ADTs Stack and Queue

Data Structures
Fall 2022

ADT Stack

Stacks of Coins and Plates

Stacks of Rocks and Books

TOP OF THE STACK

TOP OF THE STACK

Add, remove rock and book from the top, or else…

Stack at logical level

- A stack is an ADT in which elements are added and removed from only one end (i.e.,at the top of the stack).

- A stack is a LIFO “last in, first out” structure.

Stack at Logical Level

- What operations would be appropriate for a stack?

Stack Operations

Transformers

- Push
- Pop

change state

Observers

- Top
- IsEmpty
- IsFull

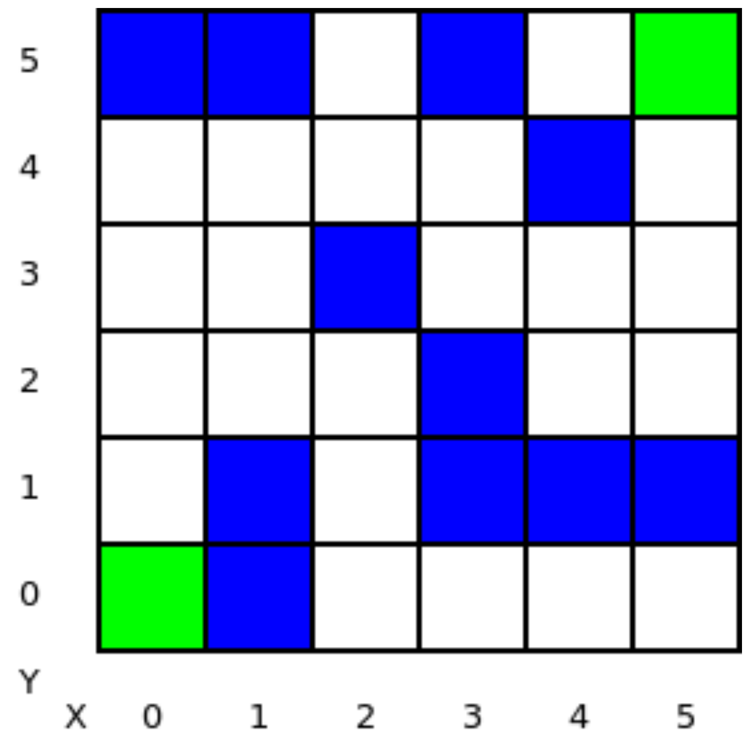
observe state

Stack at Application Level

- For what types of problems would stack be useful for?
- LIFO: good for reversing data
    - If we push a, b, c, d into a stack, and then pop all elements out, we get d, c, b, a
- In OS/language: function call stack
- Finding Palindromes
- Expression evaluation and Syntax Parsing

Use stack to backtrack

- In maze-walking algorithm, we can use stack to store all nodes that led us to current node, so that we can backtrack when needed.
- Goal: find a path via white blocks from (0,0) to (5,5)
- Need to explore: try diff. next step
- and backtrack (when reach dead-end): i.e., reverse back to previous node



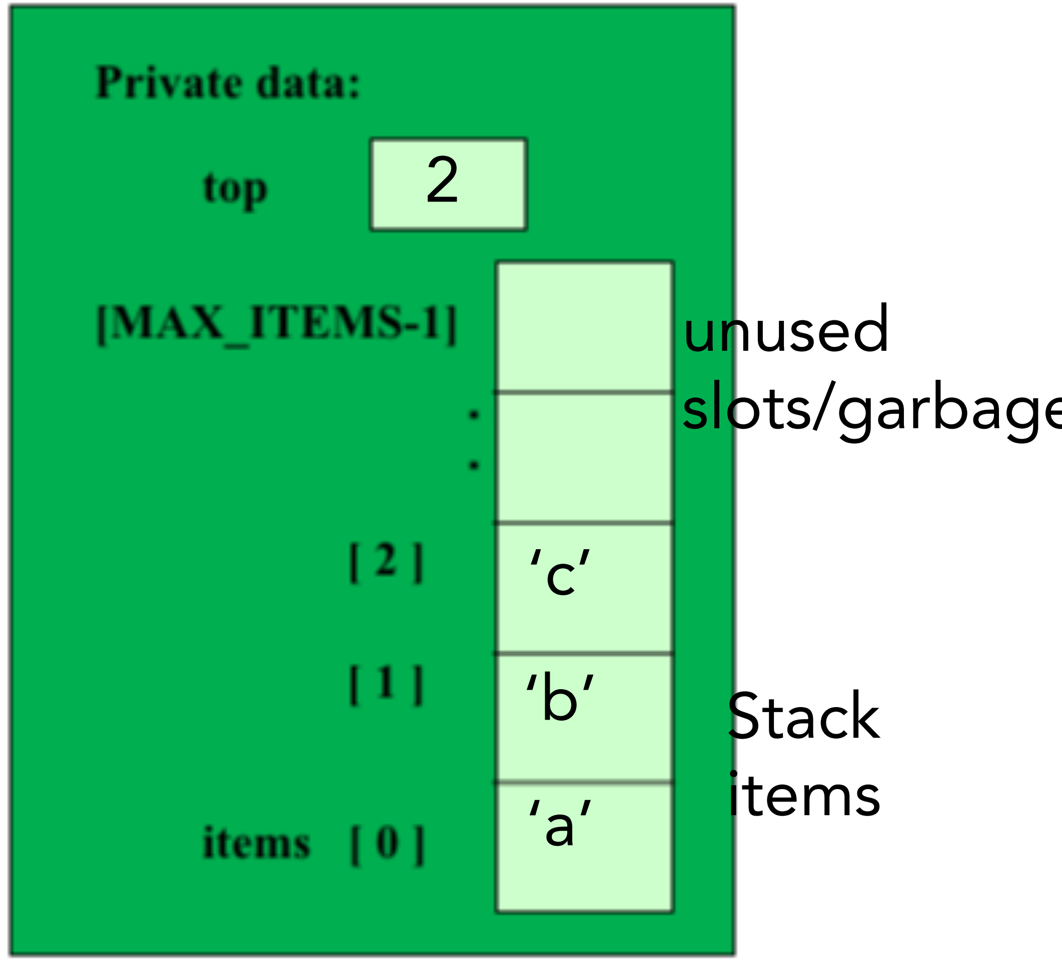
Stack of ItemTypes

```
class StackType
{
public:

  StackType( );

  bool IsFull () const;

  bool IsEmpty() const;

  void Push( ItemType item );

  void Pop();

  ItemType Top() const;



};
```

Stack Implementation

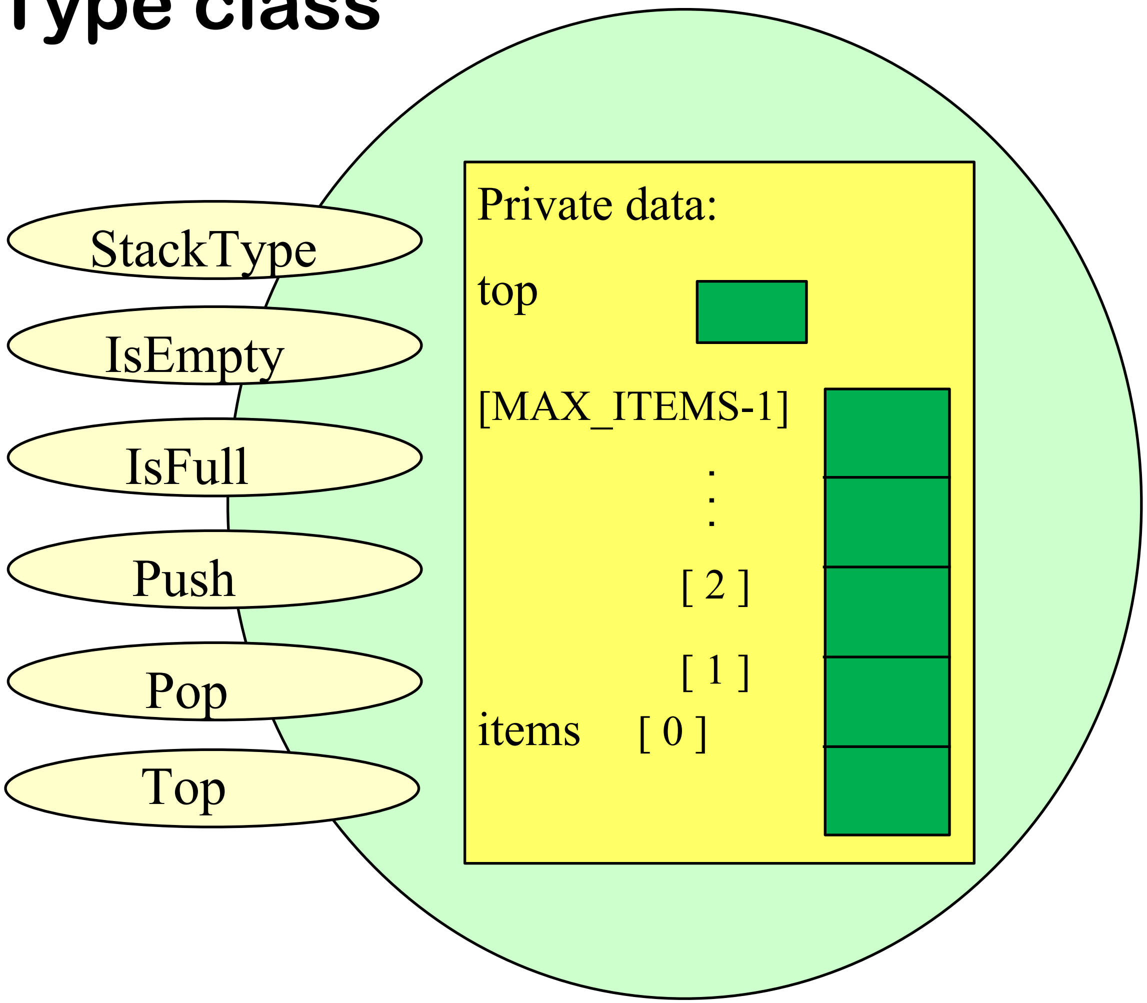
- array-based implementation: static or dynamic array
- linked-structure implementation

```
class StackType
{
public:

  StackType( );

  bool IsFull () const;

  bool IsEmpty() const;

  void Push( ItemType item );

  void Pop();

  ItemType Top();

private:

   int top;
   ItemType  items[MAX_ITEMS];
};
```



Class Interface Diagram

StackType class



Initialize stack

How to initialize data member top?

0 or -1

Depends on what top stores:

*** initialized to 0: If it's the next open slot**
*** initialized to -1: if it's the index of stack top element**

Need to be consistent in all member functions, Top(), Push(), Pop(), isfull, isEmpty()…

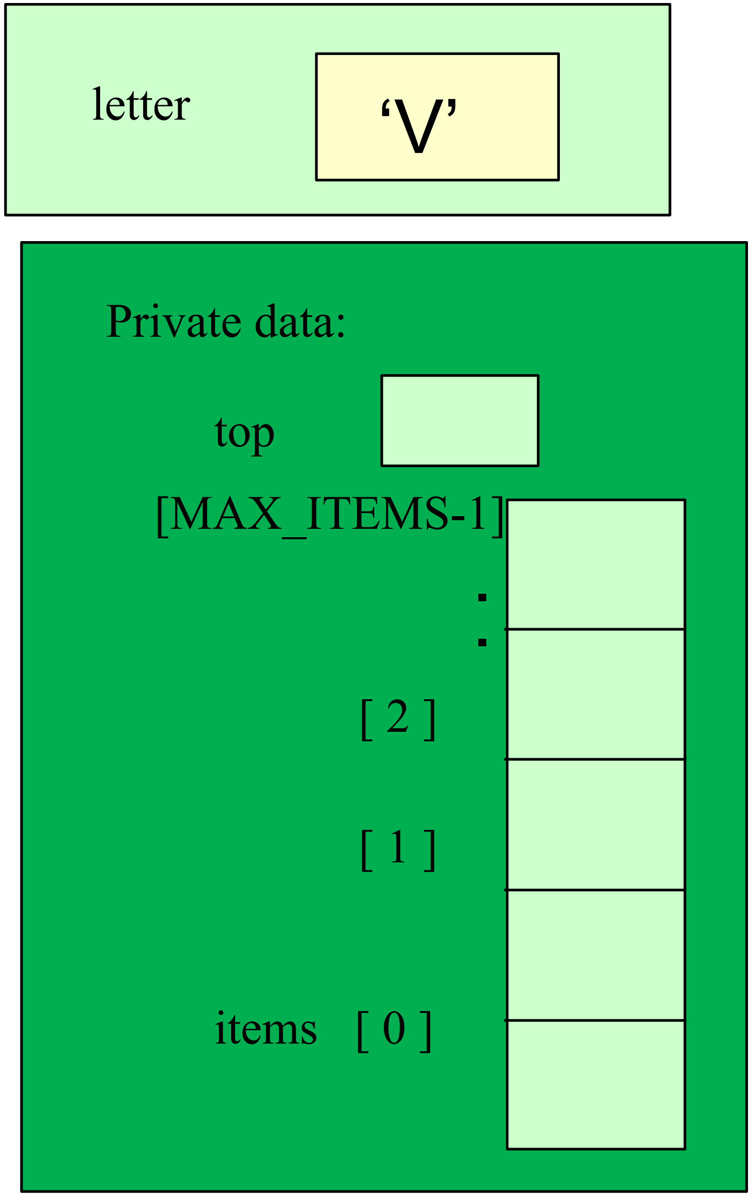
Below we use second option:

```
StackType::StackType( )
{
  top = -1; //index of top element in stack
}
```

```
// pre: the stack has been initialized
// post: return true if the stack is empty, false ow
bool StackType::IsEmpty() const
{
  return(top == -1);
}


//pre:
//post:
bool StackType::IsFull() const
{
  return (top = = MAX_ITEMS-1);
}
```

```
void StackType::Push(ItemType newItem)
{
   if( IsFull() )
      throw FullStack():
   top++;
   items[top] = newItem;
}


void StackType::Pop()
{
   if( IsEmpty() )
     throw EmptyStack();
    top--;
}


ItemType StackType::Top()
{
   if (IsEmpty())
     throw EmptyStack();
    return items[top];
}
```

Tracing Client Code




```
char        letter = 'V';
StackType  charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ))
      charStack.Pop( );
charStack.Push('K');
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop()}
```

Stack Implementation: linked structure

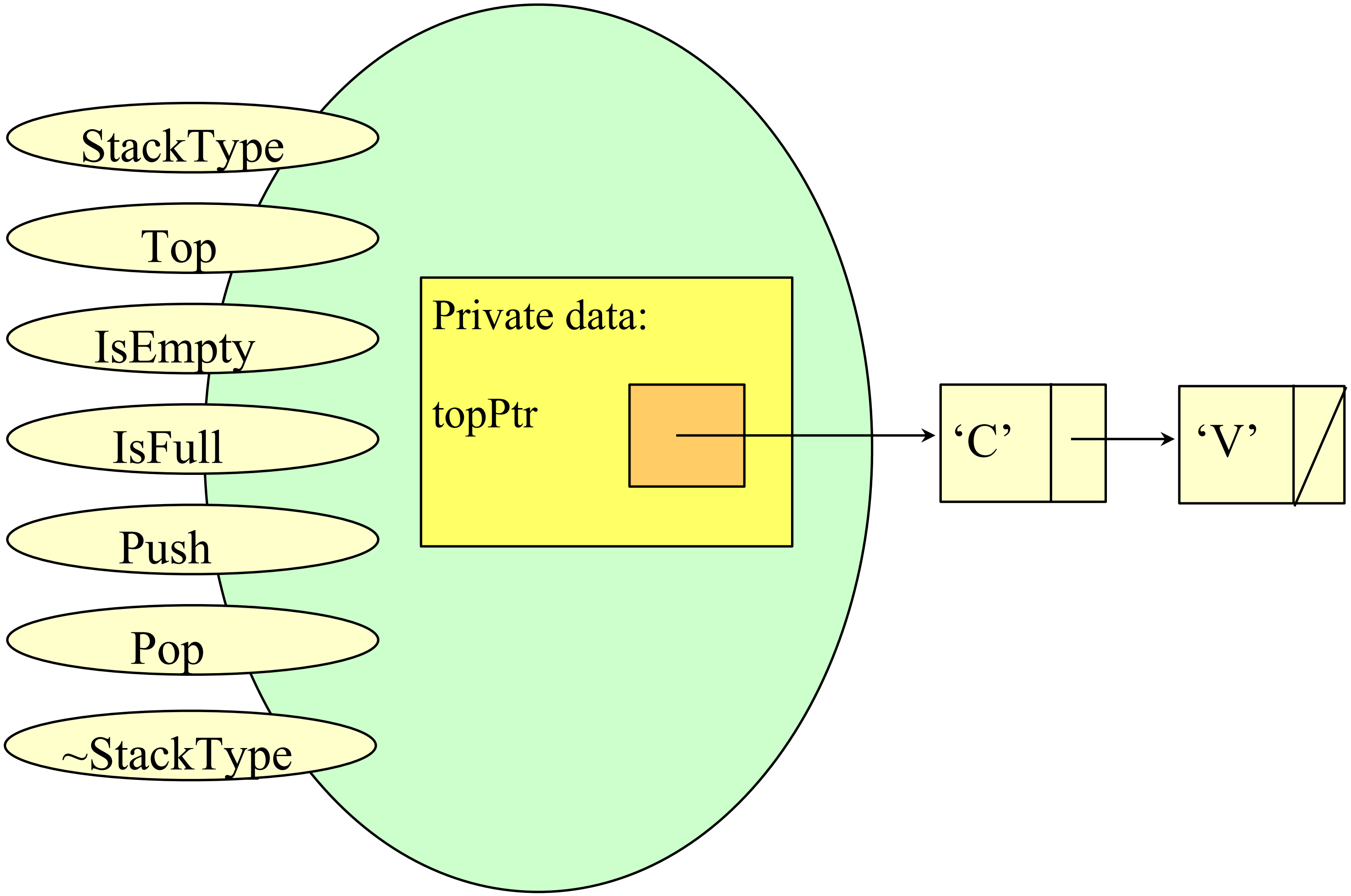
- One advantage of an ADT is that the implementation can be changed without the program using it knowing about it.

- in-object array implementation: has a fixed max. size
- dynamically allocated array (as in lab2?): can be grown when needed, but lots of copy!

- Linked structure: dynamically allocate the space for each element as it is pushed onto stack.

ItemType is char

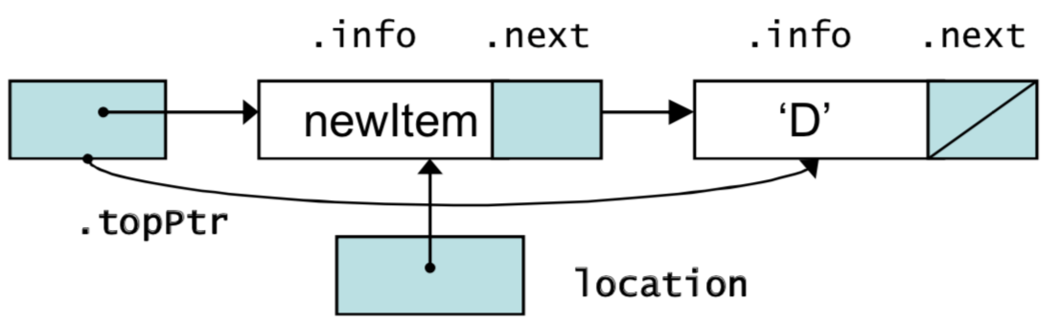
class StackType



```
// DYNAMICALLY LINKED IMPLEMENTATION OF STACK
struct NodeType;                    //Forward declaration

class StackType
{
public:
//Identical to previous implementation
/* Add: copy-constructor, destructor, assignment
   operator: since dynamic memory is used! */
private:
  NodeType* topPtr;
};
.
.
.
struct NodeType
{
   ItemType  info;
   NodeType* next;
};
```

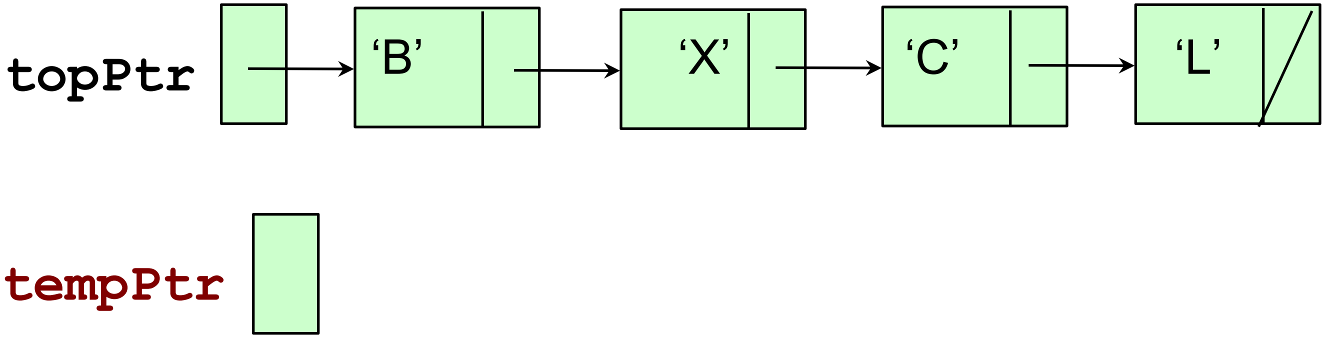
Implementing Push

```
void StackType::Push ( ItemType newItem )
  // Adds newItem to the top of the stack.
{
   NodeType*  location;
   location = new  NodeType;
   location->info = newItem;
   location->next = topPtr;
   topPtr = location;
}
```

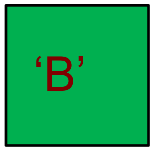


Deleting top element from the stack

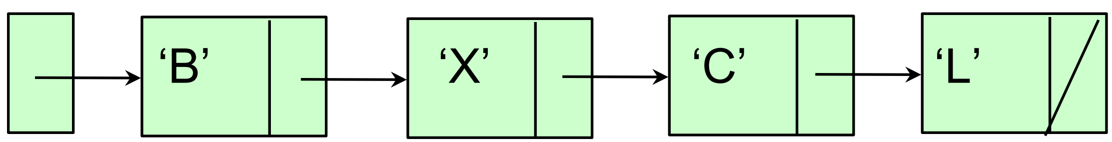
```
item

    NodeType*  tempPtr;

    item = topPtr->info;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete  tempPtr;
```

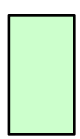


Deleting top element from the stack

item 

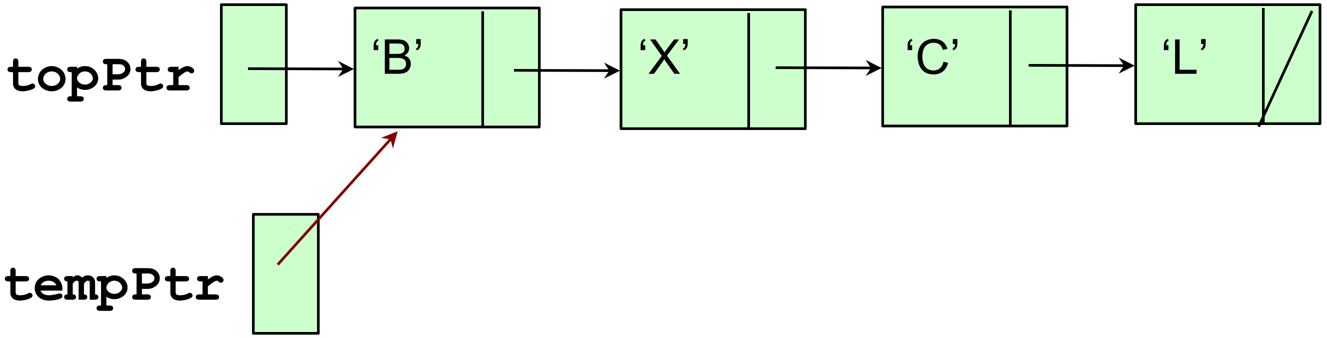

```
NodeType*  tempPtr;

item = topPtr->info;
tempPtr = topPtr;
topPtr = topPtr->next;
delete  tempPtr;
```

topPtr 

tempPtr 

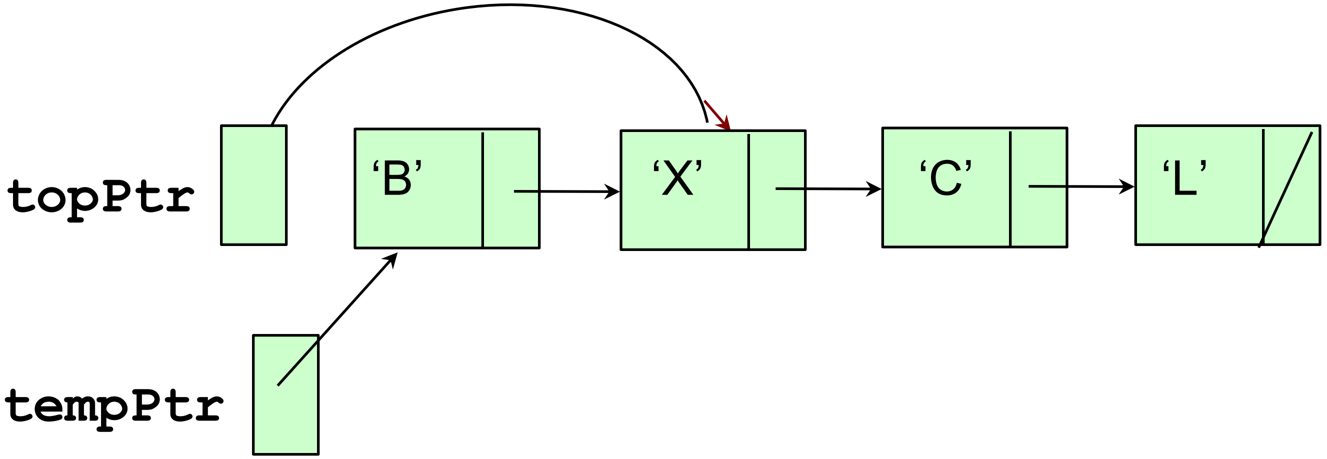
Deleting item from the stack


```
NodeType*  tempPtr;

item = topPtr->info;
tempPtr = topPtr;
topPtr = topPtr->next;
delete  tempPtr;
```



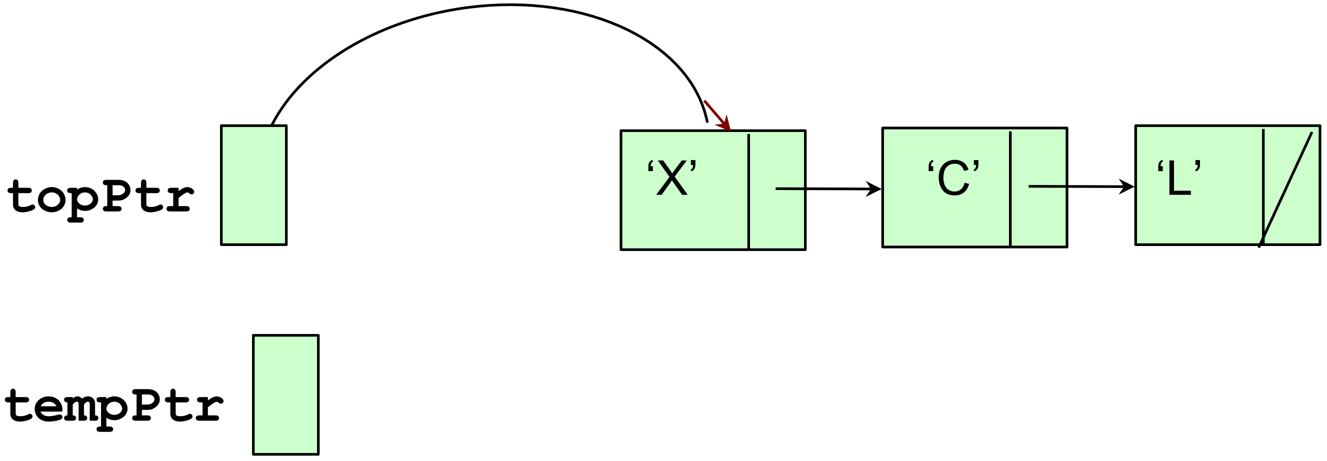
Deleting item from the stack


```
NodeType*  tempPtr;

item = topPtr->info;
tempPtr = topPtr;
topPtr = topPtr->next;
delete  tempPtr;
```



Deleting item from the stack


```
NodeType<ItemType>*  tempPtr;

item = topPtr->info;
tempPtr = topPtr;
topPtr = topPtr->next;
delete  tempPtr;
```



Implementing Pop

```
void StackType::Pop()
// Remove top item from Stack.
{
   if (IsEmpty())
     throw EmptyStack();
   else
   {
     NodeType* tempPtr;
      tempPtr = topPtr;
     topPtr = topPtr ->next;
     delete tempPtr;
   }
}
```

Implementing Top

```
ItemType StackType::Top()
// Returns a copy of the top item in the stack.
{
  if (IsEmpty())
      throw EmptyStack();
   else
      return topPtr->info;
}
```

Implementing IsFull

```
bool StackType::IsFull()  const
// Returns true if there is no room for another
// ItemType on the free store; false otherwise
{
   NodeType* location;
   try
   {
      location = new NodeType;
      delete location;
      return false;
   }
   catch(std::bad_alloc exception)
   {
      return true;
   }
}
```

Array vs Linked Structure

- Drawback of array-based implementation:
  - at any point of time, array is either filled or not
  - memory is either not enough or wasted
- Linked Structure: allocate on demand
  - Drawback: need to store lots of addresses (in pointer field of node)
  - if ItemType is small compared to pointer, then it’s not memory efficient

Efficiency comparison

Time	Array-Based Implementation	Linked Implementation
Class constructor Class destructor	O(1) O(1)	O(1) O(n)
IsFull() IsEmpty()	O(1) O(1)	O(1) O(1)
Push() Pop()	O(1) O(1)	O(1) O(1)

C++ Standard Template Library

- a set of C++ class templates that provides common programming data structures and functions
  - lists, stacks, queues, hash table, many more…
- all data structures/container are implemented as class template, which can be parameterized:
  - vector<int>, vector<double> …
  - stack<int>, stack<char>
  - the type in <> is type parameter, specifying the type of items that the stack stores…

Sample code using STL stack

```
// CPP program to demonstrate working of STL stack
#include <iostream>
#include <stack>
using namespace std;

void showstack(stack <int> s)
{
    while (!s.empty())
    {
        cout << '\t' << s.top();
        s.pop();
    }
    cout << '\n';
}
```

Sample code using STL stack

```
int main ()
{
    stack <int> s;
    s.push(10);
    s.push(30);
    s.push(20);
    s.push(5);
    s.push(1);

    cout << "The stack is : ";
    showstack(s);

    cout << "\ns.size() : " << s.size();
    cout << "\ns.top() : " << s.top();


    cout << "\ns.pop() : ";
    s.pop();
    showstack(s);

    return 0;
}
```

ADT Queue

Queues

Queue at logical level

- A queue is an ADT in which elements are added to the rear and removed from the front

- A queue is FIFO (FCFS) “First in, first out” structure.

Queue at Logical Level

- What operations would be appropriate for a queue?

Queue Operations

Transformers

- MakeEmpty
- Enqueue
- Dequeue

change state

Observers

- IsEmpty
- IsFull

observe state

Queue at Application Level

- For what types of problems would queue be useful for?
- various servers that serve requests in First Come First Serve order:
    - printer server (a queue of print jobs),
    - disk driver (a queue of disk input/output requests)
    - CPU scheduler (a queue of processes waiting to be executed)

Queue: Logical level

```
class QueueType
{
public:
  QueueType(int max);
  QueueType();
  ~QueueType();
  bool IsEmpty() const;
  bool IsFull() const;
  void Enqueue(ItemType item);
          //add newItem to the rear of the queue.
  void Dequeue(ItemType& item);
              //remove front item from queue
```

Logical Level

Array-based Implementation

```
private:
  ItemType* items; // Dynamically allocated array
  int maxQue;
  // Whatever else we need
};
```

Implementation level

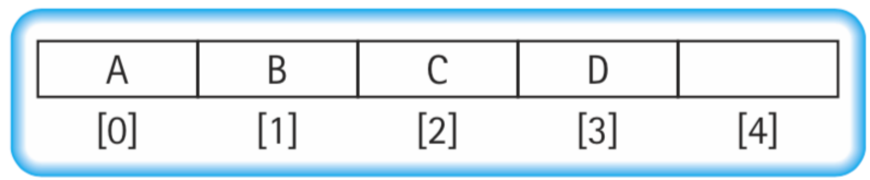
Array-based Implementation: Fixed-Front Queue

- Array-based implementation where index 0 is always the front of the queue
- Enqueue fills in the first empty slot
- Dequeue empties the first slot and moves all subsequent elements up
- Copying elements like this is inefficient

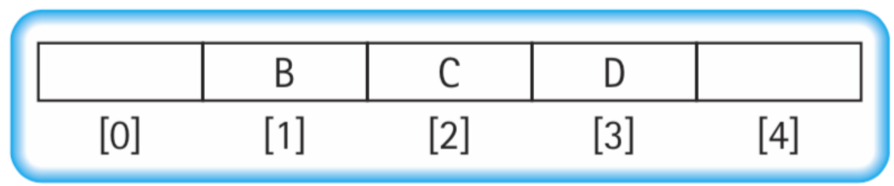
Array-based Implementation: Fixed-Front Queue

- An array with the front queue always in the first position

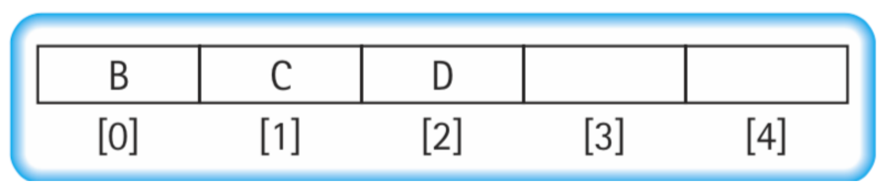
Enqueue A, B, C, D:



Dequeue:



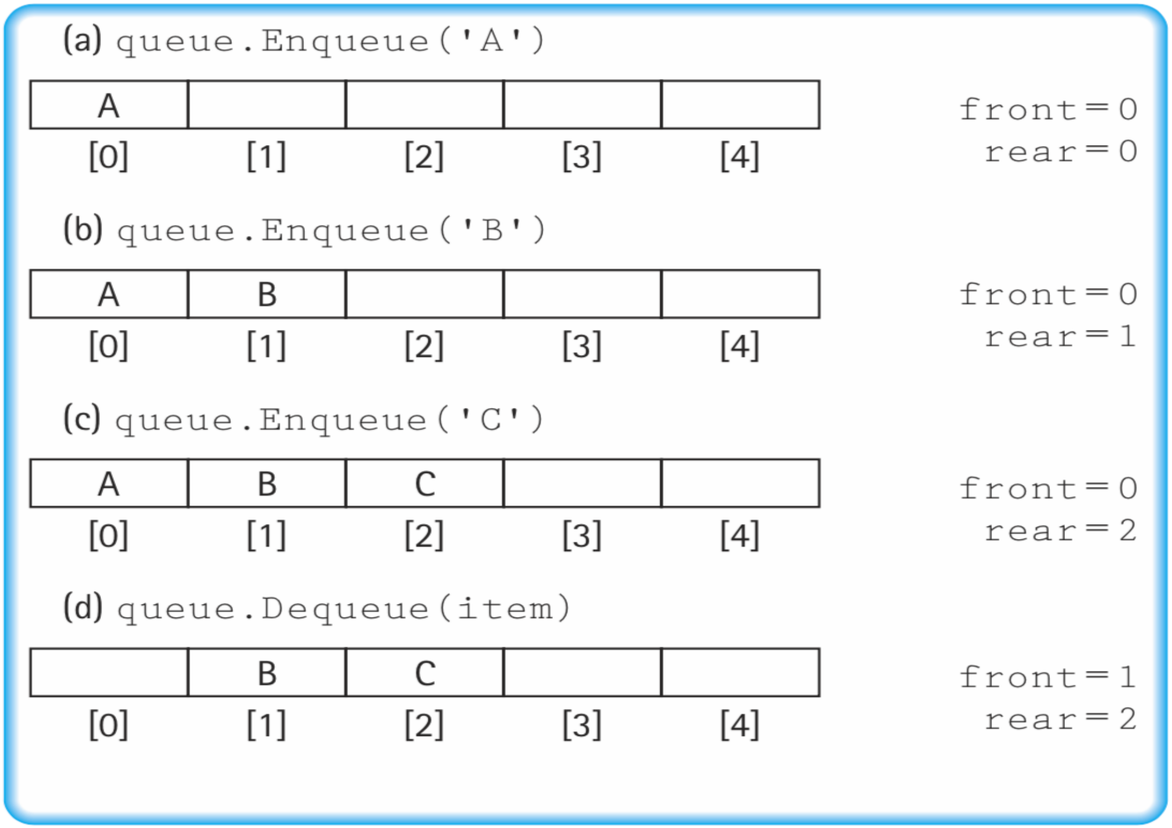
need to shift all items



Dequeue() is inefficient: it takes time linear to queue length to move all elements forward

Array-based Implementation: Floating Queue

- Both the front and end of the queue float in the array



What if we enqueue X, Y and Z?

Array-based Implementation: Floating Queue

- An array with the front **floats, circular array**

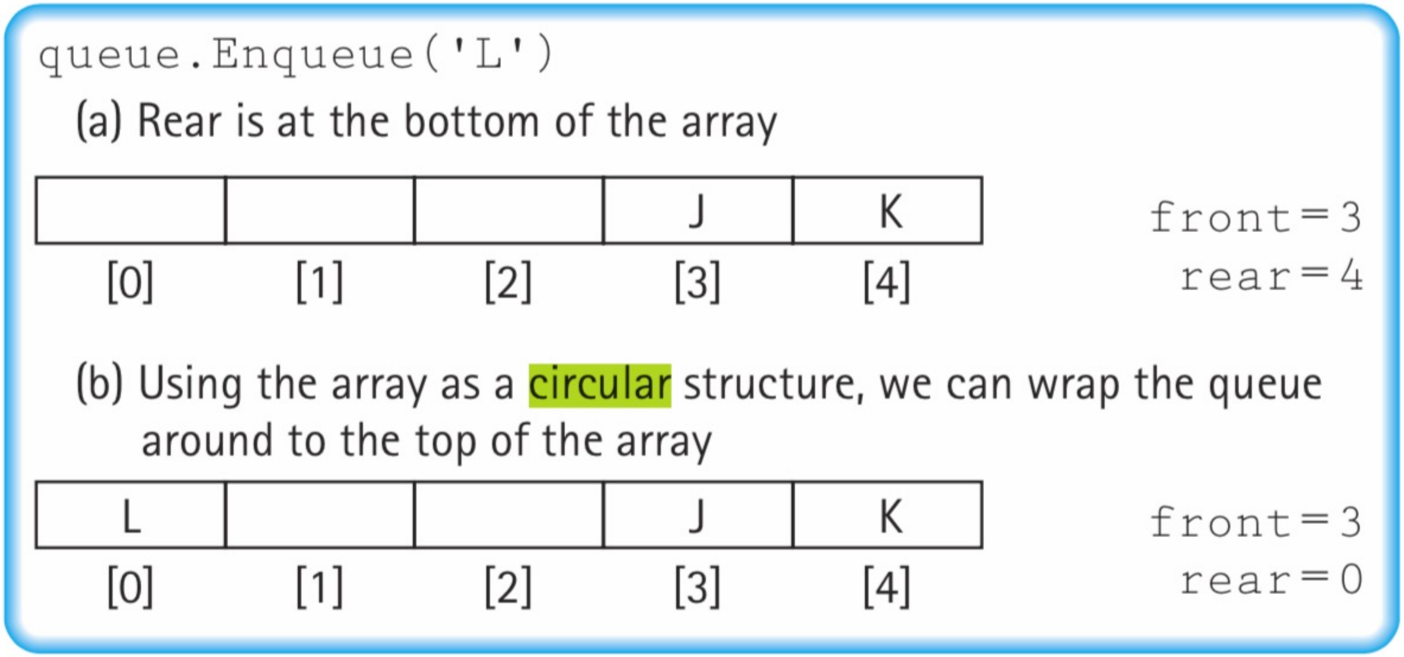


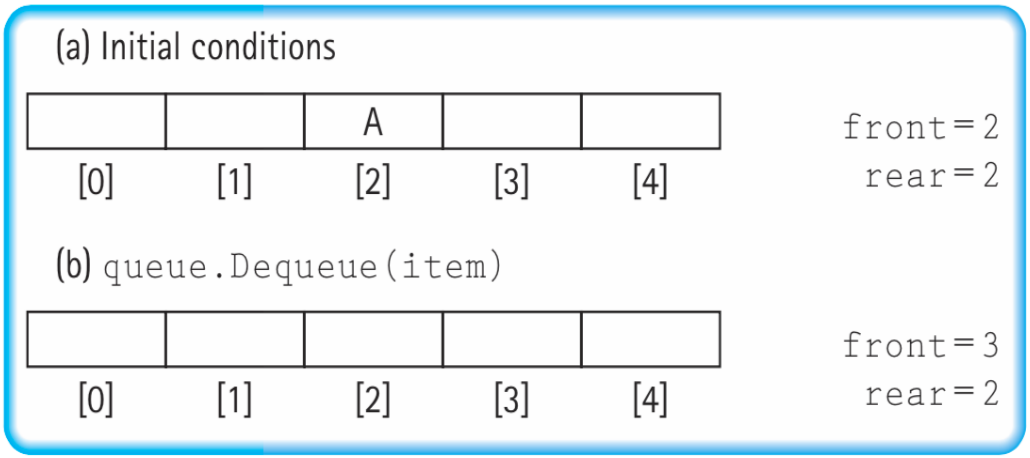
Figure 4.9 *Wrapping the queue elements around*

How to wrap around?
(rear+1) % 5

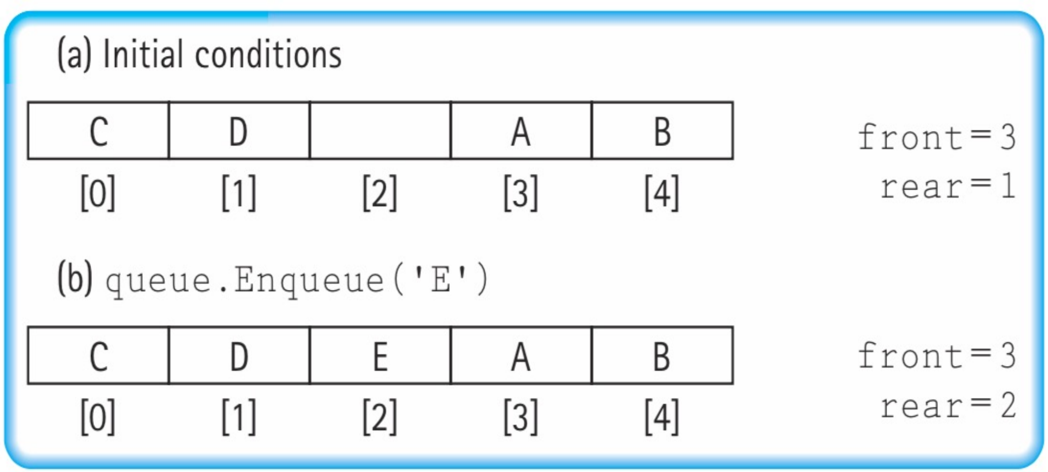
Array-based Implementation: Floating Queue

Need to differentiate!
sol:

* add a length member
* reserve an empty slot



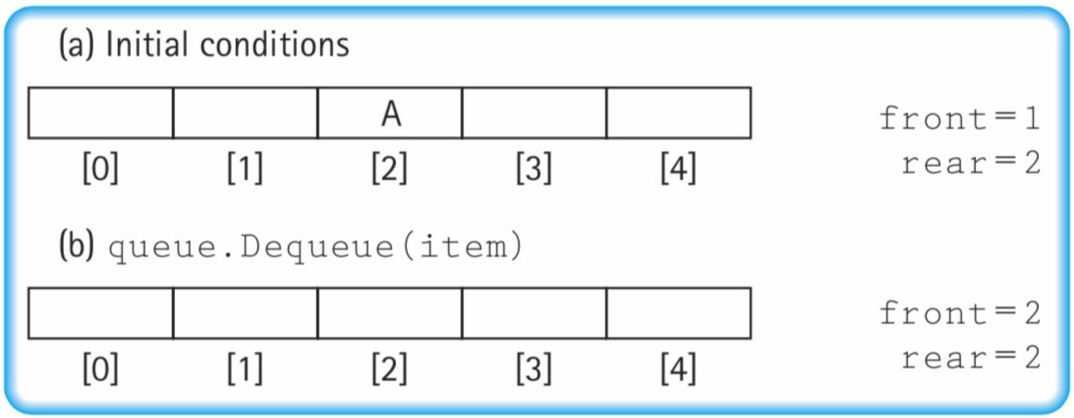
Empty Queue



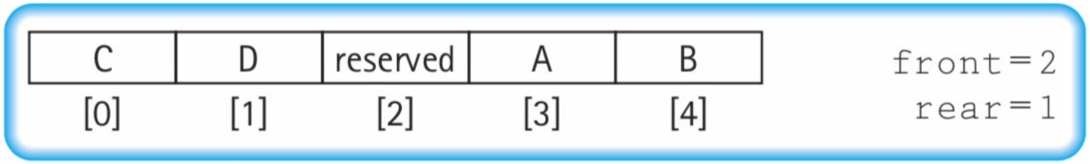
Full Queue

Array-based Implementation: Floating Queue

- An array with **front** indicate the slot before the front item, and this slot does not store anything)



Empty queue:
front==rear



Full queue:
front==rear+1

Array-based Implementation: Floating Queue

```
private:
    int front;        // index of front element -1
    int rear;         //index of queue rear element
    int maxQue;  //size of array

     ItemType * items;
};
```

Array-based Implementation: Floating Queue

```
QueType::QueType(int max=500)
// Parameterized class constructor
// Pre: maxQue, front, and rear have been initialized.
//Post: The array to hold the queue elements has been dynamically allocated.
{
  maxQue = max + 1;
  front = maxQue - 1;
  rear = maxQue - 1;
  items = new ItemType[maxQue];
}
```

Array-based Implementation: Floating Queue

```
QueType::~QueType()
{
 delete [] items;
}
```

Array-based Implementation: Floating Queue

```
void QueType::Enqueue(ItemType newItem)
// Post: If (queue is not full) newItem is at the rear of the queue;
//          Otherwise a FullQueue exception is thrown.
{
  if (IsFull())
    throw FullQueue();
  else
  {
    rear = (rear +1) % maxQue;
    items[rear] = newItem;
  }
}
```

Array-based Implementation: Floating Queue

```
void QueType::Dequeue(ItemType& item)
// Post: If (queue is not empty) the front of the queue is removed, and a copy returned in item;
//         Otherwise a EmptyQueue exception is thrown.
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    front = (front + 1) % maxQue;
    item = items[front];
  }
}
```

Array-based Implementation: Floating Queue

```
bool QueType::IsEmpty() const
// Returns true if the queue is empty; false otherwise.
{
  return (rear == front);
}

bool QueType::IsFull() const
// Returns true if the queue is full; false otherwise.
{
  return ((rear + 1) % maxQue == front);
}
```

Linked-Structure implementation of Queue

How do you define the data member of Queue?

Linked-Structure implementation of Queue

Enqueue(newItem)
Set newNode to the address of newly allocated node
Set Info(newNode) to newItem
Set Next(newNode) to NULL
If queue is empty
    Set front to newNode
else
    Set Next(rear) to newNode
Set rear to newNode

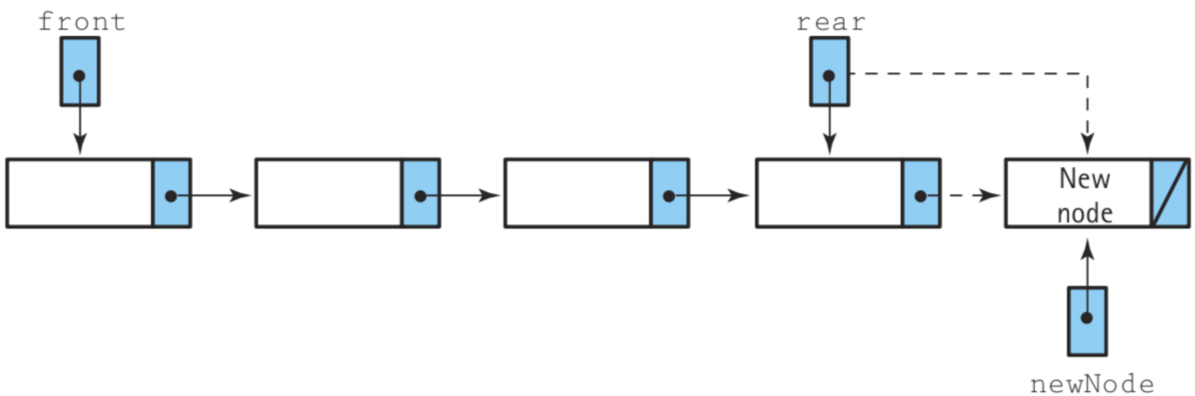


Figure 5.16 *The* `Enqueue` *operation*

Linked-Structure implementation of Queue

Dequeue(item)
Set tempPtr to front
Set item to Info(front)
Set front to Next(front)
if queue is now empty
Set rear to NULL
Deallocate Node(tempPtr)

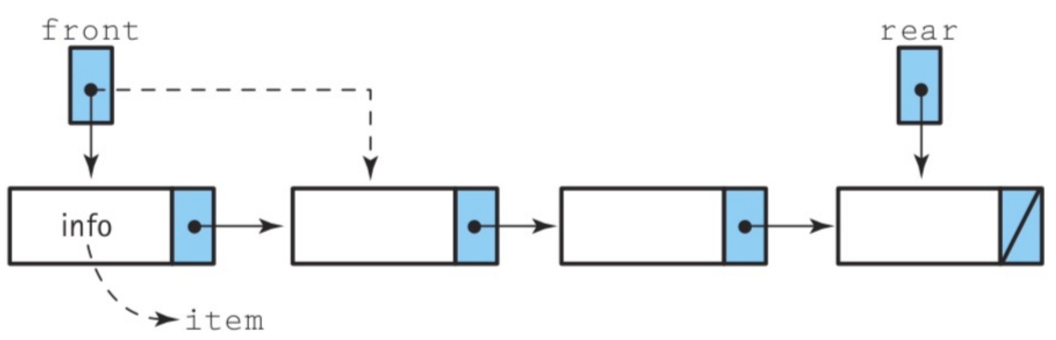


Figure 5.18 *The* Dequeue *operation*