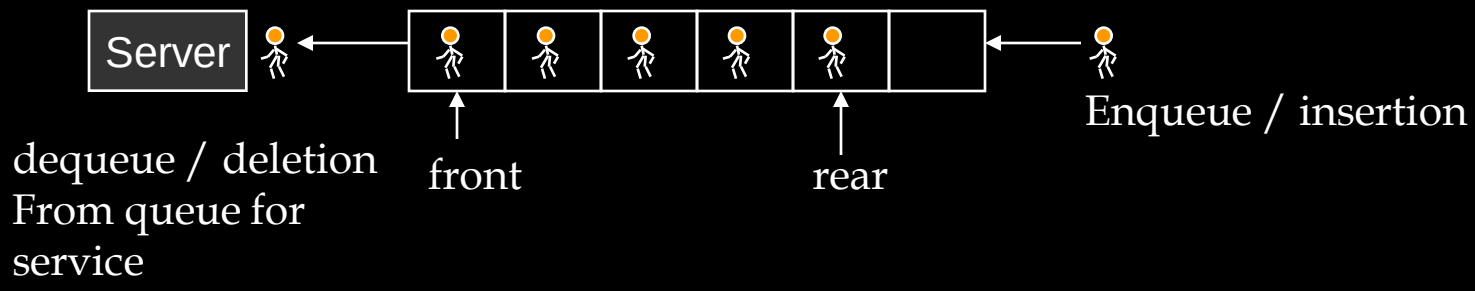
Data Structures and Algorithms

Lecture 7

QUEUES

Queues

- FIFO - first in, first out
- Insert at rear, remove from front
- Remove the item for service that has been in the queue the longest
- Maintain front and rear pointers



Examples of queues

1. Lines: " Ticket line, amusement park line, etc”
2. Access to shared resources (e.g., printer queue)
3. Phone calls to large companies
4. Waiting list for adding classes
5. Computer processes waiting for hardware resources

Queues

Main queue operations:

enqueue(Object o): inserts an element o at the end of the queue

dequeue(): remove and return the element at the front of the queue

front(): returns the element at the front without removing it

Auxiliary queue operations:

size(): returns the number of elements stored

isEmpty(): returns 1 indicating queue is empty
returns 0 indicating queue is not empty

isFull(): return 1 indicating queue is full
return 0 indicating queue is not full

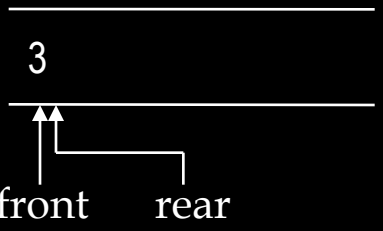
Queue Implementation: Array

Problem with linear array: the size of array not only depends on the max queue size but also on number of enqueue() and dequeue() operations

Suppose our Queue can hold upto max 9 elements, so we declare an array of 9 elements

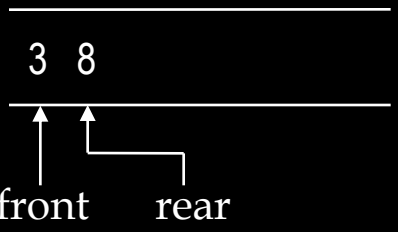
maxsize = 8 front = 0 rear = -1 size = 0

enqueue(3);



maxsize = 8 front = 0 rear = rear+1=1 size = 1

enqueue(8);

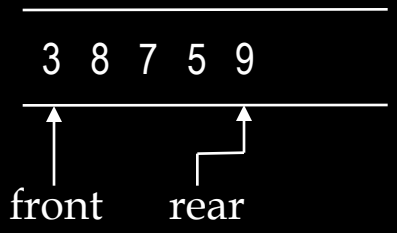


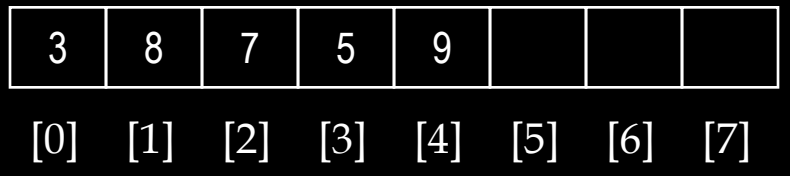
maxsize = 8 front = 0 rear = rear+1=1 size = 2

enqueue(7);
enqueue(5);

Queue Implementation: Array

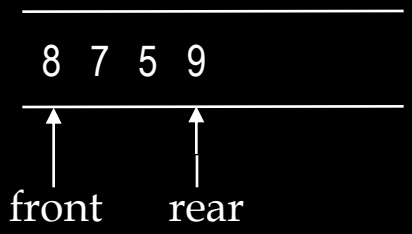
enqueue(9);

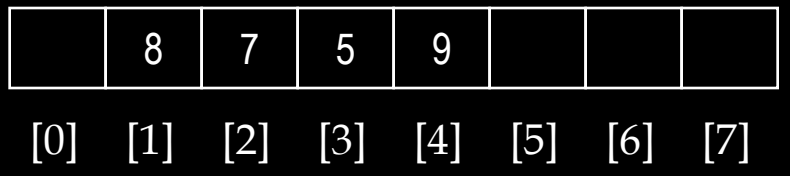




maxsize = 8 front = 0 rear = rear+1= 4 size = 5

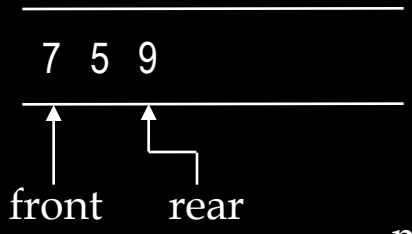
C=dequeue();

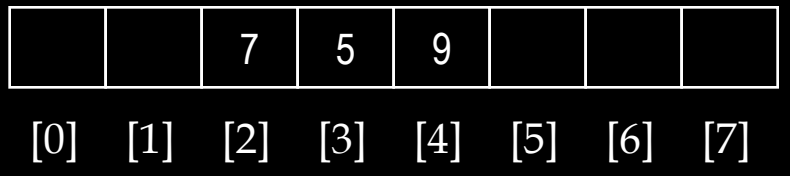




maxsize = 8 front = front+1=1 rear = 4 size = 4

C=dequeue();



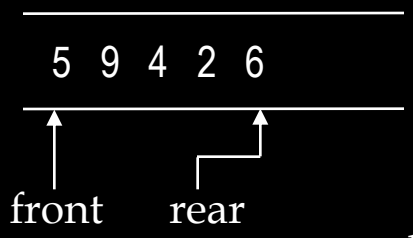


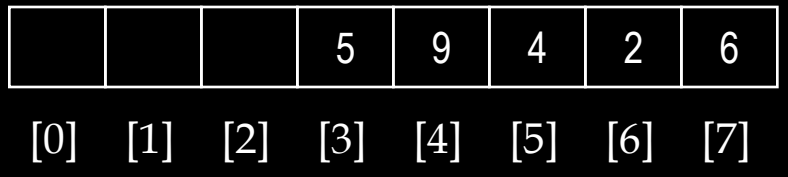
maxsize = 8 front = front+1=2 rear = 4 size = 3

```
enqueue(4);
enqueue(2);
enqueue(6);
```

Queue Implementation: Array

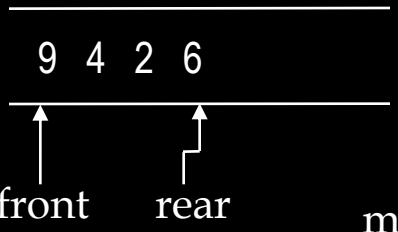
C=dequeue();

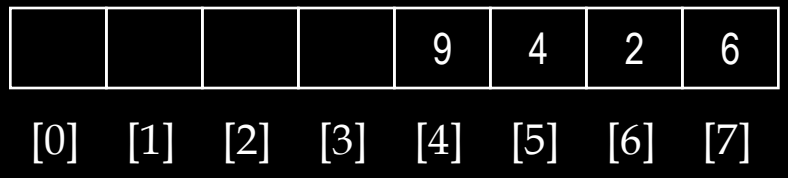




maxsize = 8 front = front+1=3 rear = 7 size = 5

C=dequeue();





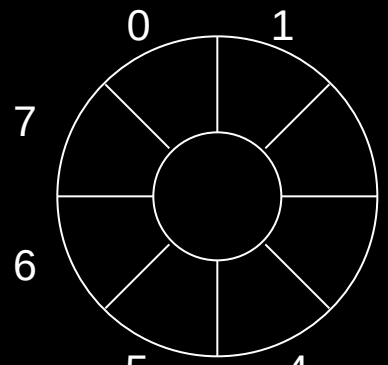
maxsize = 8 front = front+1=4 rear = 7 size = 4

enqueue(7); ?????

cannot further enqueue an element rear exceeds the array size, although there are only 4 elements in the queue and array size is 8

The solution is to wrap around (circular) array

Queue Implementation: Circular Array



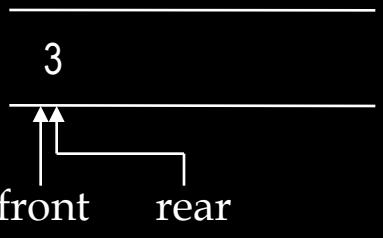
size = 0

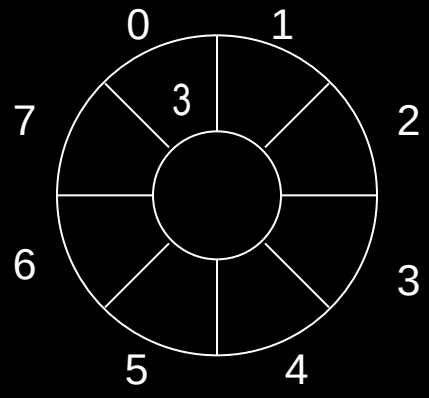
front = 0

maxsize = 8

rear = -1

enqueue(3);





size = 1

front = 0

maxsize = 8

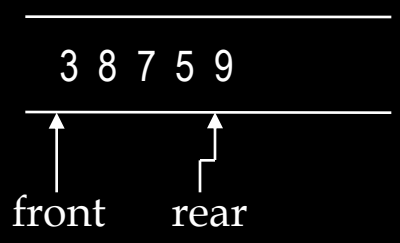
rear = (rear+1)% maxsize = 0

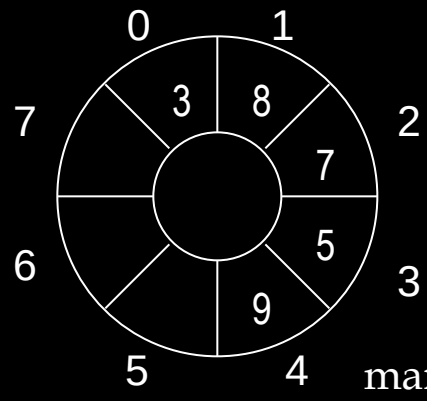
enqueue(8);
enqueue(7);
enqueue(5);

```
void enqueue(int x)
{
     rear = (rear+1)%maxsize;
     array[rear] = x;
      size++;
}
```

Queue Implementation: Circular Array

enqueue(9);





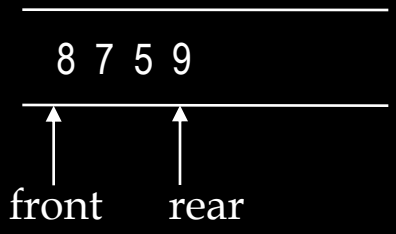
size = 5

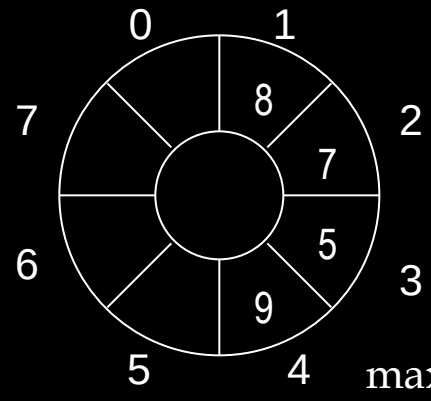
front = 0

maxsize = 8

rear =(rear+1)% maxsize = 4

dequeue();





size = 5

front = (front+1)% maxsize = 1

maxsize = 8

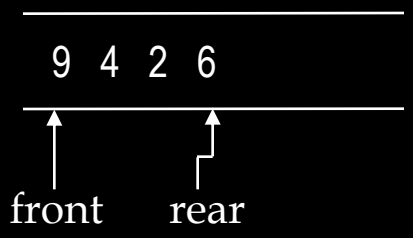
rear = 4

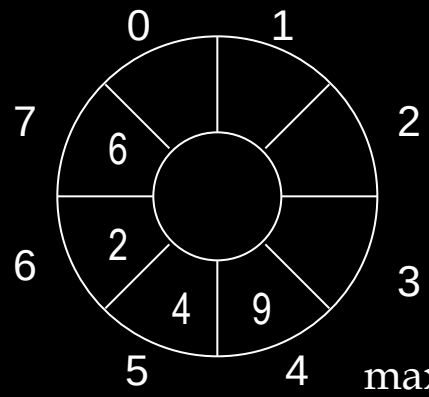
```
C = dequeue();
enqueue(4);
enqueue(2);
enqueue(6);
C = dequeue();
```

```
int dequeue()
{    int x = array[front];
     front = (front+1)%maxsize;
     size--;
     return x;
}
```

Queue Implementation: Circular Array

dequeue();

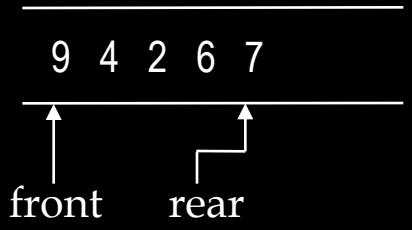


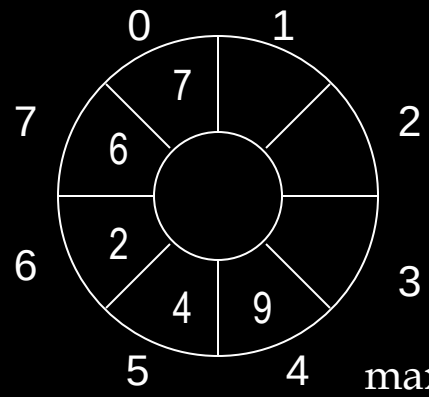


size = 4 maxsize = 8

front = (front+1)% maxsize = 4 rear = 7

enqueue(7);





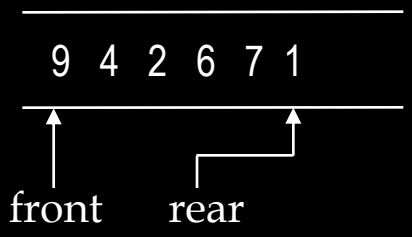
size = 5

front = 4

rear =(rear+1)% maxsize
= (7+1)% 8 = 0

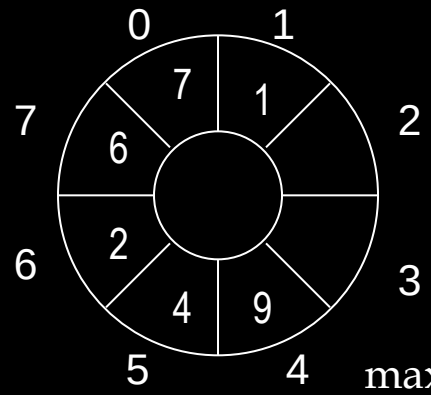
Queue Implementation: Circular Array

enqueue(1);




```
C=dequeue();
C=dequeue();
C=dequeue();
```

C = dequeue();

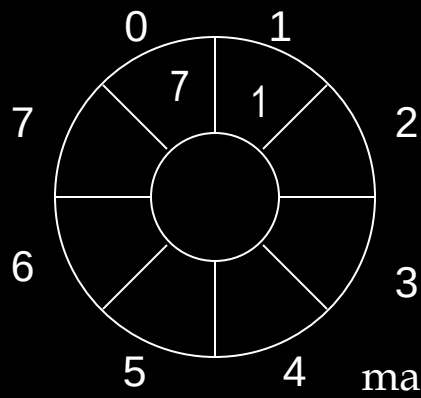


size = 6

front = 4

maxsize = 8

rear =(rear+1)% maxsize
= (8+1)% 8 = 1



size = 2

maxsize = 8

front = (front+1)% maxsize
= (7+1)% 8
= 0

rear = 1

Queue Implementation: Circular Array

```
#include<iostream.h>
class queue
{
private:
   int front, rear, qsize, maxqsize, printptr;
   int *array;

public:
queue(int x)
{
   maxqsize =  x;
   array = new int[maxqsize];
   printptr = 0;   qsize=0;
   rear = -1;      front = 0;
}
```

Queue Implementation: Circular Array

```
void enqueue(int x)
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

int dequeue()
{
   int x = array[front];
   front = (front+1)% maxqsize;
   qsize--;
   return x;
}
```

Queue Implementation: Circular Array

```
int atfront() { return array[front]; }

int IsFull() { return (qsize == maxqsize); }

int IsEmpty() { return (qsize == 0); }

int size() { return qsize; }

void print()
 {
  printptr=front;
  do {
      cout << array[printptr] << " ";
      printptr=((printptr+1)%maxqsize);
     } while (printptr!=rear);
  cout << array[printptr];
 }
};
```

Queue Implementation: Circular Array

```
void main()
{
 queue Q(10);

 char a='$';
 while ((Q.IsFull()==0)&&(a!='@'))
 {
 cout << "Enter Element to enqueue, '@' to terminate: ";
 cin >> a;
 if (a!='@') Q.enqueue(a);
 }

if (Q.IsEmpty()==0)
     { cout << "Queue is: ";
       Q.print();
      }
```

Queue Implementation: Circular Array

```
int choice=1;
while (!((choice>2)||(choice<1)))
 {
  cout << "\n1. enqueue      2. dequeue     3. Exit    :Enter your choice  ";
  cin  >> choice;
  if (choice==1)
      {if (Q.IsFull()==0)
         { cout << "\nEnter Element to enqueue: "; cin >> a; Q.enqueue(a);
           cout << "\nQueue is: "; Q.print(); }
       else cout << "\nQueue is Full";
      }
   if (choice==2)
      {if (Q.IsEmpty()==0)
          { a=Q.dequeue();cout << "\n" << a << " has been dequeued from Q";
            cout << "\nQueue is ";
            if (Q.IsEmpty()==1) cout << "Empty";
            else Q.print(); }
       else
            cout << "\nQueue is empty";
      }
  }
}
```

Queue Implementation: Array

```
 //Queue implementation using Arrays
#include <iostream>
using namespace std;
class Queue
{
public:
Queue();
~Queue();
void enqueue(int);
void dequeue();
void DisplayQueue();
private:
int rear, front;
int Qu[10];
};
Queue::Queue()
{
rear = front = -1;
}
```

Queue Implementation: Array

```
Queue::~Queue()
{
}
void Queue::enqueue(int element)
{
if (rear == 9)
{
cout << "Queue overflow" << endl;
system("pause>0");
return;
}
else
{
rear++;
Qu[rear] = element;
}
if (front == -1)
{
front = 0;
}
}
```

Queue Implementation: Array

```
void Queue::dequeue()
{
if (front == -1)
{
cout << "Queue is empty, Queue underflow" << endl;
system("pause>0");
return;
}
cout << "Element " << Qu[front] << " is removed from the queue" << endl;
system("pause>0");
Qu[front] = NULL;
front++;
if (front > rear)
{ rear = front = -1; }
}
void Queue::DisplayQueue()
{
if (front == -1)
{ cout << "Queue is empty" << endl;
system("pause>0");
return;}
```

Queue Implementation: Array

```
for (int i = front; i <= rear; i++)
{ cout << Qu[i] << "\t";
}
system("pause>0");
}
int main()
{
Queue obj;
int choice, element;
while (true)
{ system("cls");
cout << "1 for inserting an element into Queue" << endl;
cout << "2 for removing an element from a Queue" << endl;
cout << "3 for displaying the elements of Queue" << endl;
cout << "4 for exit the program" << endl;
cout << "Enter your choice [1 - 4] ";
cin >> choice;
switch (choice)
{
```

Queue Implementation: Array

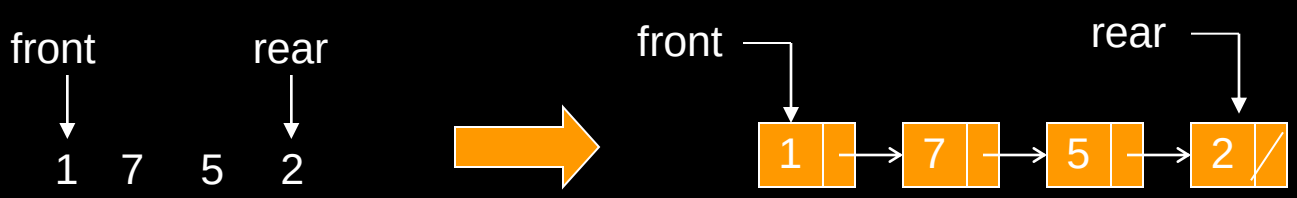
```
case 1:
cout << "Enter an element to be inserted into Queue ";
cin >> element;
obj.enqueue(element);
break;
case 2:
obj.dequeue();
break;
case 3:
obj.DisplayQueue();
break;
case 4:
exit(0);
break;
default:
cout << "invalid choice" << endl;
}
}
system("pause");
return 0;
}
```

Implementing Queue

- Using linked List: *Recall*
- Insert works in constant time for either end of a linked list.
- Remove works in constant time only.
- Seems best that head of the linked list be the front of the queue so that all removes will be from the front.
- Inserts will be at the end of the list.

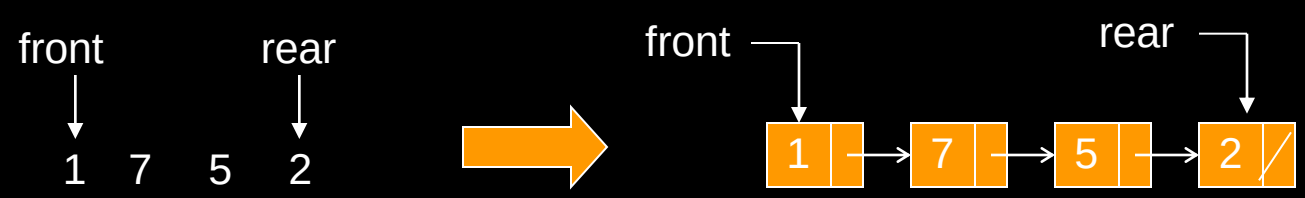
Implementing Queue

- Using linked List:

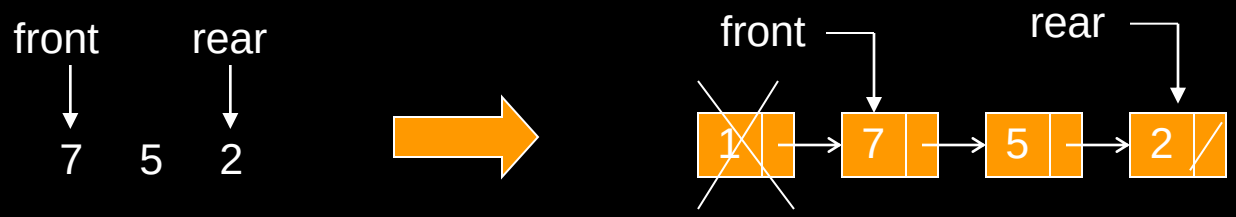


Implementing Queue

- Using linked List:

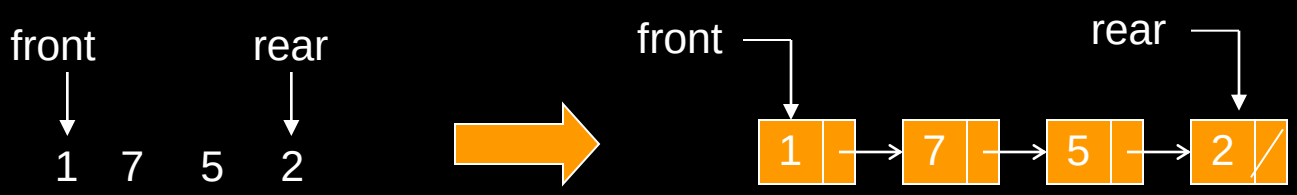


dequeue()

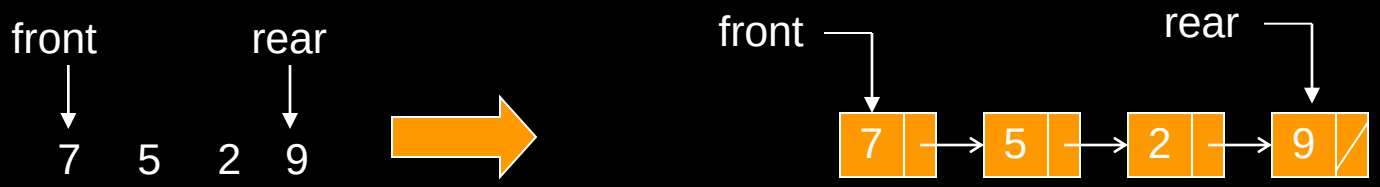


Implementing Queue

- Using linked List:



enqueue(9)



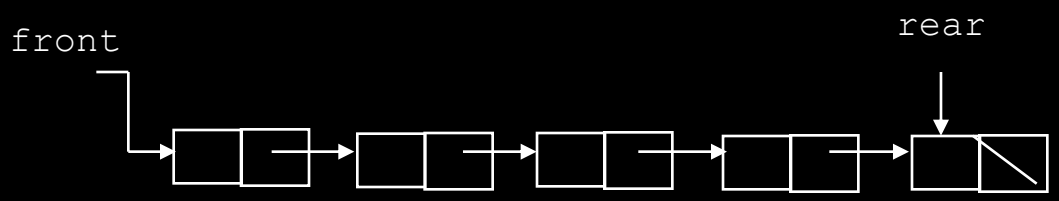
Implementing Queue

```
int dequeue()
{
    int x = front->get();
    Node* p = front;
    front = front->getNext();
    delete p;
    return x;
}
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    rear->setNext(newNode);
    rear = newNode;
}
```

Implementing Queue

```
int front()
{
    return front->get();
}


int isEmpty()
{
    return ( front == NULL );
}
```

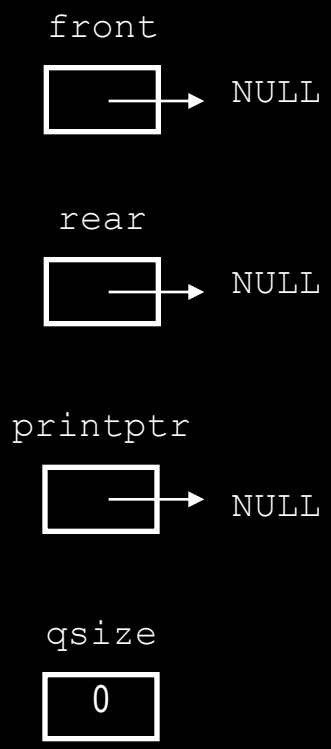
Queue Implementation: Linked List




```
#include <iostream>

class Node {
public:
   int get() { return object; };
   void set(int object) { this->object = object; };
   Node *getNext() { return nextNode; };
   void setNext(Node *nextNode){ this->nextNode = nextNode; };
private:
   int object;
   Node *nextNode;
};
class queue {
private:
   int qsize;
  Node *front,*rear;
  Node *printptr;
```

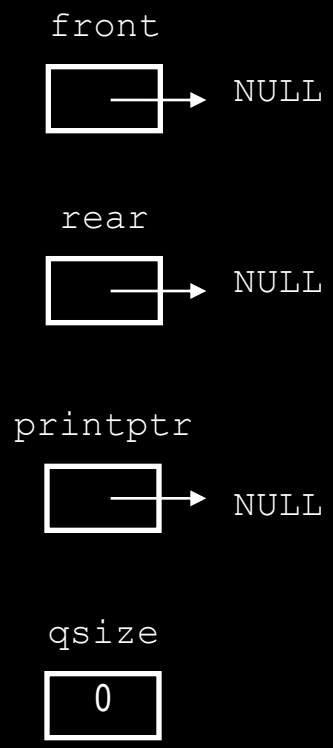
Queue Implementation: Linked List

```
public:
   // Constructor
   queue() {
        front = NULL;
        rear = NULL;
        printptr = NULL;
        qsize = 0;
   };
```



Queue Implementation: Linked List

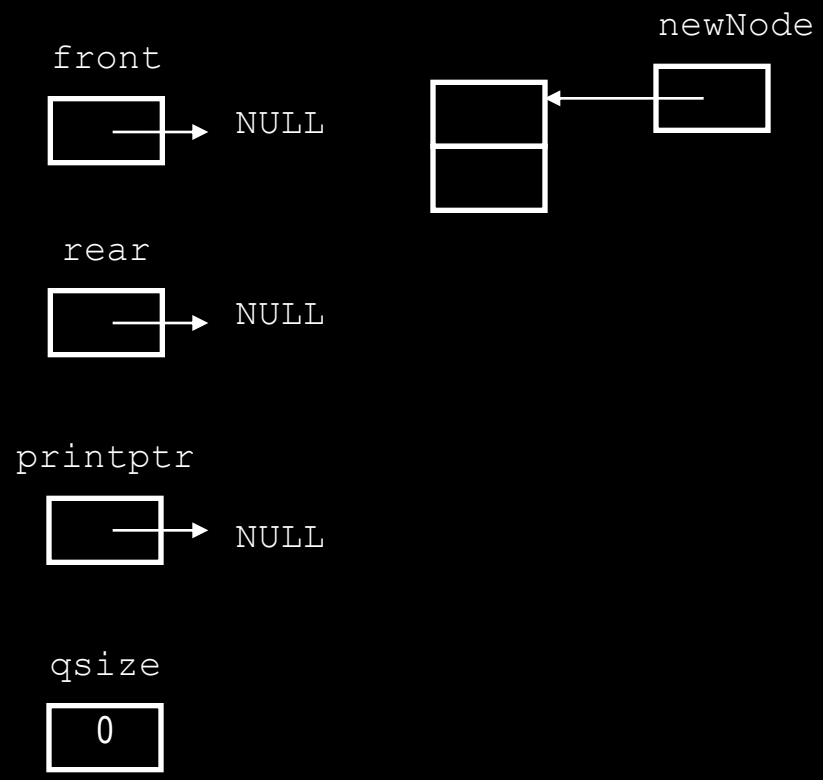
```
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    if (rear!=NULL)
        rear->setNext(newNode);
    rear = newNode;
    if (front==NULL) front=rear;
    qsize++;
}
```



Queue is Empty

Queue Implementation: Linked List

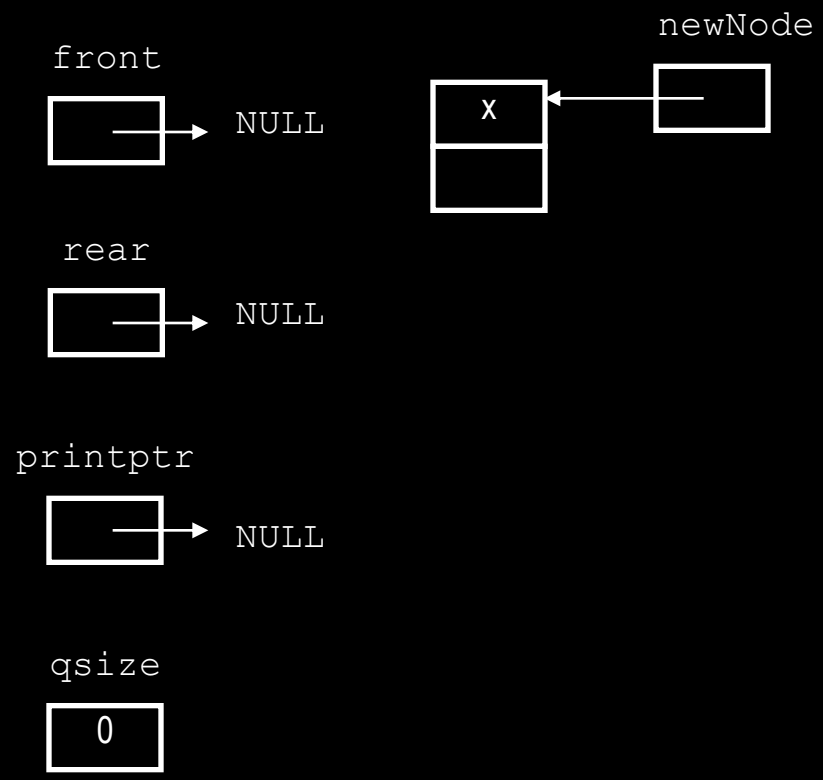
```
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    if (rear!=NULL)
        rear->setNext(newNode);
    rear = newNode;
    if (front==NULL) front=rear;
    qsize++;
}
```



Queue is Empty

Queue Implementation: Linked List

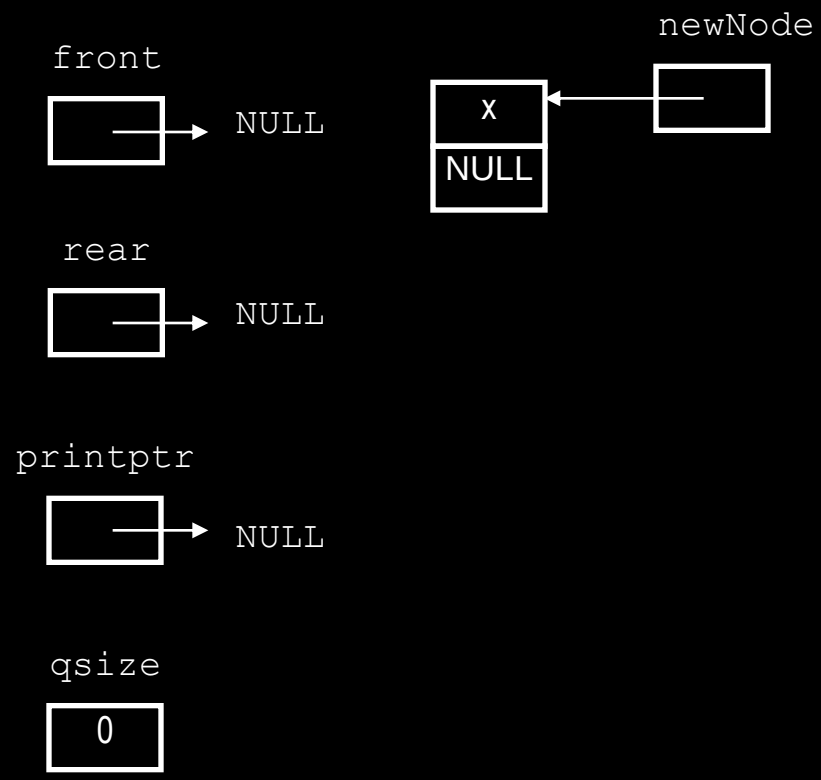
```
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    if (rear!=NULL)
        rear->setNext(newNode);
    rear = newNode;
    if (front==NULL) front=rear;
    qsize++;
}
```



Queue is Empty

Queue Implementation: Linked List

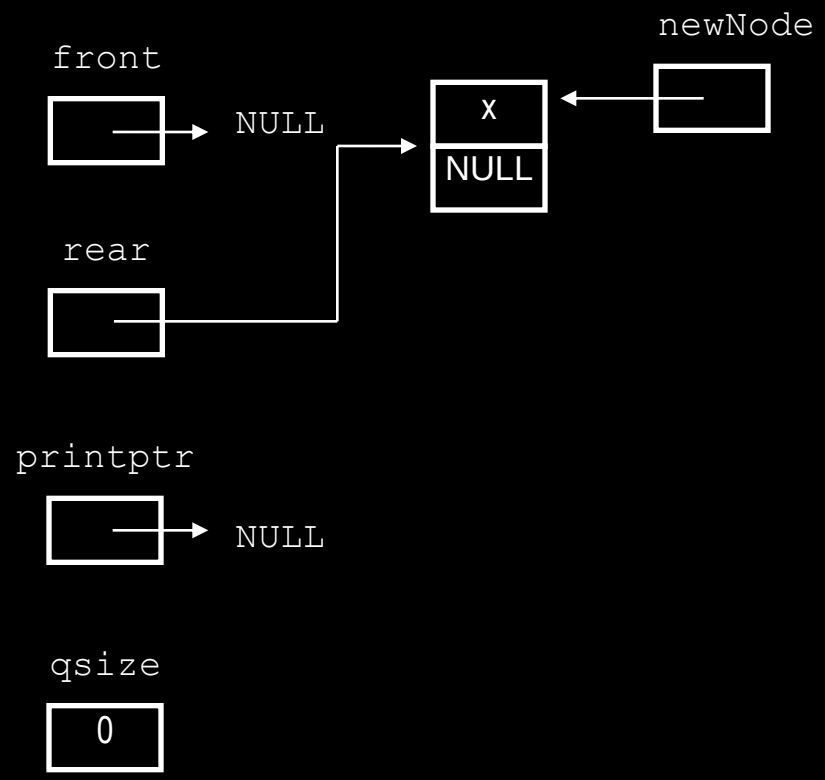
```
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    if (rear!=NULL)
        rear->setNext(newNode);
    rear = newNode;
    if (front==NULL) front=rear;
    qsize++;
}
```



Queue is Empty

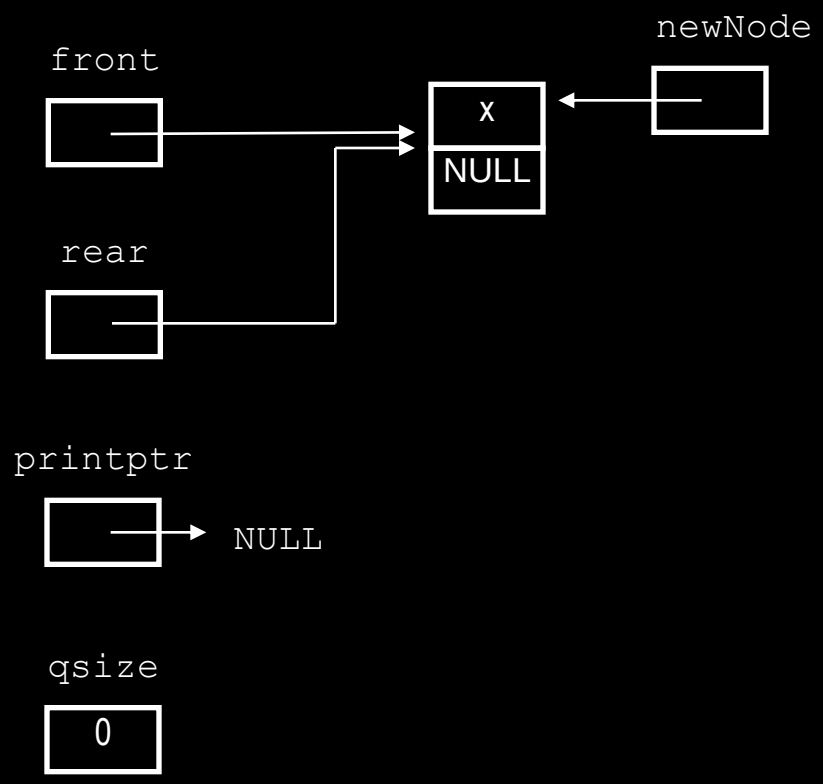
Queue Implementation: Linked List

```
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    if (rear!=NULL)
        rear->setNext(newNode);
    rear = newNode;
    if (front==NULL) front=rear;
    qsize++;
}
```



Queue Implementation: Linked List

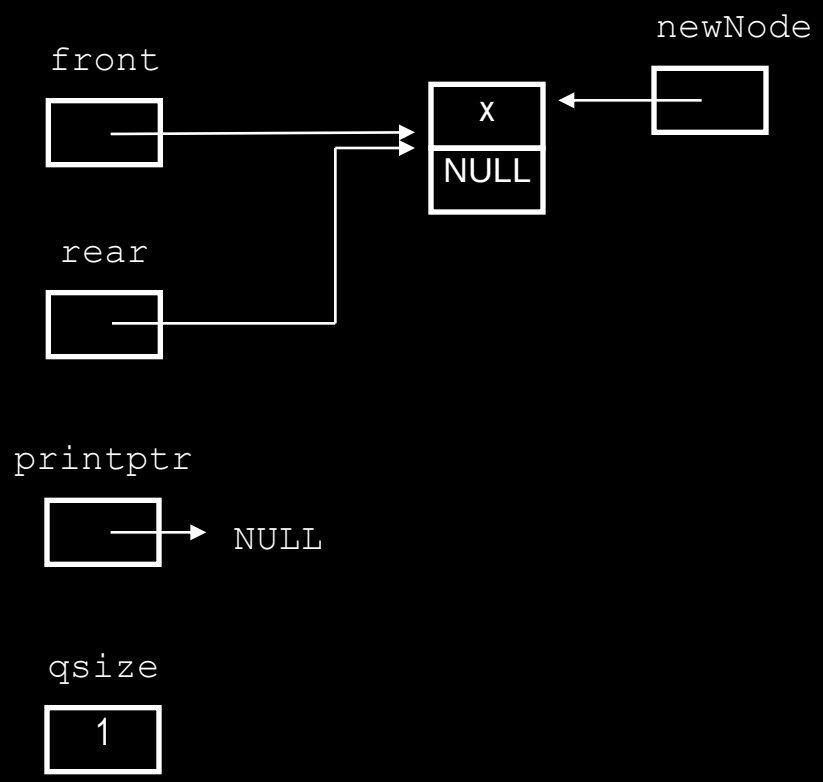
```
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    if (rear!=NULL)
        rear->setNext(newNode);
    rear = newNode;
    if (front==NULL) front=rear;
    qsize++;
}
```



Queue is Empty

Queue Implementation: Linked List

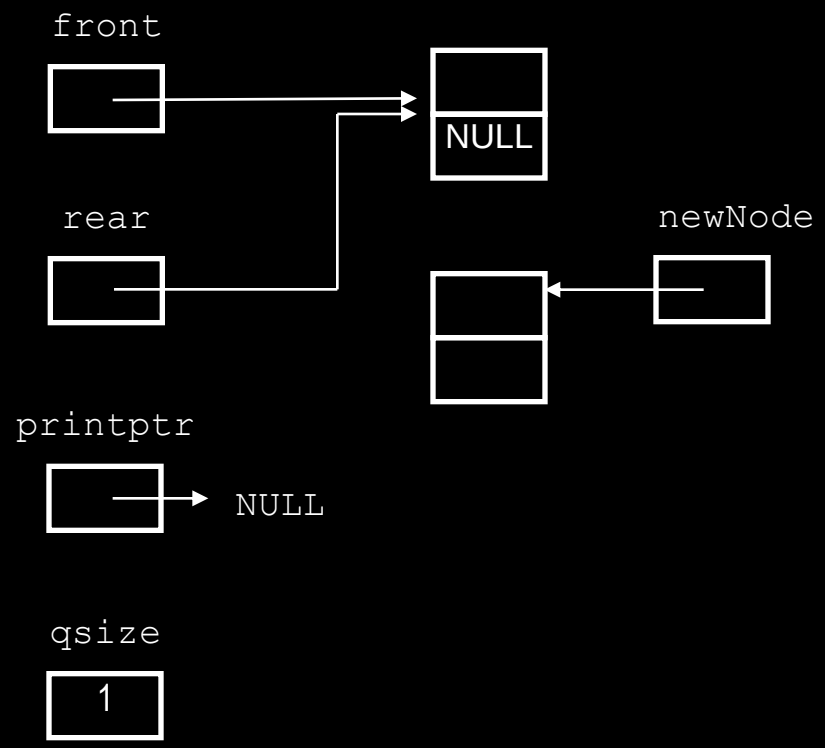
```
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    if (rear!=NULL)
        rear->setNext(newNode);
    rear = newNode;
    if (front==NULL) front=rear;
    qsize++;
}
```



Queue is Empty

Queue Implementation: Linked List

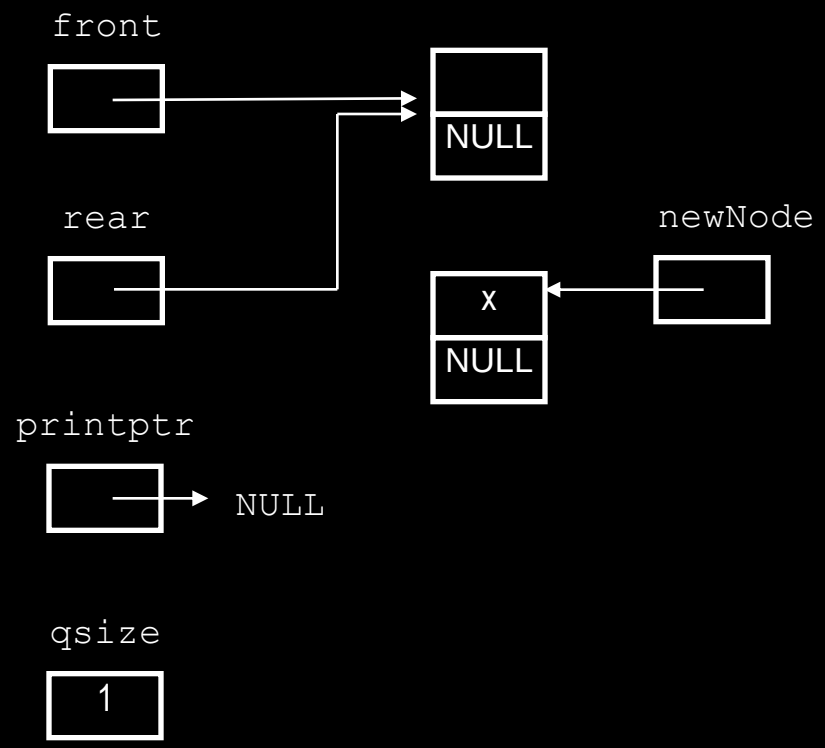
```
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    if (rear!=NULL)
        rear->setNext(newNode);
    rear = newNode;
    if (front==NULL) front=rear;
    qsize++;
}
```



Queue is NOT Empty

Queue Implementation: Linked List

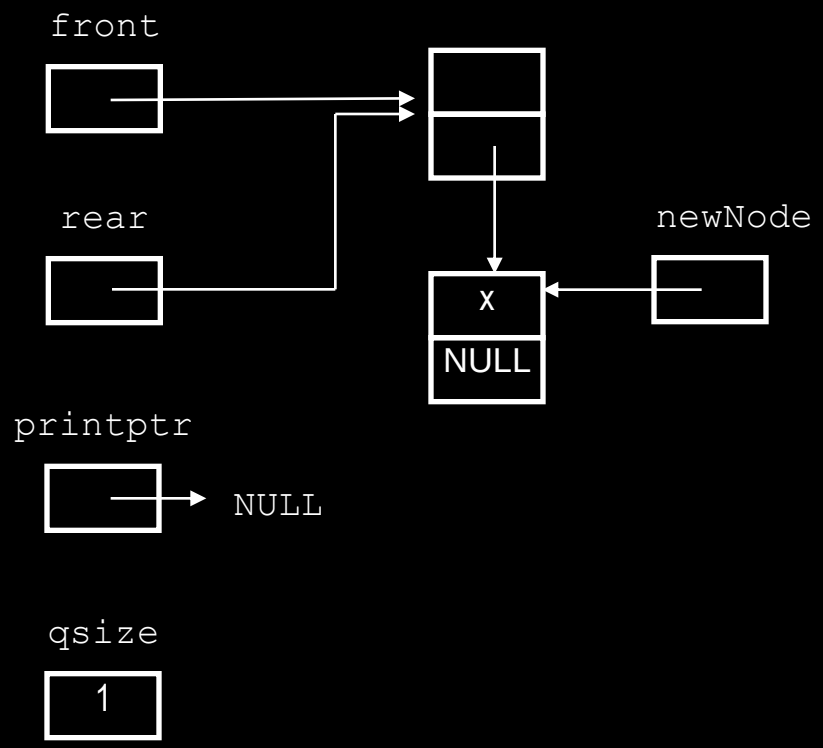
```
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    if (rear!=NULL)
        rear->setNext(newNode);
    rear = newNode;
    if (front==NULL) front=rear;
    qsize++;
}
```



Queue is NOT Empty

Queue Implementation: Linked List

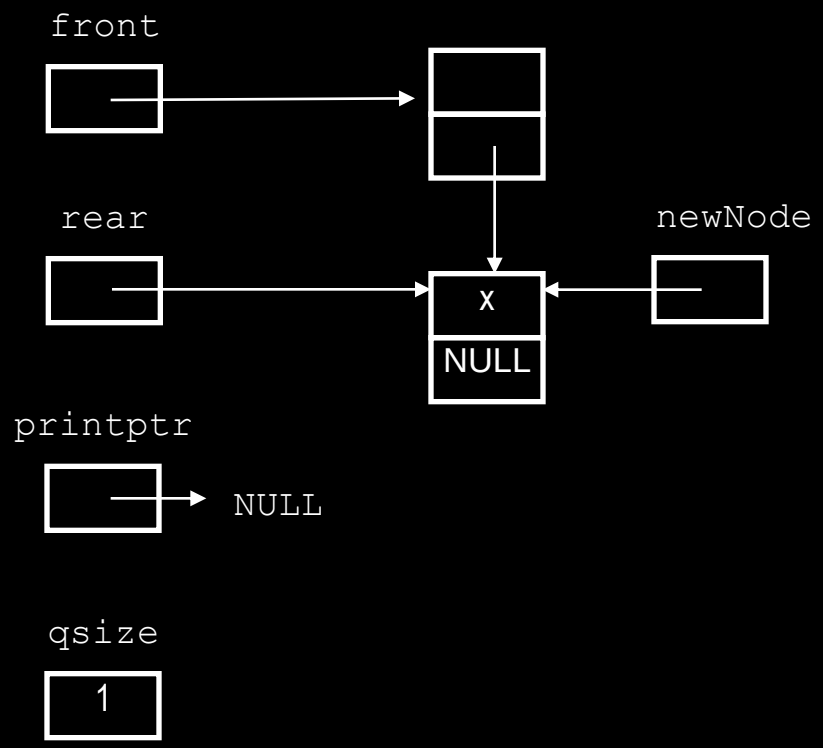
```
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    if (rear!=NULL)
        rear->setNext(newNode);
    rear = newNode;
    if (front==NULL) front=rear;
    qsize++;
}
```



Queue is NOT Empty

Queue Implementation: Linked List

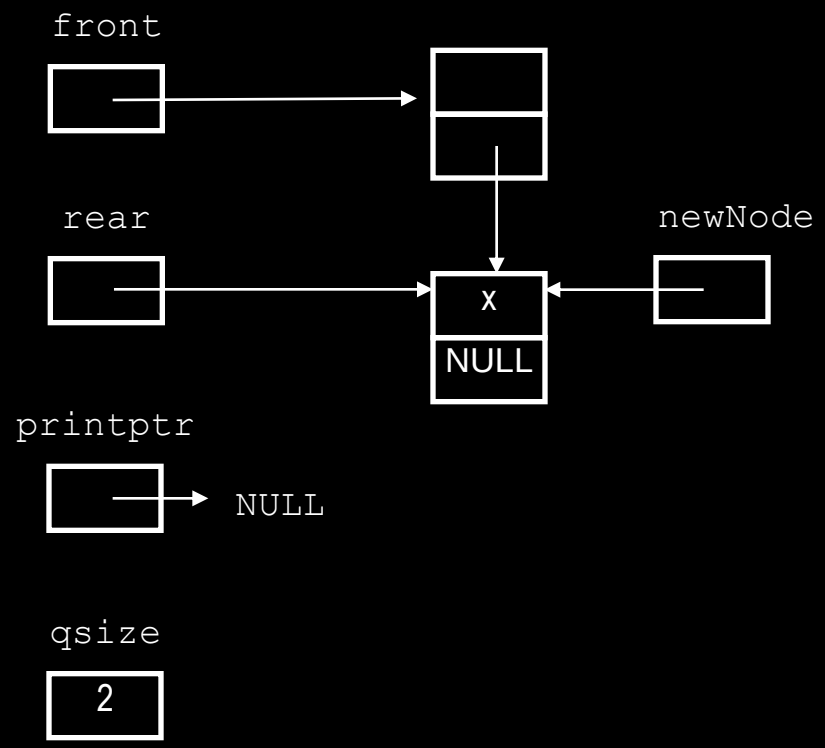
```
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    if (rear!=NULL)
        rear->setNext(newNode);
    rear = newNode;
    if (front==NULL) front=rear;
    qsize++;
}
```



Queue is NOT Empty

Queue Implementation: Linked List

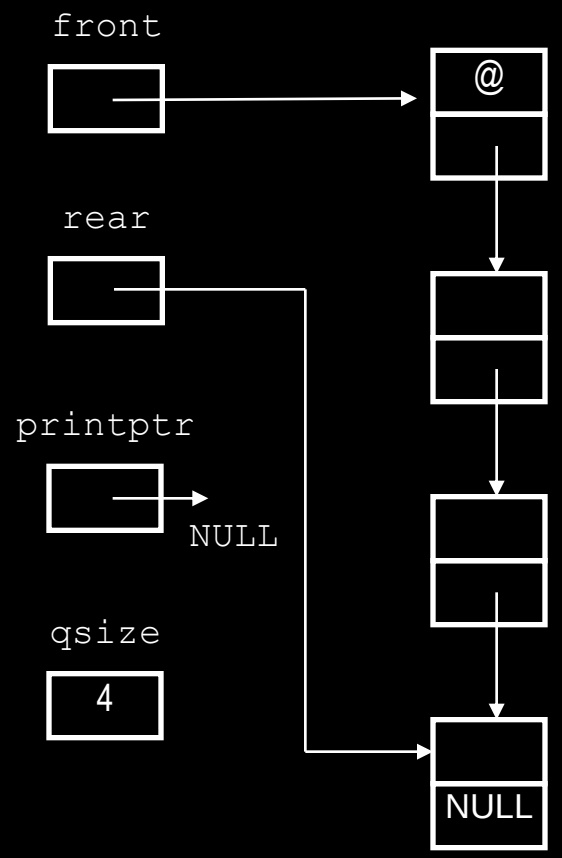
```
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    if (rear!=NULL)
        rear->setNext(newNode);
    rear = newNode;
    if (front==NULL) front=rear;
    qsize++;
}
```



Queue is NOT Empty

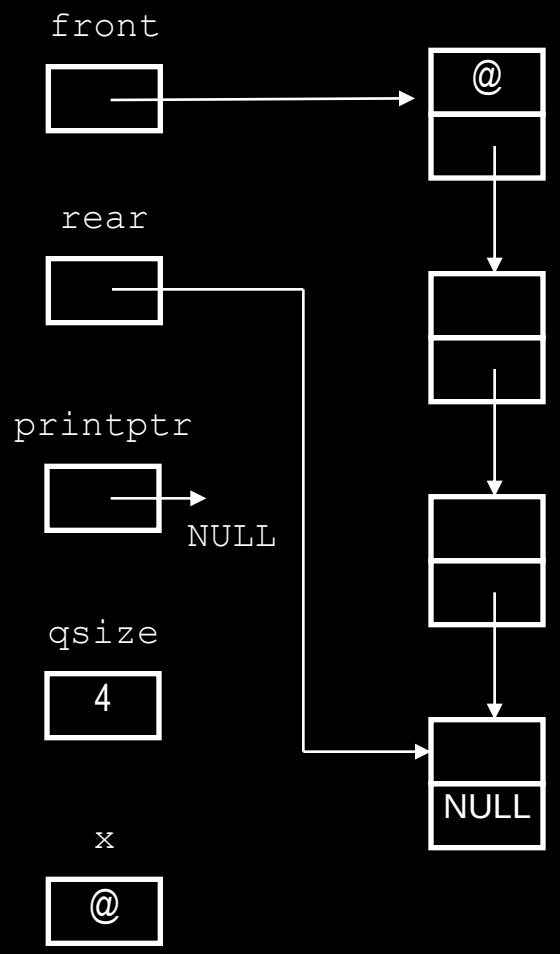
Queue Implementation: Linked List

```
int dequeue()
{
    int x = front->get();
    Node* p = front;
    front = front->getNext();
    delete p;
    qsize--;
    return x;
}
```



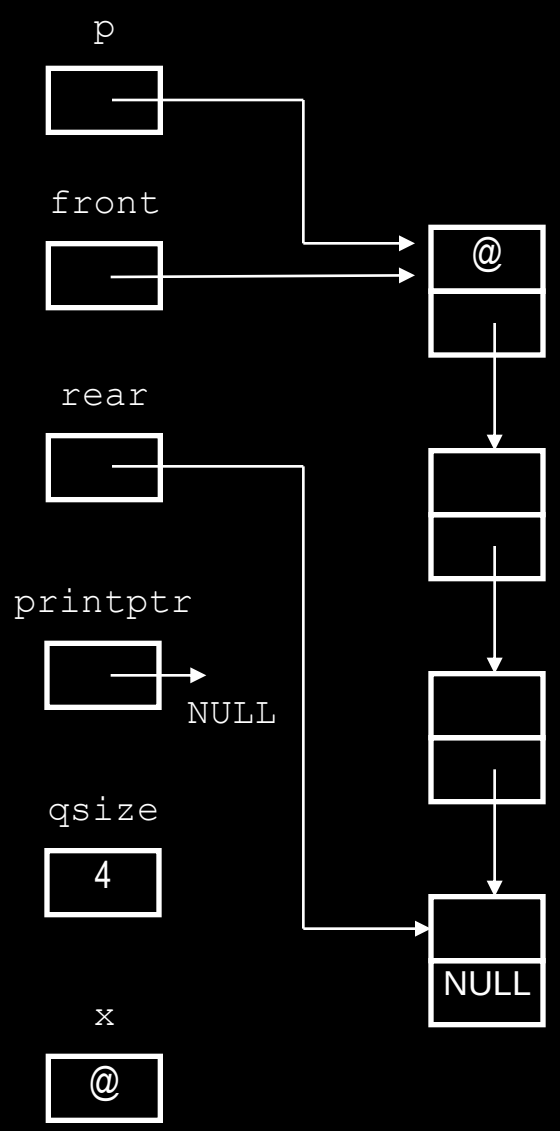
Queue Implementation: Linked List

```
int dequeue()
{
    int x = front->get();
    Node* p = front;
    front = front->getNext();
    delete p;
    qsize--;
    return x;
}
```



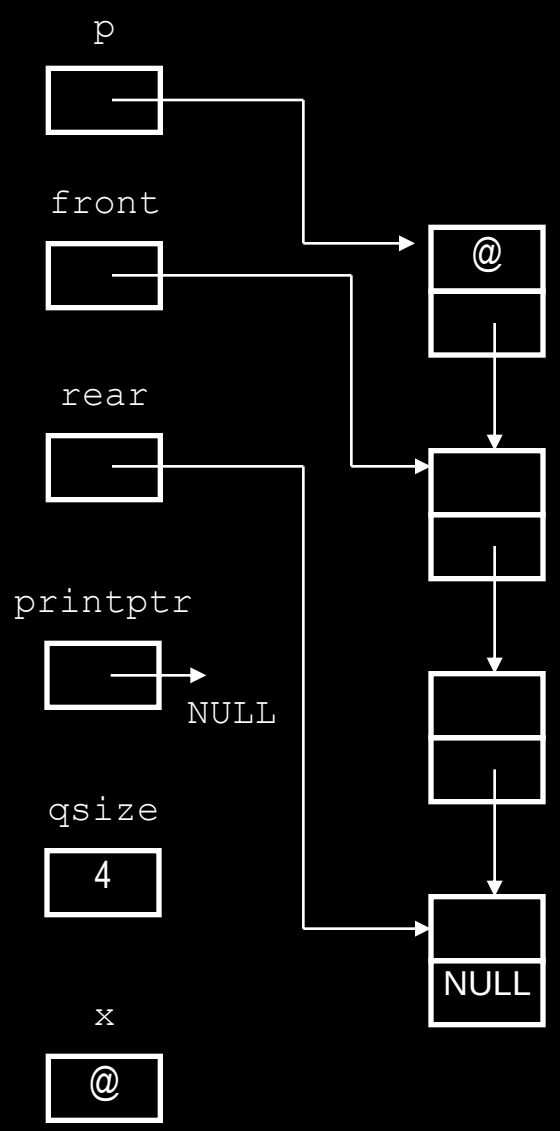
Queue Implementation: Linked List

```
int dequeue()
{
    int x = front->get();
    Node* p = front;
    front = front->getNext();
    delete p;
    qsize--;
    return x;
}
```



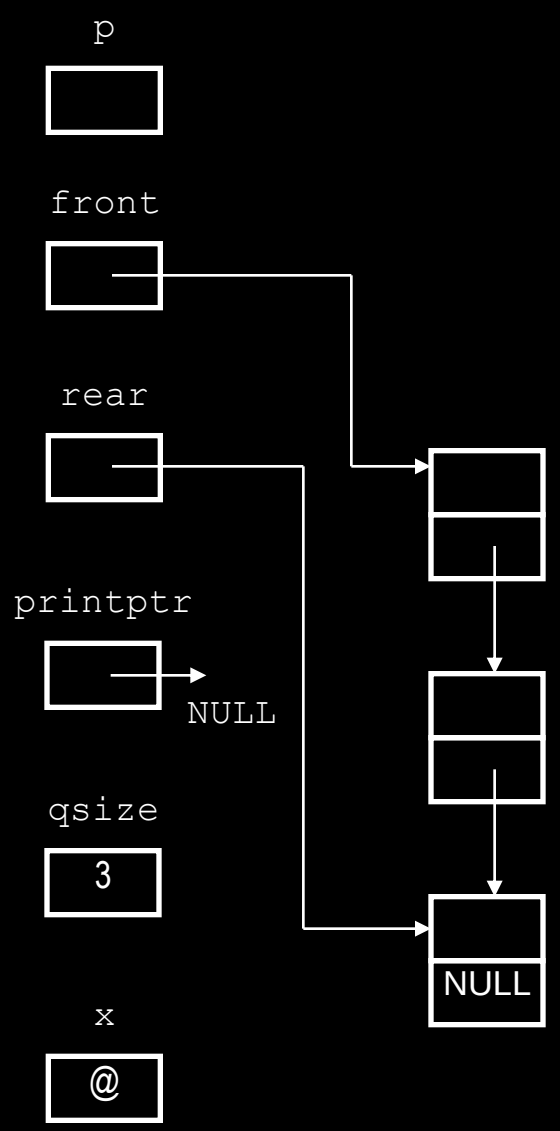
Queue Implementation: Linked List

```
int dequeue()
{
    int x = front->get();
    Node* p = front;
    front = front->getNext();
    delete p;
    qsize--;
    return x;
}
```



Queue Implementation: Linked List

```
int dequeue()
{
    int x = front->get();
    Node* p = front;
    front = front->getNext();
    delete p;
    qsize--;
    return x;
}
```



Queue Implementation: Linked List

```
int atfront()  {  return front->get();  }

int IsEmpty()  {  return ( front == NULL ); }

int size()     {  return qsize;  }

void print()
  {
  printptr = front;
  while (printptr!=NULL)
    {
    cout << printptr->get() << " ";
    printptr=printptr->getNext();
    }
  }
};
```

Queue Implementation: Linked List

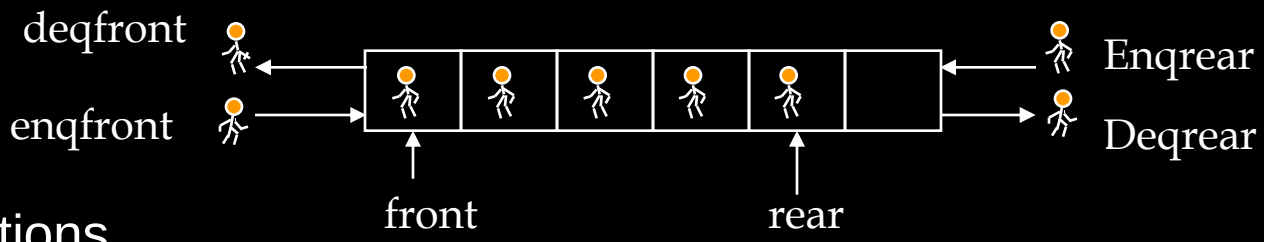
```
int main()
{
 queue Q;

 char a='$';
 while (a!='~')
   {
    cout << "Enter value to enqueue, '~' to terminate: ";
    cin >> a;
    if (a!='~') Q.enqueue(a);
   }

 if (Q.IsEmpty()==0)
     { cout << "Queue is: ";
       Q.print();
     }
```

Queue Implementation: Linked List

```
int choice=1;
while (!((choice>2)||(choice<1)))
 {
   cout << "\n1. enqueue      2. dequeue     3. Exit    :Enter your
   choice  ";
   cin  >> choice;
   if (choice == 1)
   { cout << "\nEnter No. to enqueue: "; cin >> a; Q.enqueue(a);
     cout << "\nQueue is: "; Q.print(); }

   if (choice == 2)
     {if (Q.IsEmpty() == 0)
       { a=Q.dequeue();cout << "\n" << a << " has been dequeued from Q";
         cout << "\nQueue is ";
         if (Q.IsEmpty()==1) cout << "Empty";
         else Q.print(); }
      else
         cout << "\nQueue is empty";
     }
 }
}
```

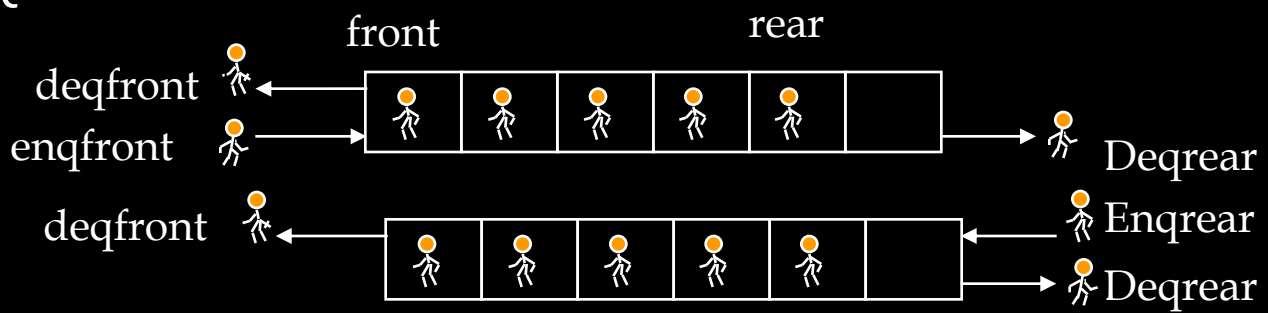
Double Ended Queue - DEQ

- Insert at both rear & front, remove from both rear & front
- Maintain front and rear pointers

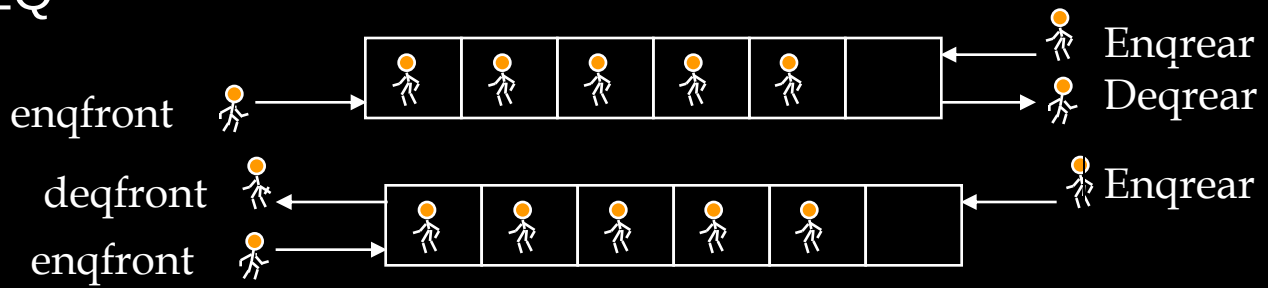


Two variations

- Input – Restricted DEQ



- output – Restricted DEQ



Double Ended Queues

Main DEQ operations:

enqfront(Object o): inserts an element o at the front of the queue

enqrear(Object o): inserts an element o at the front of the queue

deqfront(): remove and return the element at the front of the queue

deqrear(): remove and return the element at the front of the queue

front(): returns the element at the front without removing it

Rear(): returns the element at the rear without removing it

Auxiliary DEQ operations:

size(): returns the number of elements stored

isEmpty(): returns 1 indicating queue is empty
returns 0 indicating queue is not empty

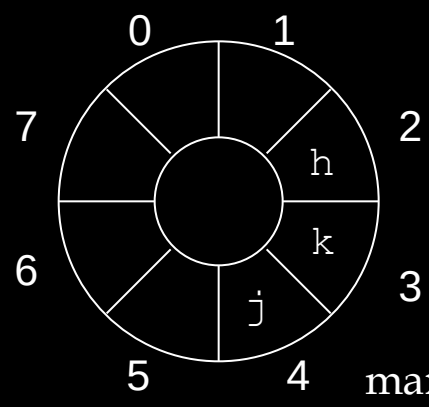
isFull(): return 1 indicating queue is full
return 0 indicating queue is not full

Double Ended Queue Implementation: Array

```
#include<iostream.h>
class queue
{
private:
   int front, rear, qsize, maxqsize, printptr;
   int *array;

public:
queue(int x)
{
   maxqsize =  x;
   array=new int[maxqsize];
   printptr = 0;
   qsize=0;
   rear = -1;
   front = 0;
}
```

Double Ended Queue Implementation: Array

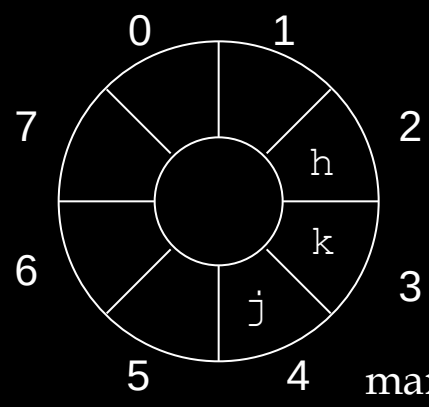
```
void enqrear(int x) // Example 1
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 3
front = 2
maxsize = 8
rear = 4

Double Ended Queue Implementation: Array

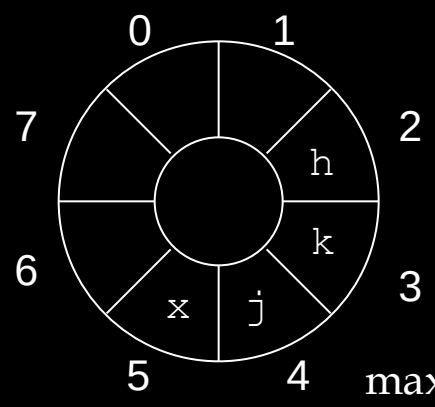
```
void enqrear(int x) // Example 1
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 3
front = 2
maxsize = 8
rear = 5

Double Ended Queue Implementation: Array

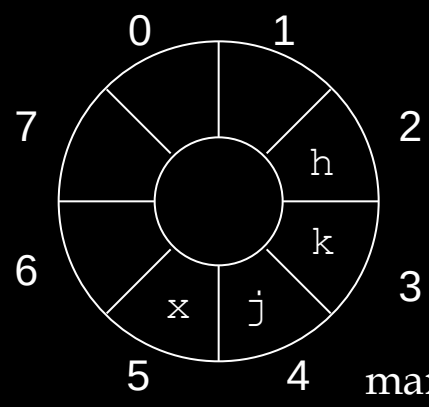
```
void enqrear(int x)  // Example 1
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 3
front = 2
maxsize = 8
rear = 5

Double Ended Queue Implementation: Array

```
void enqrear(int x)  //Example 1
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 4

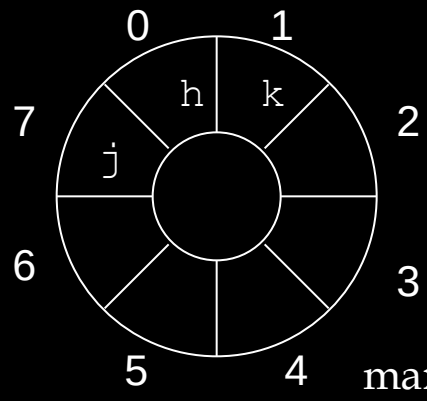
front = 2

maxsize = 8

rear = 5

Double Ended Queue Implementation: Array

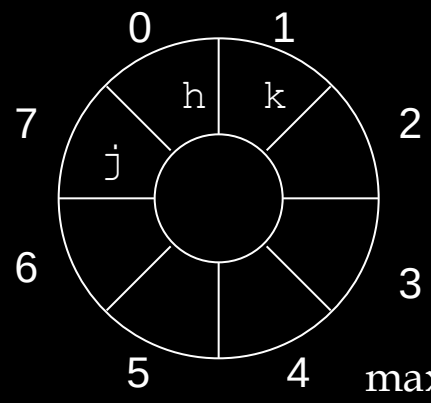
```
void enqrear(int x)  //Example 2
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 3
front = 7
maxsize = 8
rear = 1

Double Ended Queue Implementation: Array

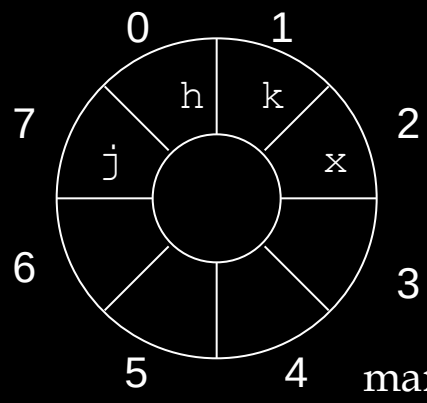
```
void enqrear(int x)  //Example 2
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 3
front = 7
maxsize = 8
rear = 2

Double Ended Queue Implementation: Array

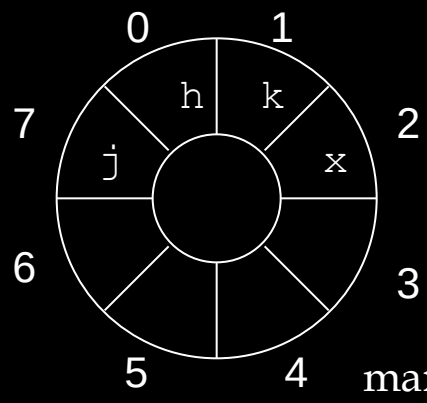
```
void enqrear(int x)  //Example 2
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 3
front = 7
maxsize = 8
rear = 2

Double Ended Queue Implementation: Array

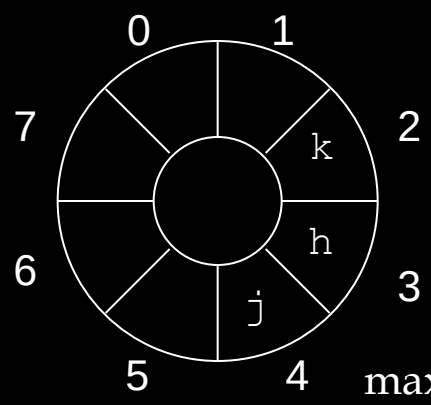
```
void enqrear(int x)  //Example 2
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 4
front = 7
maxsize = 8
rear = 2

Double Ended Queue Implementation: Array

```
void enqrear(int x)  //Example 2
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)//Example 1
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 3

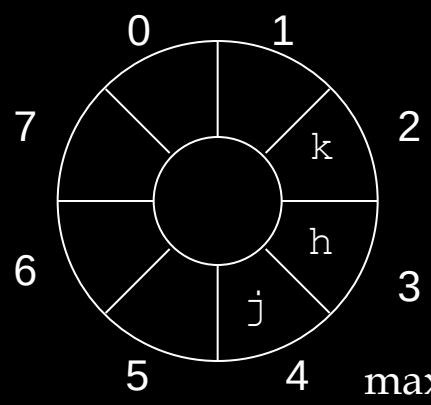
front = 2

maxsize = 8

rear = 4

Double Ended Queue Implementation: Array

```
void enqrear(int x)  //Example 2
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)//Example 1
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 3

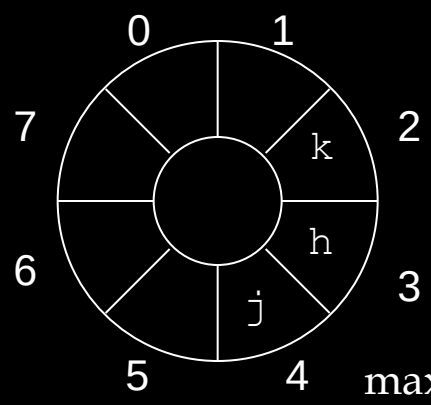
front = 1

maxsize = 8

rear = 4

Double Ended Queue Implementation: Array

```
void enqrear(int x)  //Example 2
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)//Example 1
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 3

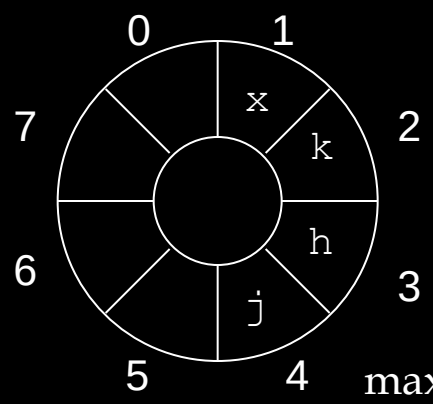
front = 1

maxsize = 8

rear = 4

Double Ended Queue Implementation: Array

```
void enqrear(int x)  //Example 2
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)//Example 1
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 3

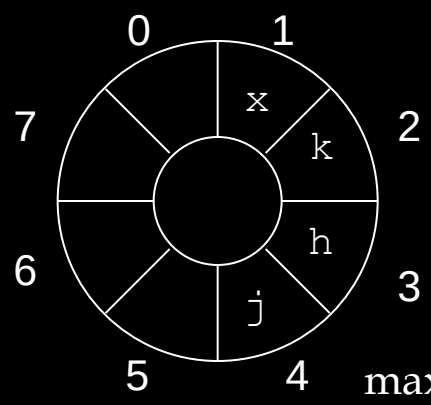
front = 1

maxsize = 8

rear = 4

Double Ended Queue Implementation: Array

```
void enqrear(int x)  //Example 2
{
   rear = (rear+1)% maxqsize;
   array[rear] = x;
   qsize++;
}

void enqfront(int x)//Example 1
{
   front = front-1;
   if (front<0) front=maxqsize+front;
   array[front] = x;
   qsize++;
}
```



size = 4

front = 1

maxsize = 8

rear = 4

Double Ended Queue Implementation: Array

```
int deqrear()
{
   int x=array[rear];
   rear = rear-1;
   if (rear<0) rear=maxqsize+rear;
   qsize--;
   return x;
}

int deqfront()
{
   int x = array[front];
   front = (front+1)% maxqsize;
   qsize--;
   return x;
}
```

Double Ended Queue Implementation: Array

```
int atfront() { return array[front]; }

int atrear() {return array[rear]; }

int IsFull() { return (qsize == maxqsize); }

int IsEmpty() { return (qsize == 0); }

int size() { return qsize; }
```

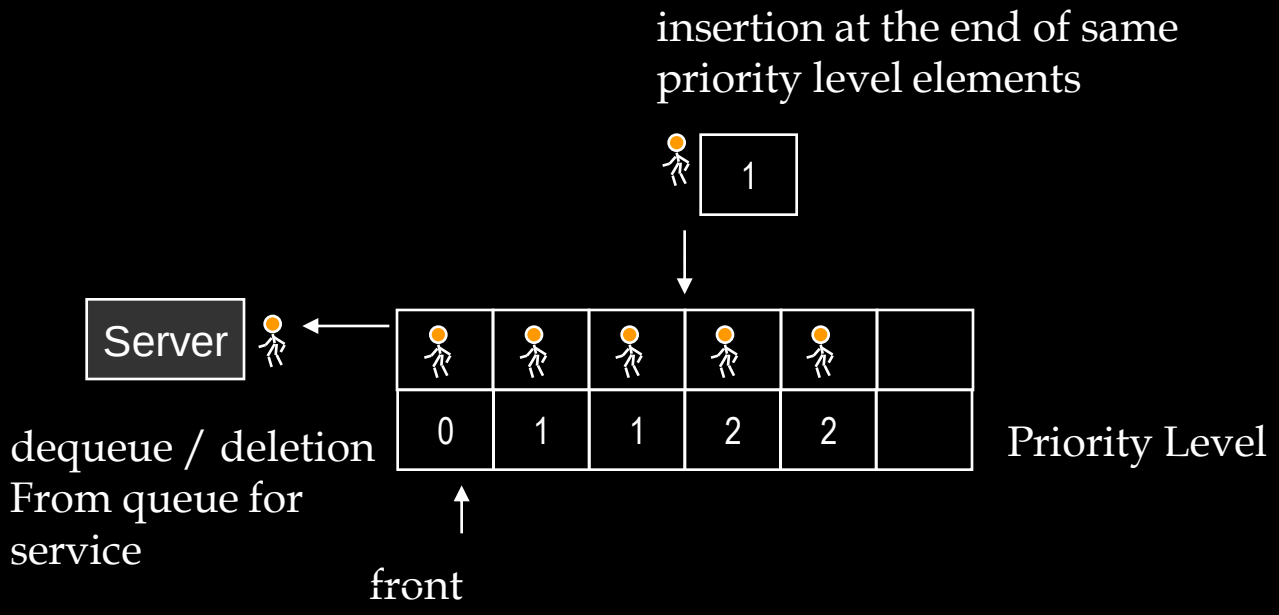
Double Ended Queue Implementation: Array

```
void print()
{
printptr=front;
while (printptr!=rear)
    {
    cout << array[printptr] << " ";
    printptr=( (printptr+1)%maxqsize );
    }
cout << array[printptr];
}
};
```

Priority Queues

A Priority Queue is a collection of elements such that each element is assigned a priority and the order of removal (for service or processing) comes from the following rules

1. An element of higher priority is processed before any element of lower priority
2. Two elements with the same priority are processed according to the order in which they are inserted in the Queue

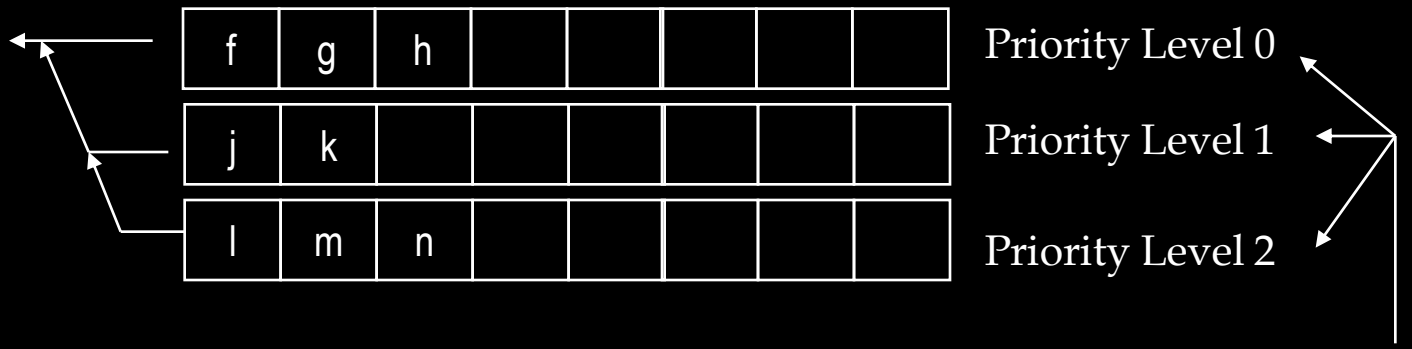


Generally 0 is the highest Priority level

Priority Queues - Implementation

1. One Dimentional Circular Arrays

Use a separate Queue for each level of Priority, each such Queue is represented in circular fashion and has it's own front and rear pointers



Dequeue

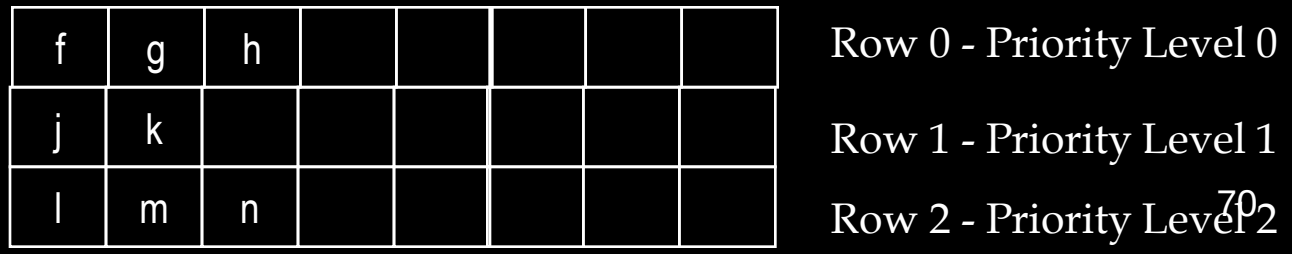
Remove from front of Q-0, if there are elements
Remove from front of Q-1, if Q-0 is empty
Remove from front of Q-2 is Q-0 and Q-1 are empty

Enqueue

Insert at rear of the Q of same Priority Level

2. Two dimentional Circular Arrays

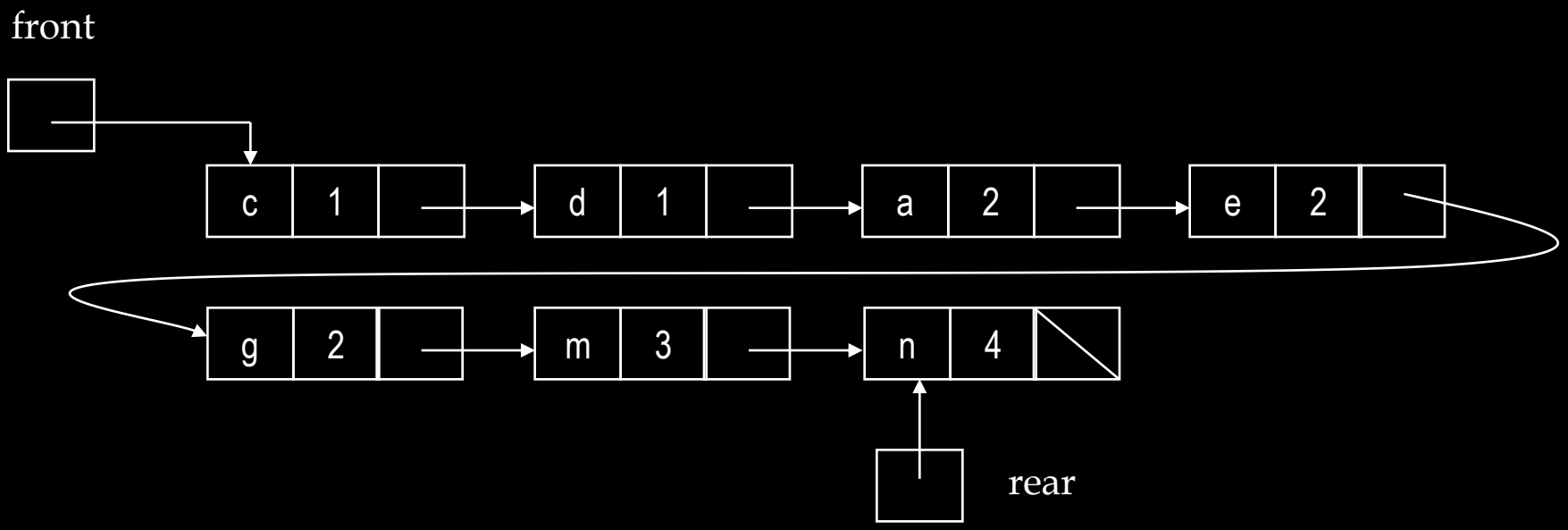
Use a single Two Dimensional Array, where row number represents the priority level and each row represents (in circular manner) a Queue, with separate pointers (front and rear)



Priority Queues - Implementation

1. Single Linked List

One additional field for Priority level in each node,



2. Double Linked List

Can You describe the advantage & Disadvantage of using Double Linked List over Single Linked List after the end of this Session,

Priority Queues

Main queue operations:

enqueue(Object o): inserts an element o at the end of the elements of same priority in the queue

dequeue(): remove and return the element at the front of the queue

front(): returns the element at the front without removing it

Auxiliary queue operations:

size(): returns the number of elements stored

isEmpty(): returns 1 indicating queue is empty
returns 0 indicating queue is not empty

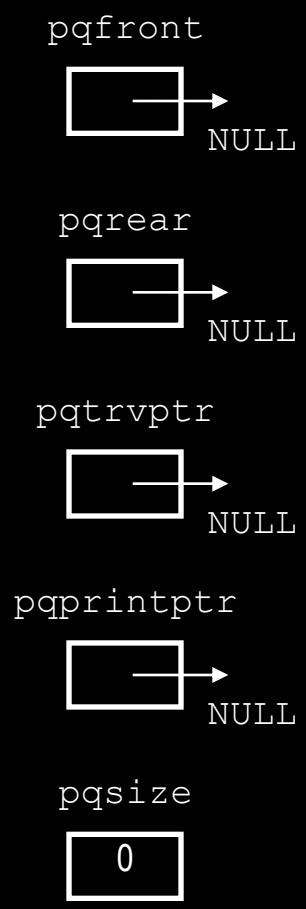
isFull(): return 1 indicating queue is full
return 0 indicating queue is not full

Priority Queue Implementation: Linked List

```
#include <iostream.h>
class pqueue {
private:
   class Node {
     private:
         int data;
         int Priority;
         Node *nextNode;

     public:
         int get()  { return data; };
         int getP() { return Priority;};
         void set(int data, int Priority)
                   { this->data = data; this->Priority=Priority;};

         Node* getNext()       { return nextNode; };
         void setNext(Node *nextNode)
                              { this->nextNode = nextNode; };
             };
   int pqsize;
   Node *pqfront,*pqrear;
   Node *pqprintptr,*pqtrvptr;
```

Priority Queue Implementation: Linked List

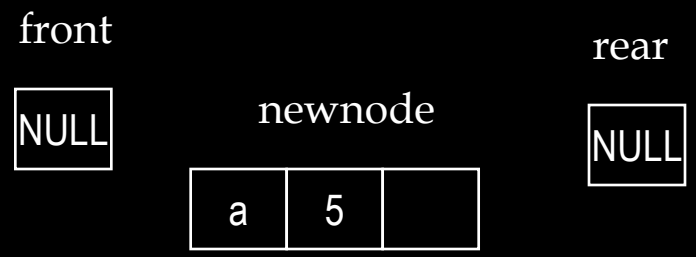
```
public:
   // Constructor
   pqueue() {
        pqfront = NULL;
        pqrear = NULL;
        pqtrvptr = NULL;
        pqprintptr = NULL;
        pqsize = 0;
   };
```

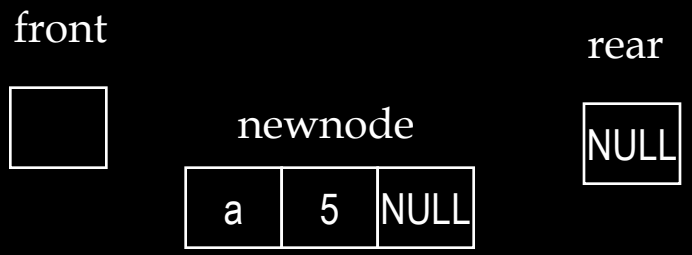


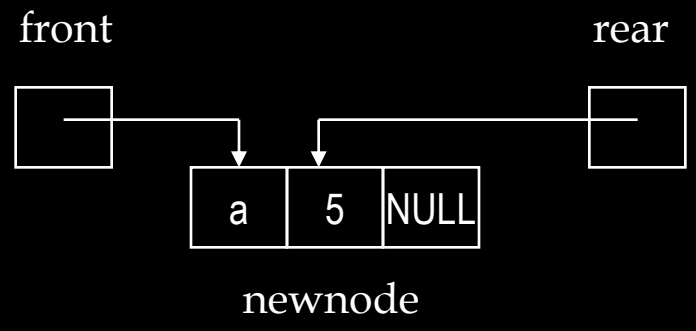
Enqueue in Priority Queue - Implementation

Four Cases: enqueue(a,5)

1. List is empty

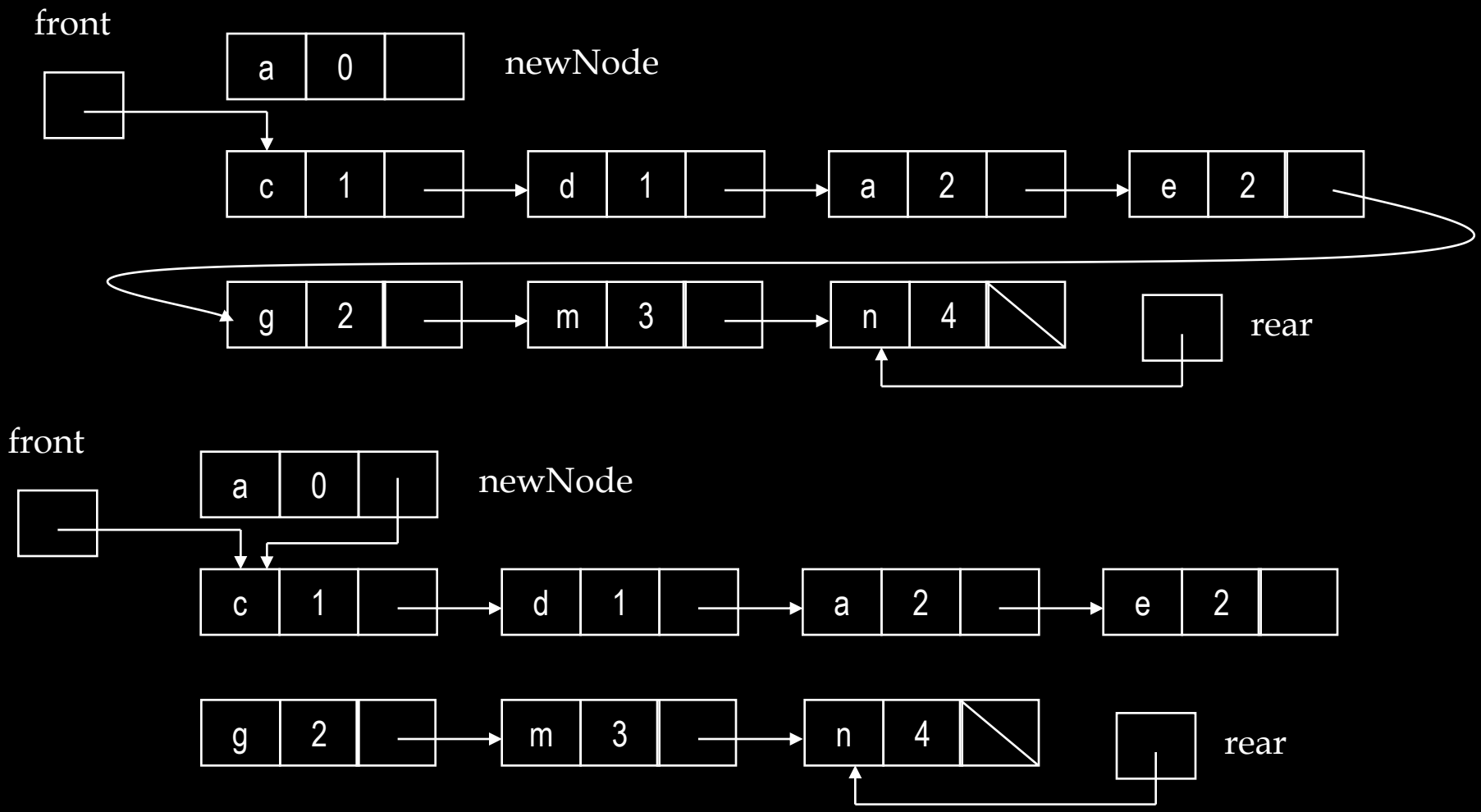






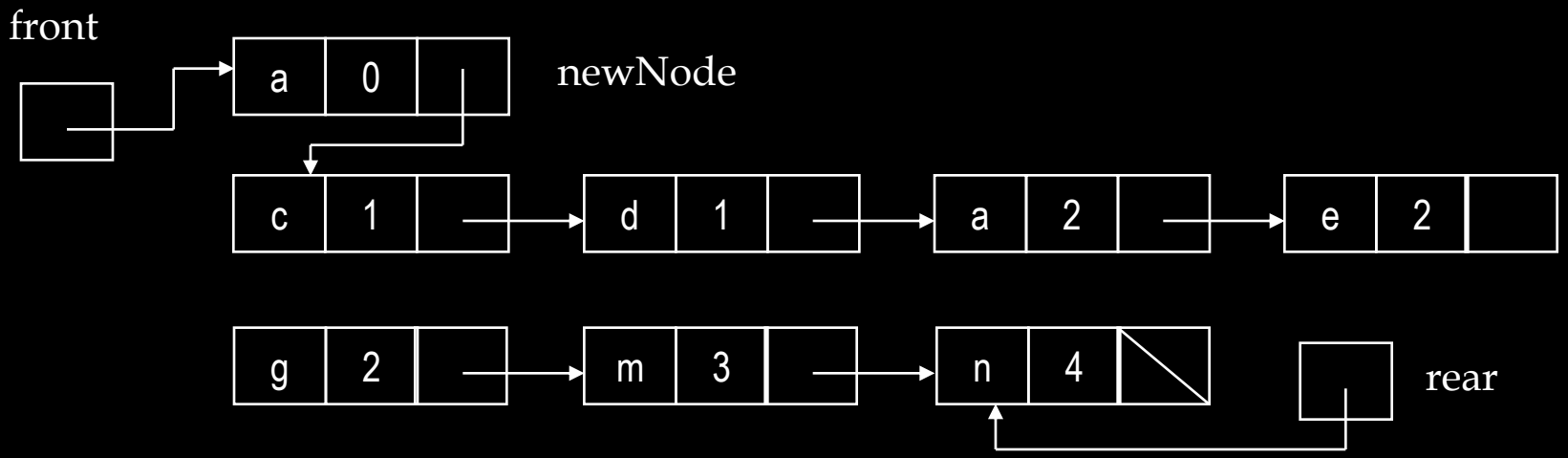
Enqueue in Priority Queue - Implementation

2. newNode priority > front node priority, insert on front - enqueue(a,0)



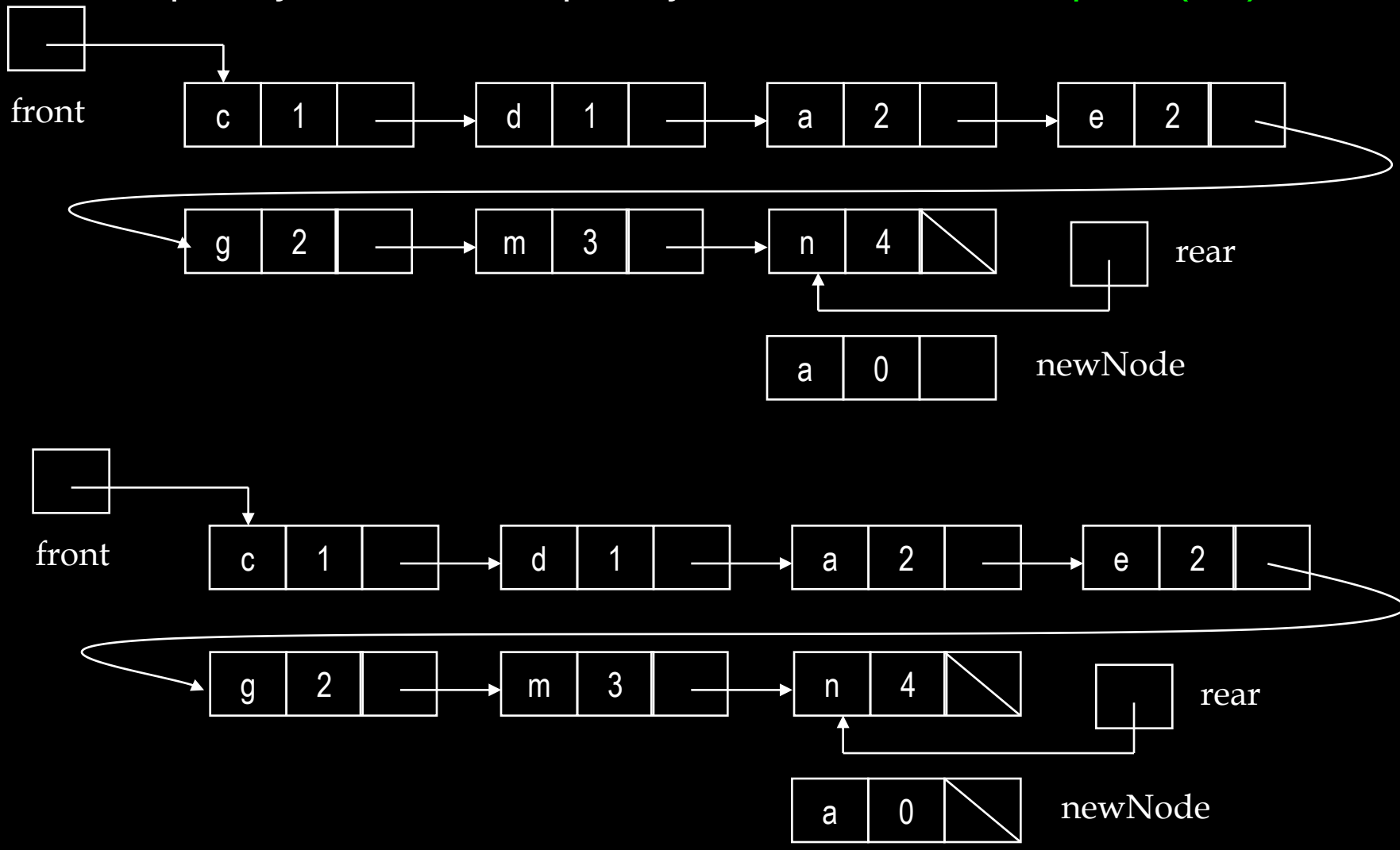
Enqueue in Priority Queue - Implementation

2. **(cont..)** newNode priority > front node priority, insert on front - enqueue(a,0)



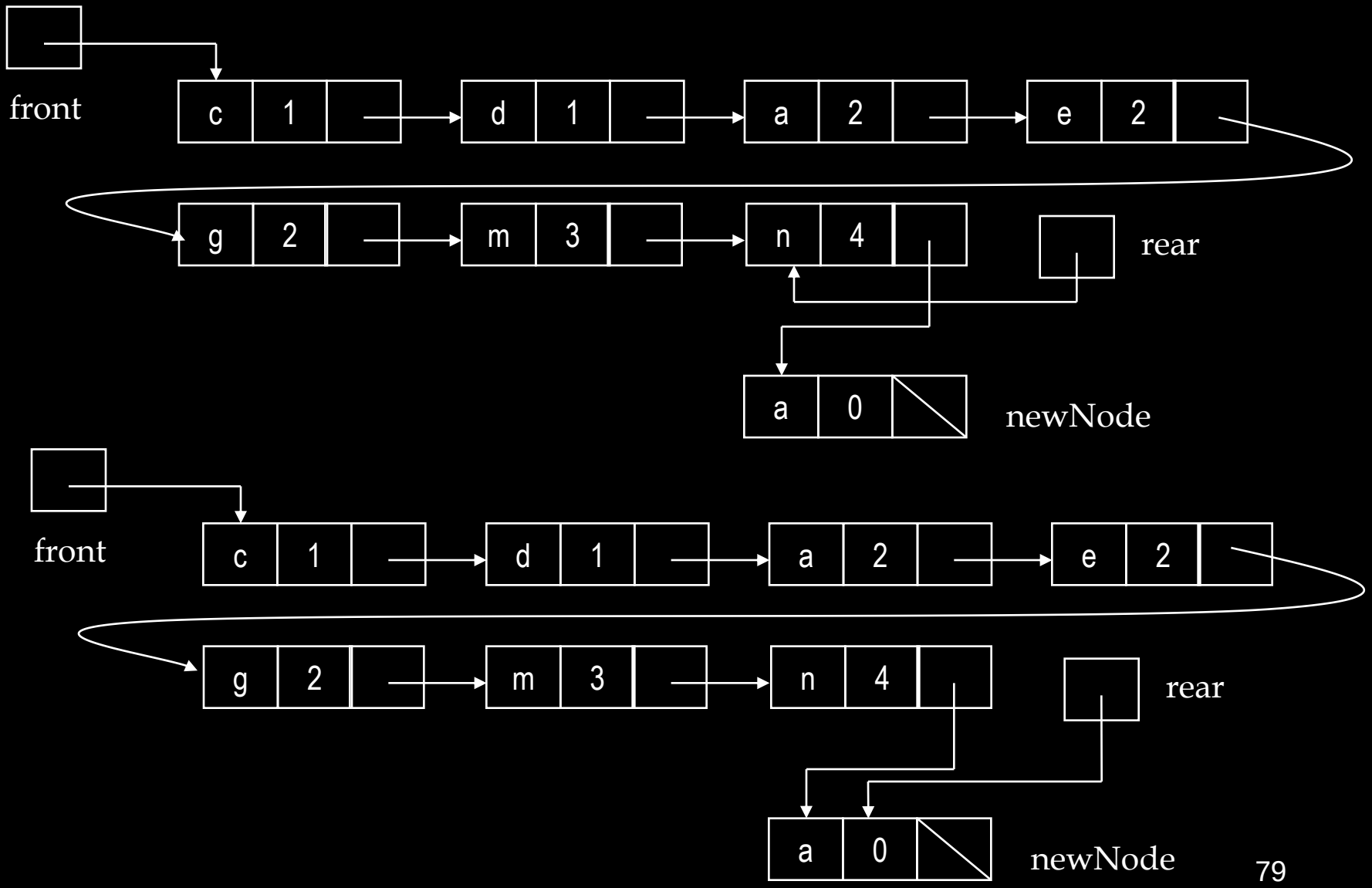
Enqueue in Priority Queue - Implementation

3. newNode priority <= rear node priority, insert on rear - enqueue(a,4)



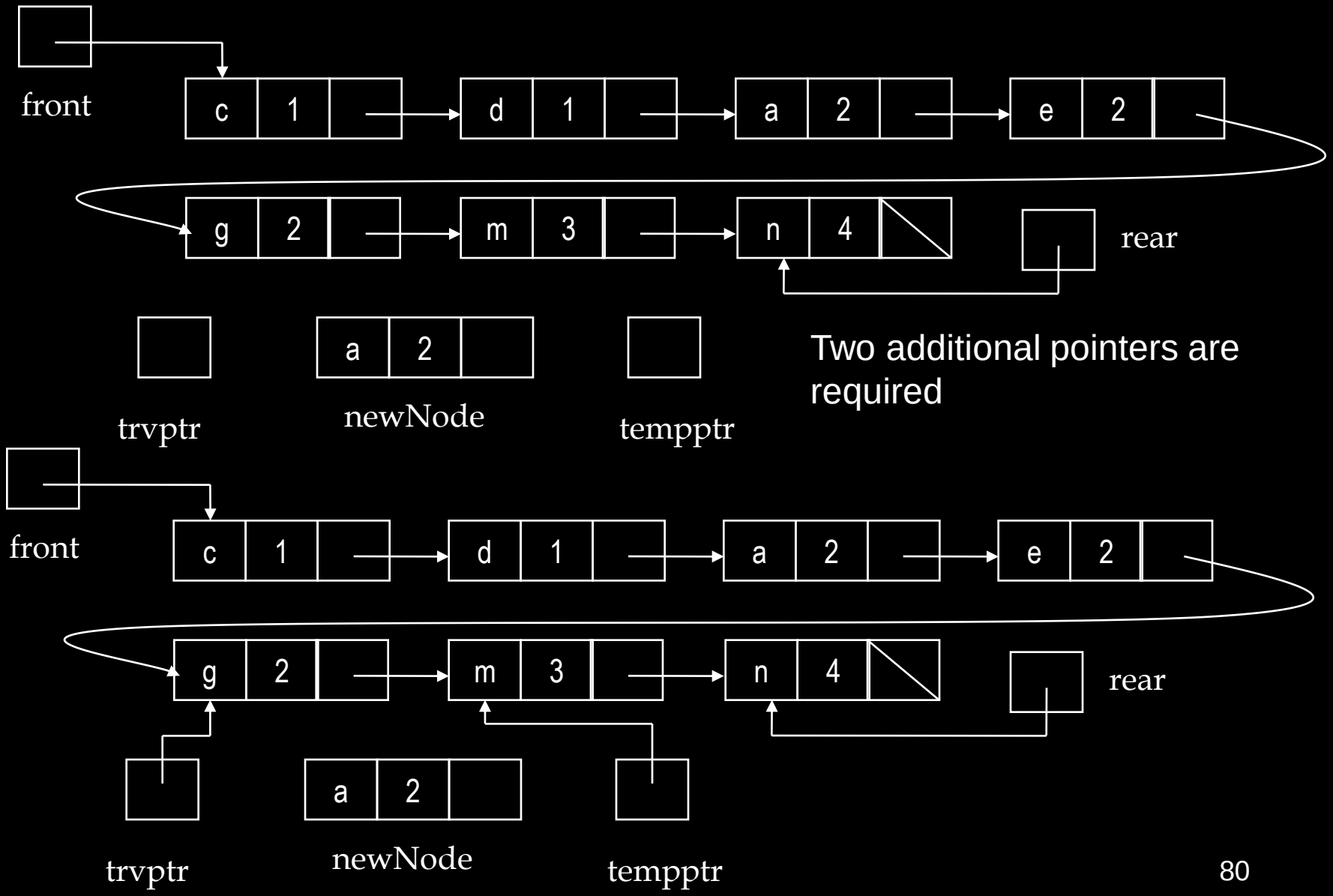
Enqueue in Priority Queue - Implementation

3. (cont..) newNode priority <= rear node priority, insert on rear - enqueue(a,4)



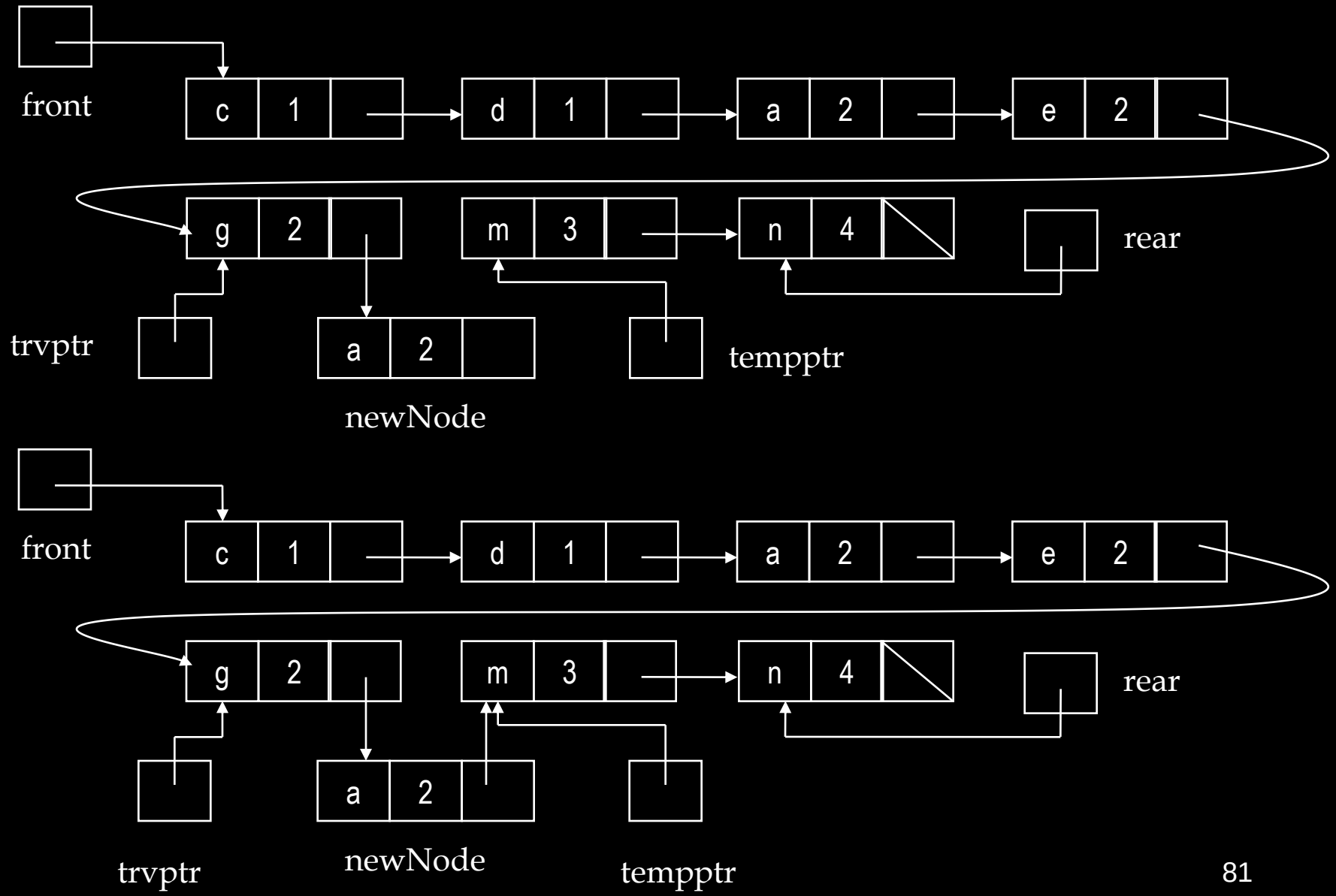
Enqueue in Priority Queue - Implementation

4. newNode priority <= front node priority AND >rear node priority-enqueue(a,2)



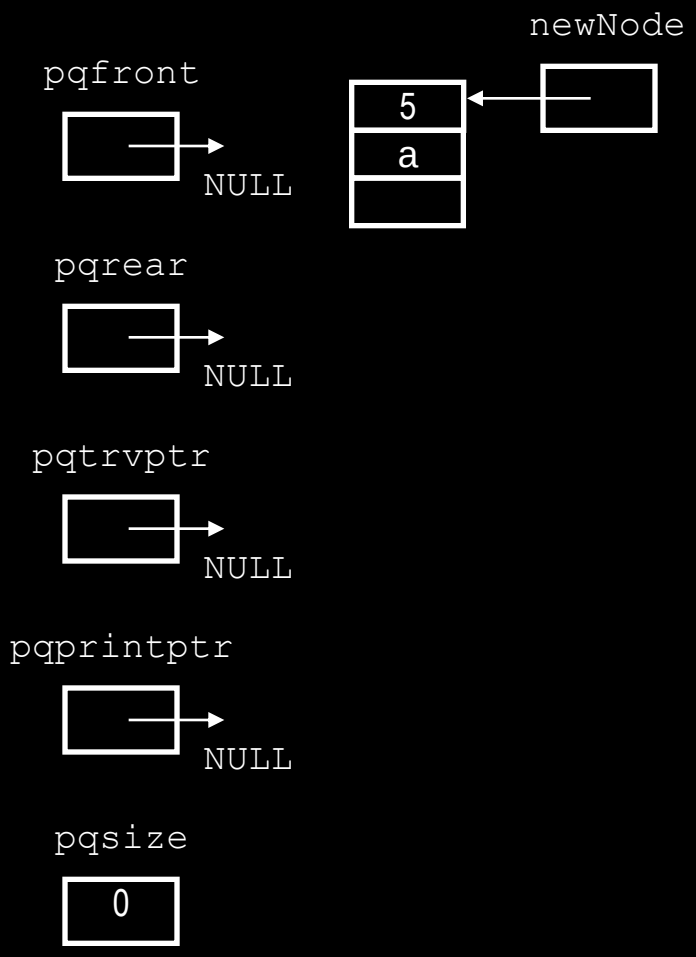
Enqueue in Priority Queue - Implementation

4. (cont..)newNode priority <= front node priority AND >rear node priority



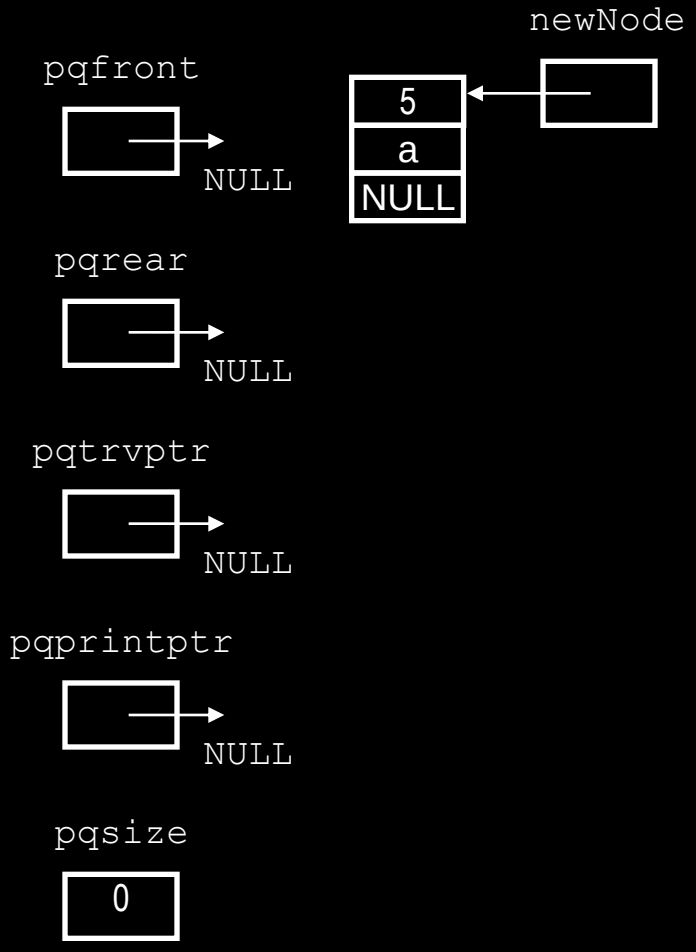
Priority Queue Implementation: Linked List

```
void enqueue(int x, int P)
{
    Node* newNode = new Node();
    newNode->set(x,P);
```



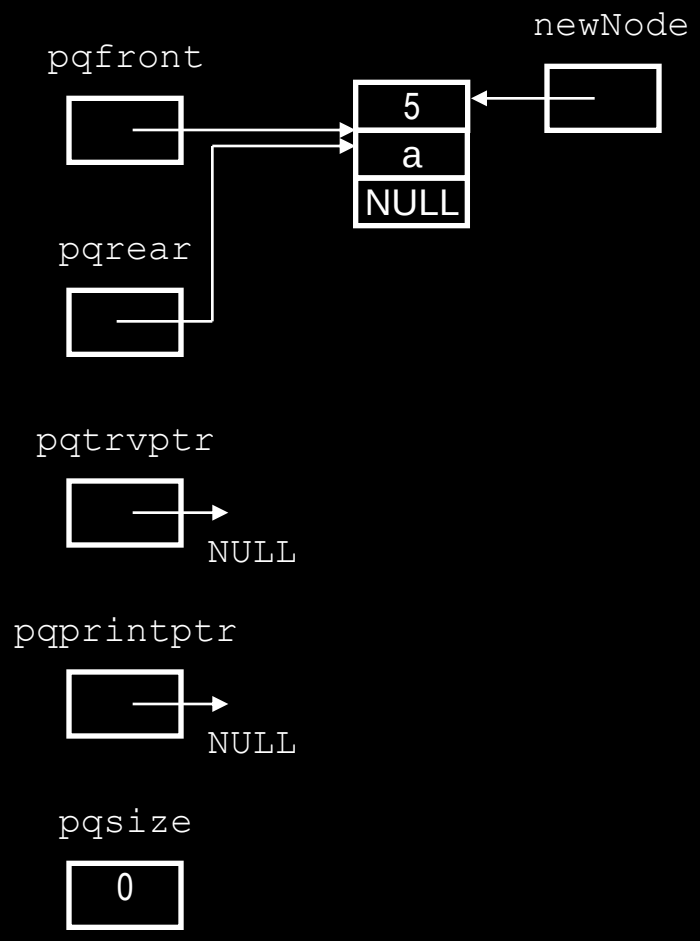
Priority Queue Implementation: Linked List

```
//Q is Empty -1-1
    if ((pqfront==NULL)||(pqrear==NULL))
    {
    newNode->setNext(NULL);
    pqfront=newNode;
    pqrear=newNode;
    }

//enqueue(a,5);
```



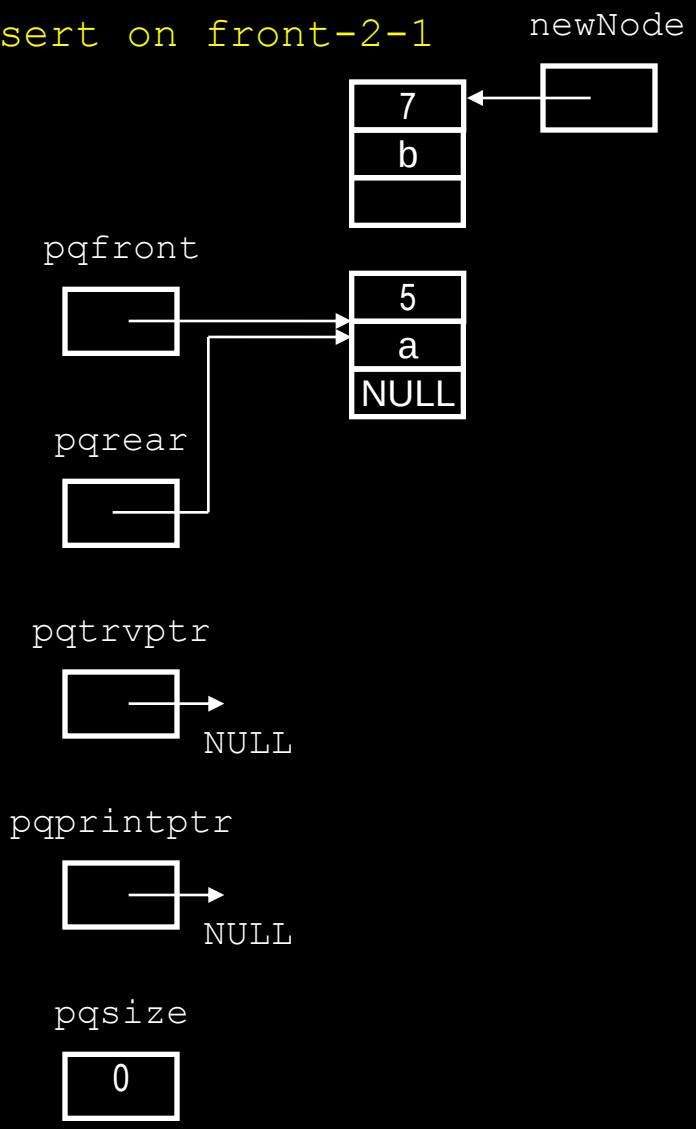
Priority Queue Implementation: Linked List

```
//Q is Empty -1-2
    if ((pqfront==NULL)||(pqrear==NULL))
    {
    newNode->setNext(NULL);
    pqfront=newNode;
    pqrear=newNode;
    }

//enqueue(a,5);
```



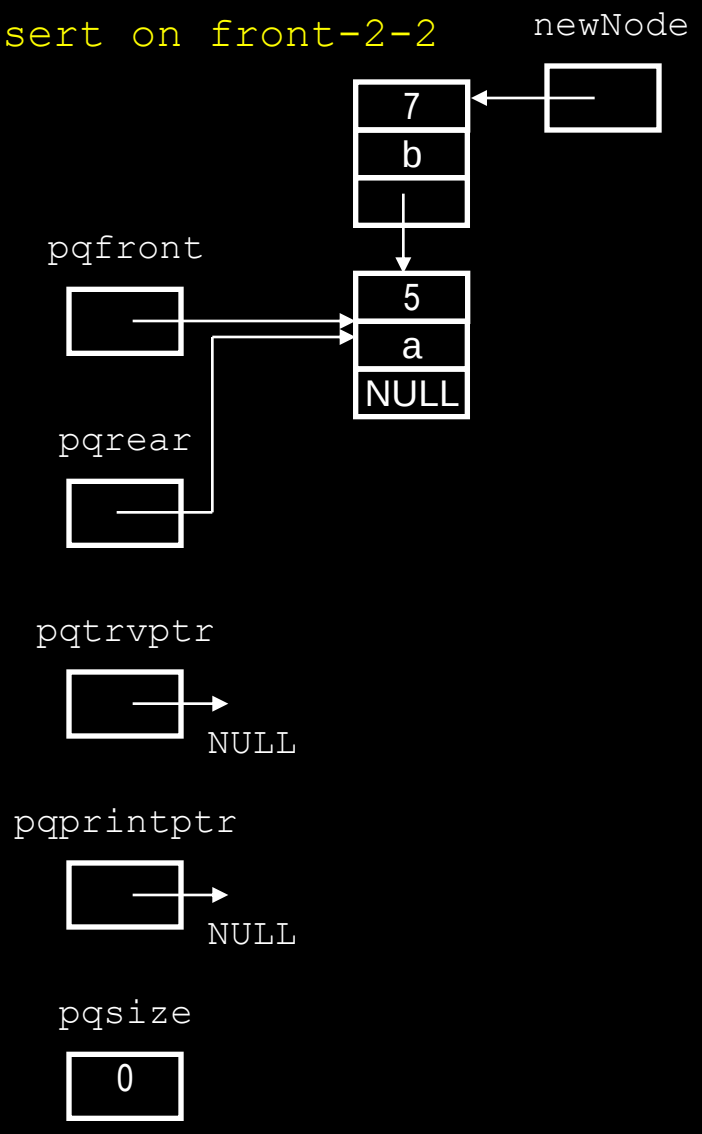
Priority Queue Implementation: Linked List

```
//newNode priority > front node priority, insert on front-2-1
    else if (P>pqfront->getP())
    {
    newNode->setNext(pqfront);
    pqfront=newNode;
    }

//enqueue(b,7)
```



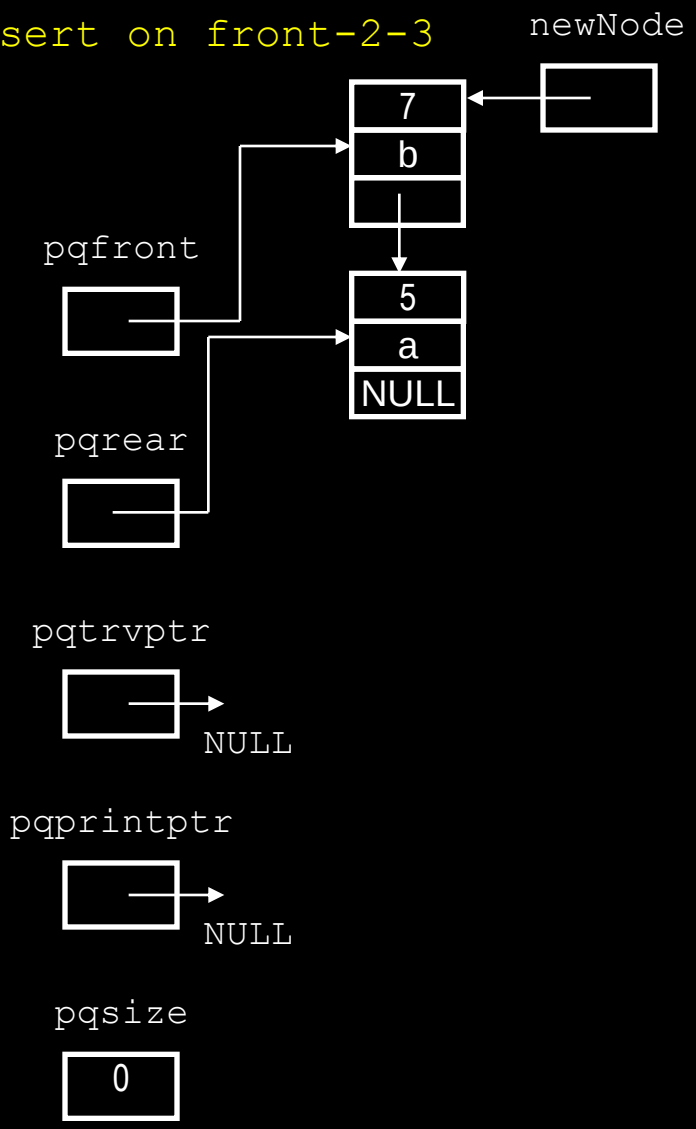
Priority Queue Implementation: Linked List

```
//newNode priority > front node priority, insert on front-2-2
    else if (P>pqfront->getP())
    {
    newNode->setNext(pqfront);
    pqfront=newNode;
    }

//enqueue(b,7)
```



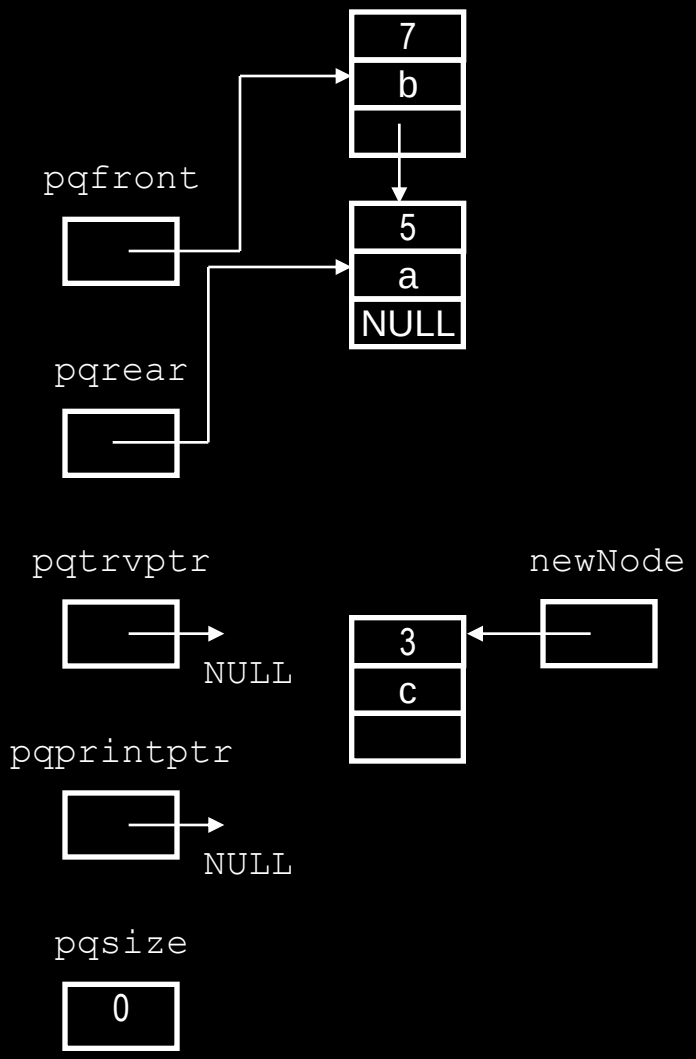
Priority Queue Implementation: Linked List

```
//newNode priority > front node priority, insert on front-2-3
    else if (P>pqfront->getP())
    {
    newNode->setNext(pqfront);
    pqfront=newNode;
    }

//enqueue(b,7)
```



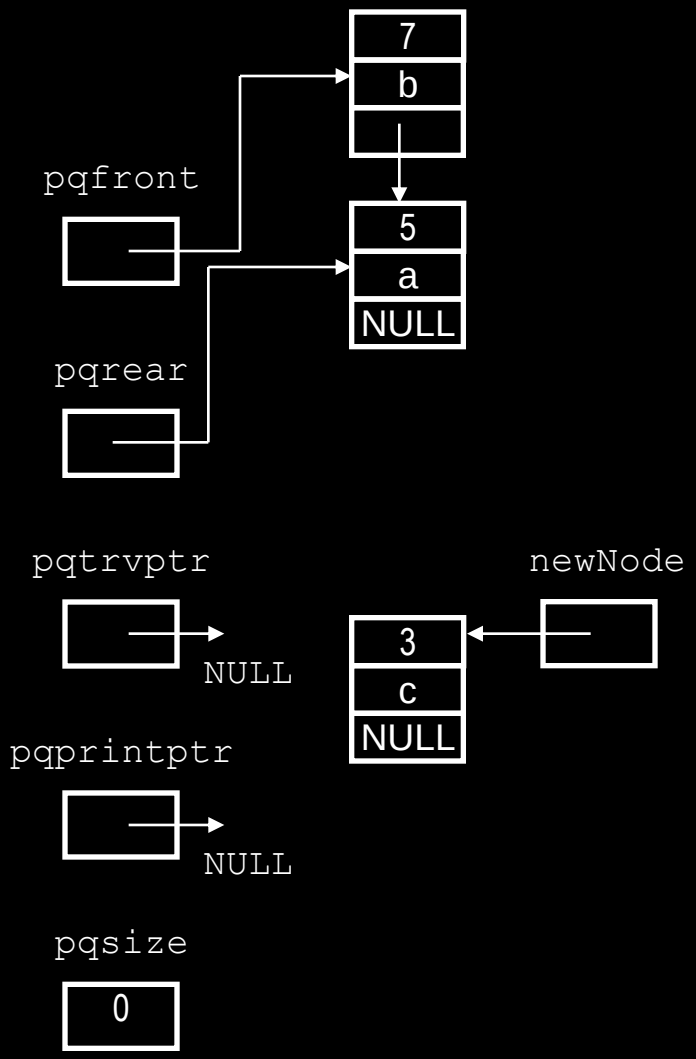
Priority Queue Implementation: Linked List

```
//newNode priority <= rear node priority, insert on rear-3-1
    else if (P<=pqrear->getP())
    {
    newNode->setNext(NULL);
    pqrear->setNext(newNode);
    pqrear=newNode;
    }

//enqueue(c,3)
```



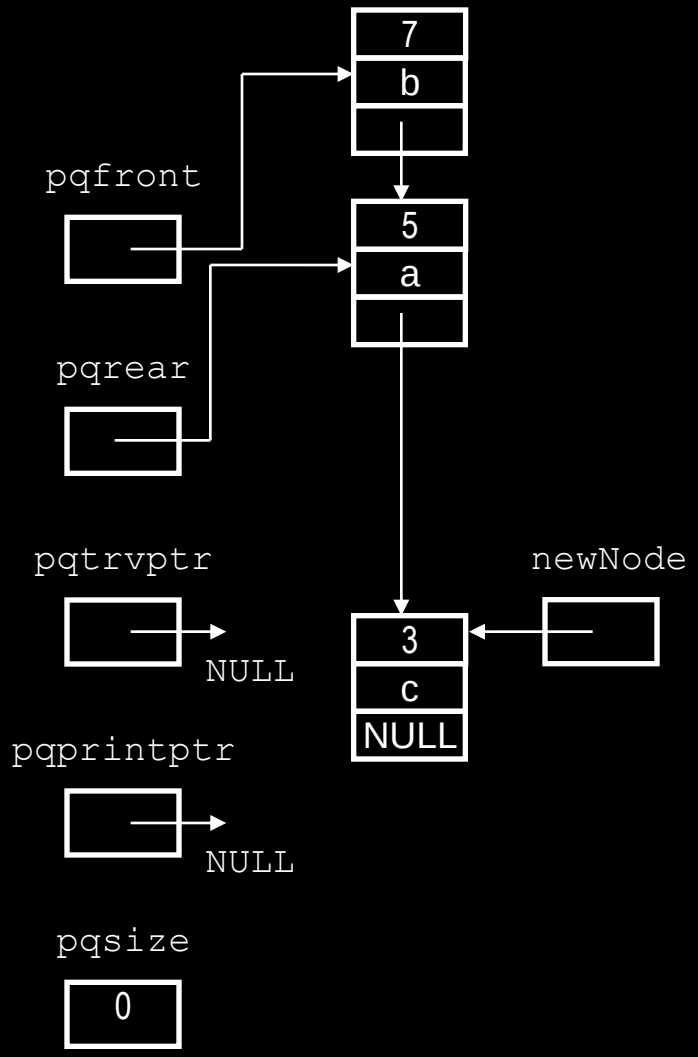
Priority Queue Implementation: Linked List

```
//newNode priority <= rear node priority, insert on rear-3-2
    else if (P<=pqrear->getP())
    {
    newNode->setNext(NULL);
    pqrear->setNext(newNode);
    pqrear=newNode;
    }

//enqueue(c,3)
```



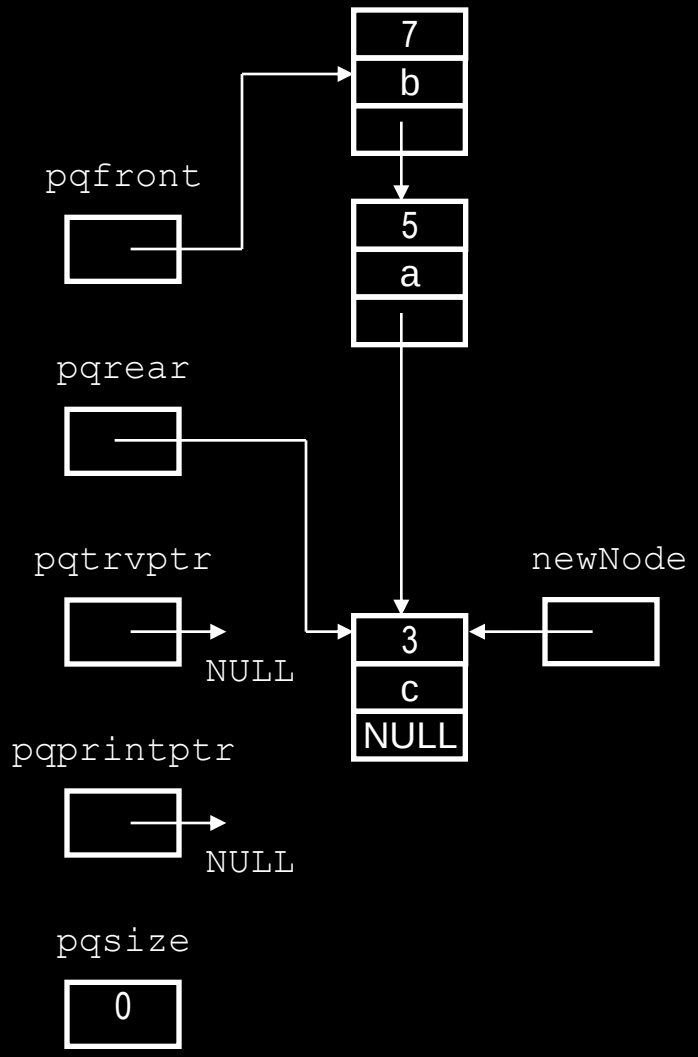
Priority Queue Implementation: Linked List

```
//newNode priority <= rear node priority, insert on rear-3-3
    else if (P<=pqrear->getP())
    {
    newNode->setNext(NULL);
    pqrear->setNext(newNode);
    pqrear=newNode;
    }

//enqueue(c,3)
```



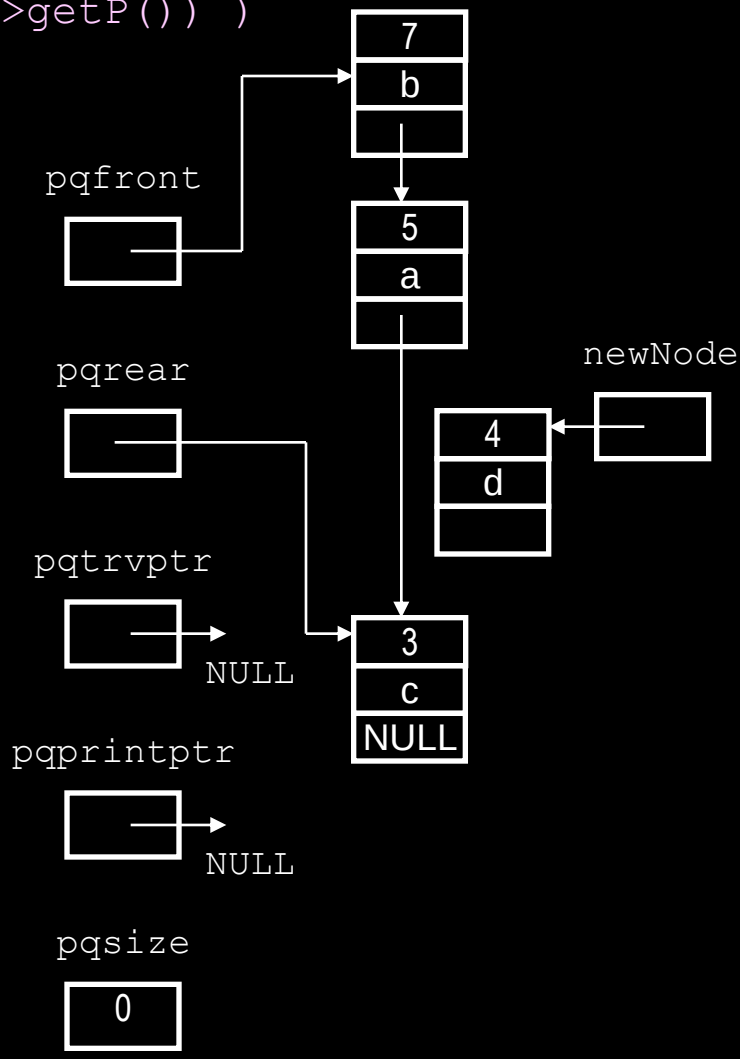
Priority Queue Implementation: Linked List

```
//newNode priority <= rear node priority, insert on rear-3-4
    else if (P<=pqrear->getP())
    {
    newNode->setNext(NULL);
    pqrear->setNext(newNode);
    pqrear=newNode;
    }

//enqueue(c,3)
```



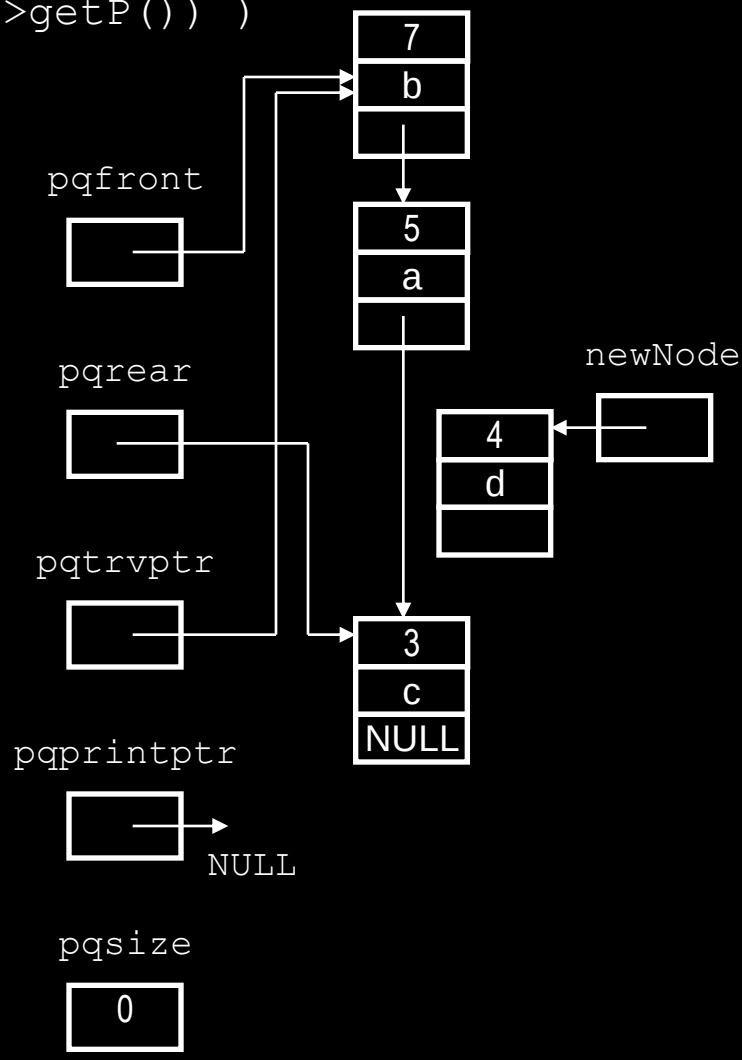
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-1.1
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



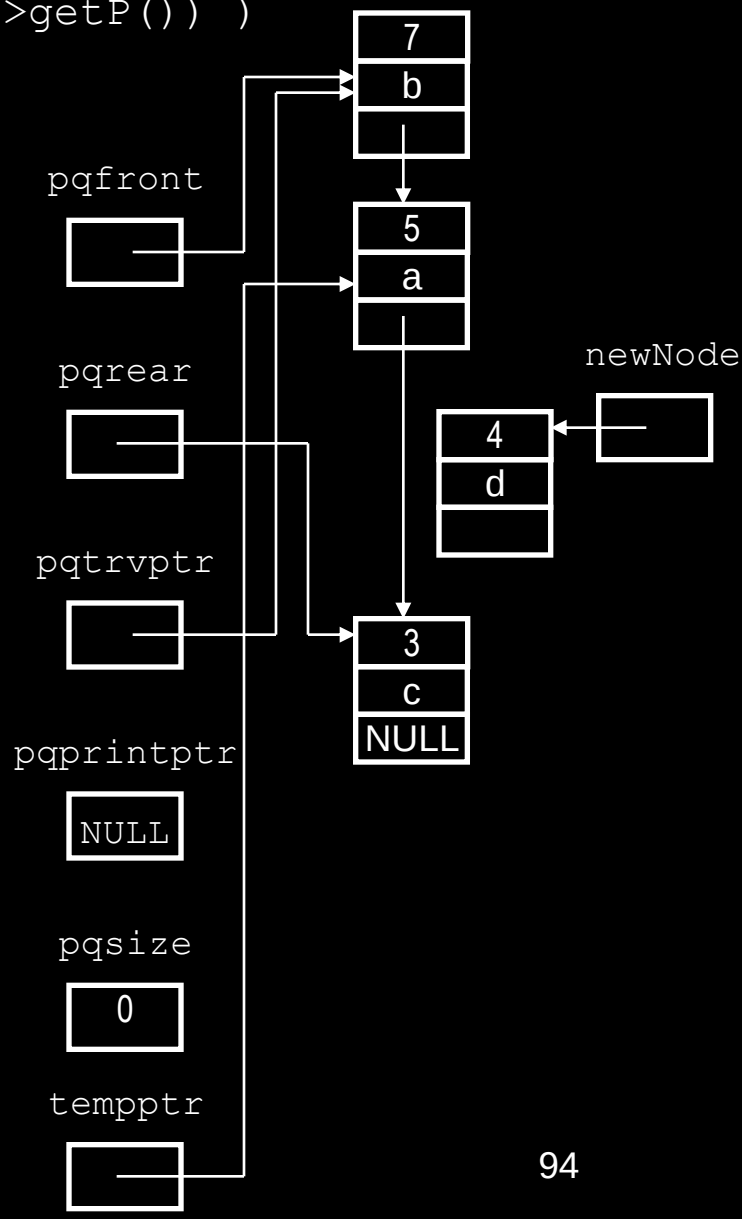
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-1.2
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



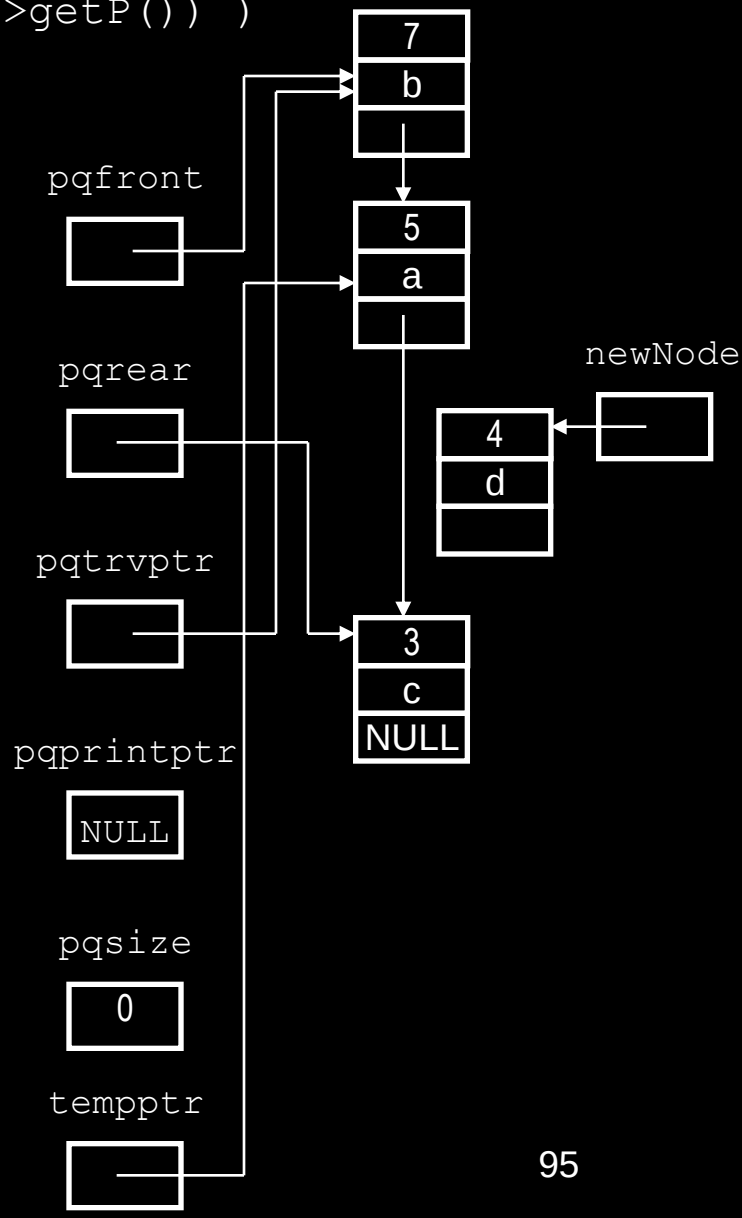
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-1.3
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



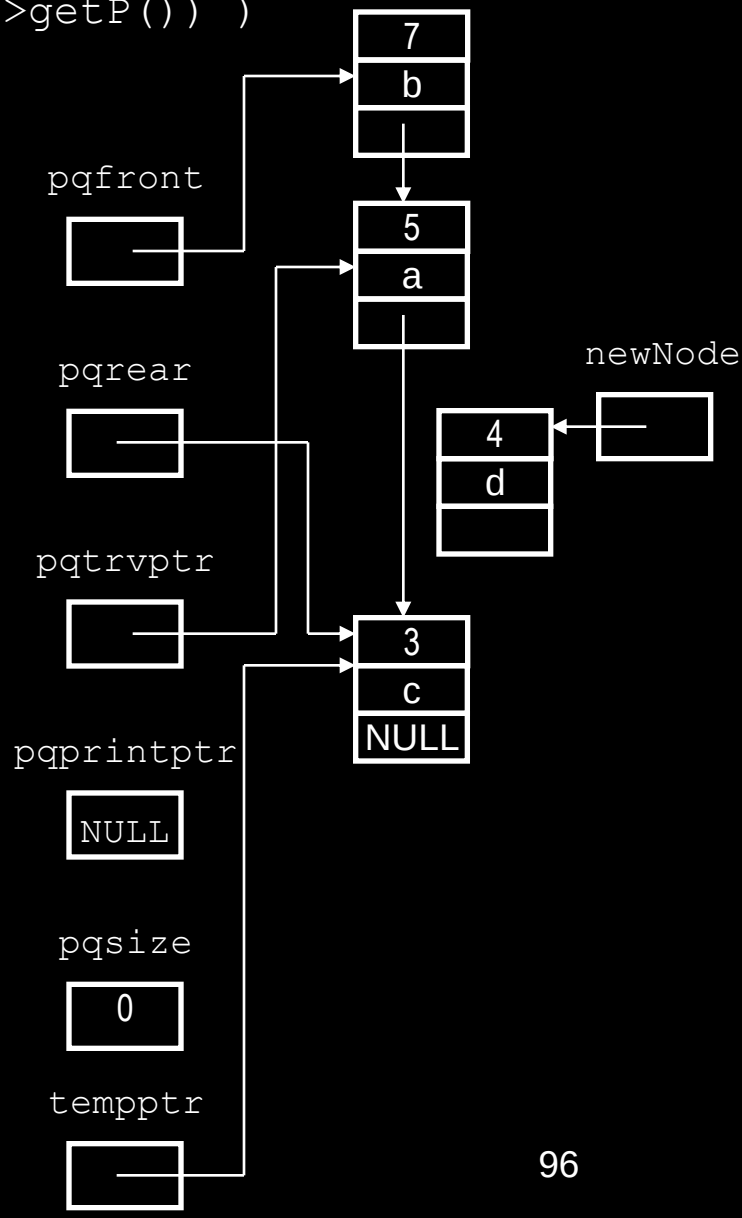
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-1.4
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



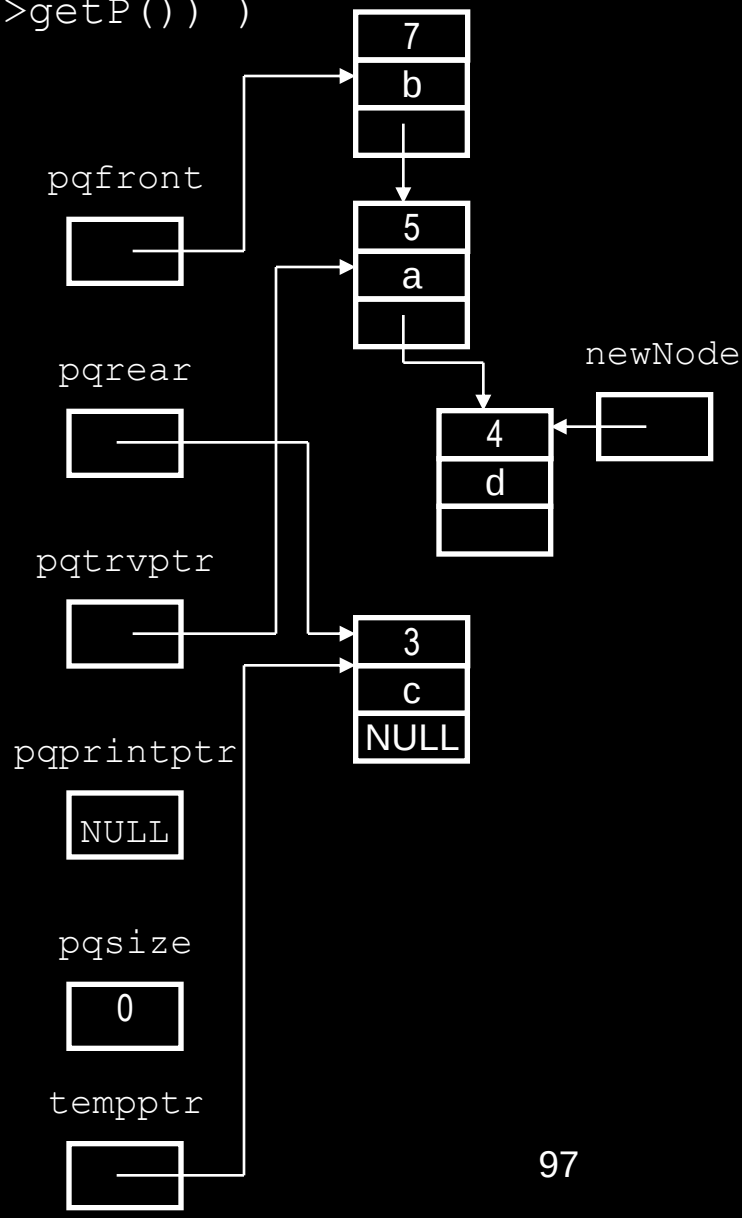
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-1.5
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



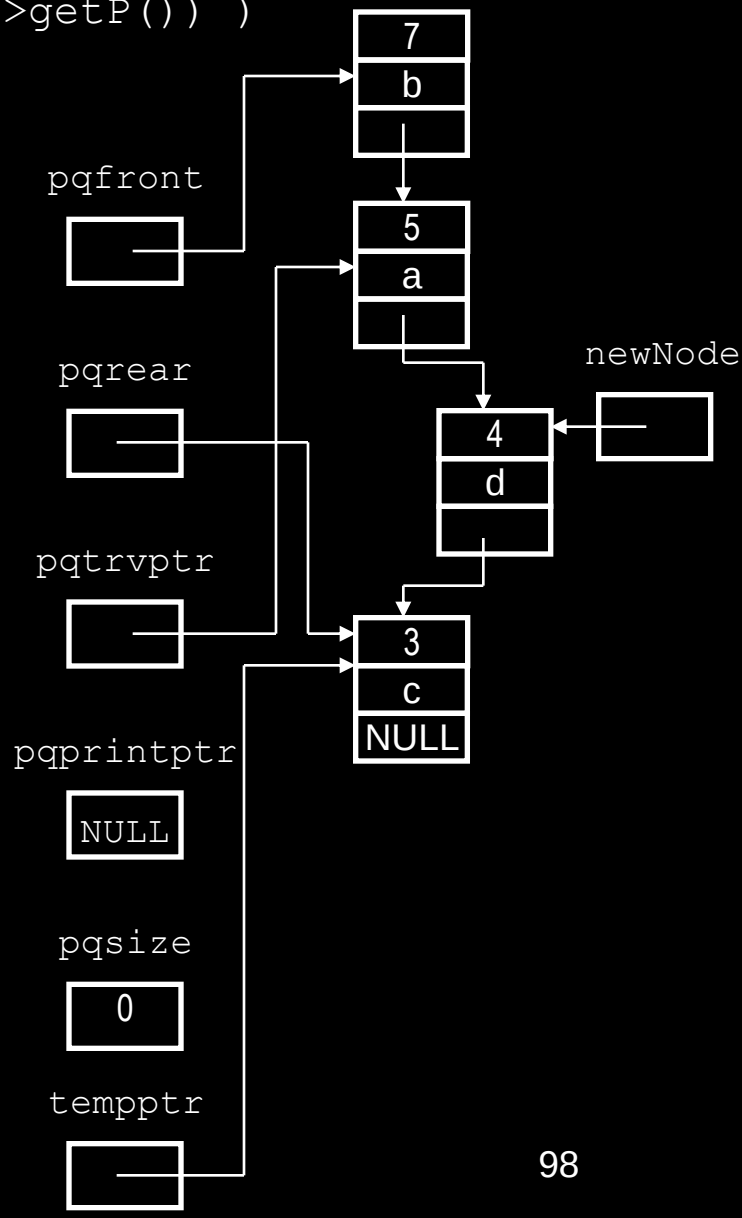
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-1.6
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



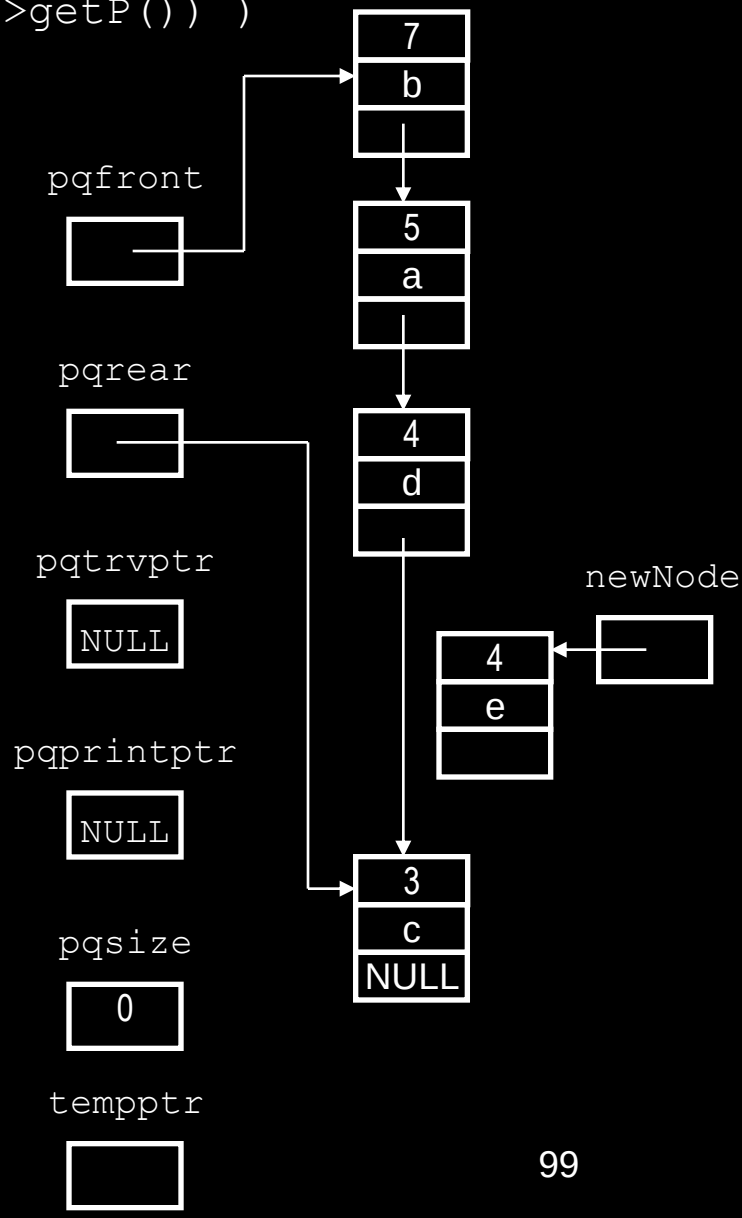
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-1.7
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



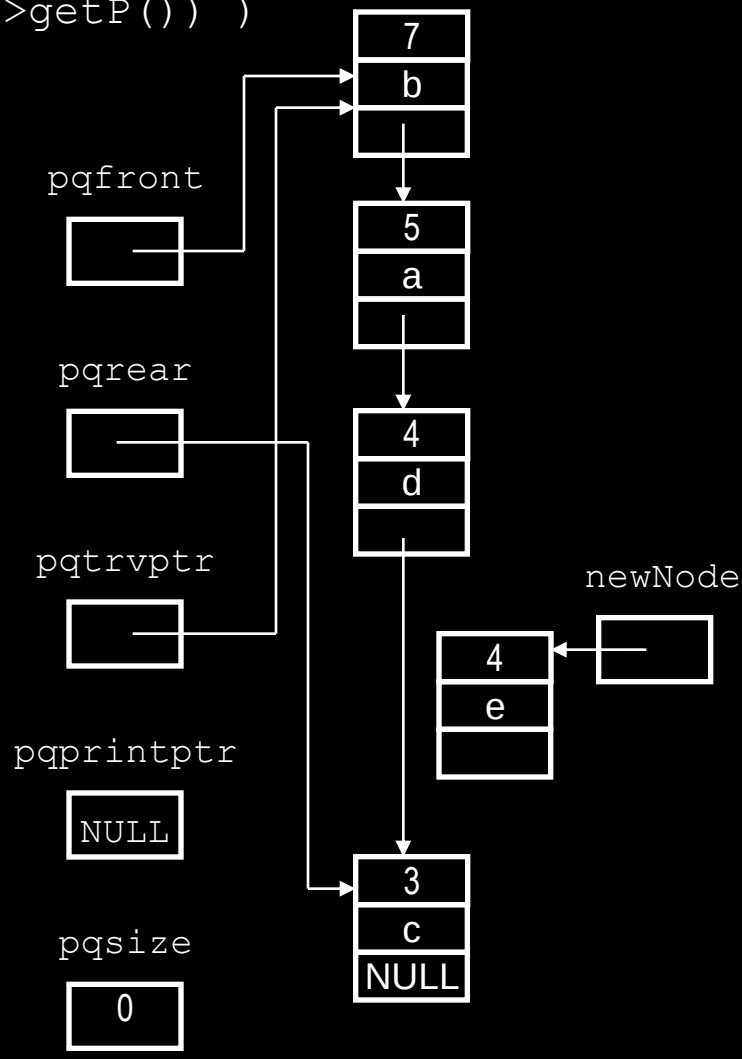
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-2.1
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



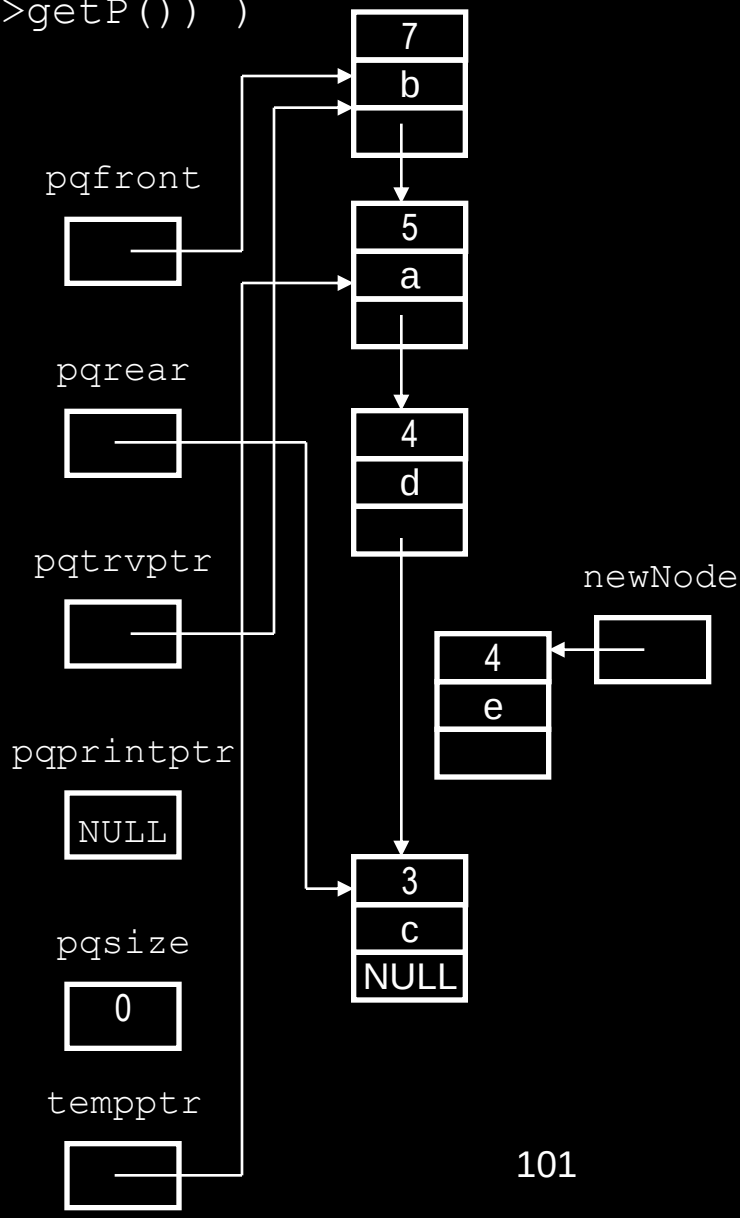
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-2.2
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



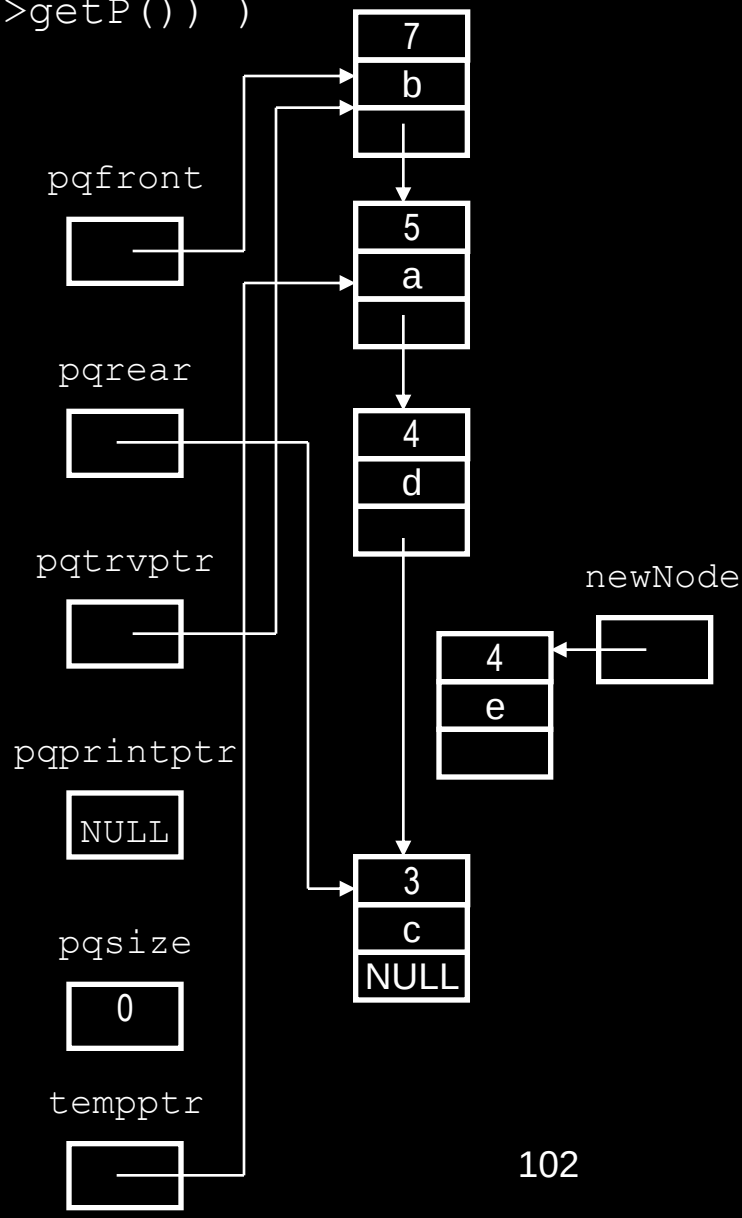
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-2.3
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



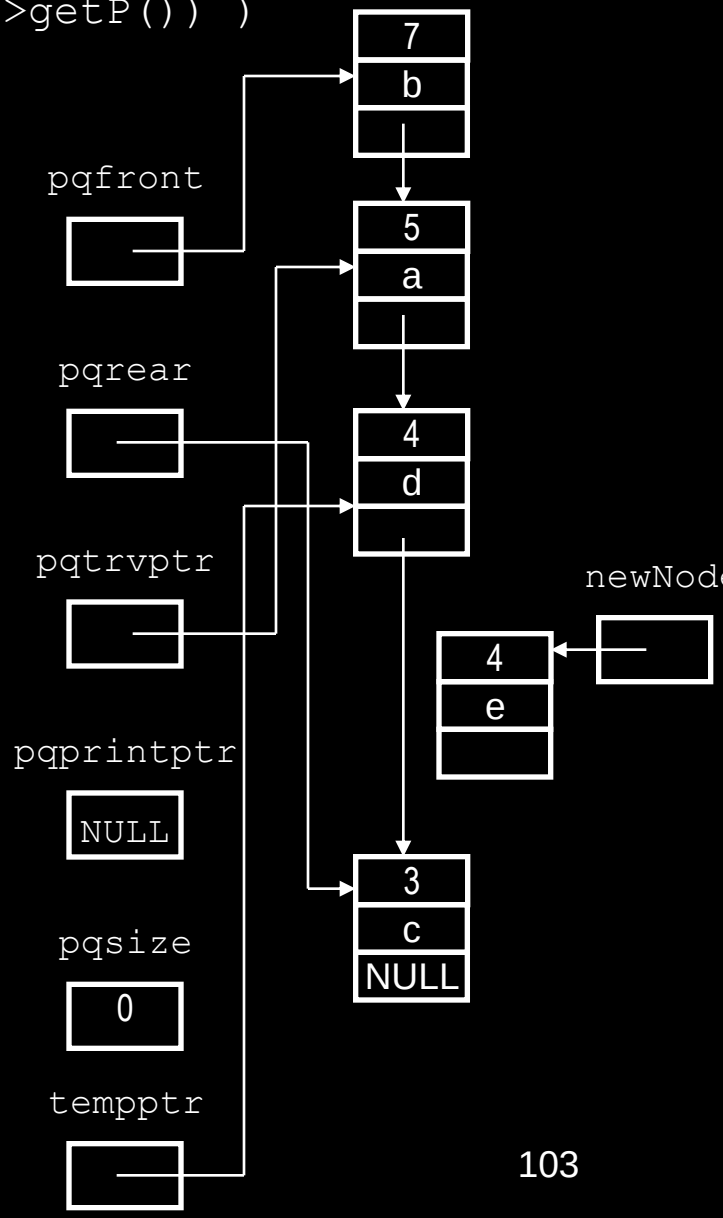
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-2.4
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



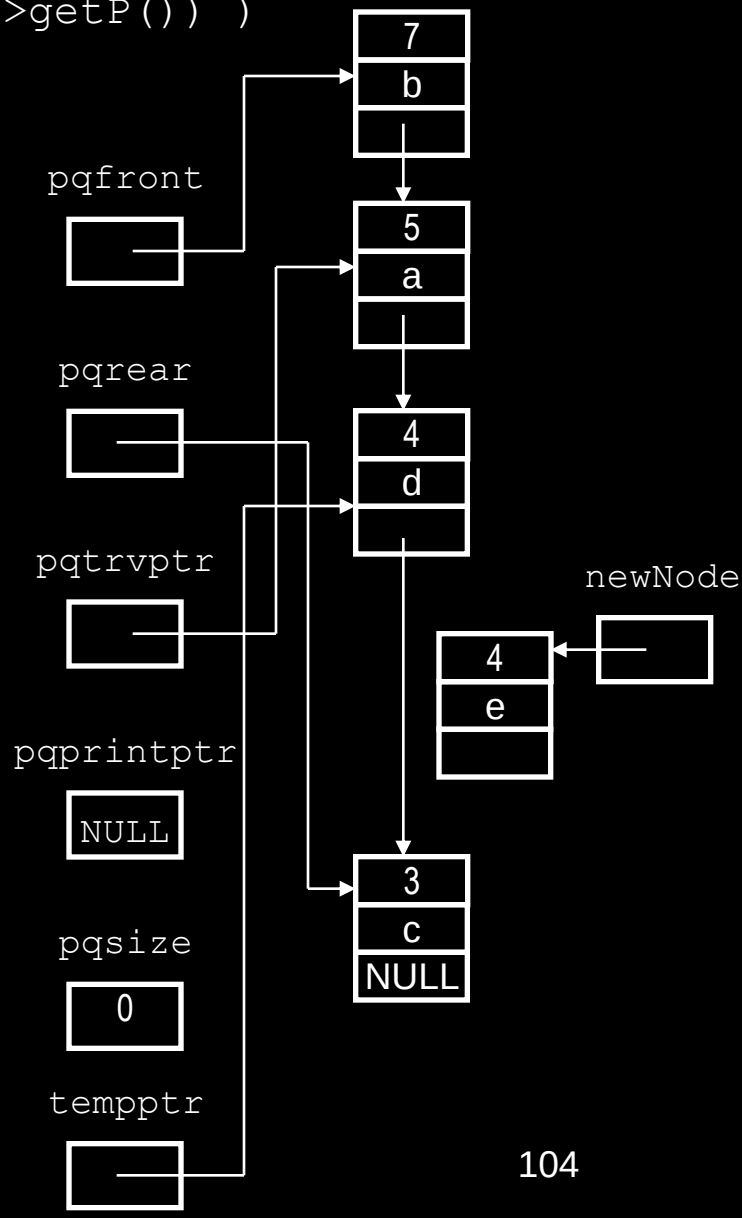
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-2.5
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



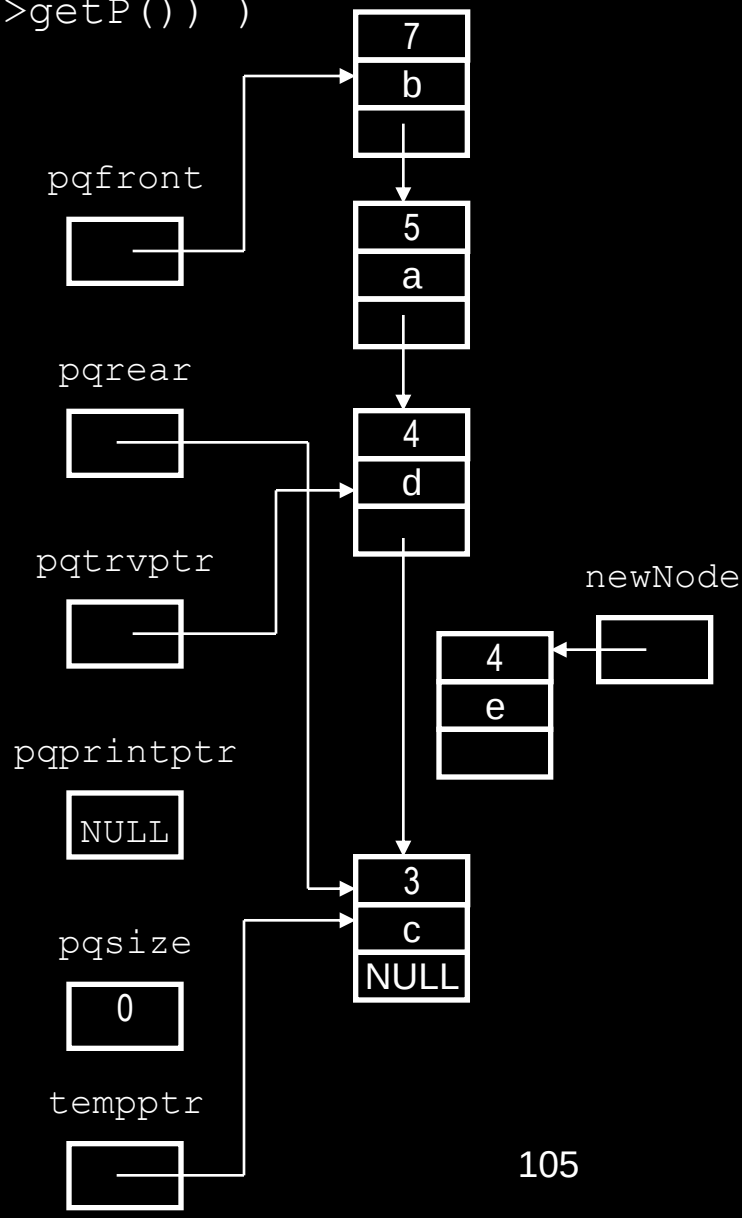
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-2.6
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



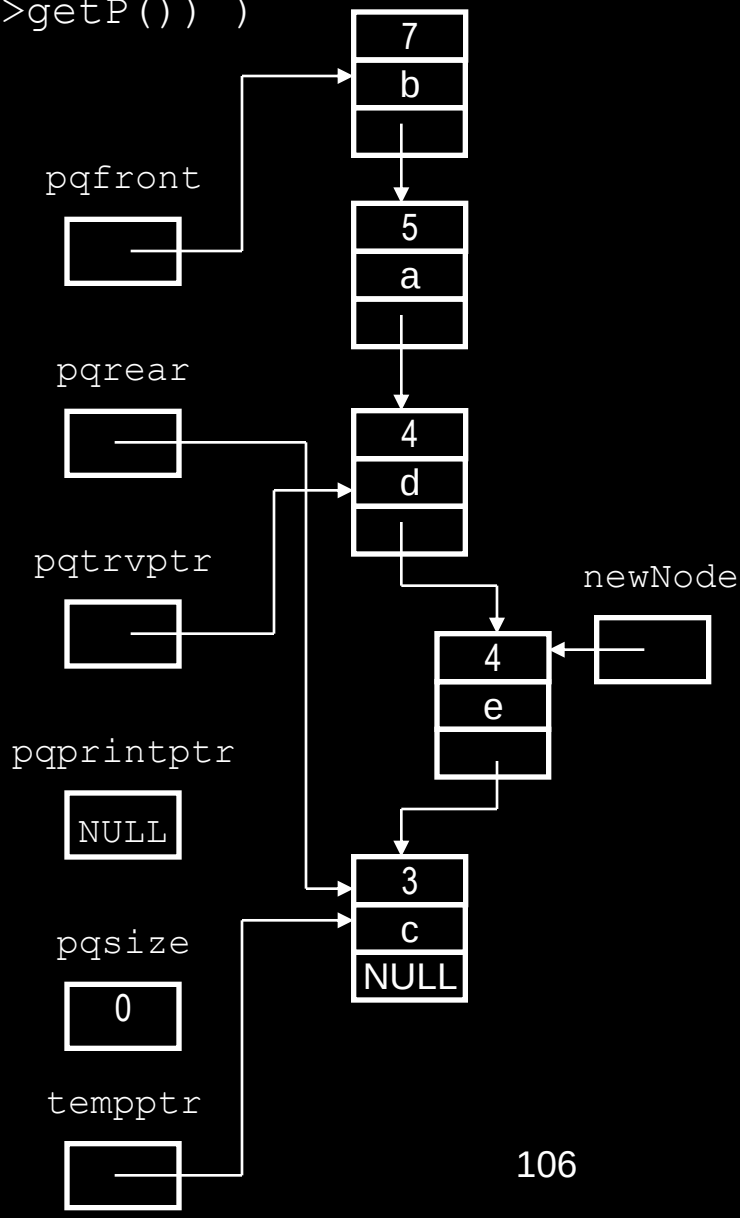
Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-2.7
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



Priority Queue Implementation: Linked List

```
//newNode priority <= front node priority AND >rear node priority-4-2.8
else if ( (P<=pqfront->getP()) && (P>pqrear->getP()) )
{
pqtrvptr=pqfront;
Node *tempptr=pqtrvptr->getNext();
while (P<=tempptr->getP())
  {
  pqtrvptr=pqtrvptr->getNext();
  tempptr=pqtrvptr->getNext();
  }
pqtrvptr->setNext(newNode);
newNode->setNext(tempptr);
}
```



Queue Implementation: Linked List

```
int dequeue()
{
    int x = pqfront->get();
    Node* p = pqfront;
    pqfront = pqfront->getNext();
    delete p;
    pqsize--;
    return x;
}
```

Queue Implementation: Linked List

```
int atfront()   {  return pqfront->get(); }

int IsEmpty() {  return ( pqfront == NULL ); }

int size()    {  return pqsize; }

void print()
{
  pqprintptr=pqfront;
  while (pqprintptr!=NULL)
    {
    cout << pqprintptr->get() << "," << pqprintptr->getP() << " ";
    pqprintptr=pqprintptr->getNext();
    }
}
};
```