t item;

typedef struct Node Node;

struct Node {
    Node *next;
    item data;
};

typedef struct Queue {
    Node *head;
    Node *tail;
} Queue;

Queue *queue_create(void);

void queue_delete(Queue **q);

bool queue_empty(Queue *q);

bool enqueue(Queue *q, item i);

bool dequeue(Queue *q, item *i);

#endif
```

queue_create()

- Both the head and tail are NULL to begin with.

```
Queue *queue_create(void) {
    Queue *q = (Queue *)malloc(sizeof(Queue));
    q->head = q->tail = NULL;
    return q;
}
```

`enqueue()`

- First, create the node to enqueue.
- If the tail is `NULL`, then the queue is empty, and so the tail and head should be the created node.
- Otherwise, add the node at the end and update the tail.

```c
static Node *node_create(item i) {
    Node *n = (Node *)malloc(sizeof(Node));
    if (n) {
        n->data = i;
        n->next = NULL;
    }
    return n;
}

bool enqueue(Queue *q, item i) {
    Node *n = node_create(i);
    if (!n) {
        return false;
    }

    if (q->tail == NULL) {
        q->tail = q->head = n;
        return true;
    }

    q->tail->next = n;
    q->tail = n;
    return true;
}
```

dequeue()

- First, save the value of the node being dequeued.
- The head of the queue is moved forward one node.
- We save a pointer to it so we can free the memory it pointed to.
- If the head is now `NULL`, then the queue is empty, and so the tail should be `NULL` as well.

```c
static void node_delete(Node **n) {
    free(*n);
    *n = NULL;
}

bool dequeue(Queue *q, item *i) {
    if (q->head == NULL) {
        return false;
    }

    *i = q->head->data;
    Node *n = q->head;
    q->head = q->head->next;
    node_delete(&n);

    if (q->head == NULL) {
        q->tail = NULL;
    }

    return true;
}
```

Differences between stacks and queues

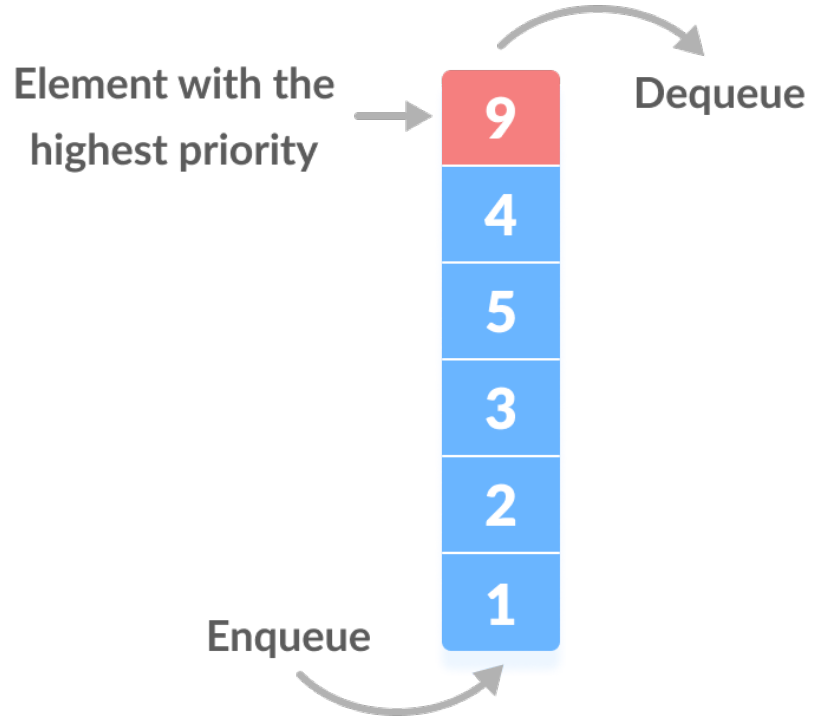
- Removing an element:
  - Stack removes the most recently added element.
  - Queue removes the least recently added element.

Queue implementations

- Commonly Implemented using vector-based data structures:
  - Arrays, linked-lists, vectors (used in C++), etc.
- Some queue implementations feature an algorithm, or a rule called *move-to-front:*
  - Elements that have been *previously searched* for in a queue are placed at the *front* of the queue.
  - Typically used as an optimization, as the recent past is a good indicator of the near future.

Priority Queue

- Implemented like a generic queue, *BUT* every element has a priority associated with it.
- Element of highest priority is dequeued before elements with lower priority.
- In the case of two elements of the same priority, preference is given to the position of the element in the queue.



Summary

- Stacks operate with the LIFO policy
- Queues operate with the FIFO policy
- Both can be implemented using basic data structures such as arrays or linked lists.