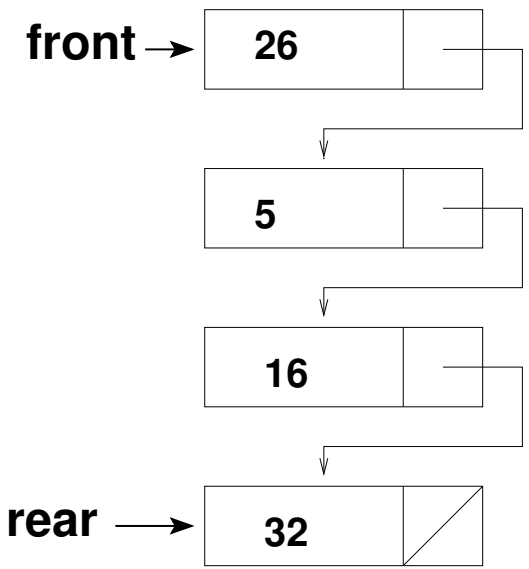
A Queue Based on a Linked List

The queue can be implemented based on a linked list. The references `rear` and `front` are references to the end and the front of the list.



The Queue OPerations

Special situations in a linked-list implementation of a queue:

Condition	**Special situation**
`front == NULL`	Empty queue
`back == front`	One-entry queue
no memory available	Full queue

We can implement the class `LinkedQueue` in many ways. We start with the implementation which is similar to what is done for the class `LinkedList`. This is *not* a good approach because we do not reuse the class `LinkedList`. The class `ListNode` is defined in an implementation of Linked List ADT.

The LinkedQueue Class

```
class LinkedQueue {
private:
   ListNode *front, *rear;
   void deleteLinkedQueue(); // private function
public:
   LinkedQueue() {  front = rear = NULL; } // constructor
   ~LinkedQueue() { deleteLinkedQueue(); } //destructor
   bool isEmpty( ) const {  return front == NULL;  }
   int first( ) throw(QueueEmptyException);
   void enqueue(int x);
   int dequeue( ) throw(QueueEmptyException);
};
```

The Class Implementation

```
int LinkedQueue::first( ) throw(QueueEmptyException)
{
   if ( isEmpty() )
      throw QueueEmptyException("Access to an empty queue"
   return front->getElem();
}
void LinkedQueue::enqueue(int x)
{
   if ( isEmpty() ) //make one-element queue
      rear = front = new ListNode(x, NULL);
   else { //regular case: we need to use setNext()
      rear->setNext(new ListNode(x, NULL));
      rear = rear->getNext();
   }
}
```

Class Implementation (cont.)

```
int LinkedQueue::dequeue( ) throw(QueueEmptyException)
{  if ( isEmpty() )
      throw QueueEmptyException("Access to an empty queue"
   int item = front->getElem();
   ListNode *node = front;
   front = front->getNext();
   delete node;
   return item;
}
void LinkedQueue::deleteLinkedQueue()
{  ListNode *node;
   while (front != NULL) {
      node = front;
      front = front->getNext();
      delete node;
   }
   rear = NULL;
}
```

A Better Implementation of the Queue

As with the stack we can provide an efficient implementation of the queue ADT using the class `LinkedList`. As previously, we choose the front of the queue to be at the head of the linked list, and the rear of the queue to be at the tail of the linked list.
We use now the adapter design pattern where members of the queue are implemented as the corresponding linked list operations.

Queue Method	Linked List Implementation
`isEmpty()`	`isEmpty()`
`first()`	`first()`
`enqueue(e)`	`insertLast(e)`
`dequeue()`	`removeFirst()`

Adapter Design Pattern

```
class LinkedQueue {
private:
   LinkedList ll; // internal LinkedList object
public:
   LinkedQueue() : ll() { } // constructor
   ~LinkedQueue() { ll.~LinkedList(); } // destructor
   bool isEmpty( ) const {  return ll.isEmpty();  }
   int first( ) const throw(QueueEmptyException);
   void enqueue(int elem) { ll.insertLast(elem); }
   int dequeue( ) throw(QueueEmptyException);
};
```

Adapter Design Pattern **(cont.)**

```
int LinkedQueue::first( ) const throw(QueueEmptyException)
{
   if ( ll.isEmpty() )
      throw QueueEmptyException("Access to an empty queue"
   return ll.first();
}
int LinkedQueue::dequeue( ) throw(QueueEmptyException)
{
   if ( ll.isEmpty() )
      throw QueueEmptyException("Access to an empty queue"
   return ll.removeFirst();
}
```