FIFO Queues

CSE 3318 – Algorithms and Data Structures
Alexandra Stefan
University of Texas at Arlington

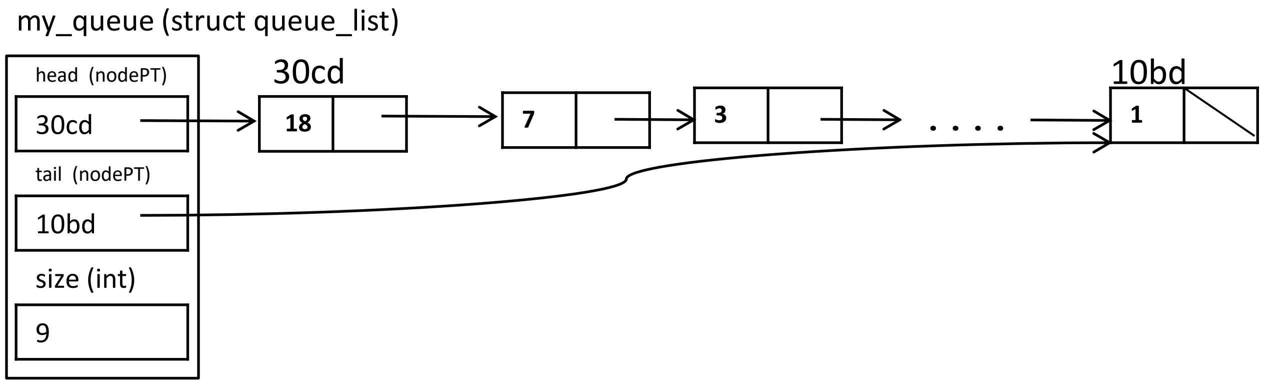
FIFO Queues

- *First-in first-out (FIFO)* queues.
- Examples of usage of FIFO queues:
  - Program execution:
    - Requests for access to memory, disk, network...
  - Resource allocation:
    - Forwarding network traffic in network switches and routers.
  - Search algorithms.
    - E.g. part of BFS in Graphs, level-order traversal for trees. (See later in the course)
- Main operations:
  - **put** - inserts an item at the end of the queue. (add/offer/enqueue/insert)
  - **get** - removes the item from the head of the queue. (remove/poll/dequeue)
- 2 implementations for FIFO queues: single linked list & array

Linked List Implementation for FIFO Queues

- A FIFO queue is essentially a list.
- **put(&queue, item)** inserts that item at the **end** of the list. - O(1)
  - Assumption: the list data type contains a pointer to the last element.
- **get(&queue)** removes (and returns) the item at the **beginning** of the list. - O(1)
- See animation

```
typedef struct node * nodePT;
struct queue_list {
   nodePT head;
   nodePT tail;
   int size;
};
```



Array-Based Queue: Example

```
struct queue_array {
   int capacity;
   int size;
   int head;  // index OF first item
   int tail;   // index AFTER last item
   int *items;
};
```

Conventions:

—— place where the new item will be added (tail).

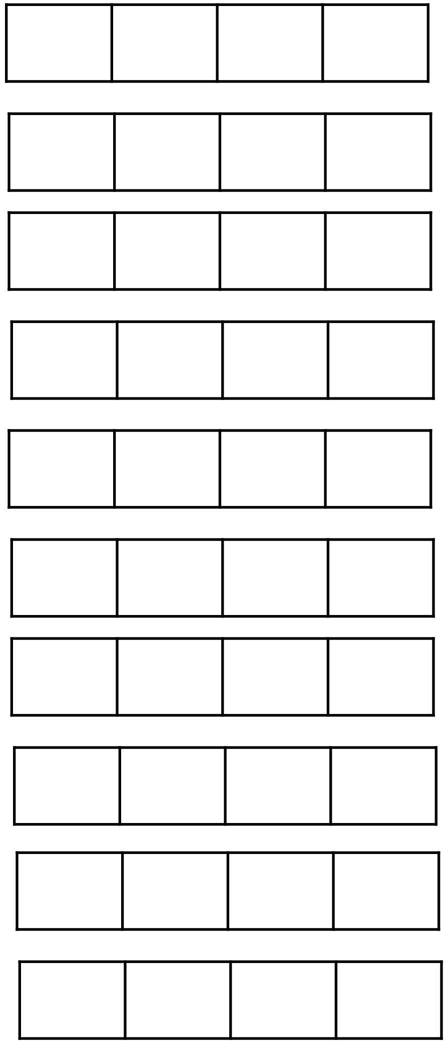
underline: first item in the queue (head).

x – put(x)

* – get()

See animation

put(15)
put(20)
get()
put(30)
put(7)
get()
put(12)
get()
get()



Array Implementation for FIFO Queues

```
struct queue_array {
   int capacity;
   int size;
   // index of first item
   int head;
    // index AFTER last item
   int tail;
   int *items;
};
typedef struct queue_array Queue;
```

```
bool put(Queue * Q, int val){
  if ((Q==NULL)||(Q->size == Q->capacity-1)) {// full
     return false;
  }
  Q->size++;
  Q->items[Q->tail] = val;
  Q->tail = (Q->tail+1)%Q->capacity;
  return true;
}

bool get(Queue * Q, int* ret){
  if ((Q==NULL) || (Q->size == 0)){
    return false;
  }
  *ret = Q->items[Q->head];
  Q->head = (Q->head+1)%Q->capacity;
  Q->size--;
  return true;
}
```

25

Issues with reallocation

Assume that if the queue is full and a put operation is called you will NOT refuse the insertion, instead you will reallocate the array to make a bigger queue. (Assume the initial max_size is 10)

Q1. How do reallocate?

a) +10 (an extra 10 spaces)

b) *2 (double the space)

Assume you allocate a queue of max size 10 at first. The user keeps inserting items until they put M items in the queue. (M can be a 1000000)

For Q1 a) and b) above answer:

- how many reallocations are needed
- Time complexity of all the reallocations and data copying to put all M items in

Q2. How do you reallocate memory and how do you “copy”? Consider these options:

- (Realloc) or (malloc with memcopy)
- Malloc with copy one by one (e.g. newArr[i] = oldArr[i])
- Reinsert one by one

Array Implementation for FIFO Queues

If for the put() operation, when the queue is full, we want to reallocate, how will we copy the data from the old array into the new one?
We must do it in such a way that whatever sequence of operations follows, the data is processed in the correct order, in particular, it will be removed from the queue in the FIFO order.
E.g. if we follow with: 100,*,*,*,*,*
We should get the data in order: 20,30,7,100

```
struct queue_array {
   int capacity;
   int size;
   // index of first item
   int head;
    // index AFTER last item
   int tail;
   int * items;
};
typedef struct queue_array Queue;
```

7	_	20	30

good

7	_	20	30

bad
