Chapter 6

Queues

Topics

- FIFO (first0in-first0out) structure
- Implementations: formula-based and linked classes
- Applications:
  - railroad-switching problem
  - **shortest path for a wire that is to connect two points**
  - **pixels labeling**
  - machine shop simulation

Queues

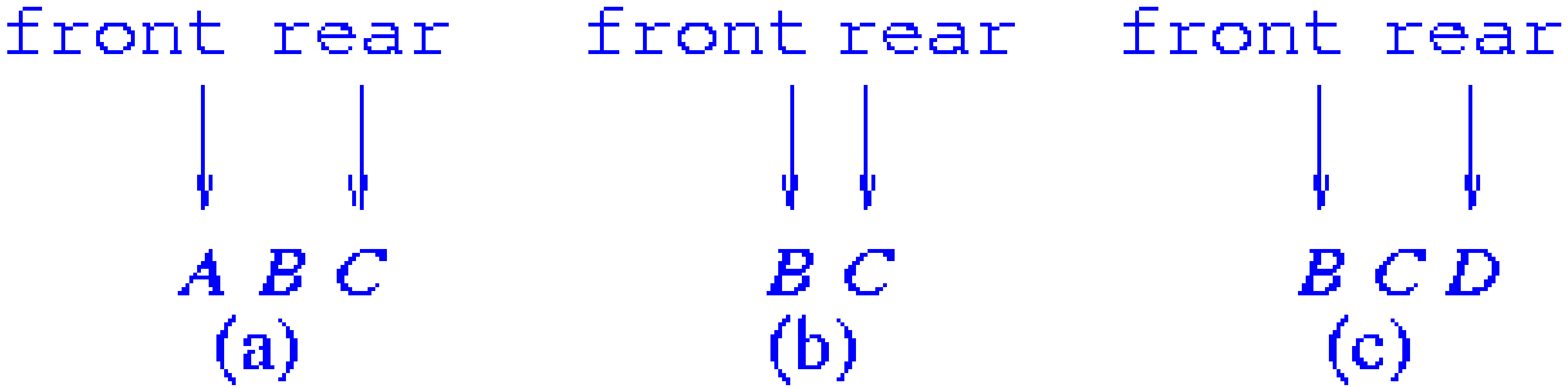


Figure 6.1 Sample queues

Abstract data type

AbstractDataType *Queue* {
instances
ordered list of elements; one end is called the `front`; the other is the `rear`;
operations:
Create (): Create an empty queue;
IsEmpty (): Return `true` if queue is empty, return `false` otherwise;
IsFull (): Return `true` if queue is full, return `false` otherwise;
First (): Return first element of queue;
Last (): Return last element of queue;
Add (x): Add element x to the queue;
Delete (x): Delete front element from queue and put it in x;
}

ADT 6.1 The abstract data type queue

Queues 1

- *location(i) = i - 1*
- add - *O(1)*
- delete - *Θ(n)* *(need to slide items to front)*

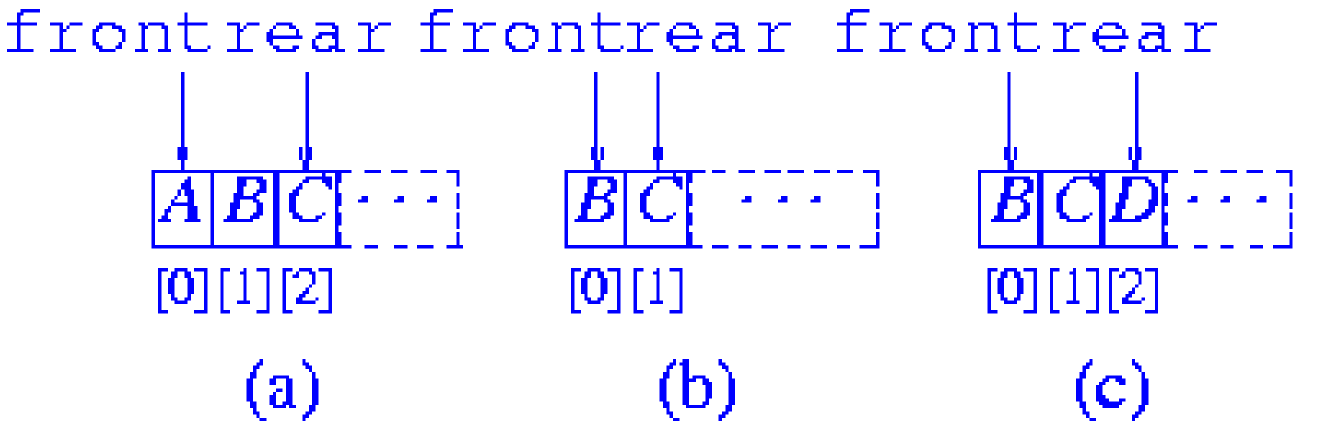


Figure 6.2 Queues of Figure 6.1 using formula (6.1)

Queues 2

- $location(i) = location(1) + i - 1$
- add - worst-case $\Theta(n)$ *(when buffer is full)*
- delete - $O(1)$

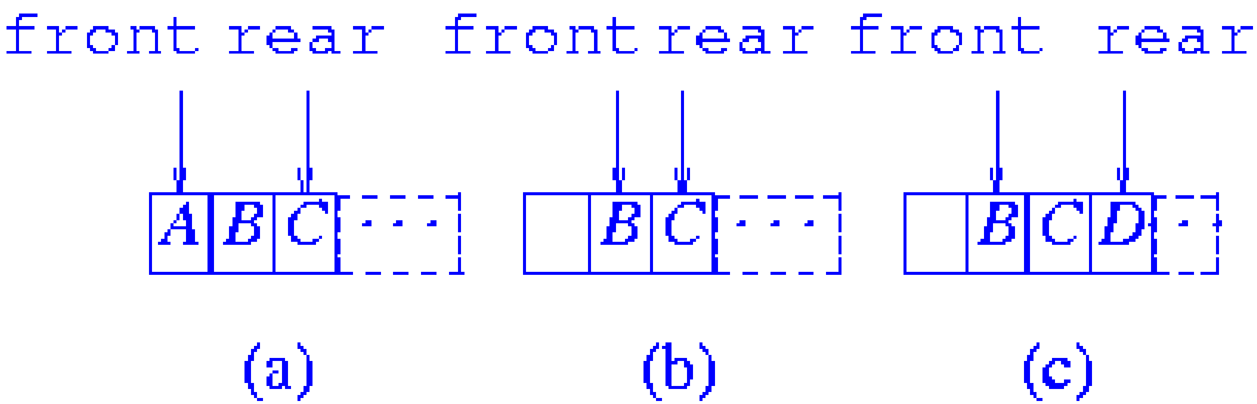


Figure 6.3 Queues of figure 6.1 using formula 6.2

Shifting
when *rear=MaxSize-1* and *front > 0*

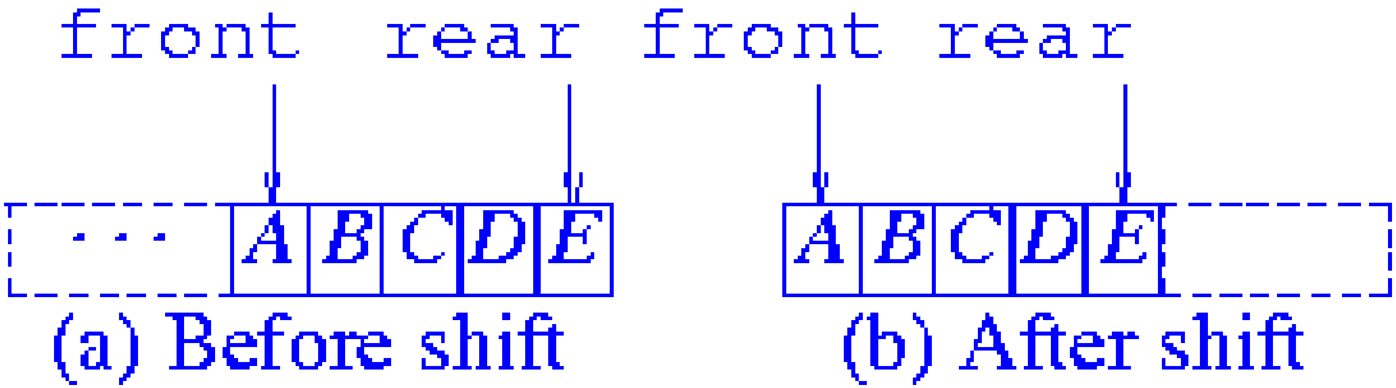


Figure 6.4 Shifting a queue

Queues 3

- *location(i) = (location(1) + i - 1) % MaxSize*
- add - *Θ(1)*
- delete - *Θ(1)*

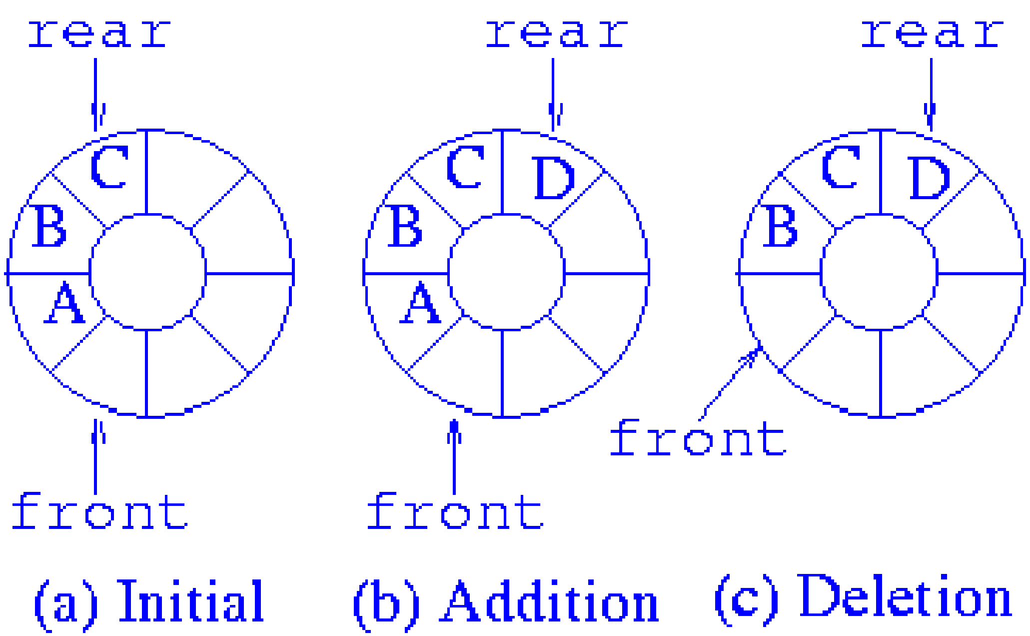


Figure 6.5 Circular queues

Empty and full queue

- Empty queue: *front=rear*
- full queue: can only hold up to *MaxSize-1* elements

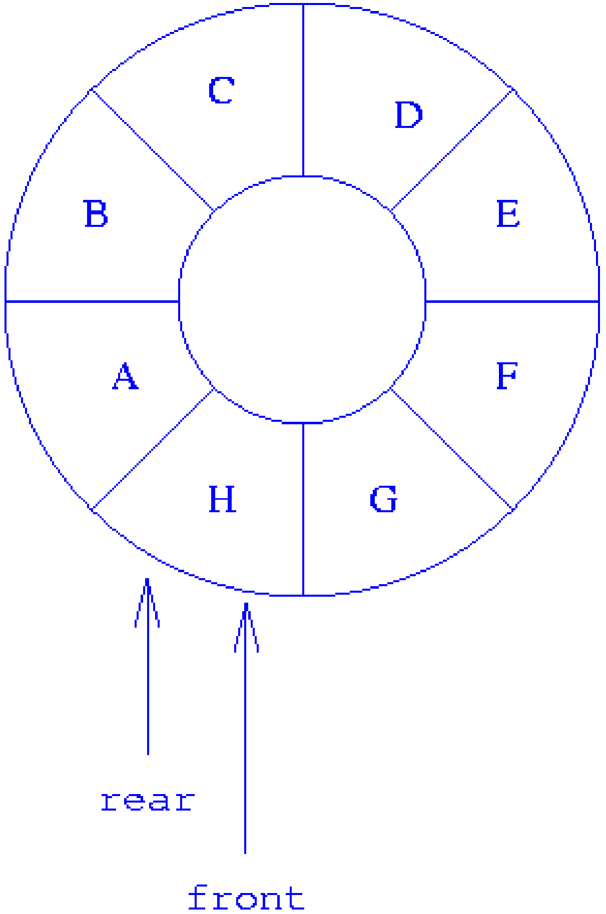


Figure 6.6 A circular queue with `MaxSize` elements

Formula-based class Queue

```
template <class T>
class Queue {
// FIFO objects
   public :
     Queue(int MaxQueueSize = 10);
     ~LinkedQueue() {delete [] queue;}
     bool IsEmpty() const {return front == rear;}
     bool IsFull() const
       {return(((rear+1) % MaxSize == front)?1:0);}
     T First() const ; // return front element
     T Last() const ; // return last element
     Queue<T>& Add(const T& x);
     Queue<T>& Delete(T& x);
   private :
     int front; // one counterclockwise from first
     int rear; // last element
     int MaxSize; // size of array queue
     T *queue; // element array
};
```

Program 6.1 Formula-based class Queue

Constructor - $\Theta(1)$ when T is internal; $O(MaxStackSize)$ when T is user-defined

First - $\Theta(1)$

```
template <class T>
Queue<T>::Queue(int MaxQueueSize)
{// Create an empty queue whose capacity
// is MaxQueueSize.
   MaxSize = MaxQueueSize + 1;
   queue = new T[MaxSize];
   front = rear = 0;
}

template <class T>
T Queue<T>::First() const
{// Return first element of queue.  Throw
// OutOfBounds exception if the queue is empty.
   if (IsEmpty()) throw OutOfBounds();
   return queue[(front + 1) % MaxSize];
}
```

Last - $\Theta(1)$

```
template <class T>
T Queue<T>::Last() const
{// Return last element of queue.  Throw
// OutOfBounds exception if the queue is empty.
   if (IsEmpty()) throw OutOfBounds();
   return queue[rear];
}
```

Program 6.2 Queue functions using formula-based representation (continues)

Add and Delete - $\Theta(1)$

```
template <class T>
Queue<T>& Queue<T>::Add(const T& x)
{// Add x to the rear of the queue.  Throw
// NoMem exception blue if the queue is full.
   if (IsFull()) throw NoMem();
   rear = (rear + 1) % MaxSize;
   queue[rear] = x;
   return *this ;
}

template <class T>
Queue<T>& Queue<T>::Delete(T& x)
{// Delete first element and put it in x.  Throw
// OutOfBounds exception if the queue is empty.
   if (IsEmpty()) throw OutOfBounds();
   front = (front + 1) % MaxSize;
   x = queue[front];
   return *this ;
}
```

Program 6.3 Queue functions using formula-based representation (concluded)

Linked presentation

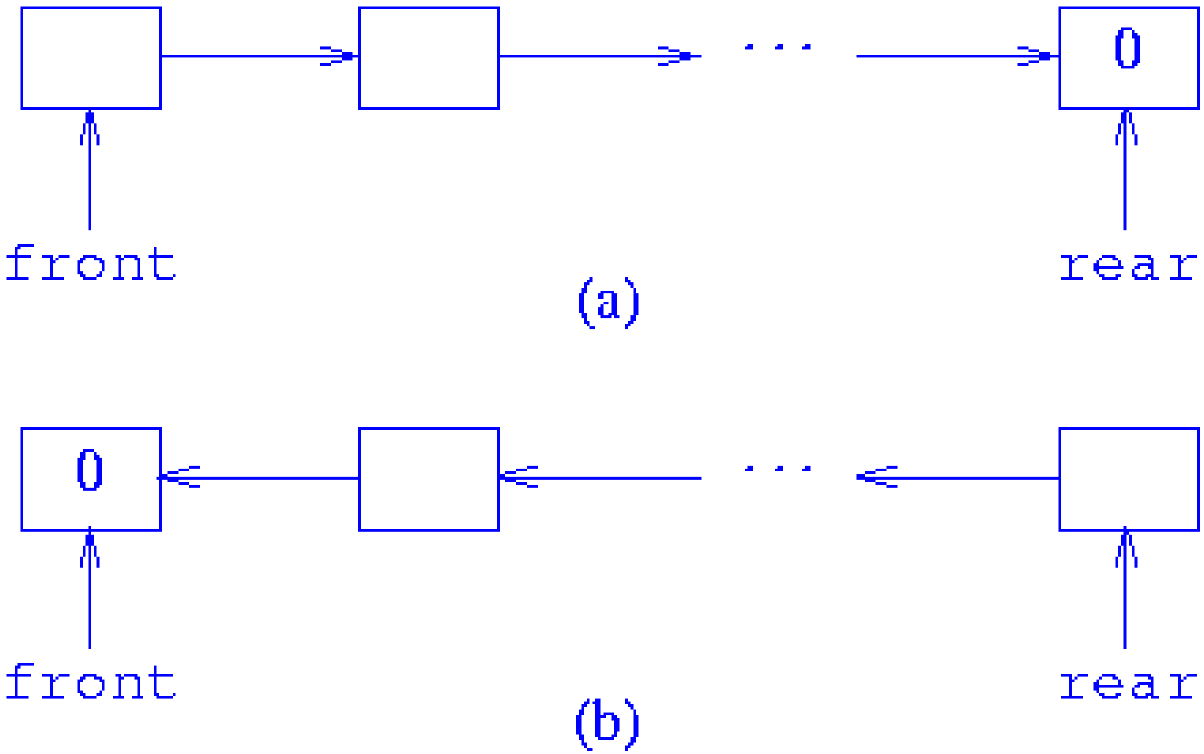
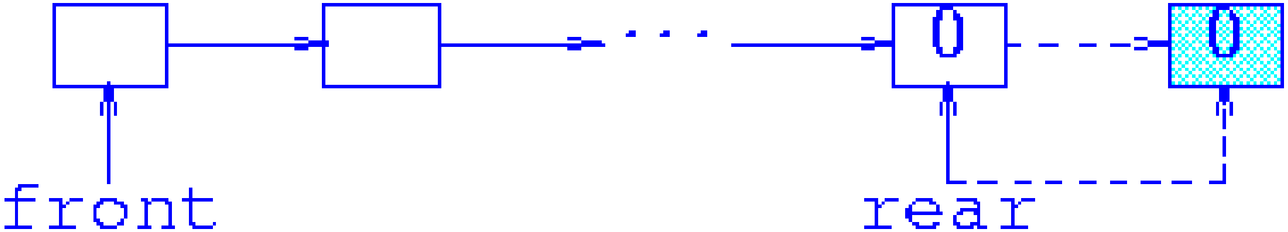
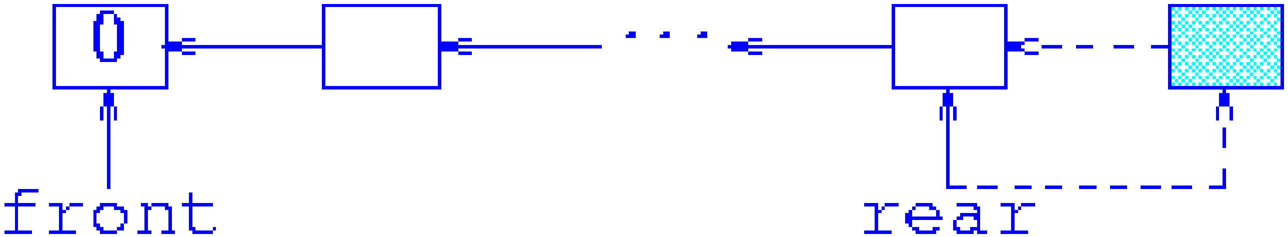


Figure 6.7 Linked queues

Addition



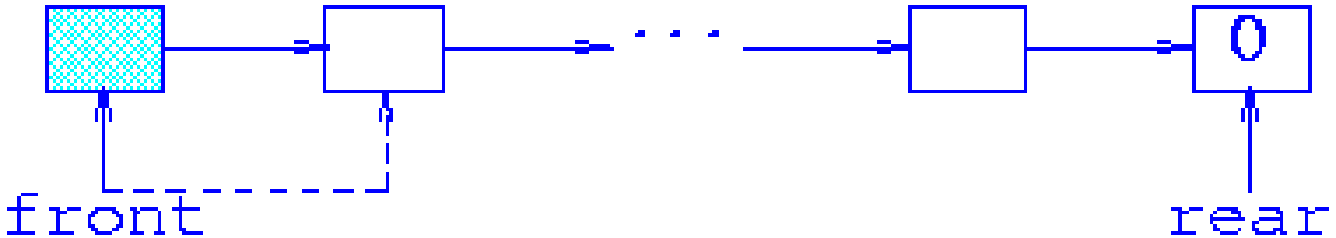
(a) Addition to figure6.7 (a)



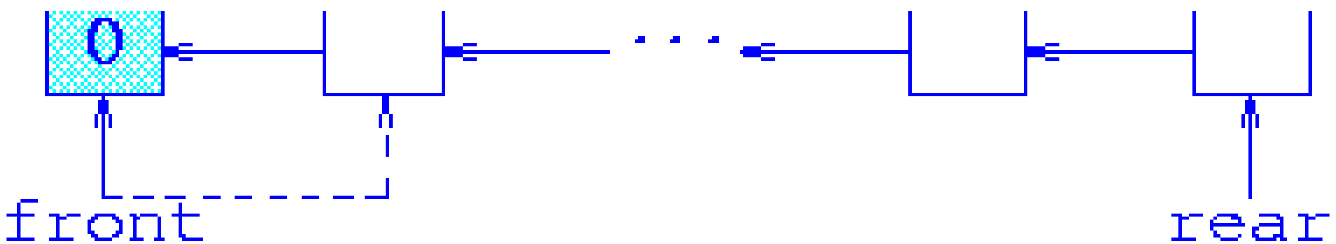
(b) Addition to figure 6.7(b)

Figure 6.8 Addition to a linked queue

Deletion



(a) Deletion from figure 6.7(a)



(b) Deletion from figure6.7 (b)

Figure 6.9 Deletion from a linked queue

Class definition

```
template <class T>
class LinkedQueue {
// FIFO objects
  public :
    //constructor
    LinkedQueue() {front = rear = 0;}
    ~LinkedQueue(); // destructor
    bool IsEmpty() const
       {return ((front) ?  false :  true);}
    bool IsFull() const ;
    T First() const ; // return first element
    T Last() const ; // return last element
    LinkedQueue<T>& Add(const T& x);
    LinkedQueue<T>& Delete(T& x);
  private :
    Node<T> *front; // pointer to first node
    Node<T> *rear; // pointer to last node
};
```

Program 6.4 Class definition for a linked queue

Destructor - $\Theta(n)$

```
template <class T>
LinkedQueue<T>::~LinkedQueue()
{// Queue destructor.  Delete all nodes.
   Node<T> *next;
   while (front) {
     next = front->link;
     delete front;
     front = next;
  }
}
```

IsFull - $\Theta(1)$

```
template <class T>
bool LinkedQueue<T>::IsFull() const
{// Is the queue full?
   Node<T> *p;
   try {p = new Node<T>;
      delete p;
      return false;}
   catch (NoMem) {return true;}
}
```

First and Last - $\Theta(1)$

```
template <class T>
T LinkedQueue<T>::First() const
{// Return first element of queue.  Throw
// OutOfBounds exception if the queue is empty.
   if (IsEmpty()) throw OutOfBounds();
   return front->data;
}

template <class T>
T LinkedQueue<T>::Last() const
{// Return last element of queue.  Throw
// OutOfBounds exception if the queue is empty.
   if (IsEmpty()) throw OutOfBounds();
   return rear->data;
}
```

Program 6.5 Linked queue function implementations (continues)

Add - $\Theta(1)$

```
template <class T>
LinkedQueue<T>& LinkedQueue<T>::Add(const T& x)
{// Add x to rear of queue.  Do not catch
  // possible NoMem exception thrown by new .

   // create node for new element
   Node<T> *p = new Node<T>;
   p->data = x;
   p->link = 0;

   // add new node to rear of queue
   if (front) rear->link = p; // queue not empty
   else front = p; // queue empty
   rear = p;

   return *this ;
}
```

Delete - $\Theta(1)$

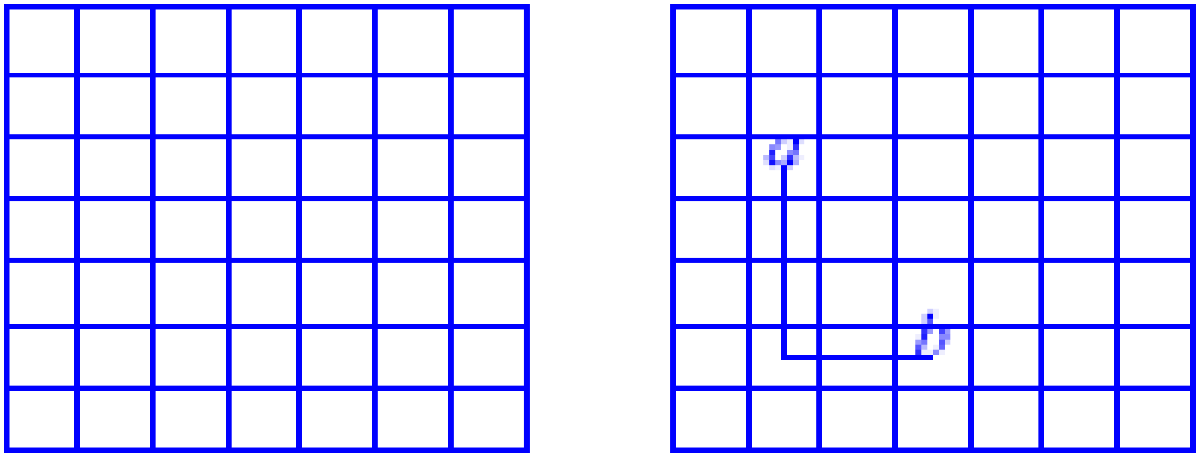
```
template <class T>
LinkedQueue<T>& LinkedQueue<T>::Delete(T& x)
{// Delete first element and put it in x.  Throw
// OutOfBounds exception if the queue is empty.

   if (IsEmpty()) throw OutOfBounds();

   // save element in first node
   x = front->data;

   // delete first node
   Node<T> *p = front;
   front = front->link;
   delete p;

   return *this ;
}
```

Program 6.6 Linked queue function implementations (concluded)

Wire-Routing Problem

- Need to route wires from A to B using the shortest-path.
- Routing region can be represented with a grid: an nxm matrix of squares
- wire runs midpoint of one square to midpoint of another making only right-angles.
- Grid squares that already have wires in them are blocked.

Wire-Routing Problem



(a) A 7 × 7 grid (b) A wire between *a* and *b*

Figure 6.11 Wire-routing example

Wire-Routing Problem

- Begin at source a
- label its reachable neighbors with 1
- next the reachable neighbors of distance 1 squares are labeled 2.
- Continue labeling processing until either reach destination *b* or have no more reachable neighbors.
- If *b* is reached, label it with its distance.

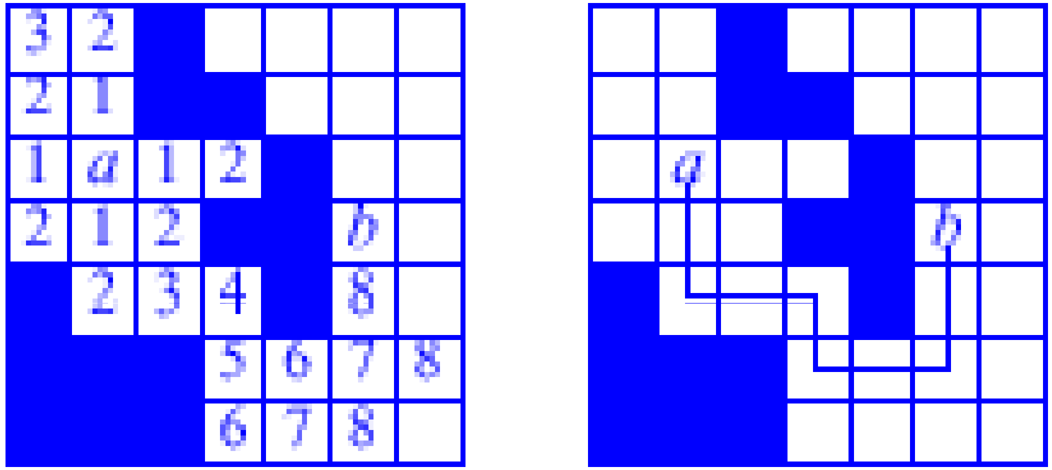
Wire-Routing Problem

To construct shortest path,

- Start from *b* and move to anyone of its neighbors labeled 1 less than *b*'s label.
- From the neighbor found in previous step, move to one of its neighbors whose label is 1 less
- repeat until square *a* is reached

The sequence of neighbors collected forms the shortest path.

Wire-Routing Problem



(a) Distance labeling (b) Wire path

Figure 6.12 Wire routing

```
bool FindPath(Position start, Position finish,
        int & PathLen, Position * &path)
{// Find a path from start to finish.
 // Return true if successful, false if impossibl
 // Throw NoMem exception if inadequate space.

  if ((start.row == finish.row) &&
                  (start.col == finish.col))
   {PathLen = 0;return true;}//start=finish

  // initialize wall of blocks around grid
  for (int i = 0; i <= m+1; i++) {
    // bottom & top
    grid[0][i] = grid[m+1][i] = 1;
   // left & right
    grid[i][0] = grid[i][m+1] = 1;      }
```

```
// initialize offsets
Position offset[4];
offset[0].row = 0; offset[0].col = 1; // right
offset[1].row = 1; offset[1].col = 0; //down
offset[2].row = 0; offset[2].col = -1; //left
offset[3].row = -1; offset[3].col = 0; //up

int NumOfNbrs = 4;//neighbors of a grid position
Position here, nbr;
here.row = start.row;
here.col = start.col;
grid[start.row][start.col] = 2; // block
```

Program 6.9 Find a wire route

Railroad Car Rearrangement

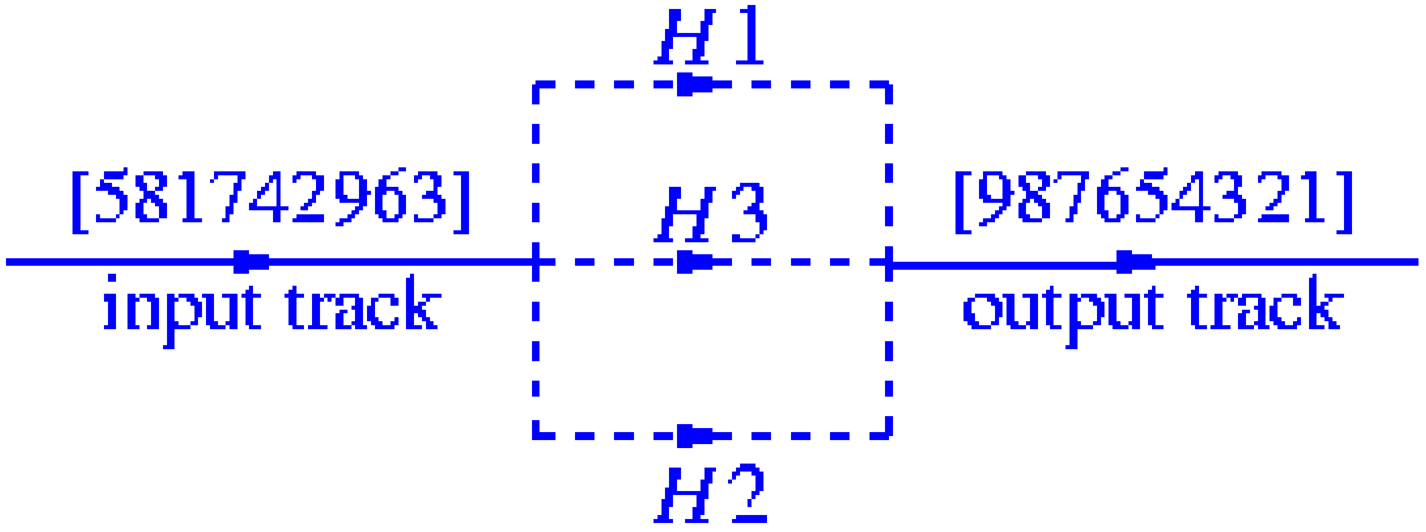


Figure 6.10 A three-track example

Rules

- Reserve *Hk* for moving cars directly from the input track to the output track
- Move car *c* to a holding track that contains only cars with a smaller label; if there are several such tracks, select one with largest label at its left end; otherwise, select an empty track (if one remains)

Output

```
void Output(int & minH, int & minQ,
      LinkedQueue<int > H[], int k, int n)
{// Move from hold to output and update minH
                                    and minQ.
   int c; // car index
   //delete smallest car minH from queue minQ
   H[minQ].Delete(c);
   cout << "Move car" << minH
        << "from holding track"
        << minQ << " to output" << endl;
   // find new minH and minQ
   // by checking front of all queues
   minH = n + 2;
   for (int i = 1; i <= k; i++)
     if (!H[i].IsEmpty() &&
        (c = H[i].First()) < minH) {
        minH = c;
        minQ = i;}
}
```

Program 6.7 Rearranging cars using queues (continues)

Hold

```
bool Hold(int c, int & minH, int &minQ,
LinkedQueue<int > H[], int k)
{// Add car c to a holding track.
// Return false if no feasible holding track.
// Throw NoMem exception if no queue space.
// Return true otherwise.

   // find best holding track for car c
   // initialize
   int BestTrack = 0, // best track so far
      BestLast = 0, // last car in BestTrack
      x; // a car index
```

Hold (continue)

```
// scan holding tracks
for (int i = 1; i <= k; i++)
  if (!H[i].IsEmpty()) {// track i not empty
    x = H[i].Last();
    if (c > x && x > BestLast) {
     // track i has bigger car at end
     BestLast = x;
     BestTrack = i;}
    }
  else // track i empty
    if (!BestTrack) BestTrack = i;

if (!BestTrack) // no track available
 return false;
```

Hold (continue)

```
  // add c to best track
  H[BestTrack].Add(c);
  cout << "Move car " << c << " from input "
       << "to holding track " << BestTrack
       << endl;

  // update minH and minQ if needed
  if (c < minH) {minH = c;
                 minQ = BestTrack;}

  return true;
}
```

Program 6.7 Rearranging cars using queues (concluded)

Output without using queue

```
void Output(int NowOut, int Track, int & Last)
{// Move car NowOut from hold to output,
 //update Last.
  cout << "Move car " << NowOut
       << " from holding track "
       << Track << " to output" << endl;
  if (NowOut == Last) Last = 0;
}
bool Hold(int c, int last[],
                        int track[],int k)
{// Add car c to a holding track.
// Return false if no feasible holding track.
// Return true otherwise.
  // find best holding track for car c
  // initialize
  int BestTrack = 0, // best track so far
    BestLast = 0; // last car in BestTrack
```

Output without using queue (continue)

```
// scan holding tracks
for (int i = 1; i <= k; i++) // find best track
  if (last[i]) {// track i not empty
    if (c > last[i] && last[i] > BestLast){
     // track i has bigger car at end
     BestLast = last[i];
     BestTrack = i;}
    }
  else // track i empty
    if (!BestTrack) BestTrack = i;
```

Output without using queue (continue)

```
    if (!BestTrack) // no track available
     return false;
    // add c to best track
    track[c] = BestTrack;
    last[BestTrack] = c;
    cout << "Move car " << c << " from input "
         << "to holding track " << BestTrack
         << endl;

    return true;
}
```

Program 6.8 Rearranging cars without the use of a queue (continues)

Railroad without using queue - *O(nlogk)*

```
bool Railroad(int p[], int n, int k)
{// k track rearrangement of car order p[1:n].
// Return true if successful, false if impossible.
// Throw NoMem exception if inadequate space.

   // initialize arrays last and track
   int *last = new int [k + 1];
   int *track = new int [n + 1];
```

Railroad without using queue (continue)

```
for (int i = 1; i <= k; i++)
  last[i] = 0; // track i is empty
for (int i = 1; i <= n; i++)
  track[i] = 0; // car i is on no track
k--; // keep track k open for direct moves

// initialize index of next car
// that goes to output
int NowOut = 1;
// output cars in order
for (int i = 1; i <= n; i++)
  if (p[i] == NowOut) {// send straight to output
    cout << "Move car " << p[i]
         << "from input to output" << endl;
    NowOut++;
```

Railroad without using queue (continue)

```
// output from holding tracks
while (NowOut <= n && track[NowOut]) {
 Output(NowOut, track[NowOut], last[NowOut]);
 NowOut++;
 }
}
else {// put car p[i] in a holding track
 if (!Hold(p[i], last, track, k))
 return false;}

return true;
}
```

Program 6.8 Rearranging cars without the use of a queue (concluded)

Image-Component labeling

		1				
		1	1			
				1		
			1	1		
	1			1		1
1	1	1				1
1	1	1			1	1

(a) A 7 × 7 image

		2				
		2	2			
				3		
			3	3		
	4			3		5
4	4	4				5
4	4	4			5	5

(b) Labeled components

Figure 6.13 Image-component labeling

Offset

move	direction	offset[move].row	offset[move].col
0	right	0	1
1	down	1	0
2	left	0	−1
3	up	−1	0

Figure 5.18 Table of offsets

Wall of blank pixels (0)
Note: change 1 to 0

1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	0	0	0	0	1
1	0	0	0	0	0	1	0	1	0	0	1
1	0	0	0	1	0	1	0	0	0	0	1
1	0	1	0	1	0	1	0	1	1	0	1
1	0	1	0	1	0	1	0	1	0	0	1
1	0	1	1	1	0	1	0	1	0	1	1
1	0	1	0	0	0	1	0	1	0	1	1
1	0	1	0	1	1	1	0	1	0	0	1
1	1	0	0	0	0	0	0	1	0	0	1
1	0	0	0	0	1	1	1	1	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1

Figure 5.15 Maze of figure 5.8 with wall of ones around it

Component labeling

```
void Label()
{// Label image components.
  // initialize wall of 0 pixels
  for (int i = 0; i <= m+1; i++) {
    // bottom & top
    pixel[0][i] = pixel[m+1][i] = 0;
    // left & right
    pixel[i][0] = pixel[i][m+1] = 0;
    }
  // initialize offsets
  Position offset[4];
  offset[0].row = 0; offset[0].col = 1; // right
  offset[1].row = 1; offset[1].col = 0; // down
  offset[2].row = 0; offset[2].col = -1; // left
  offset[3].row = -1; offset[3].col = 0; // up

 int NumOfNbrs = 4;//neighbors of a pixel position
  LinkedQueue<Position> Q;
  int id = 1; // component id
  Position here, nbr;
```

Component labeling (continue)

```
//scan all pixels labeling components
for (int r = 1; r <= m; r++) // row r of image
  for (int c = 1; c <= m; c++) // column c
   if (pixel[r][c] == 1) {// new component}
    pixel[r][c] = ++id; // get next id
    here.row = r; here.col = c;
    do {// find rest of component
    for (int i = 0; i < NumOfNbrs; i++) {
    // check all neighbors of here
     nbr.row = here.row + offset[i].row;
     nbr.col = here.col + offset[i].col;
    if (pixel[nbr.row][nbr.col] == 1) {
     pixel[nbr.row][nbr.col] = id;
     Q.Add(nbr);}} // end of if and for
    //any unexplored pixels in component?
    if (Q.IsEmpty()) break ;
     Q.Delete(here); //a component pixel
    } while (true);
   }} // end if, for c, for r, and Label
```

Program 6.10 Component labelling being the use of a queue

Evaluation

- Initialize the wall - $\Theta(m)$
- Initialize offsets - $\Theta(1)$
- identifying and labeling pixels of one component - $\Theta(\#\ of\ pixels\ in\ component)$
- identifying and labeling nonseed component - $\Theta(\#\ of\ pixels\ in\ component\ pixels\ in\ image) = O(m^2)$
- Overall complexity - $O(m^2)$

End of Chapter 6