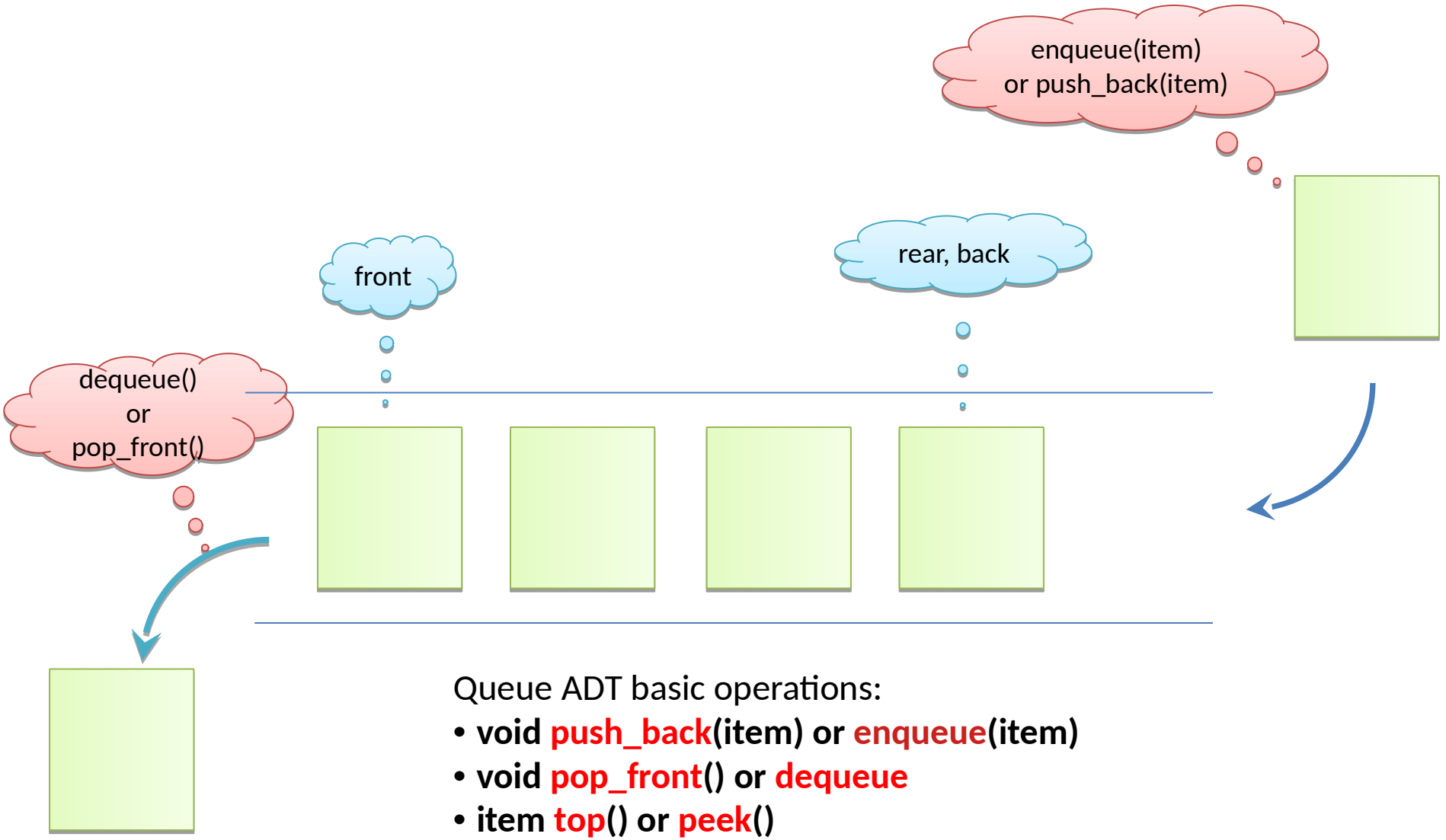
Queue ADT

Queue

- In computer science, a queue is a collection of entities that are maintained in a sequence and can be modified by the addition of entities at one end of the sequence and the removal of entities from the other end of the sequence.
- The operation of adding an element to the rear of the queue is known as enqueue, and the operation of removing an element from the front is known as dequeue.
- Other operations may also be allowed, often including a peek or front operation that returns the value of the next element to be dequeued without dequeuing it.
- The operations of a queue make it a first-in-first-out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed.
- A queue is an example of a linear data structure or, more abstractly, a sequential collection.
- Queues are standard in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure, or in object-oriented languages as classes.
- Typical implementations are circular buffers (using an array) and linked lists.

Queue working principle

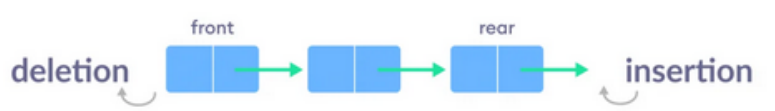


Queue ADT basic operations:

- **void push_back(item) or enqueue(item)**
- **void pop_front() or dequeue**
- **item top() or peek()**
- **void clear()**
- **bool empty()**

Linked implementation

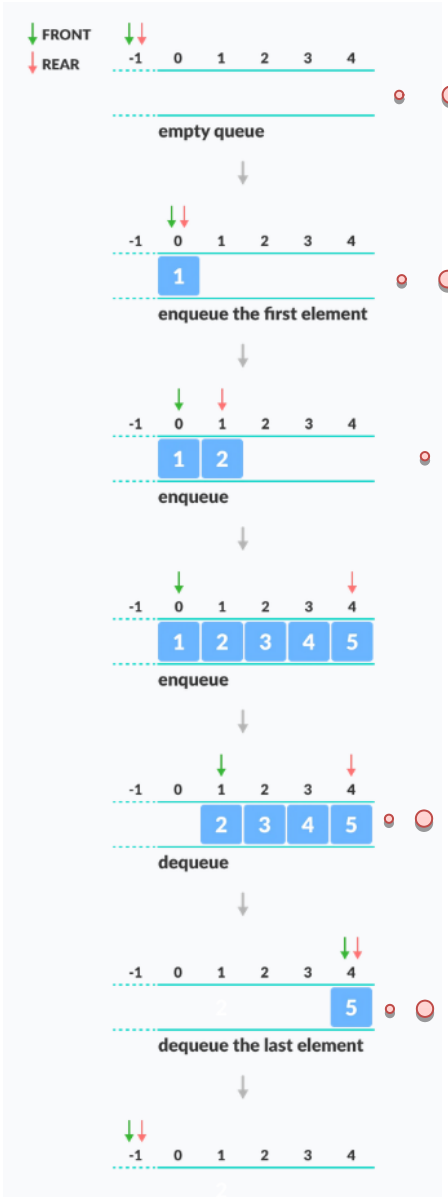
- A singly linked list can be used to implement a queue.
- There are two pointers involved for the realization
- One is used to add elements to back of the queue, the other is used to remove the elements from the front of the queue.
- Adding and removing elements takes O(1) time.




```
template<typename T> class Queue {
private:
Node<T> *front;
Node<T> *back; //rear
int count;
public:
Queue() {
  front = back = NULL;
  count = 0;
}
int size() const {
return count;
}
//return the item at the front of the queue
const T& peek() const {
  if (count == 0)  throw EmptyQueueException();
  return front->item;
}
void enqueue(const T &item) { //add an item to queue
  Node<T> *last = new Node<T>(item);
  if (count == 0) front = back = last;
  else {
    back->next = last;
    back = last;
  }
  count++;
}
void dequeue() { //remove
  if (count == 0)  throw EmptyQueueException();
  Node<T> *del_front = front;
  front = front->next;
  delete del_front;
  count--;
}

void clear() {
   while (count != 0)
   dequeue();
}
bool empty() const {
   return count == 0;
}
```

Array based implementation



Initially the queue is empty.
front=-1, rear=-1, or front=0,
rear=-1

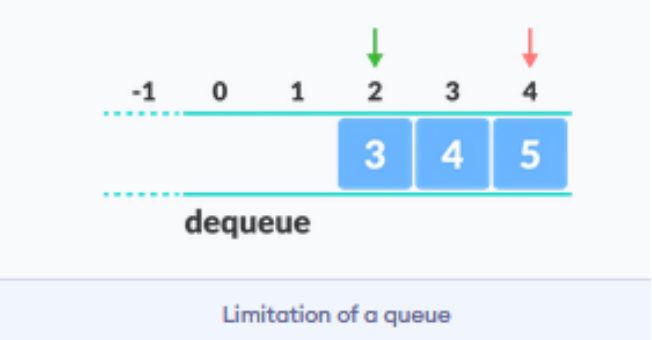
After enqueue(1), there is
one element.
front=0, rear=0

After enqueue(2), there
are two elements.
front=0, rear=1

After dequeue()
front=1, rear=4

After a number of
dequeue()
front=4, rear=4

Array based implementation

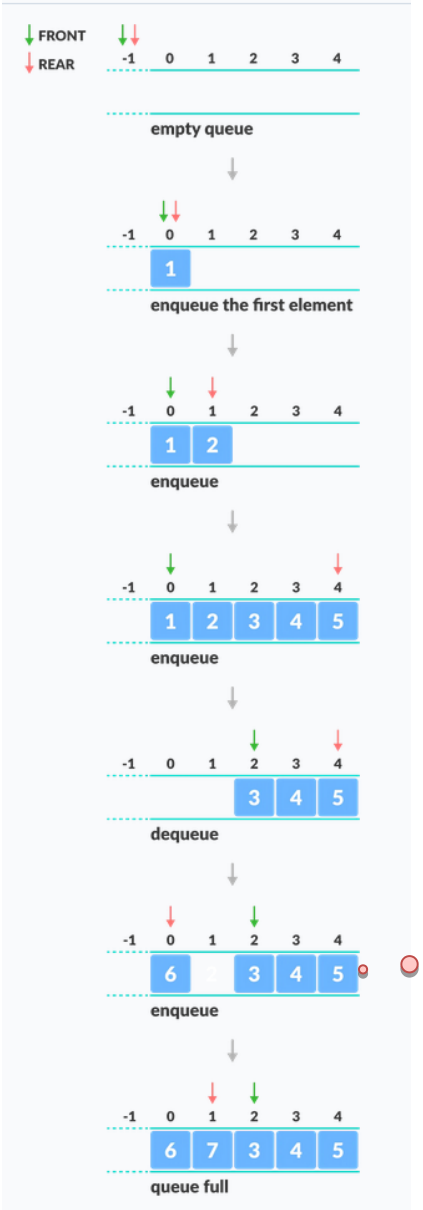


Limitation of a queue

- As you can see in the image below, after a bit of enqueuing and dequeuing, the size of the queue has been reduced.

Circular queue representation

Array based implementation



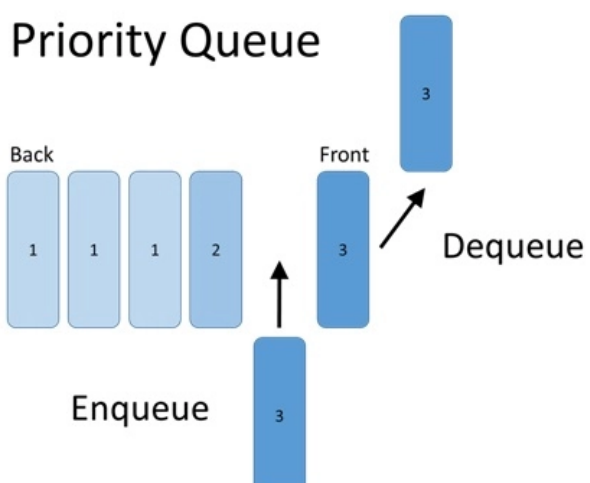
If capacity is not full, new elements are placed from the start again.

```
template <class Object> class Queue{
private:
Object *items;
int front,rear,length,capacity;
void resize(int newcap){
  Object *temp=new Object[newcap];
  for (int var = 0; var < length; ++var) {
        temp[var]=items[(var+front)%capacity];
  }
  capacity=newcap;
  front=0;
  rear=length-1;
  delete[] items;
  items=temp;
}
public:
Queue(int capacity=1){
   this->capacity=capacity;
   items=new Object[capacity];
   front=0;
   rear=-1;
   length=0;
}
void enqueue(const Object &item){
  if (length==capacity) resize(2*capacity);
  rear=(rear+1)%capacity;//circular
  items[rear]=item;
  length++;
}
const Object &peek()const{
    if (length==0) throw EmptyQueueException();
    return items[front];
}
void dequeu(){
    if (length==0) throw EmptyQueueException();
    front=(front+1)%capacity;
    length--;
}
int size()const{
   return length;
}
bool empty(){
   return rear==-1;
}
~Queue(){
   delete[] items;
}
```

Priority Queues

- In computer science, a priority queue is an abstract data-type similar to a regular queue or stack data structure in which each element additionally has a "priority" associated with it.
- In a priority queue, an element with high priority is served before an element with low priority.
- In some implementations, if two elements have the same priority, they are served according to the order in which they were enqueued; in other implementations ordering of elements with the same priority remains undefined.

https://en.wikipedia.org/wiki/Priority_queue

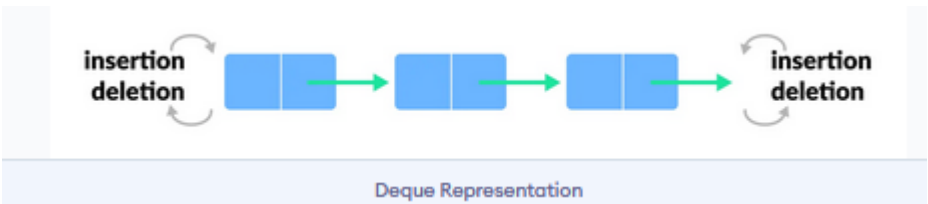


Implementation:
We can think of various ways of implementing the priority queues. We can choose various data structures like an array, linked list, BST, heaps to make the operations of the priority queues efficient.

Source: https://www.includehelp.com/ds/implementation-of-priority-queue-using-linked-list.aspx

Deque (Double Ended Queue)

- In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.



Source: https://www.programiz.com/dsa/types-of-queue