Data Structures

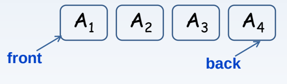
Queue

Queue Structure

- The queue structure is a simple structure like the stack.
- The difference from the stack is that the first element to enter the queue gets processed first.
- That is, service requests get added to the end of the list. That is why, it is also defined as First In First Out (FIFO) storage.
- The queue is an ordered waiting list.
- Those requesting a specific service enter an ordered list to get that service.
- The first element to get added to the list gets served first (First-Come First-Served "FCFS").

Queue Structure (continued)

- It is possible to access the structure from two points:
  - a back pointer (back) that marks the position where new arrivals will be added
  - a front pointer (front) to designate the element that will get served



Queue elements get removed only from the front and new elements are added to the back.

Areas of Application

- In computer systems, queues have many areas of application, including:
  - Single-processor computers can only serve one user at a time. User requests are kept in a queue structure.
  - In operating systems, many timing operations are performed using the queue structure.
  - Printing to a printer uses a queue.

Queue Operations

- enqueue: operation of adding a new element to the very end of the queue.
  - The added element becomes the last element in the queue.

    enqueue(...)
- dequeue: operation of removing the foremost element from the queue.
  - The element waiting behind this element gets into the foremost place in the queue.

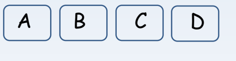
    dequeue()
- checking for emptiness: operation of checking if the queue is empty.

    isempty()

Exercise

Perform these operations on a queue in order:

- enqueue('A');
- enqueue('B');
- enqueue('C');
- dequeue();
- enqueue('D');
- dequeue();
- dequeue();



Realizing Queues on Arrays

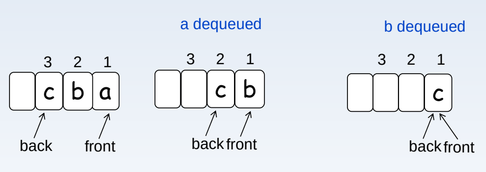
- It is possible to implement a queue on an array.
- We will first look at two implementations, which have some drawbacks.
- Then, we will discuss the proper way.

Realization on an Array 1: Shifting Elements

- Let the queue "front" pointer always be the first element of the array.
- To add a new element, the "back" pointer is incremented once.
- When removing an element, it is always the first element that gets removed.
- But, this is an expensive solution: when removing each element, all other elements have to be shifted once.

Example:

Drawback: Elements must be shifted



Each time, all elements of the array have to be shifted forward once.

Realization on an Array 2: Linear Array

- It is possible to configure front and back pointers in different ways. For example:
  - "back" points to the empty space where the element will be added. To add an element, add (write) and increment the "back" pointer.
  - To remove an element, read the element by using the "front" pointer and then increment the pointer.
  - If we had an array of <u>infinite</u> size, the problem could have been solved as follows:

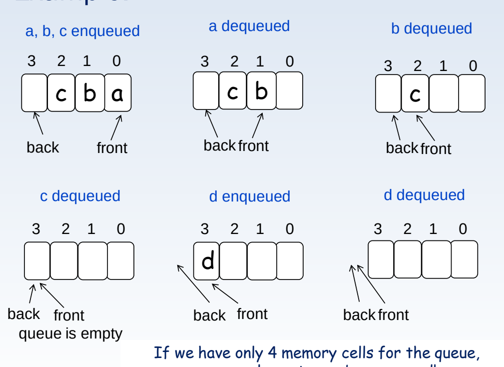
Not possible!

```
int front = 0; int back = 0;
char queue[∞];  //infinite!

bool isempty(){
  return (front == back);
}
```

```
void enqueue(char x){
   queue[back++] = x ;
}

char dequeue(){
  return (queue[front++]);
}
```


Example:

Drawback: It wastes memory.



If we have only 4 memory cells for the queue, no new elements can be enqueued!
It looks like there are no empty spaces in the array.


```
#ifndef QUEUE_H
#define QUEUE_H

#define QUEUESIZE 10

typedef int QueueDataType;
struct Queue {
  QueueDataType element[QUEUESIZE];
  int front;
  int back;
  void create();
  void close();
  bool enqueue(QueueDataType);
  QueueDataType dequeue();
  bool isempty();
};
#endif
```

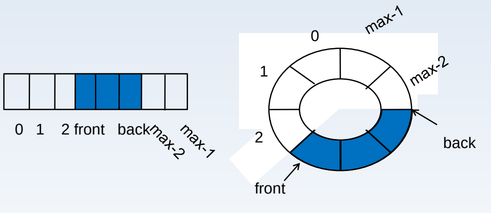
```
void Queue::create() {
   front = 0; back = 0;
}
void Queue::close() {
}
bool Queue::enqueue(QueueDataType newdata) {
   if (back < QUEUESIZE) {
         element[back++] = newdata;
         return true;
   }
   return false;
}
QueueDataType Queue::dequeue() {
   return element[front++];  // actually, isempty() check is necessary
}
bool Queue::isempty(){
   return (front == back);
}
```

Even though the array has space, a "queue full" error might return .

Realization on an Array 3: Circular Array

- This is the proper implementation of a queue on an array.
- We realize the array as a circular array rather than a linear array.
- The first element follows the last element of the array.
- We achieve continuity.
- At different times, there will be elements in different parts of the array.
- As long as the array is not completely full, there will be no "out of space" message.

Circular Array



Circular Array Realization

- An array with max number of elements is used.
  Queue[0 … max - 1]
- A transition from the element with index "max - 1" to the element with index 0 is desired.

```
if (i == max - 1)
      i = 0;
else
      i++;
```

i = (i + 1) % max

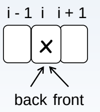
Boundary Conditions

Configuration of front and back pointers:

back: indicates the last element,
increment the "back" pointer, make an assignment.

front: indicates the first element,
do a read, increment the "front" pointer.

<u>If there is a single element in the queue</u>, "front" and "back" pointers will point to this element.

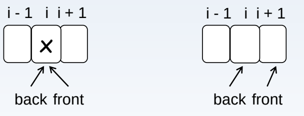


Remember: It is also possible to configure the back and front pointers in different ways.
"back" can point to the empty space instead of the last written element.

Boundary Conditions (continued)

"Queue is empty" condition:

- When this element is removed, the "front" pointer will get incremented once and pass "back".



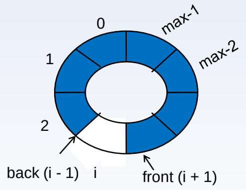
"Queue is empty" condition:

front == back + 1

Boundary Conditions (continued)

"Queue is full" condition:

- If there is only a single empty slot in the queue, and a new element is added, the "front" will be one more than "back".



"Queue is full" condition: "front" pointer one more than "back"

front == back + 1

Boundary Conditions (continued)

Empty and Full queue conditions become the same!

Same problem exists even if you use other back and front configurations.

Solutions:

1. Keep array capacity one less. Leave one space empty.
2. Use a logical variable that will indicate that the queue is full.
   Or, use a variable that contains the number of elements in the queue.
3. Assign a special value to the pointer to show the state of being empty.

Realization Using Special Index Values

- Initial state conditions: front = 0, back = -1
- When the queue becomes empty, return to initial state conditions.
- If the queue is full, front == back + 1 and back != -1.


```
void Queue::create() {
  front = 0; back = -1;
}

void Queue::close() {
}

bool Queue::enqueue(QueueDataType newdata) {
  if ( back != -1 && (front == (back + 1) % QUEUESIZE ) )
       return false;
  else {
       back = (back + 1) % QUEUESIZE;
       element[back] = newdata;
       return true;
  }
}
```



```
QueueDataType Queue::dequeue() {
  QueueDataType el = element[front];
  if ( front == back ) {
       front = 0; back = -1;
  }
  else front = (front + 1) % QUEUESIZE;
  return el;
}

bool Queue::isempty() {
  return (front == 0 && back == -1);
}
```

Realization of Queues on Arrays

- Realizations of queues on the array presents problems similar to realization of stacks on the array:
  The fixed size of the queue results in an error if we try to add more entries into the queue than expected.
  This restriction can be eliminated by using linked lists.
- On the other hand, the implementation on an array may run faster than the implementation with linked lists.

Realization Using Lists

- The restrictions imposed by array implementations can be eliminated by using linked lists.
- In list operations, adding to and removing from the beginning of the list is faster.
  - However, the whole list has to be traversed to find the end of the list.
- To add to the back of the queue, moving each time from the beginning of the list to the end is an expensive operation. Therefore we set the back pointer to point to the last element of the list.

Realization Using Lists – Our Design

- The front of the queue is the first element of the list, the back of the queue is the last element of the list.
- Adding to queue: adding an element to the end of the list
- Removing from the queue: removing an element from the beginning of the list.


```
typedef char QueueDataType;

struct Node{             // nodes of the list
  QueueDataType data;
  Node *next;
};

struct Queue {
  Node *front;
  Node *back;
  void create();
  void close();
  bool enqueue(QueueDataType);
  QueueDataType dequeue();
  bool isempty();
};
```



```
void Queue::create(){
   front = NULL; back = NULL;
}
void Queue::close(){
   Node *p;
   while (front) {
         p = front;
         front = front->next;
         delete p;
   }
}
void Queue::enqueue(QueueDataType newdata){
   Node *newnode = new Node;
   newnode->data = newdata;
   newnode->next = NULL;
   if ( isempty() ) {    // first element?
         back = newnode;
         front = back;
   }
   else {
         back->next = newnode;
         back = newnode;
   }
}
```

```
QueueDataType Queue::dequeue() {
   Node *topnode;
   QueueDataType temp;
   topnode = front;
   front = front->next;
   temp = topnode->data;
   delete topnode;
   return temp;
}
bool Queue::isempty() {
   return front == NULL;
}
```