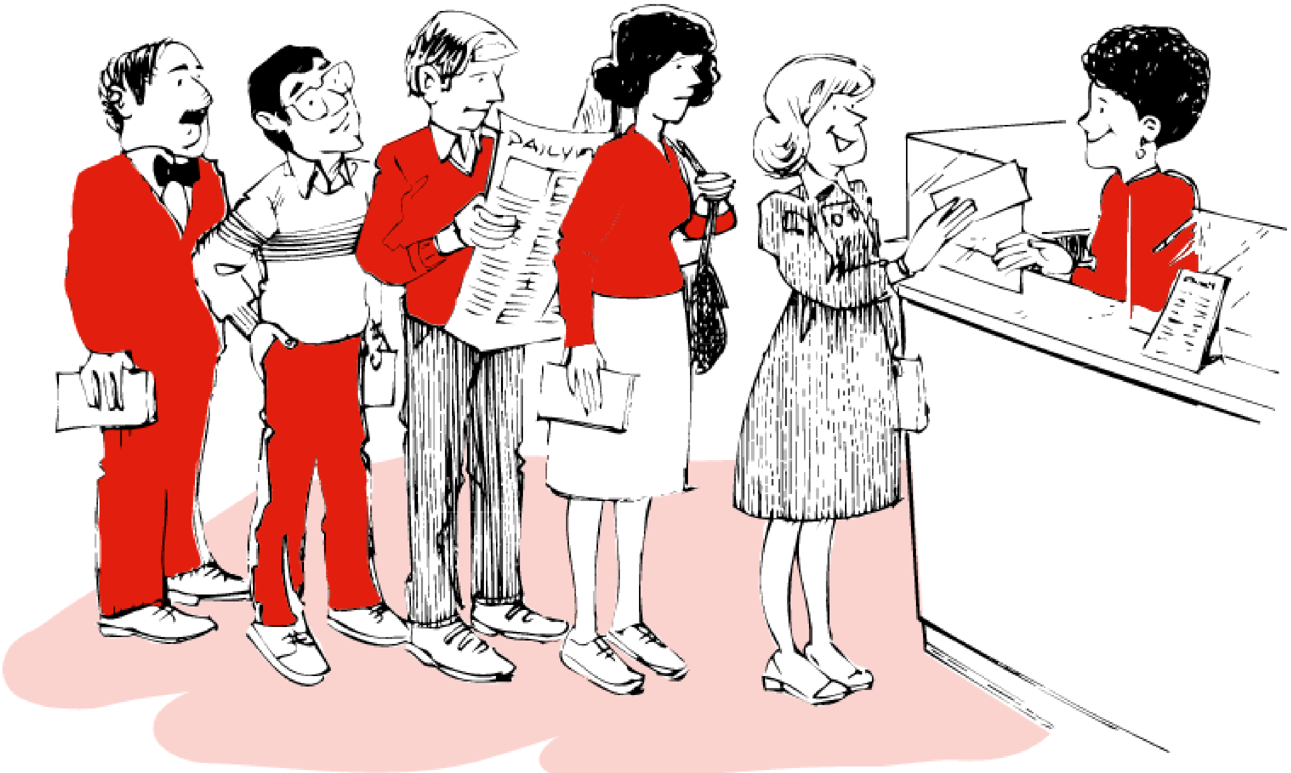
Chapter 3 Queues

信息科学与技术学院　　　黄方军

data_structures@163.com

东校区实验中心B502

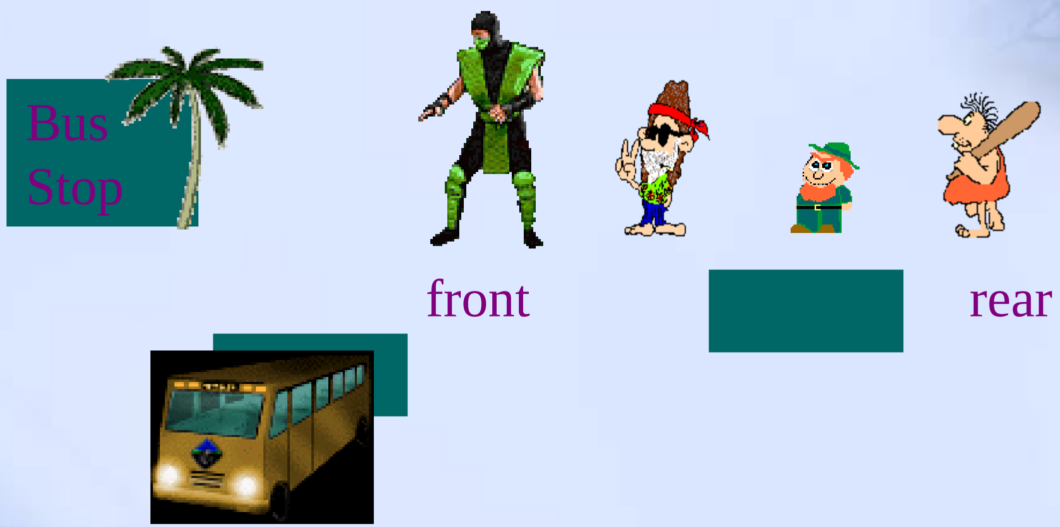
3.1 Definitions



A queue
Transp. 2, Sect. 3.1, Definitions
55
Data Structures and Program Design In C++

Bus Stop Queue

Bus Stop Queue



Bus Stop Queue

front rear

Bus Stop Queue

front

rear

3.1 Definitions

队列(Queues)是生活中“排队”的抽象。队列的特点是：

- 有穷个同类型元素的线性序列；
- 新加入的元素排在队尾，出队的元素在对头删除，即入队和出队分别在队列的两端进行；
- 先来的先得到服务；故称为先进先出表 (FIFO, First in First out).

3.1 Definitions

一个队列(*queue*) 是同类型元素的线性表，其中插入在一端进行，删除在另一端进行。

删除进行的一端称为队头（ *the front* ， *or the head*). 最新加入的元素称为队尾（*The rear or tail*）。

例子：等待打印的任务构成一个队列；等待CPU服务的任务构成一个队列等。

3.1.1 Queue Operations

The ADT Queue class:

```
class Queue {
 public:
  Queue();
  bool empty() const;
  Error_code append(const Queue_entry &x);
  Error_code serve();
  Error_code retrieve(Queue_entry &x)const;
};
```

3.1.1 Queue Operations

设 Queue_entry 表示队列元素的类型。

Constructor

Queue :: Queue( );

Post: The Queue has been created and is initialized to be empty.

Insertion
（入队）

Error_code Queue :: append(**const** Queue_entry &x);

Post: If there is space, x is added to the Queue as its rear. Otherwise an Error_code of overflow is returned.

3.1.1 Queue Operations

Deletion
出队

Error_code Queue :: serve( );

Post: If the Queue is not empty, the front of the Queue has been removed. Otherwise an Error_code of underflow is returned.

Get the front
取队头元素

Error_code Queue :: retrieve(Queue_entry &x) **const;**

Post: If the Queue is not empty, the front of the Queue has been recorded as x. Otherwise an Error_code of underflow is returned.

3.1.1 Queue Operations

Check emptiness
检查队是否空

bool Queue :: empty( ) **const;**

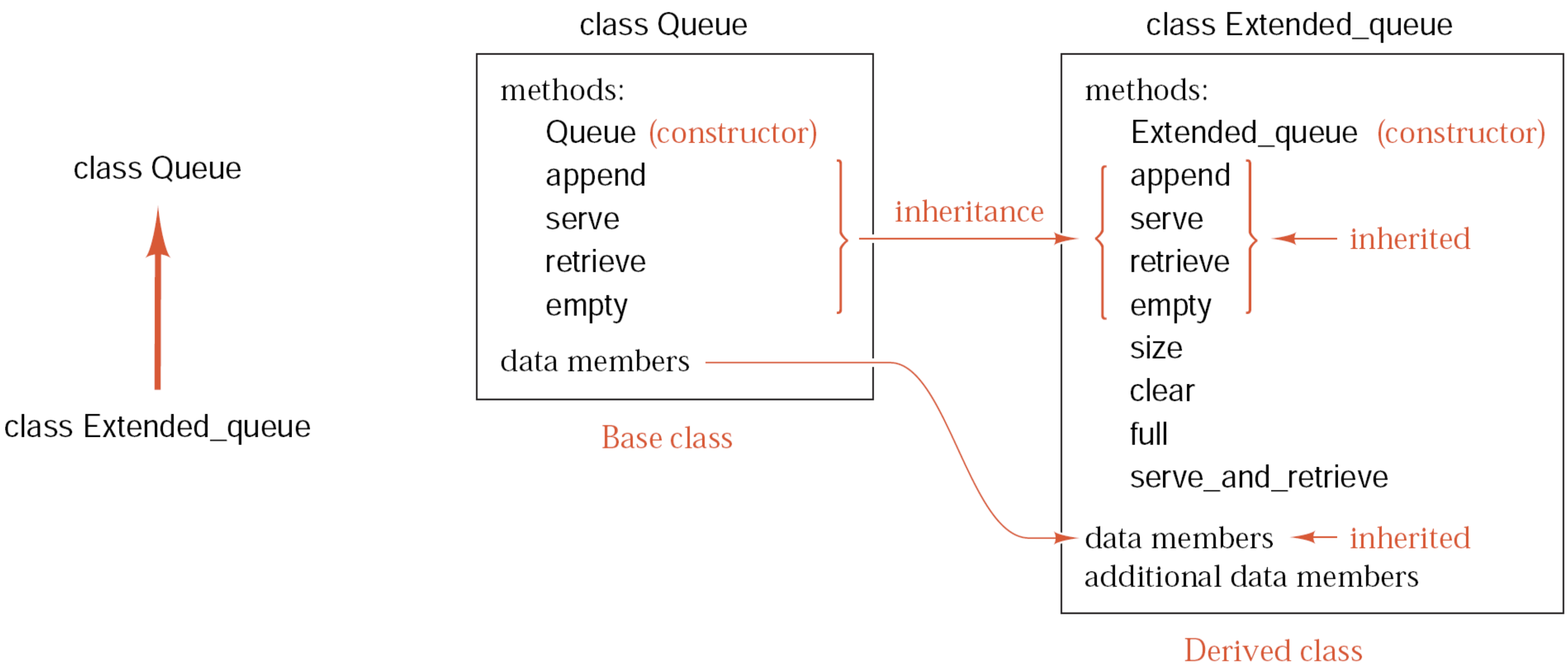
Post: Return **true** if the Queue is empty, otherwise return **false**.

3.1.2 Extended Queue Operations

If we want to add some operations on queues, for example, full, clear, serve_and_retrieve, one way is to extend the class Queue:

```
class Extended_queue:public Queue {
  public:
   bool full() const;
   int size() const;
   void clear();
  Error_code serve_and_retrieve(Queue_entry &item);
};
```

3.1.2 Extended Queue Operations



(a) Hierarchy diagram

(b) Derived class Extended_queue from base class Queue

3.1.2 Extended Queue Operations

bool Extended_queue :: full( ) **const;**

postcondition: Return **true** if the Extended_queue is full; return **false** otherwise.

void Extended_queue :: clear( );

postcondition: All entries in the Extended_queue have been removed; it is now empty.

int Extended_queue :: size( ) **const;**

postcondition: Return the number of entries in the Extended_queue.

Error_code Extended_queue :: serve_and_retrieve(Queue_entry &item);

postcondition: Return underflow if the Extended_queue is empty. Otherwise remove and copy the item at the front of the Extended_queue to item and return success.

3.2 Implementations of Queues

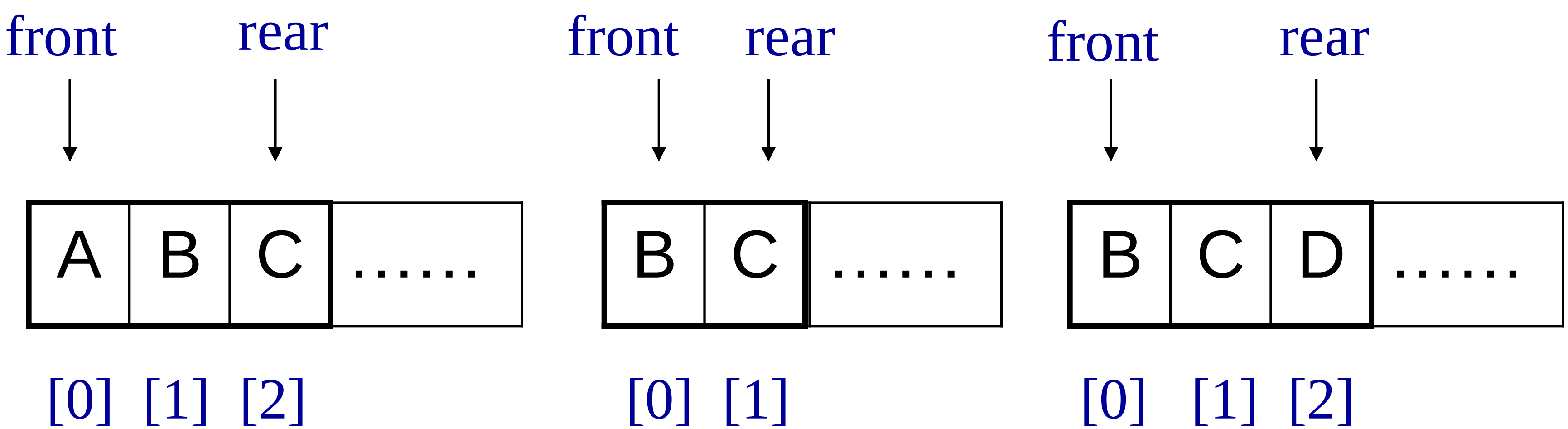
如何表示队列元素呢？考虑连续队列，即用数组存储队列元素。

1. The physical model

A linear array with *the front always in the first position* and all entries moved up the array whenever the front is removed.

Poor!

3.2 Implementations of Queues



$$\text{location}(i) = i\text{-}1$$

3.2 Implementations of Queues

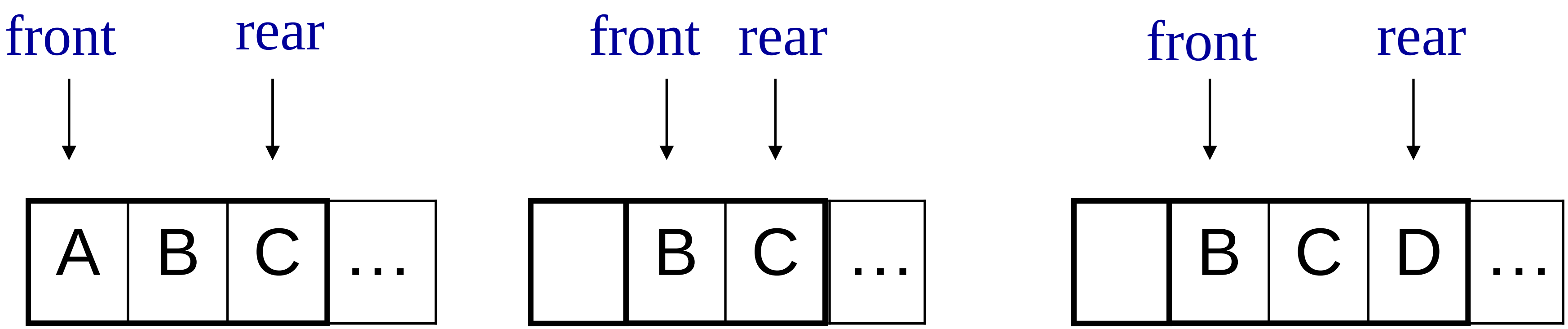
2. Linear Implementation

- Two indices（下标） to keep track of both the front and the rear of the queue.
- To serve an entry, take the entry and increase the front by one.
- To append an entry to the queue, increase the rear by one and put the entry in that position.

Problem: cannot reuse the discarded space.

When the queue is regularly emptied, this is good.

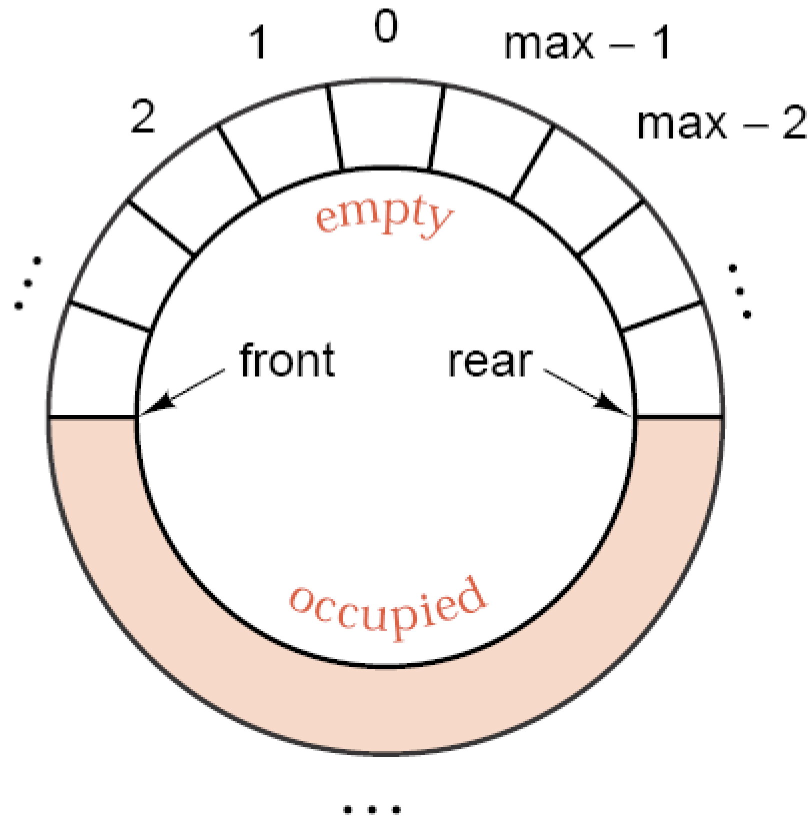
3.2 Implementations of Queues



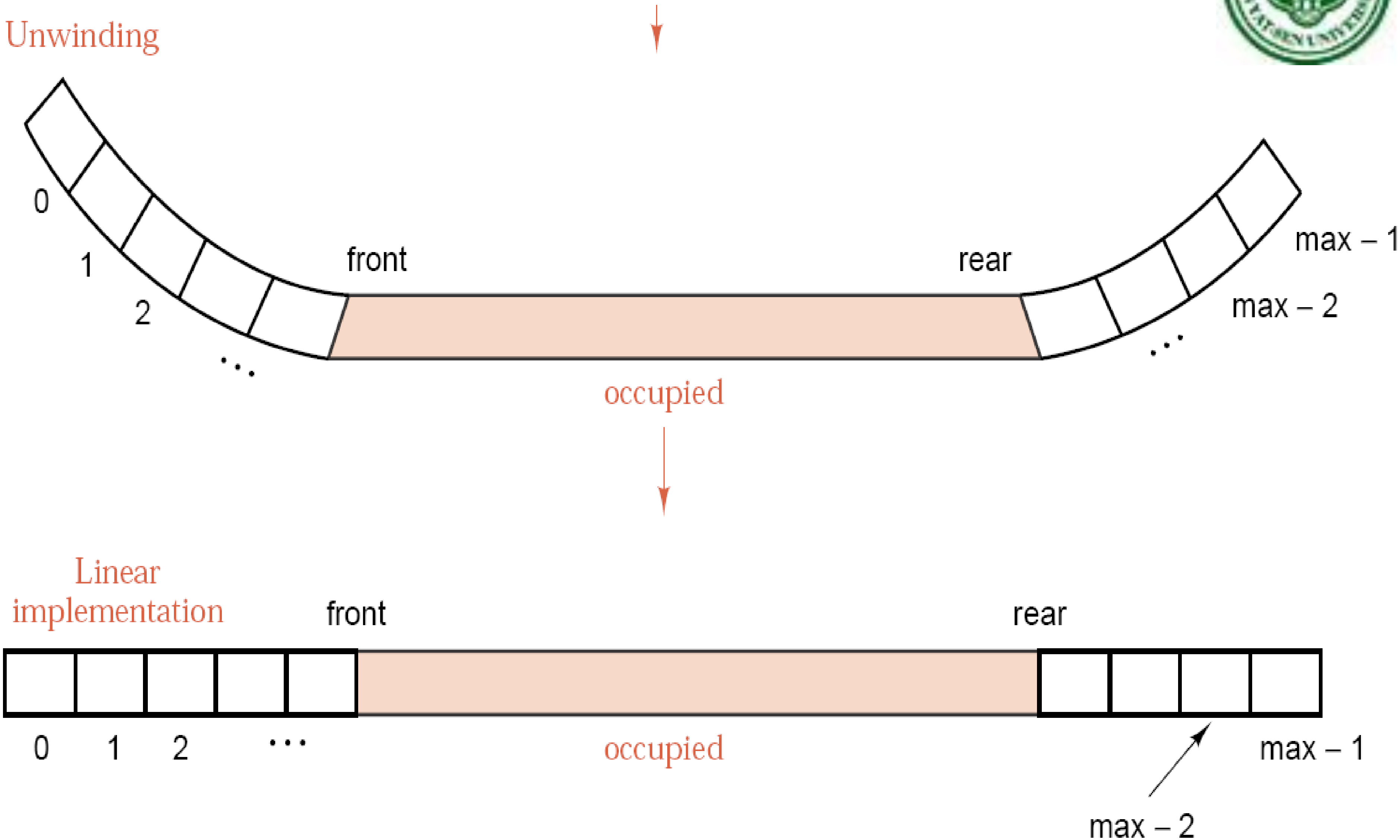
$$\text{location}(i) = \text{location}(1) + i - 1$$

3. Circular Arrays

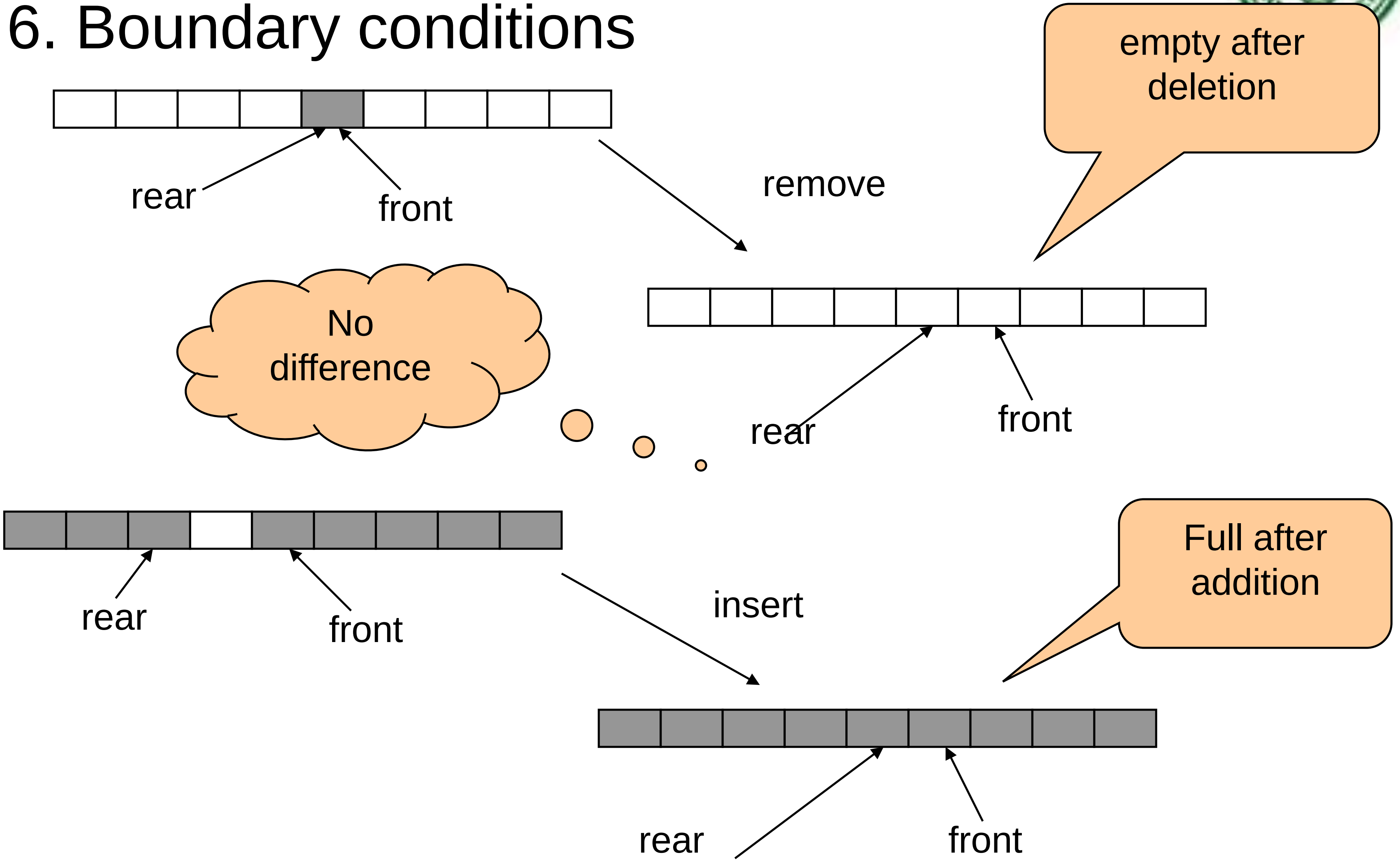
Circular
queue



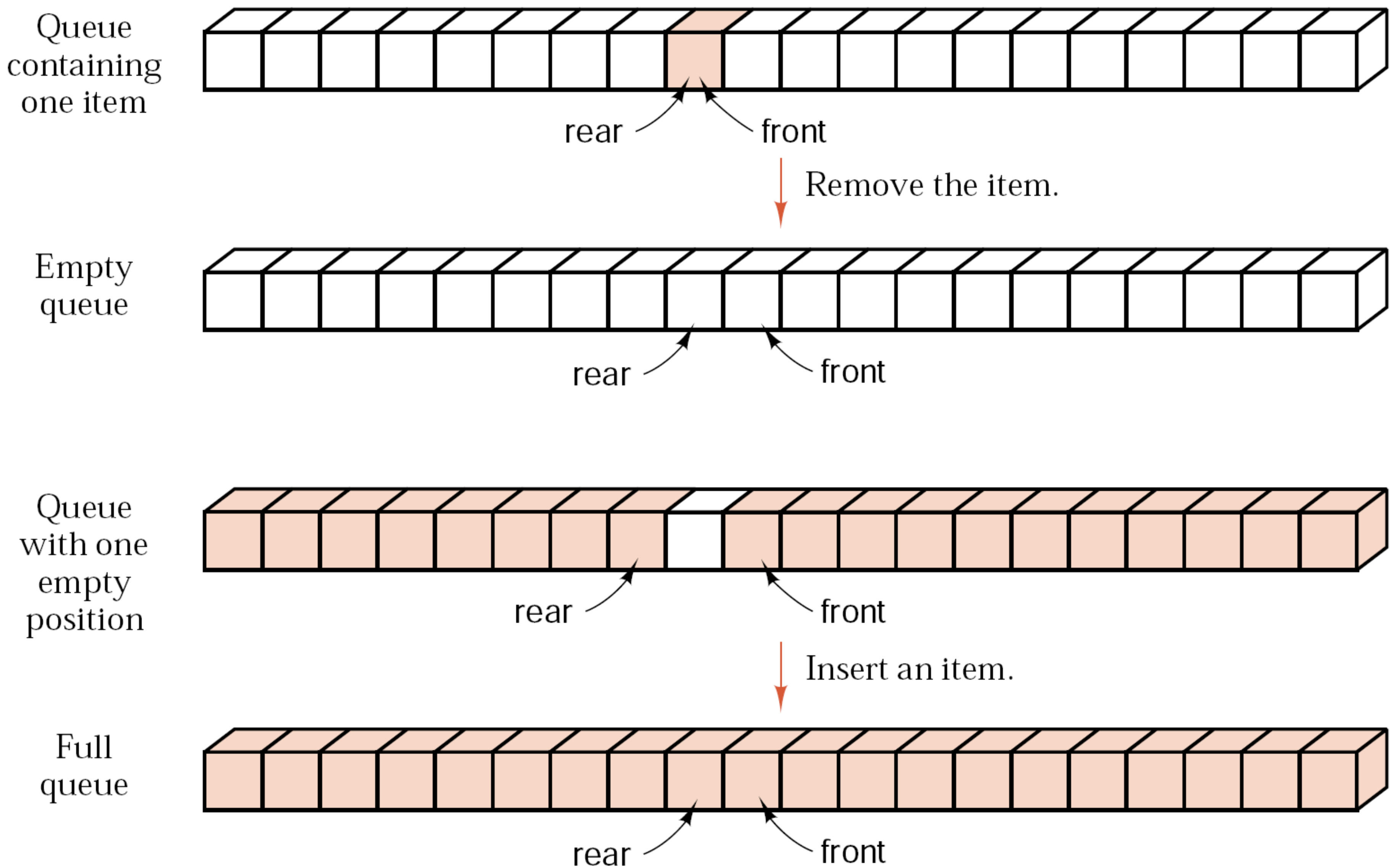
3.2 Implementations of Queues



6. Boundary conditions



3.2 Implementations of Queues



3.2 Implementations of Queues

7. Possible solutions

问题：无法区分满队列与空队列。

解决方法:

- 在数组中空一个位置;
- 使用一个布尔量表示队列是否满。当rear刚好到达front之前时，置 此标志为true.
- 使用一个计数器（ counter）以记录队列中的元素个数。

3.3 Circular Implementations of Queues

```
const int maxqueue = 10;        //    small value for testing


class Queue {
public:
    Queue( );
    bool empty( ) const;
    Error_code serve( );
    Error_code append(const Queue_entry &item);
    Error_code retrieve(Queue_entry &item) const;
protected:
    int count;
    int front, rear;
    Queue_entry entry[maxqueue];
};
```

3.3 Circular Implementations of Queues

```
Queue :: Queue( )
/* Post: The Queue is initialized to be empty. */
{
   count = 0;
   rear = maxqueue - 1;
   front = 0;
}

bool Queue :: empty( ) const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return count ==  0;
}
```

3.3 Circular Implementations of Queues

```
Error_code Queue :: append(const Queue_entry &item)
/* Post: item is added to the rear of the Queue. If the Queue is full return an
         Error_code of overflow and leave the Queue unchanged. */
{
   if (count >= maxqueue) return overflow;
   count++;
   rear = ((rear + 1) == maxqueue) ? 0 : (rear + 1);
   entry[rear] = item;
   return success;
}
```

3.3 Circular Implementations of Queues

```
Error_code Queue::serve()
/* Post: The front of the Queue is removed.  If the Queue is empty return an
         Error_code of underflow. */
{
   if (count <= 0) return underflow;
   count--;
   front = ((front + 1) == maxqueue) ? 0 : (front + 1);
   return success;
}
```

3.3 Circular Implementations of Queues

```
Error_code Queue::retrieve(Queue_entry &item) const
/* Post: The front of the Queue retrieved to the output parameter item. If the
        Queue is empty return an Error_code of underflow. */
{
   if (count <= 0) return underflow;
   item = entry[front];
   return success;
}
```

3.3 Circular Implementations of Queues

```
int Extended_queue::size() const
/* Post: Return the number of entries in the Extended_queue. */
{
   return count;
}
```

3.4 Demonstration and Testing

```
int main()
/* Post: Accepts commands from user as a menu-driven demonstration program
         for the class Extended_queue.
   Uses: The class Extended_queue and the functions introduction, get_command,
         and do_command. */

{
   Extended_queue test_queue;
   introduction();
   while (do_command(get_command(), test_queue));
}
```

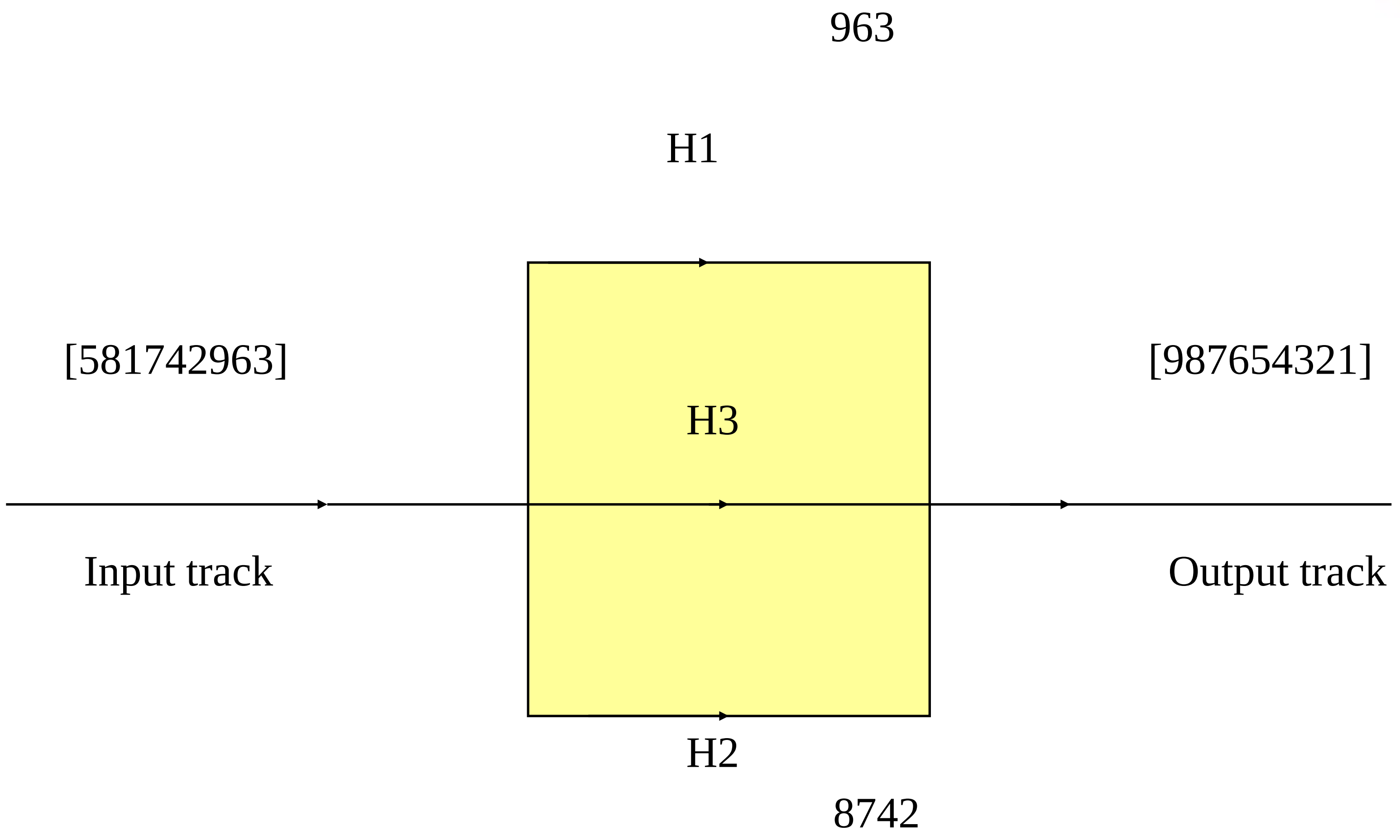
3.3 Circular Implementations of Queues

```
bool do_command(char c, Extended_queue &test_queue)
/* Pre:   c represents a valid command.
   Post:  Performs the given command c on the Extended_queue test_queue. Re-
          turns false if c == 'q', otherwise returns true.
   Uses:  The class Extended_queue. */

{
  bool continue_input = true;
  Queue_entry x;
  switch (c) {
  case 'r':
    if (test_queue.retrieve(x) == underflow)
      cout << "Queue is empty." << endl;
    else
      cout << endl
           << "The first entry is: " << x
           << endl;
    break;
  case 'q':
    cout << "Extended queue demonstration finished." << endl;
    continue_input = false;
    break;

  //  Additional cases will cover other commands.

  }
  return continue_input;
}
```

3.5 Applications

- **Railroad Car Rearrangement**
- **Wire Routing**
- **Simulation of an Airport**

3.5 Railroad Car Rearrangement



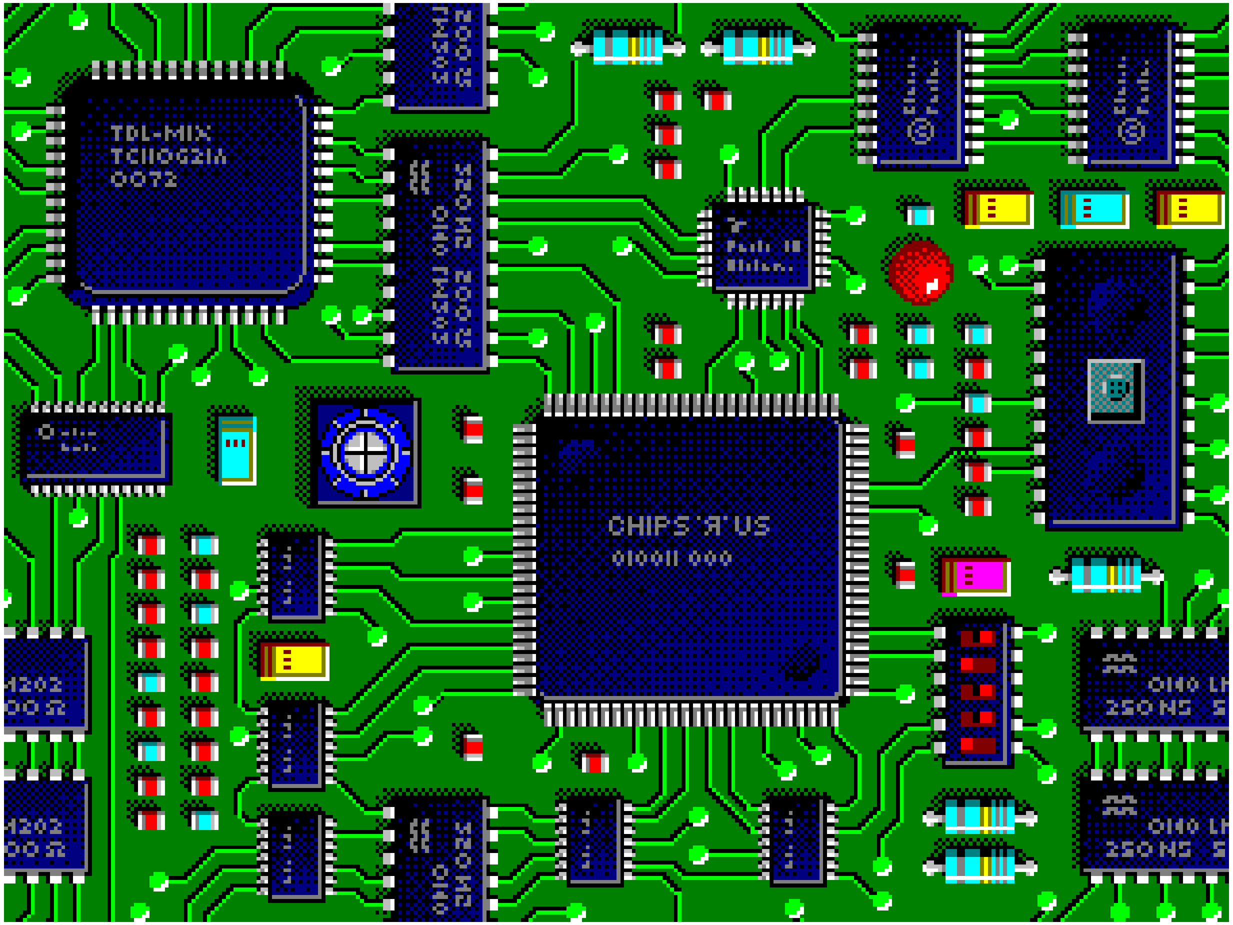
3.5 Railroad Car Rearrangement

```
bool Railroad(int p[], int n, int k)
{  LinkedQueue<int> *H; H = new LinkedQueue<int> [k];
k--;
   int NowOut = 1;   int minH = n+1;    int minQ;
   for (int i = 1; i <= n; i++)
      if (p[i] == NowOut) {// send straight out
         cout << "Move car " << p[i] <<
                  " from input to output" << endl;
         NowOut++;
         while (minH == NowOut) {
            Output(minH, minQ, H, k, n);
               NowOut++;}}
      else {// put car p[i] in a holding track
         if (!Hold(p[i], minH, minQ, H, k))
         return false;}                    return true;}
```

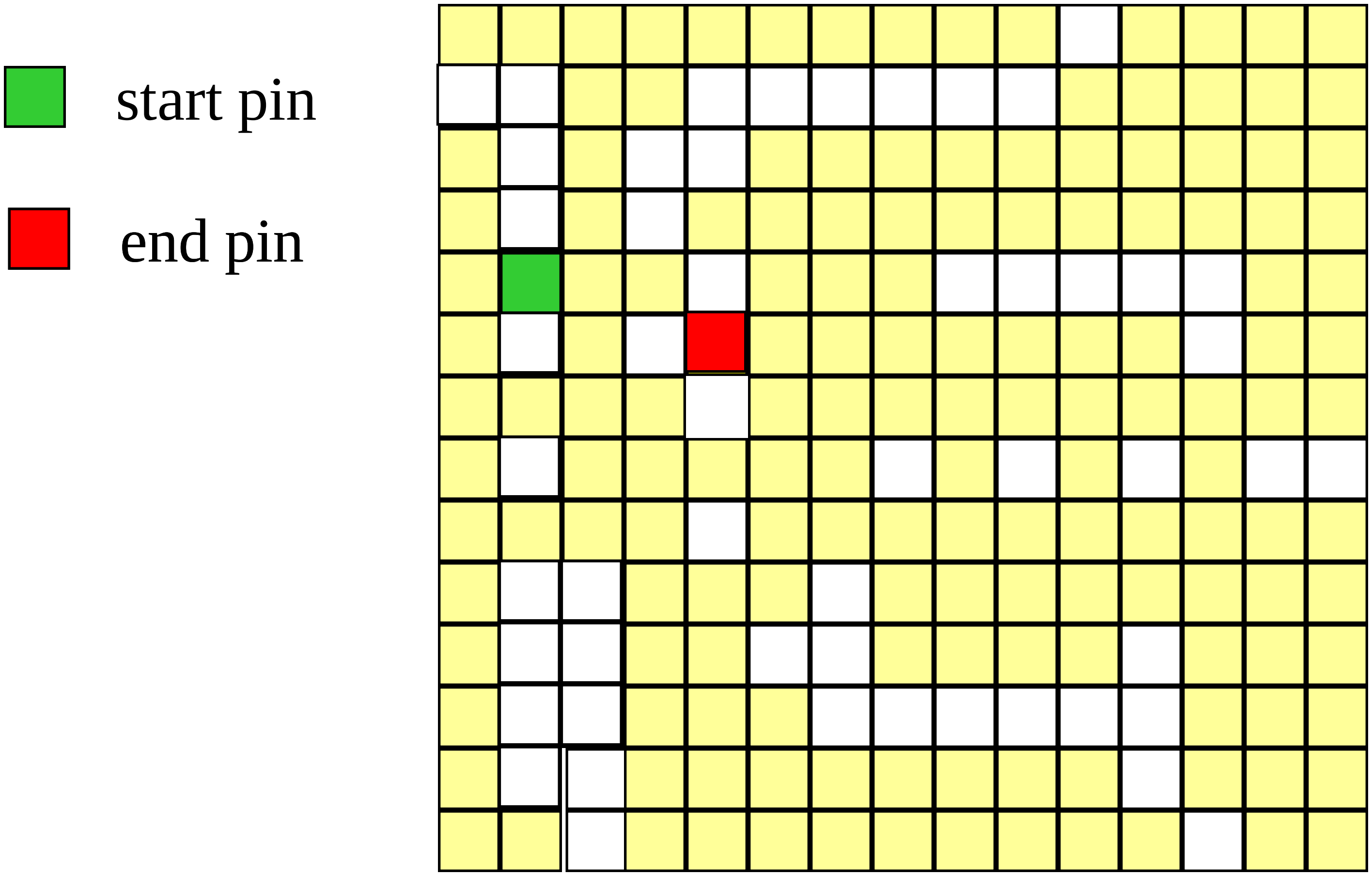
3.5 Railroad Car Rearrangement

```
void Output(int& minH, int& minQ,
        LinkedQueue<int> H[], int k, int n)
{ int c;  // car index
   H[minQ].Delete(c);
   cout << "Move car " << minH << " from holding
track " << minQ << " to output" << endl;
   minH = n + 2;
   for (int i = 1; i <= k; i++)
      if (!H[i].IsEmpty() && (c = H[i].First()) < minH)
       { minH = c;
         minQ = i; }
}
```

3.5 Wire Routing

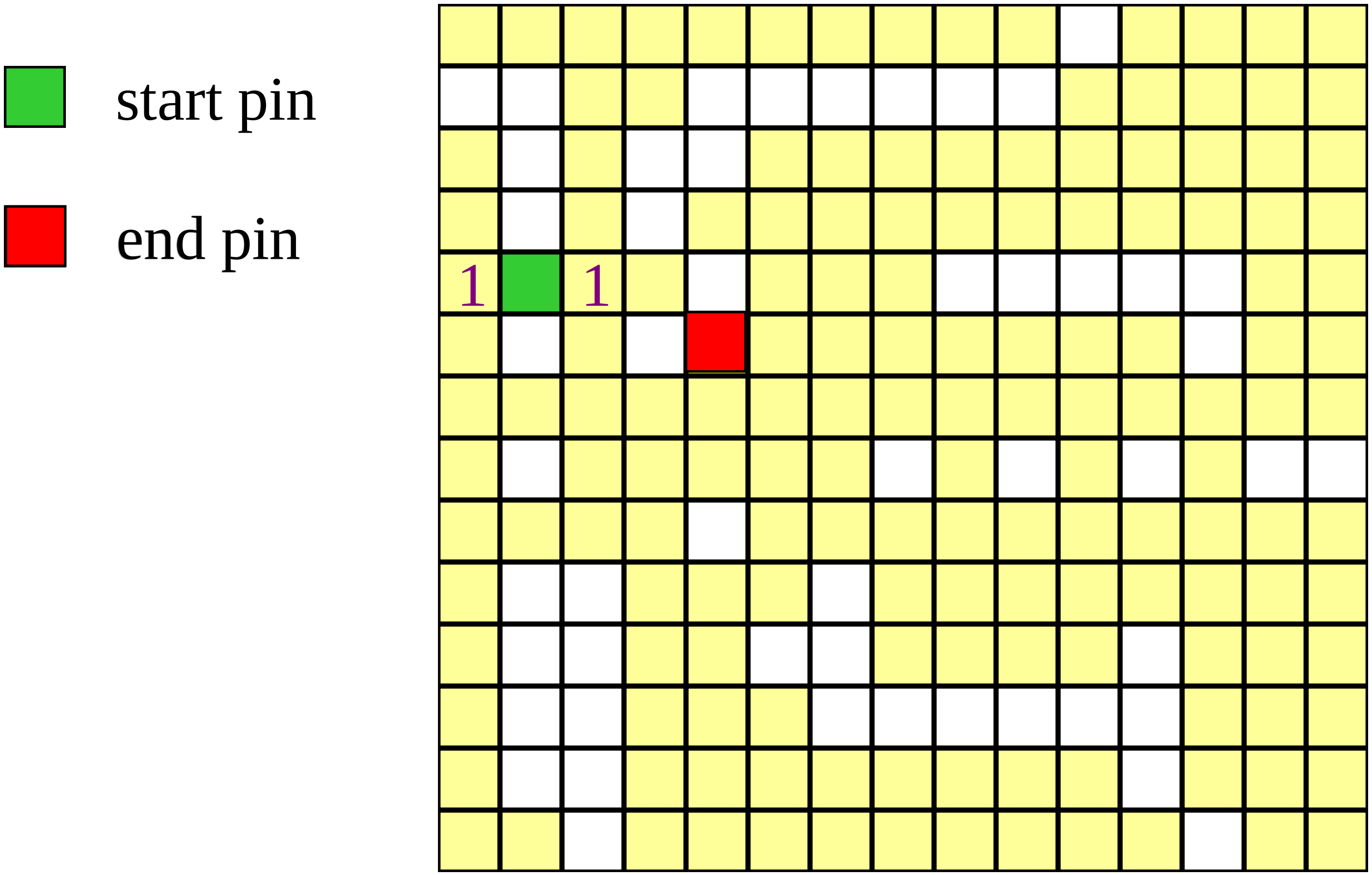


Lee’s Wire Router



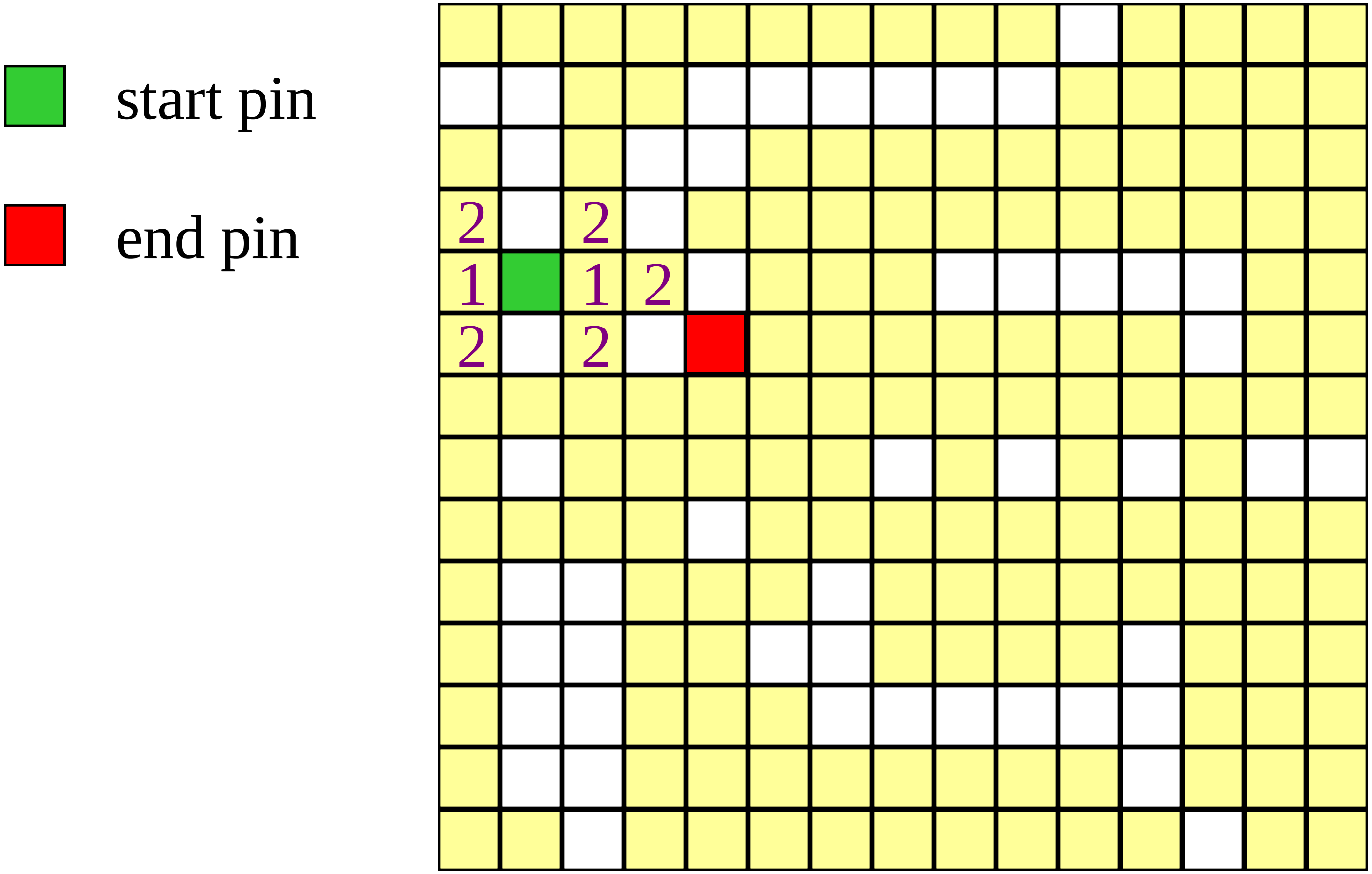
Label all reachable squares 1 unit from start.

Lee's Wire Router



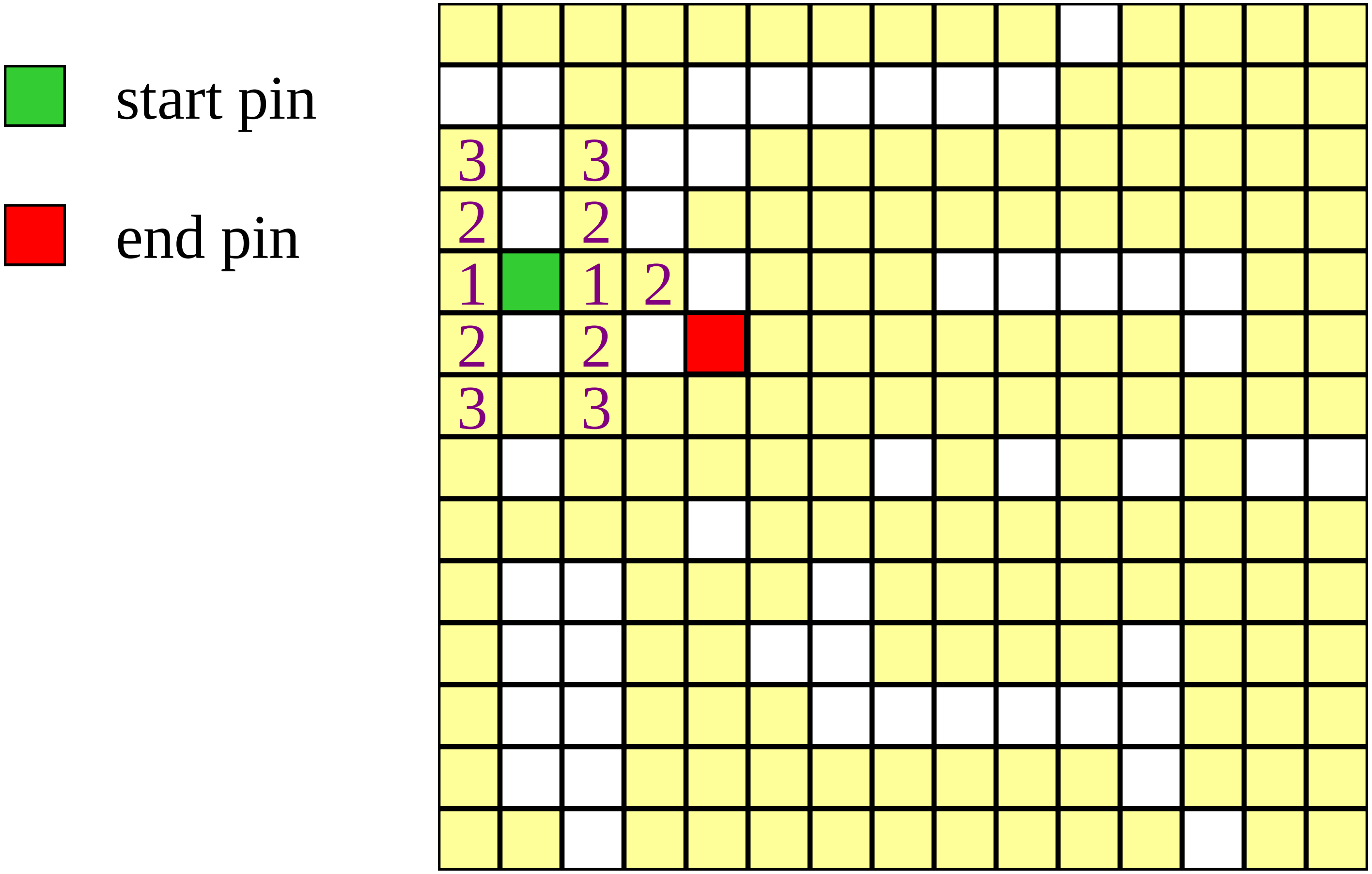
Label all reachable unlabeled squares 2 units from start.

Lee's Wire Router



Label all reachable unlabeled squares 3 units from start.

Lee's Wire Router

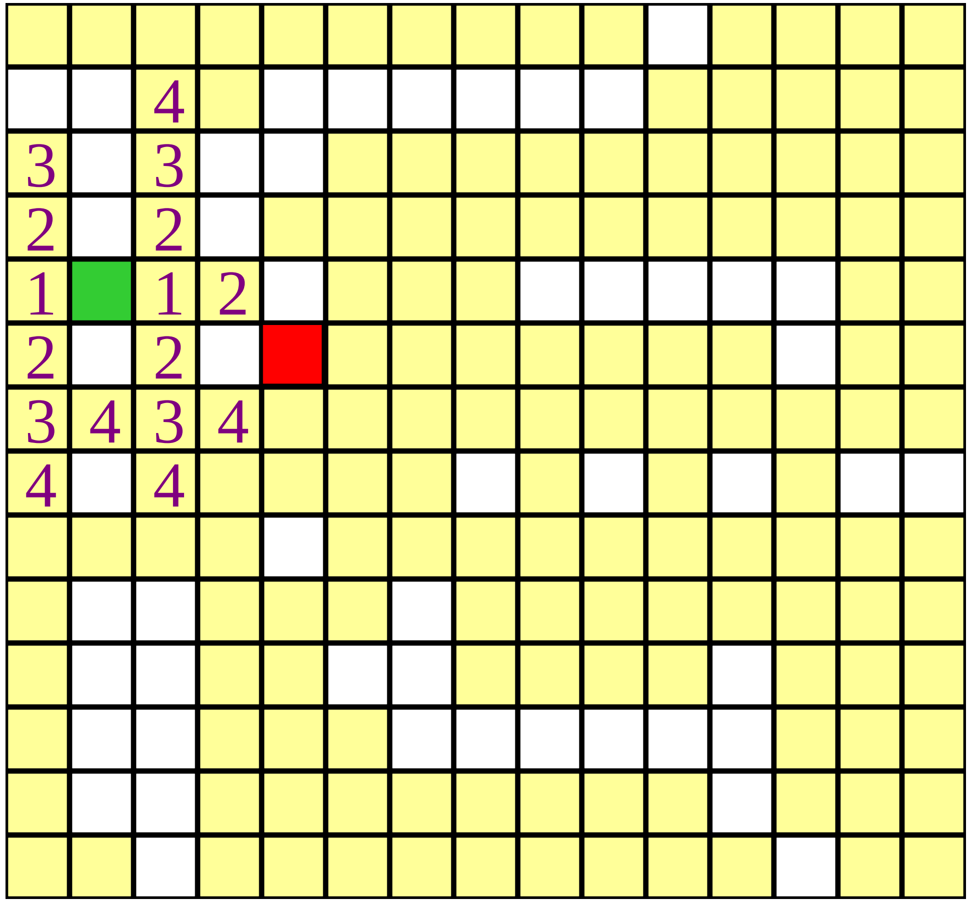


Label all reachable unlabeled squares 4 units from start.

Lee's Wire Router

start pin

end pin

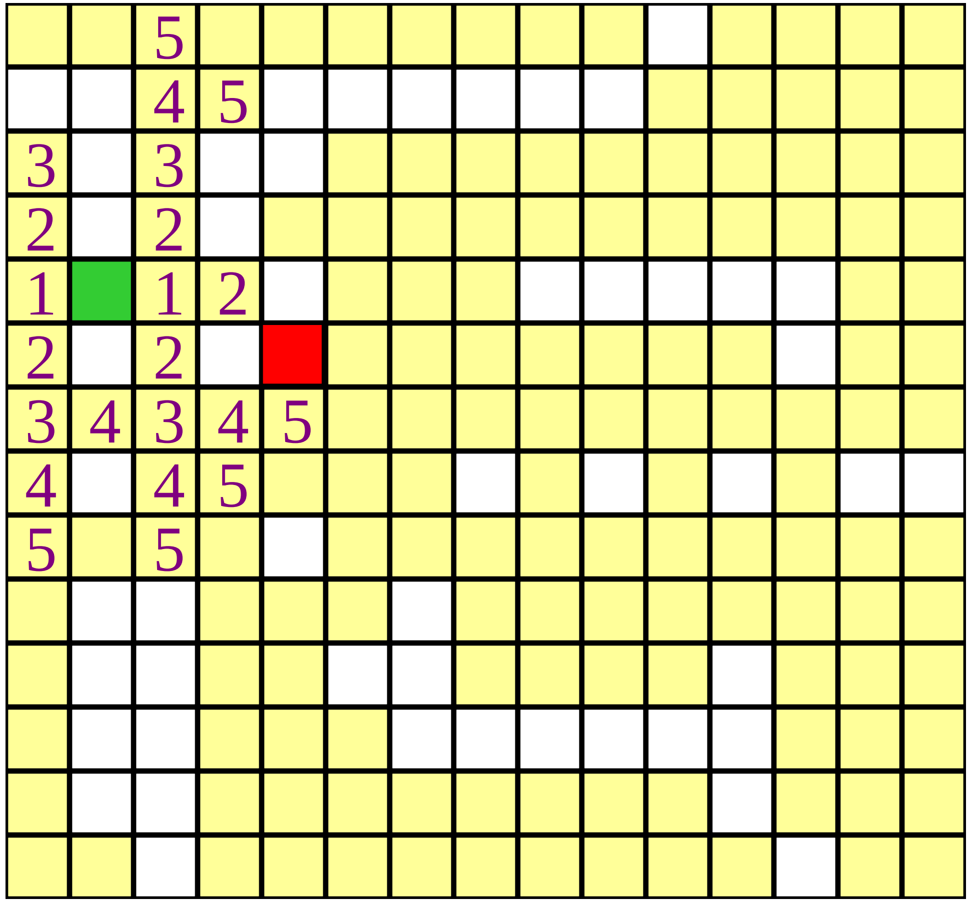


Label all reachable unlabeled squares 5 units from start.

Lee's Wire Router

start pin

end pin

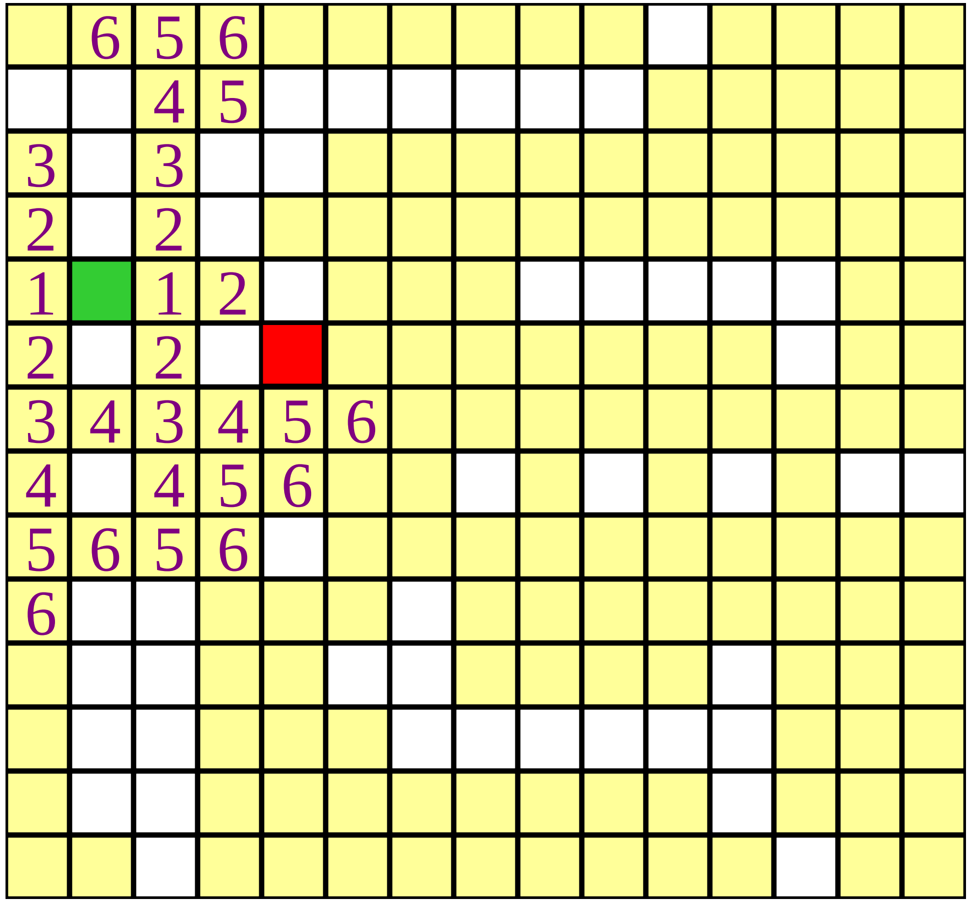


Label all reachable unlabeled squares 6 units from start.

Lee’s Wire Router

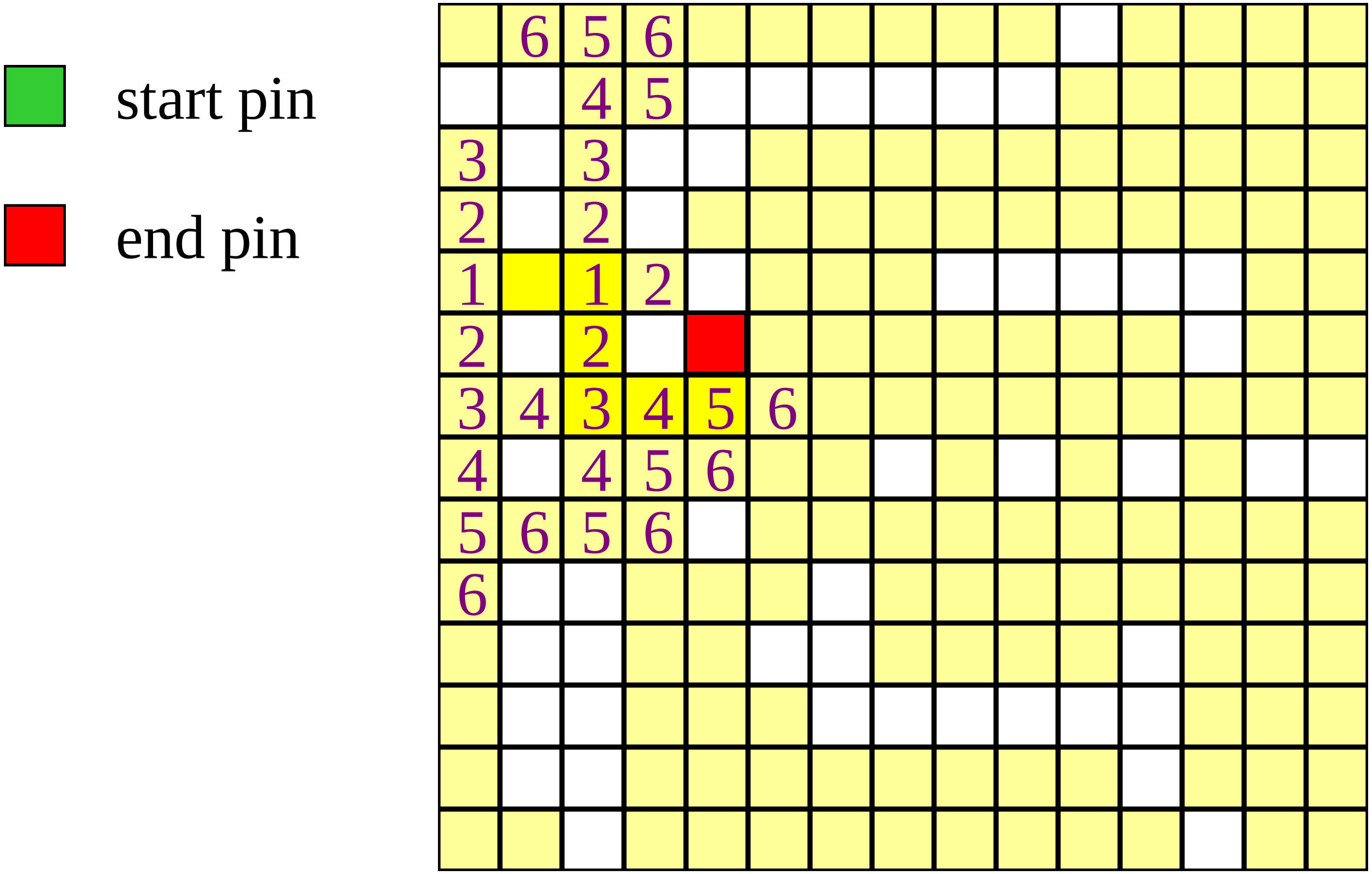
start pin

end pin



End pin reached. Traceback.

Lee's Wire Router



End pin reached. Traceback.

3.5 Wire Routing

```
LinkedQueue<Position> Q;
do {// label neighbors of here
  for (int i = 0; i < NumOfNbrs; i++) {
    nbr.row = here.row + offset[i].row;
    nbr.col = here.col + offset[i].col;
    if (grid[nbr.row][nbr.col] == 0) {
       // unlabeled nbr, label it
       grid[nbr.row][nbr.col]
         = grid[here.row][here.col] + 1;
       if ((nbr.row == finish.row) &&
         (nbr.col == finish.col)) break; // done
           Q.Add(nbr);} // end of if
    } // end of for
  if ((nbr.row == finish.row) &&
     (nbr.col == finish.col)) break; // done
  if (Q.IsEmpty()) return false; // no path
  Q.Delete(here); // get next position
} while(true);
```

3.5 Wire Routing

```
// construct path
PathLen = grid[finish.row][finish.col] - 2;
path = new Position [PathLen];
// trace backwards from finish
here = finish;
for (int j = PathLen-1; j >= 0; j--) {
   path[j] = here;
   // find predecessor position
   for (int i = 0; i < NumOfNbrs; i++) {
      nbr.row = here.row + offset[i].row;
      nbr.col = here.col + offset[i].col;
      if (grid[nbr.row][nbr.col] == j+2) break;
      }
   here = nbr;  // move to predecessor
   }
return true;
```

- Thanks for your attention!