Ve 280

Programming and Introductory Data Structures

Stacks; Queues;
Synthesized Functions

Announcement

- Due to the time limit, there will be no Project 6

- You should build demo code yourselves to understand what I teach in lecture

Outline

- Stack: Applications
- Queue
  - Implementation
  - Applications
  - Relative: Deque
- Synthesized Default Constructor, Copy Constructor, Assignment Operator, and Destructor

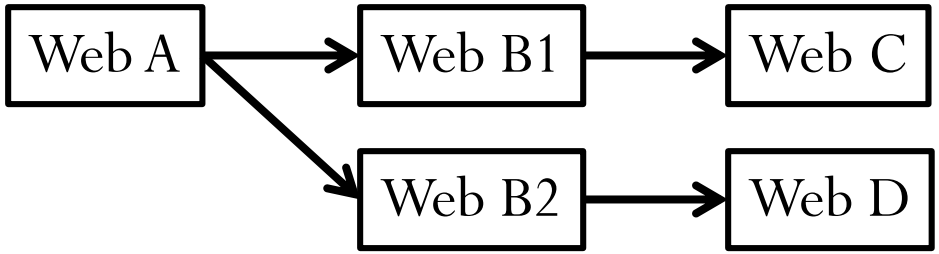
Review: Stack

- A "pile" of objects where new object is put on **top** of the pile and the top object is removed first.
- Five operations
  - size(), isEmpty(), push(), pop(), top()
- Implementation
  - Array versus linked list

Application of Stacks

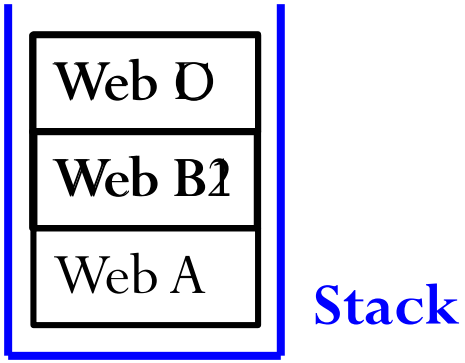
- Function calls in C++
- Web browser's "back" feature
- Parentheses Matching

Web Browser's "back" Feature



Visiting order

- Web A
- Web B1
- Web C
- Back (to Web B1)
- Back (to Web A)
- Web B2
- Web D



Parentheses Matching

- Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.

```
( ( a + b ) * c + d - e ) / ( f + g )
0 1 2 3 4 5 6 7 8 9 10  12  14  16  18
```

  - Output is: (1, 5); (0, 12); (14, 18);

```
( a + b ) ) * ( ( c + d )
0 1 2 3 4 5 6 7 8 9 10  12
```

  - Output is

    (0, 4);

    Right parenthesis at 5 has no matching left parenthesis;

    (8, 12);

    Left parenthesis at 7 has no matching right parenthesis

How to Realize Parentheses Matching?

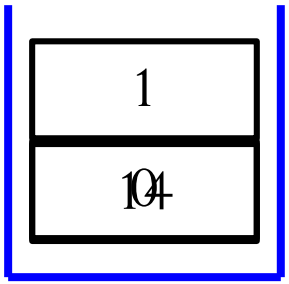
```
( ( a + b ) * c + d – e ) / ( f + g )
0 1 2 3 4 5 6 7 8 9 10  12  14  16  18
```

- Scan expression from left to right.
- When a **left** parenthesis is encountered, push its position to the stack.
- When a **right** parenthesis is encountered, pop the top position from the stack, which is the position of the **matching left** parenthesis.
  - If the stack is empty, the **right** parenthesis is not matched.
- If string is scanned over but the stack is not empty, there are not-matched **left** parentheses.

Parentheses Matching

```
( ( a + b ) * c + d – e ) / ( f + g )
0 1 2 3 4 5 6 7 8 9 10  12  14  16  18
```

Stack



(1, 5) (0, 12) (14, 18)

Outline

- Stack: Applications
- Queue
  - Implementation
  - Applications
  - Relative: Deque
- Synthesized Default Constructor, Copy Constructor, Assignment Operator, and Destructor

Queues

- A "line" of items in which the **first** item inserted into the queue is the **first** one out.
  - Restricted form of a linear list: insert at **one end** and remove from **the other**.
  - FIFO access: first in, first out.

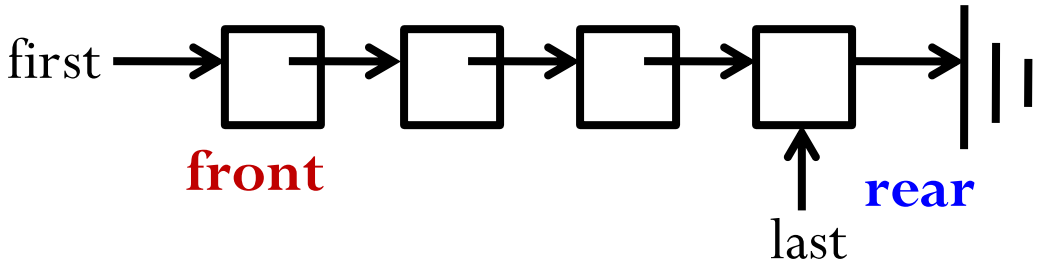
Methods of Queue

- **`size()`** : number of elements in the queue.
- **`isEmpty()`** : check if queue has no elements.
- **`enqueue(Object o)`** : add object **`o`** to the **rear** of the queue.
- **`dequeue()`** : remove the **front** object of the queue if not empty; otherwise, throw **`queueEmpty`**.
- **`Object &front()`** : return a reference to the front element of the queue.
- **`Object &rear()`** : return a reference to the rear element of the queue.

Queues Using Linked Lists

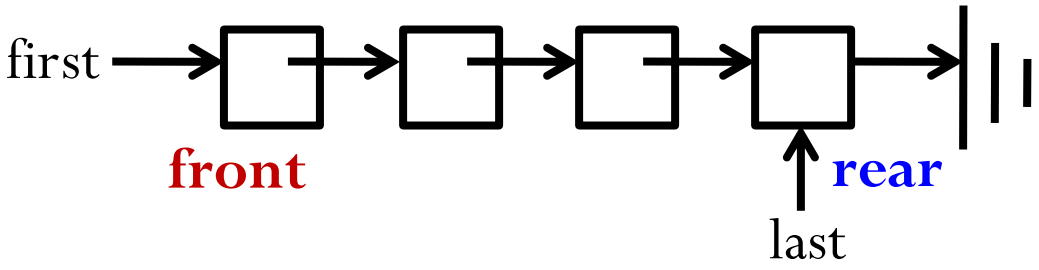
- Which type of linked list should we choose?
  - We need fast **`enqueue`** and **`dequeue`** operations.

- Double-ended singly-linked list is sufficient!



- **`enqueue(Object o)`**: append object at the end
  `LinkedList::insertLast(Object o);`
- **`dequeue()`**: remove the first node
  `LinkedList::removeFirst();`

Queues Using Linked Lists



- **`size()`**: **`LinkedList::size();`**
- **`isEmpty()`**: **`LinkedList::isEmpty();`**
- **`Object &front()`**: return a reference to the object stored in the first node.
- **`Object &rear()`**: return a reference to the object stored in the last node.

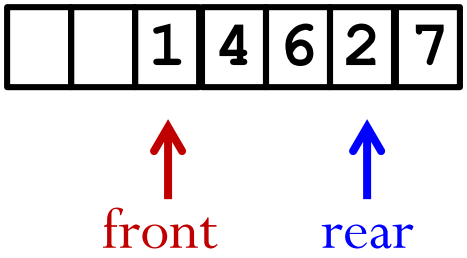
Queues Using Arrays

- If we stick to the requirement that the n elements of a queue are the **beginning** n elements of the array,
  - How many operations for **enqueue**?
    - I.e., independent of n (number of elements) or proportional to n?
  - How many operations ofr **dequeue**?

- A better way is to let the elements "**drift**" within the array.

```
enqueue(6);
dequeue();
dequeue();
```

Queues Using Arrays

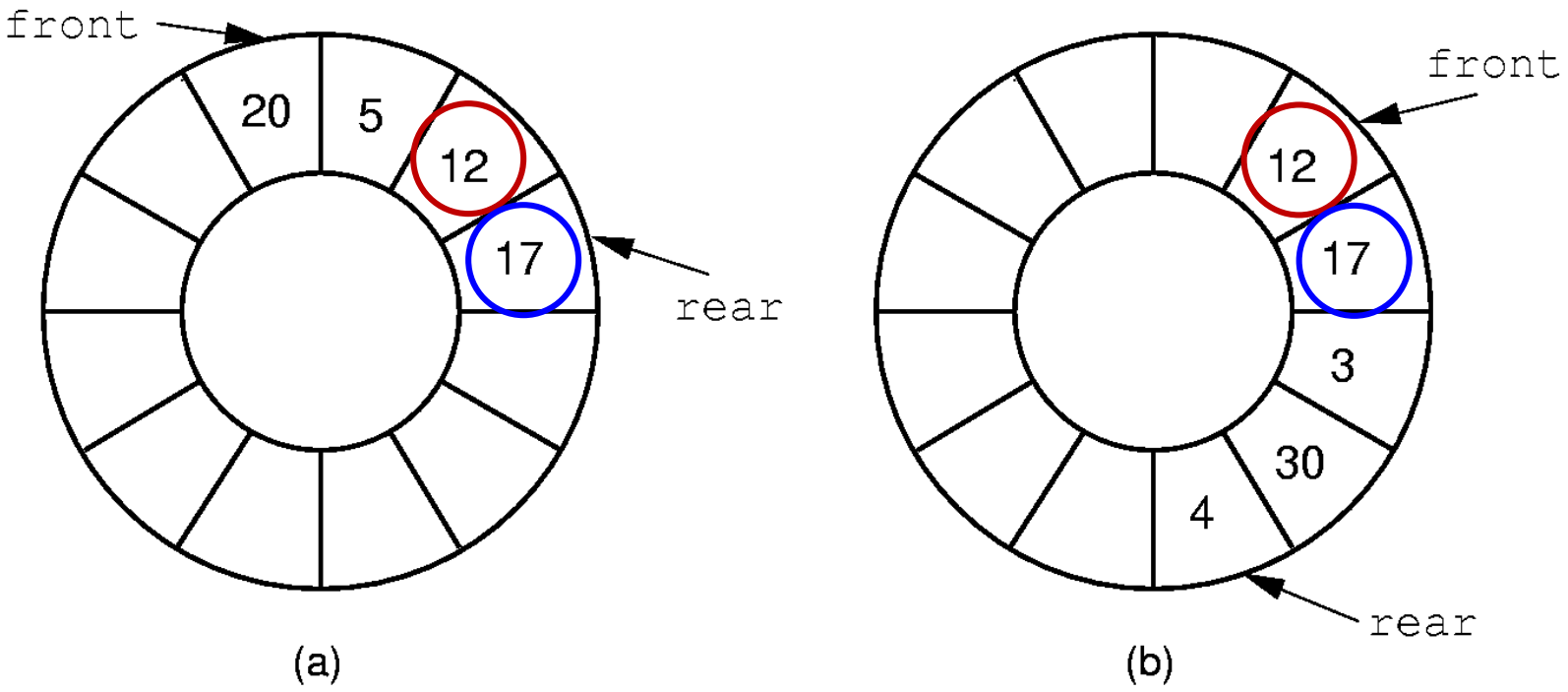


- We maintain two integers to indicate the front and the rear of the queue.

- However, as items are added and removed, the queue "drifts" toward the end.
  - Eventually, there will be no space to the right of the queue, even though there is space in the array.

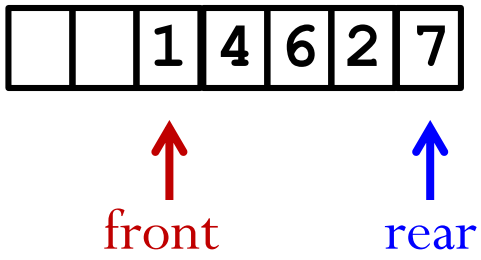
Queues Using Arrays

- To solve the problem of memory waste, we use a **circular array**.

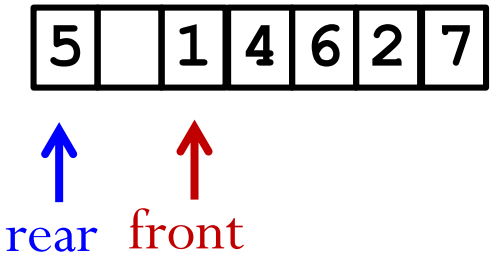


Circular Arrays

- We can implement a circular array using a plain linear array:
  - When front/rear equals the **last** index (i.e., MAXSIZE-1), increment of front/rear gives the **first** index (i.e., 0).



`enqueue(5)`



Circular Arrays

- To realize the "circular" increment, we can use modulo operation:

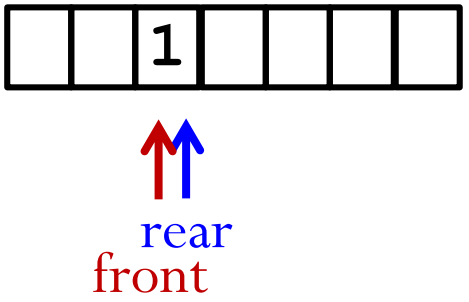
```
front = (front+1) % MAXSIZE;

rear = (rear+1) % MAXSIZE;
```

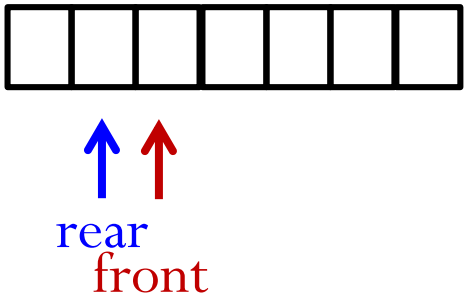
If **`front(or rear) == MAXSIZE-1`**, the statement sets **`front(or rear)`** to 0.

Boundary Conditions

- Suppose that **`front`** points to the **first** element in the queue and that **`rear`** points to the **last** element in the queue.
- What will a queue with one element look like?

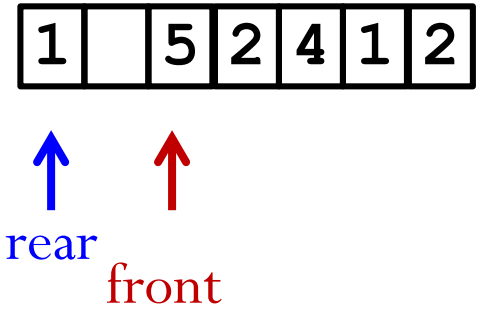


- What will an empty queue look like?

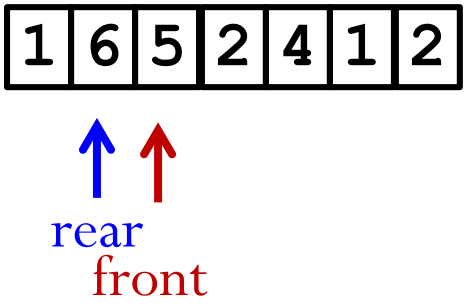


Boundary Conditions

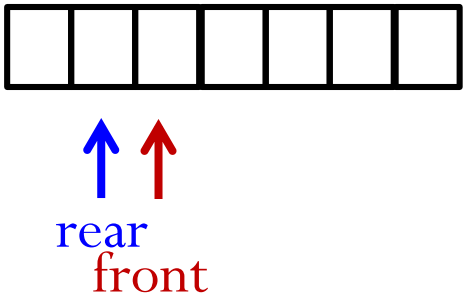
- What will a queue with one empty slot look like?



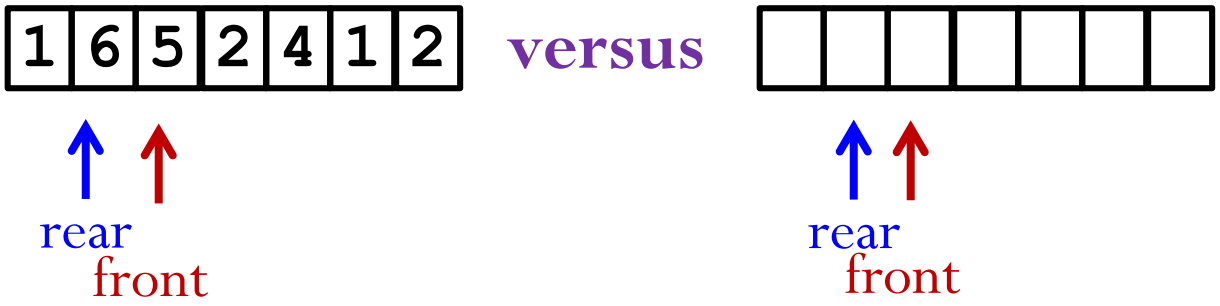
- What will a full queue look like?



versus an empty queue



Boundary Conditions



- To distinguish between the full array and the empty array, we need a flag indicating **`empty`** or **`full`**, or a **`count`** on the number of elements in the queue.

Queues Using Arrays

- **`enqueue(Object o):`** increment **`rear`**, wrapping to the beginning of the array if the end of the array is reached; if **`rear`** becomes **`front`**, reallocate arrays.

- **`dequeue():`** increment **`front`**, wrapping to the beginning of the array if the end of the array is reached; if empty, throw **`queueEmpty`**.

- **`isEmpty(): return (count == 0);`**

- **`size(): return count;`**

Outline

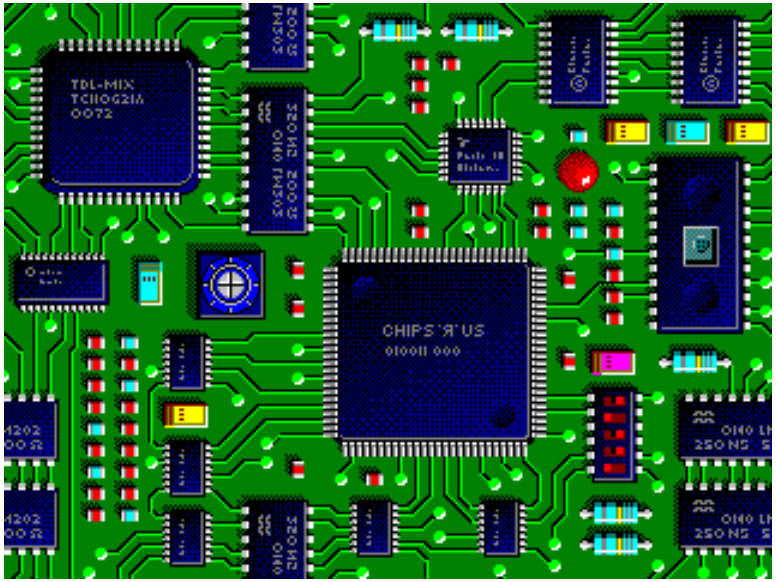
- Stack: Applications
- Queue
  - Implementation
  - Applications
  - Relative: Deque
- Synthesized Default Constructor, Copy Constructor, Assignment Operator, and Destructor

Application of Queues

- Request queue of a web server
  - Each user can send a request.
  - The arriving requests are stored in a **queue** and processed by the computer in a **first-come-first-serve** way.

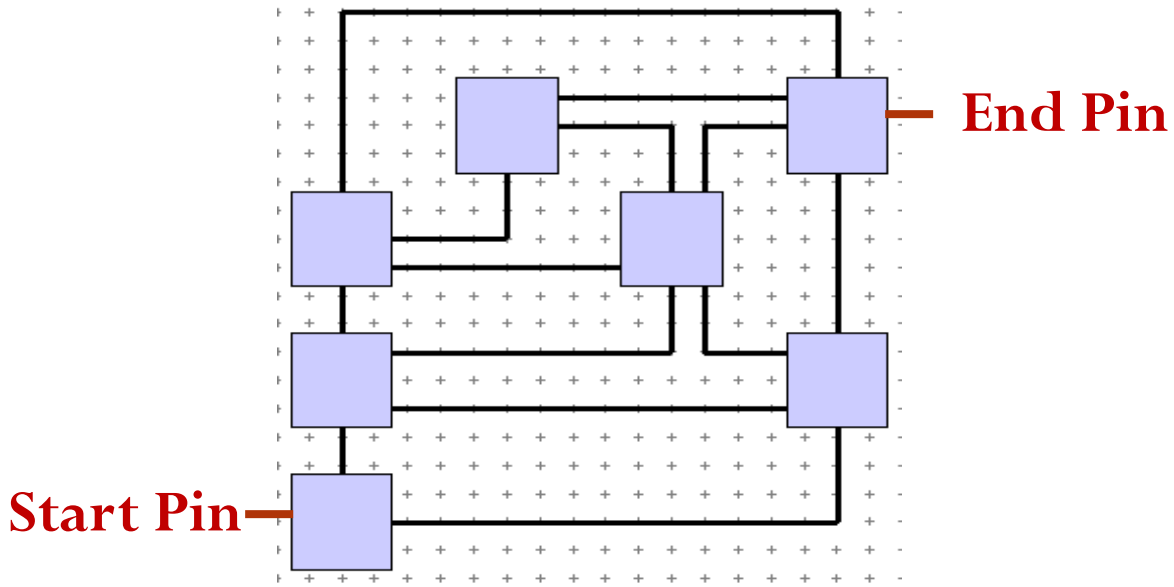
Application of Queue: Wire Routing

- Select paths to connect all pairs of pins that need to be connected together.
- An important problem in **electronic design automation**.



A Simplified Problem

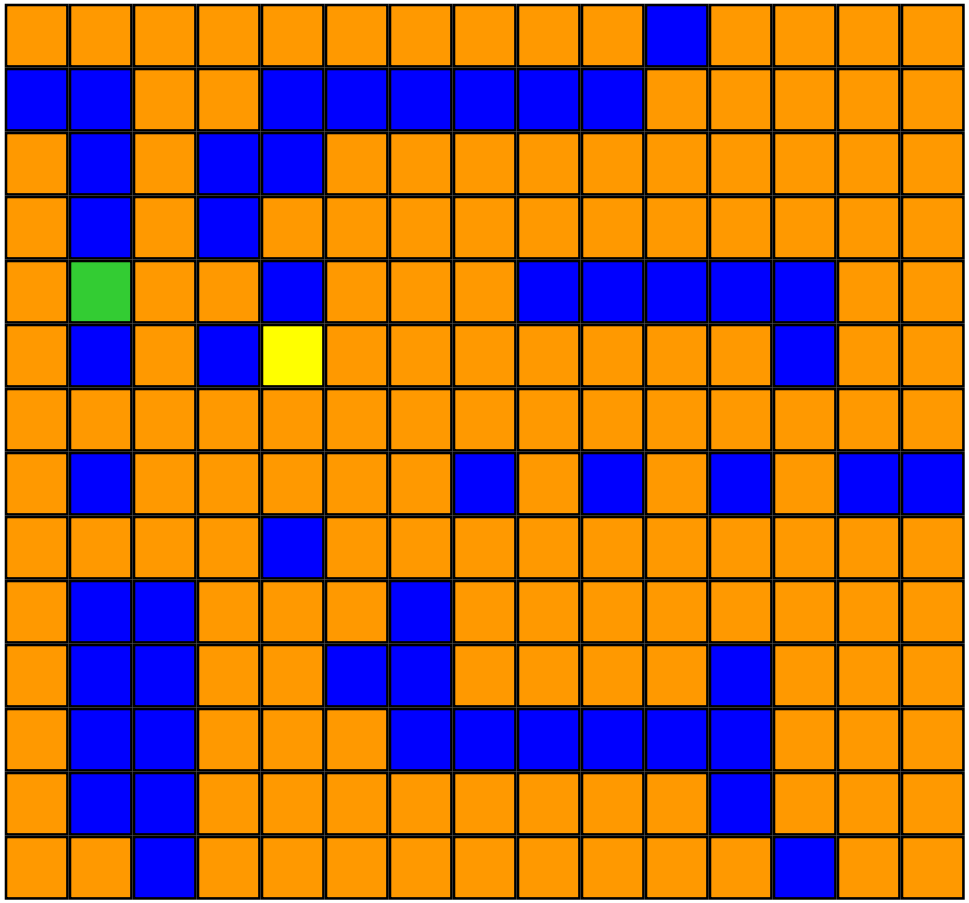
- Condition: We have all blocks laid on the chip. We also have some of the wires routed.
- Problem: We want to connect the next pair of pins.
- Constraint: we can only draw wires horizontally or vertically.



Modeling as a Grid

Start Pin

End Pin



- Blue squares are **blocked** squares.
- Orange squares are **available** to route a wire.

How to find a path from the start pin to the end pin?

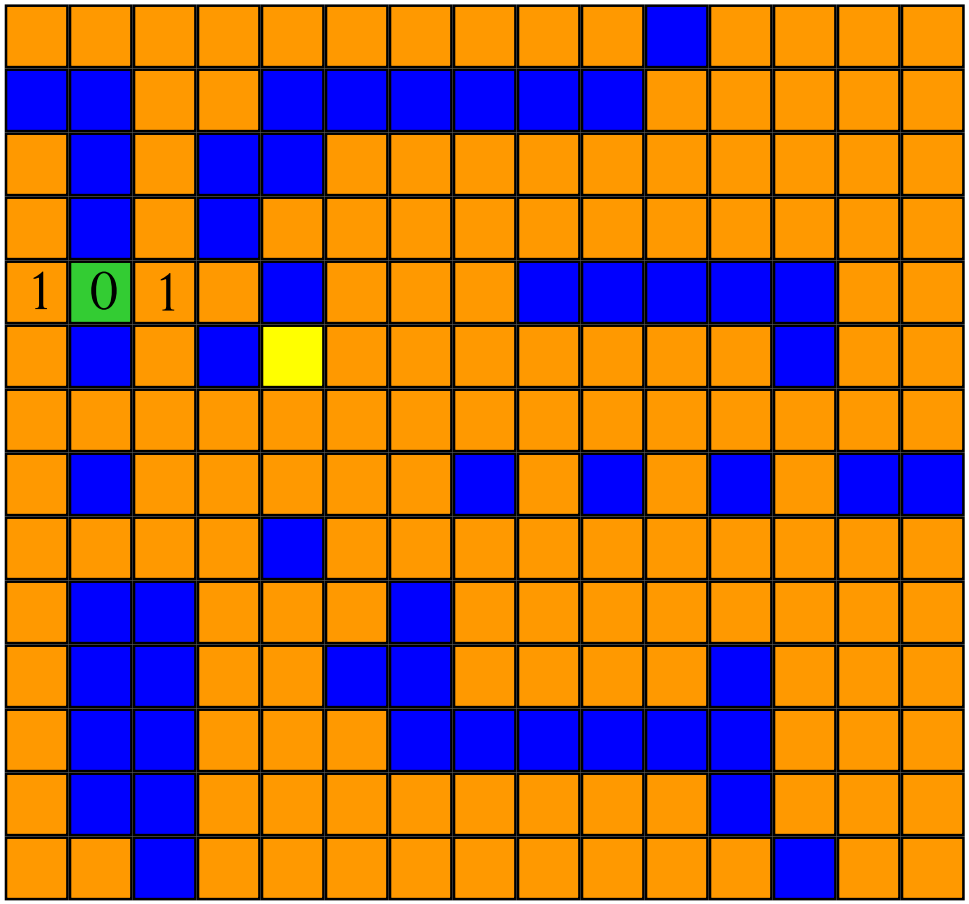
Wire Routing: Lee's Algorithm

- A **queue** of reachable squares from the start pin is used.
- The cell of the start pin is set with a distance value of 0.
- It is enqueued into an initial empty queue.
- **While** the queue is not empty.
  - A cell is **dequeued** from the queue and made the **examine cell**.
  - Is the examine cell the end pin? If yes, path found and return.
  - Otherwise, all **unreached unblocked** squares adjacent to the **examine cell** are marked with their distance (this is 1 more than the distance value of the **examine cell**) and **enqueued**.
- When queue becomes empty but not reach end pin yet, means no path found.

Illustration of Lee's Algorithm

start pin

end pin



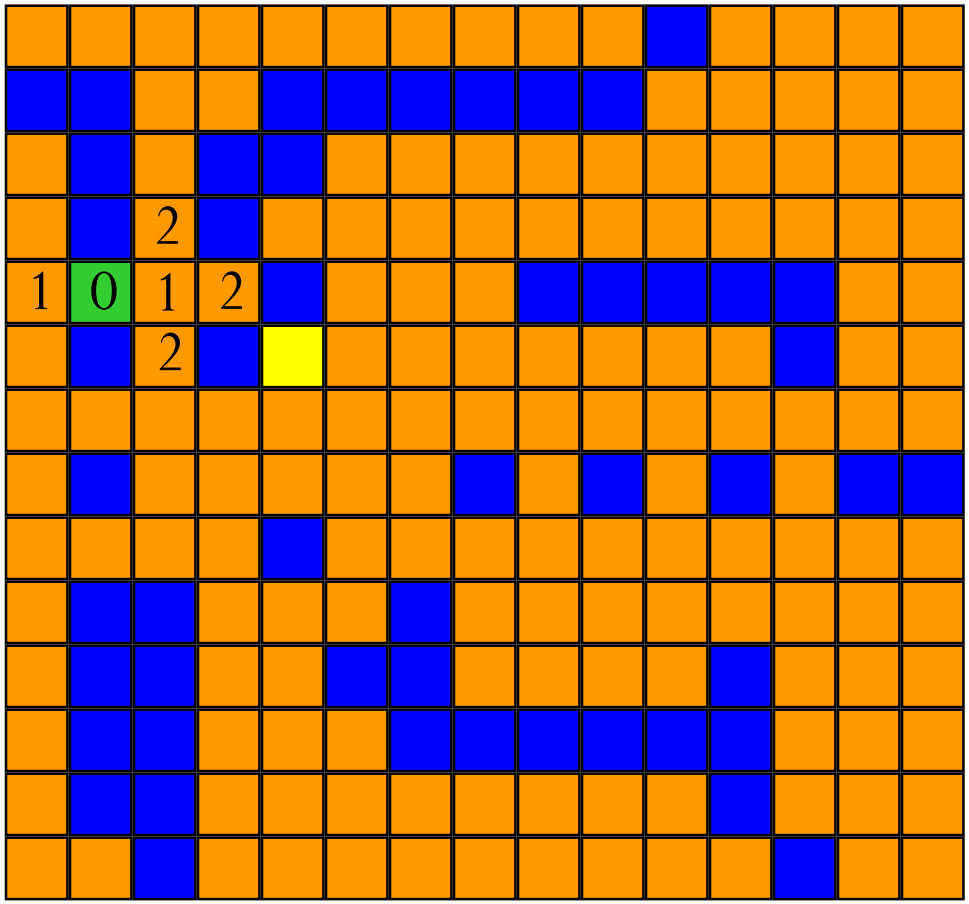
Expand "0"

queue: 0

Illustration of Lee's Algorithm

start pin

end pin



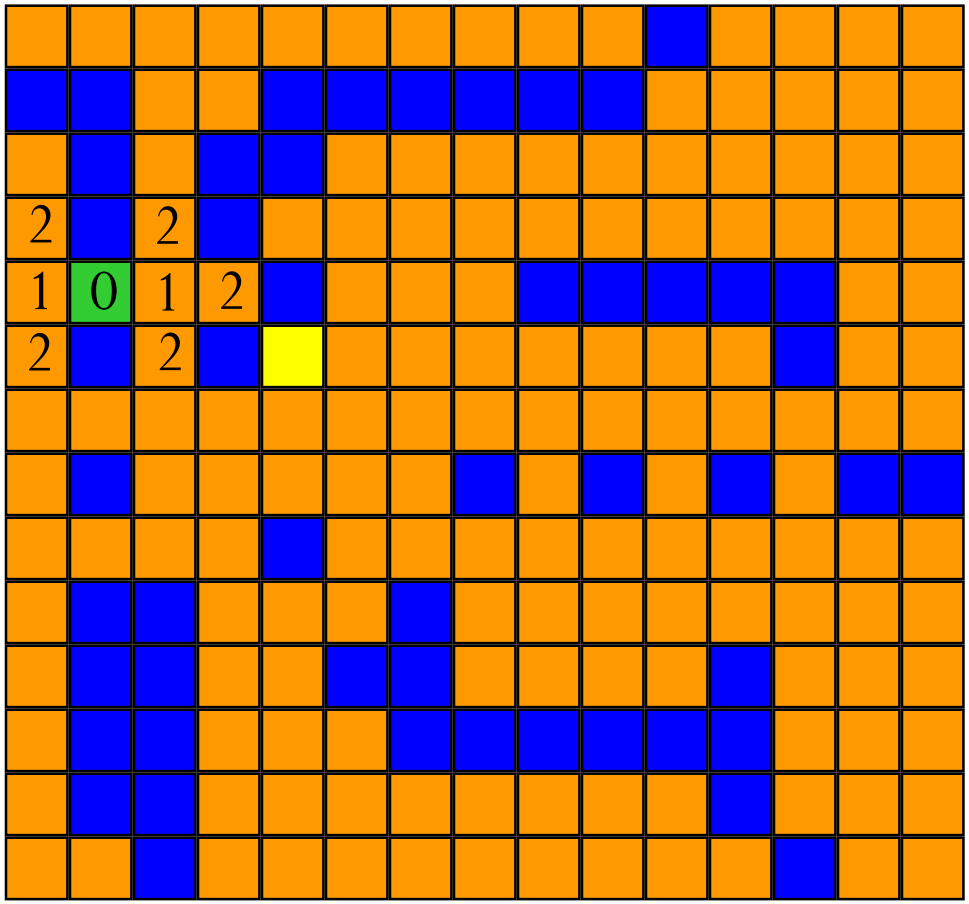
Expand right "1"

queue: 1, 1

Illustration of Lee's Algorithm

start pin

end pin



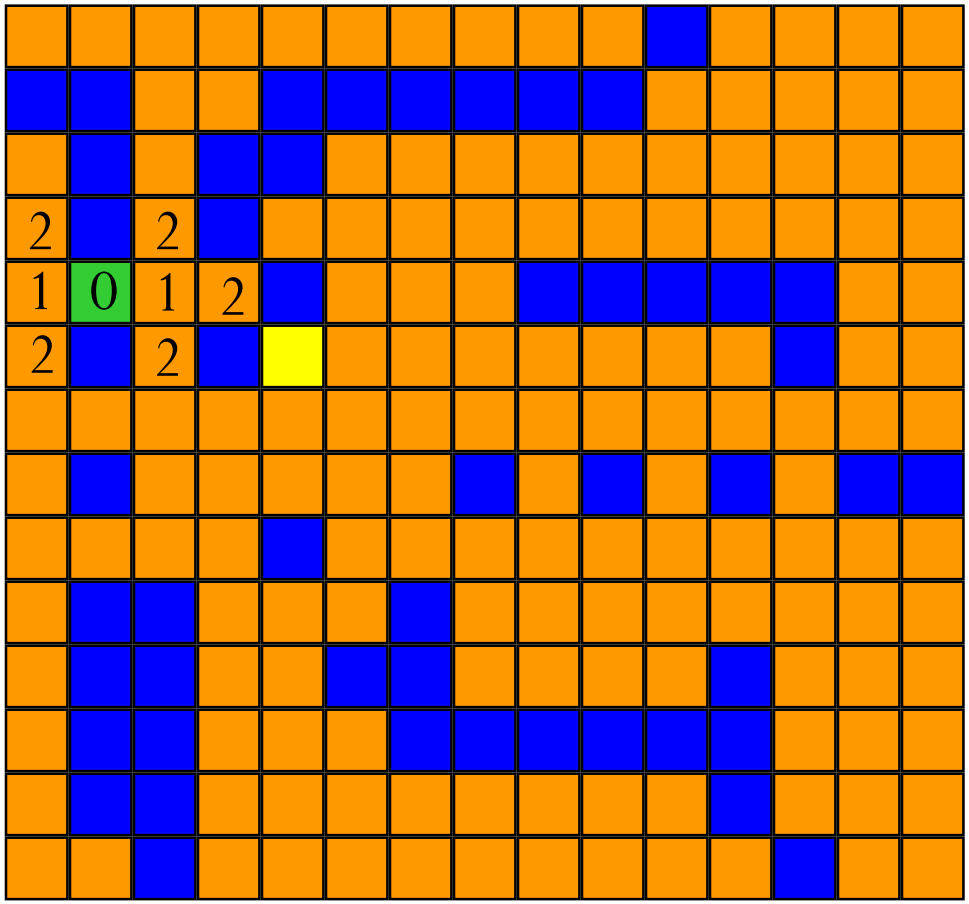
Expand left "1"

queue: 1,2,2,2

Illustration of Lee's Algorithm

start pin

end pin

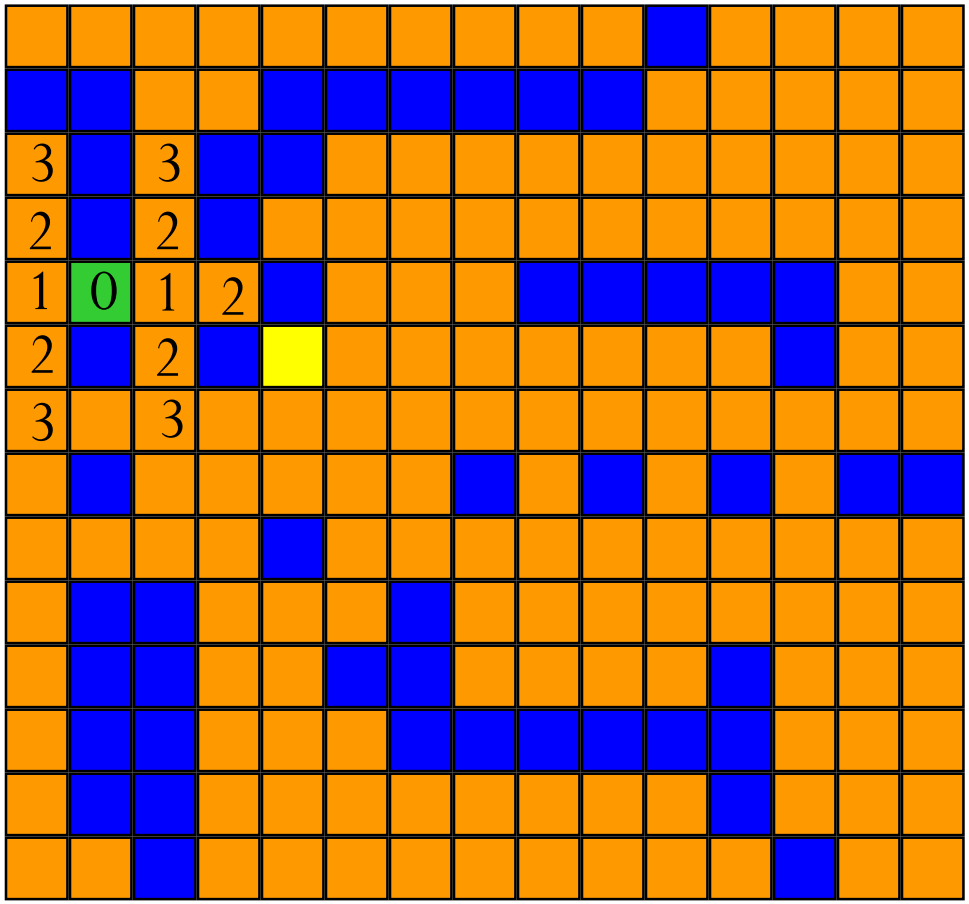


Expand and reach all squares 3 units from start.

Illustration of Lee's Algorithm

start pin

end pin

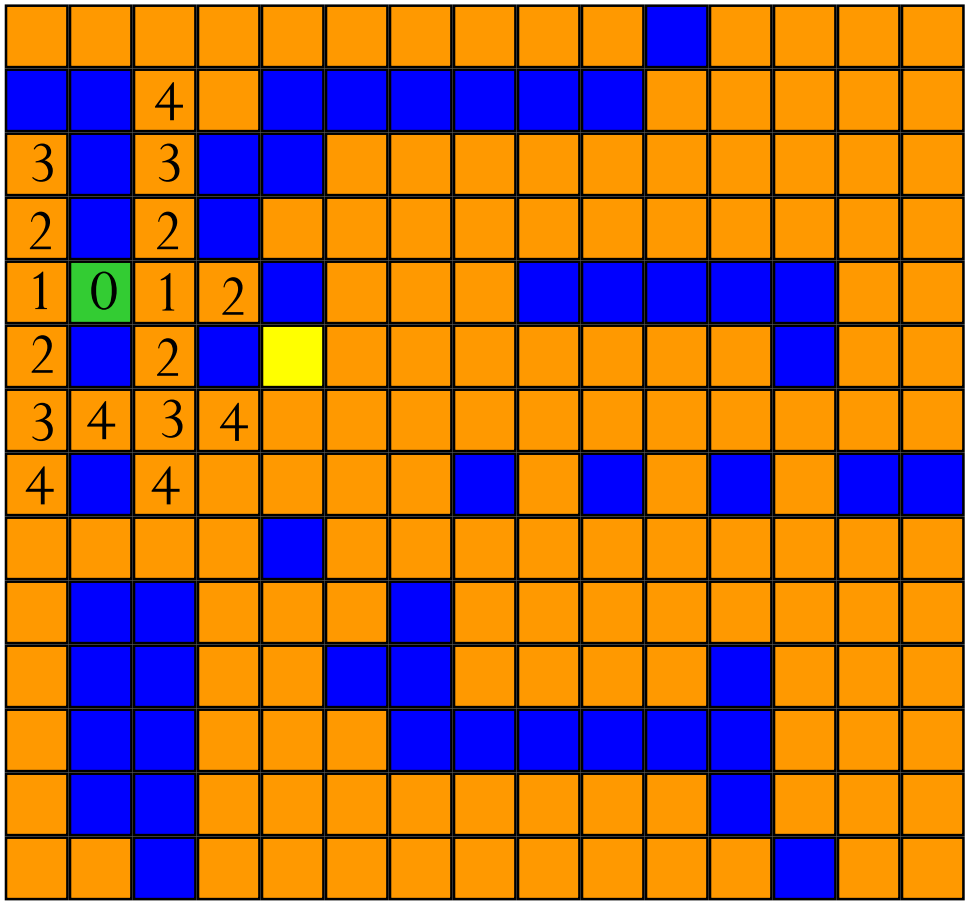


Expand and reach all squares 4 units from start.

Illustration of Lee's Algorithm

start pin

end pin

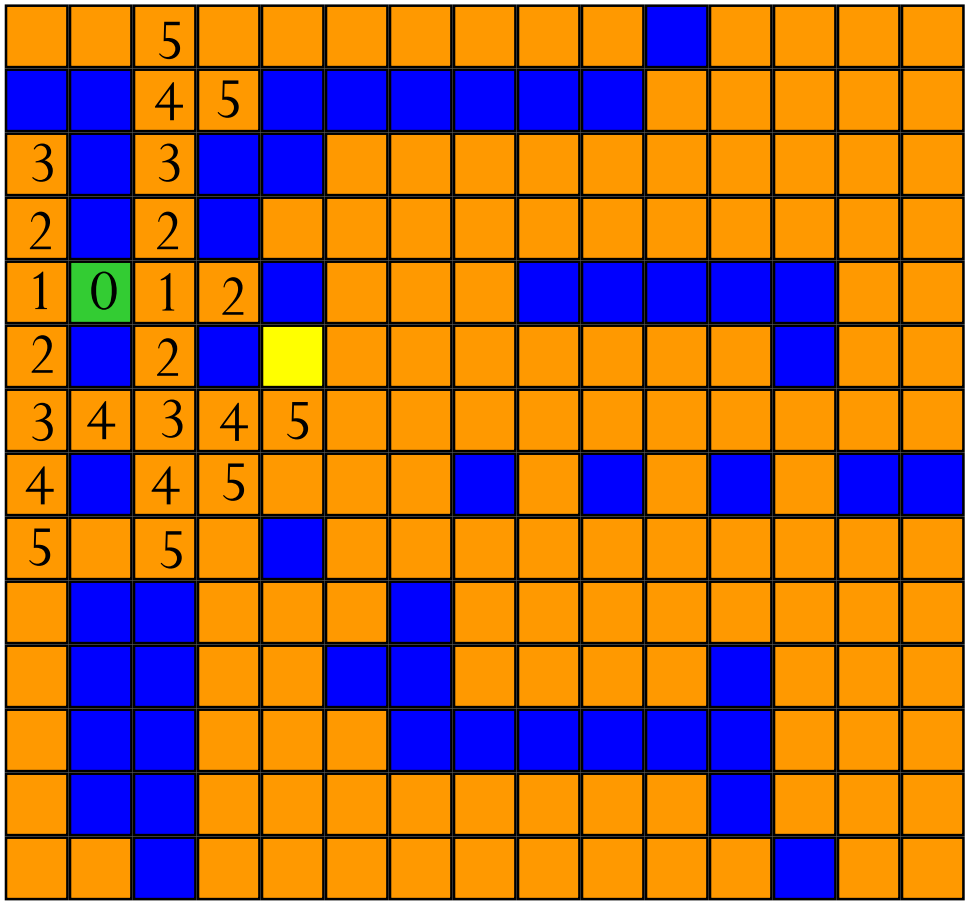


Expand and reach all squares 5 units from start.

Illustration of Lee's Algorithm

start pin

end pin

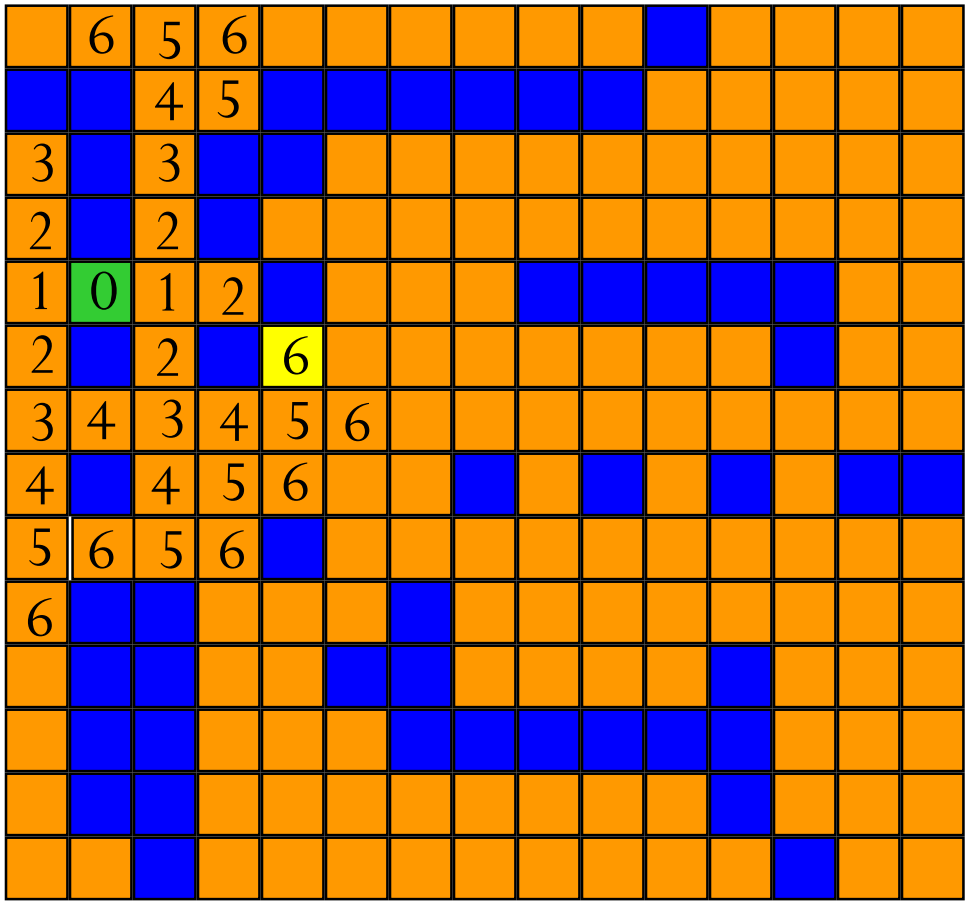


Expand and reach all squares 6 units from start.

Illustration of Lee's Algorithm

start pin

end pin

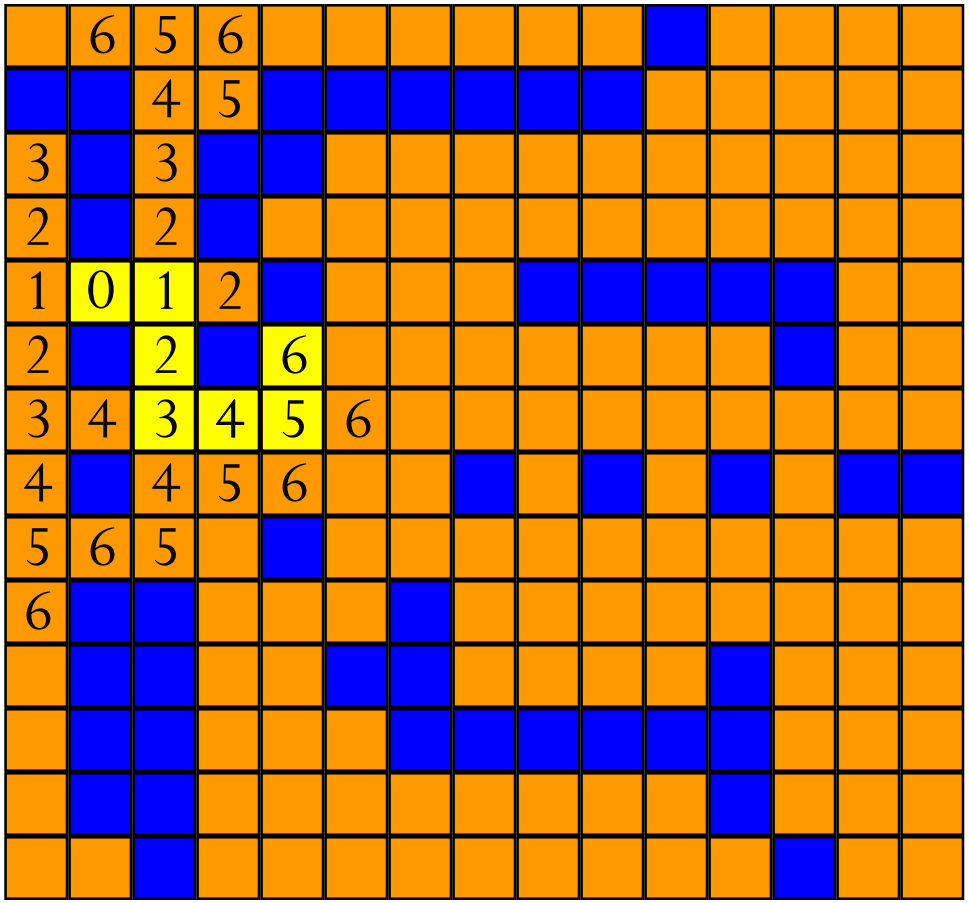


End pin reached. Trace back.

Illustration of Lee's Algorithm

start pin

end pin



Outline

- Stack: Applications
- Queue
  - Implementation
  - Applications
  - Relative: Deque
- Synthesized Default Constructor, Copy Constructor, Assignment Operator, and Destructor

Deque

- Not a proper English word, pronounced as “deck”.

- A combination of stack and queue.
  - Items can be inserted and removed from **both ends** of the list.

- Methods supported:
  - `push_front(Object o)`
  - `push_back(Object o)`
  - `pop_front()`
  - `pop_back()`

Deque Implementation

- Linked list
  - Which type of linked list will you choose to support fast insertion and removal?
  - Double-ended doubly-linked list

- Circular array
  - front and rear not only need to be incremented (**`push_back, pop_front`**), but also need to be decremented (**`push_front, pop_back`**).

Outline

- Stack: Applications
- Queue
  - Implementation
  - Applications
  - Relative: Deque
- Synthesized Default Constructor, Copy Constructor, Assignment Operator, and Destructor

Synthesized Default Constructor

- If we do not explicitly define any constructors, what will happen?
  - Answer: The compiler will automatically synthesize a default constructor, i.e., a constructor that takes no argument

```
class foo {
  int i;
  string s;
public:
  int get_int();
  string get_str();
};
```

Will synthesize a default constructor `foo::foo()`

```
foo f;
```

Call the synthesized default constructor

Synthesized Default Constructor

- Members of built-in types (int, double, etc.) are uninitialized
- Members of class types are initialized using their default constructor

```
class foo {
  int i;
  string s;
public:
  int get_int();
  string get_str();
};
```

```
foo f;
```

f.i is uninitialized
f.s is an empty string

- A good practice is to define a default constructor ourselves

Synthesized Default Constructor

- If a class defines at least one constructor, then the compiler will not generate the default constructor.

```
class foo {
  int i;
public:
  foo(int _i): i(_i) {}
};
```

```
foo f;
foo *fp = new foo;
// Both wrong: no
// default construct
// synthesized
```

Note: define a constructor taking one argument

- In practice, if other constructors are being defined, it is almost always right to provide a default constructor

Synthesized Copy Constructor

- If we don't define a copy constructor explicitly, the compiler synthesizes one for us
- Unlike the synthesized default constructor, a copy constructor is synthesized even if we define other constructors
- What's the behavior of the synthesized copy constructor?
  - Answer: Member-wise copy
  - For a built-in/pointer type, it directly copies the value
  - For a class type, it calls the copy constructor

This causes **shallow copy** when passing class object as function argument as we discussed before

Synthesized Assignment Operator

- If we don't define an overloaded assignment operator explicitly, the compiler also synthesizes one for us

- The behavior is same as synthesized copy constructor
  - For a built-in/pointer type, it directly copies the value
  - For a class type, it calls the copy constructor

Synthesized Destructor

- Surprising result: there is always a synthesized destructor even if you explicitly write one
- Effect: destroys the members in **reverse order** from which they are declared in the class
  - For class member, call the member's destructor
  - For built-in or pointer type, just destroy the member. In particular, does not delete the object pointed to by a pointer member
- What happens when we write an explicit destructor `~foo() { ... }`?
  - First, do everything in `~foo() { ... }`
  - Then, call the synthesized destructor to destroy members