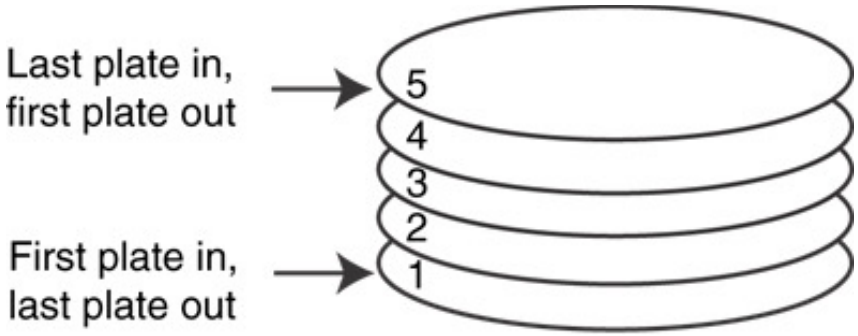
Stacks and Queues

Introduction to the Stack ADT

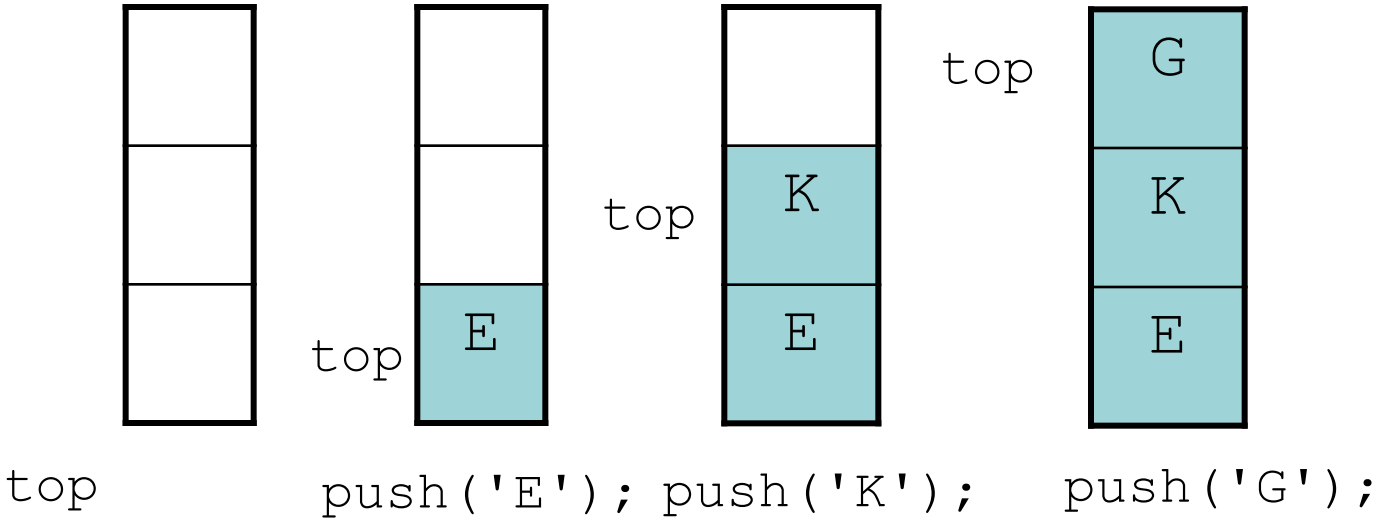
Introduction to the Stack ADT

- Stack: a LIFO (last in, first out) data structure
- Examples:
  - plates in a cafeteria
  - return addresses for function calls
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list



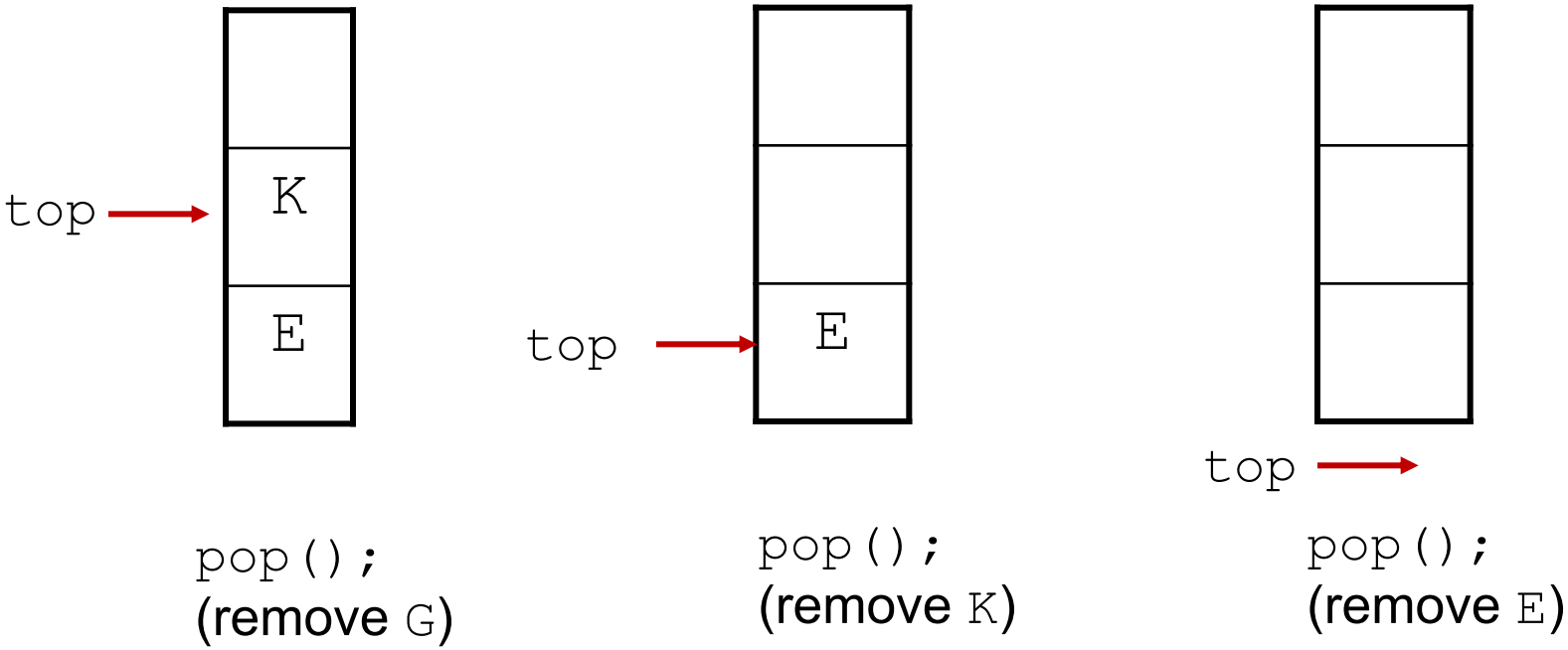
Stack Operations and Functions

- Operations:
  - push: add a value onto the top of the stack
  - pop: remove a value from the top of the stack
- Functions:
  - isFull: true if the stack is currently full, *i.e.*, has no more space to hold additional elements
  - isEmpty: true if the stack currently contains no elements
- A stack that can hold char values:



Stack Operations Example

- A stack that can hold char values:



Stack Operations Example

```
template <class T>
class Stack{
private:
    T *data;
    int sz;
    int tp;
    int capacity;

public:
    Stack()=delete;
    Stack(int cp);
    Stack(const Stack& ot);
    Stack(Stack&& ot);

    Stack& operator=(const Stack& ot);
    bool full();
    bool empty();
    bool push(T v);
    bool pop();
    T top();
    int size();
    int getcapacity();
};
```

```
Stack()=delete;

Stack(int cp)
   :capacity(cp),sz(0),tp(-1){
       data = new T[cp];
}

Stack(const Stack& ot){
       capacity=ot.capacity;
       sz= ot.sz;
       tp = ot.tp;
       data = new T[capacity];
       for (int i=0;i<capacity;i++){
           data[i] = ot.data[i];
   }
}
```

```
Stack& operator=(const Stack& ot) {
       capacity=ot.capacity;
       sz = ot.sz;
       tp = ot.tp;
       data = new T[capacity];
       for (int i=0;i<capacity;i++) {
           data[i] = ot.data[i];
       }
       return *this;
    }
```

```
Stack(Stack&& ot)
{
       capacity=ot.capacity;
       size = ot.sz;
       tp = ot.tp;
       data = ot.data;
       ot.data = nullptr;
}
```

```
bool full() {
   return sz == capacity;
}
```

```
bool empty() {
       return tp == -1;
}
```

```
bool push(T v) {
  if (!full()) {
      tp++;
      data[tp]=v;
      sz++;
      return true;
   }
    else
   return false;
}
```

Stack Operations Example

```
bool pop() {
    if (!empty()) {
        data[tp] = -1;
        tp--;
        sz--;
     }
      return false;
}
```

```
T top() {
    if (!empty())
        return data[tp];
    else
        return T();
}
```

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

```
int getcapacity() {
    return capacity;}
```

```
int main() {
    Stack<int> a(5);
    Stack<int> b(5);

    Stack<int> d(Stack<int>(3));
    cout << d.getcapacity() << endl;

    for (int i=0;i<5;i++)
        a.push(i*2);

    Stack<int> c(a);
    cout << "Stack a:\n";
    while (!c.empty()){
        cout << c.top() << ":";
        c.pop();
    }
    cout << endl;

    while (!a.empty()){
        b.push(a.top());
        a.pop();
}
```

```
    cout << "Stack b:\n";
    while (!b.empty()) {
        cout << b.top()
             << ":";
        b.pop();
    }
    cout << endl;

    return 0;
}
```

3
Stack a:
8:6:4:2:0:
Stack b:
0:2:4:6:8:

Dynamic Stacks (Inheritance)

- Grow and shrink as necessary
- Can't ever be full as long as memory is available
- Implemented as a linked list

```
template <class T>
class Stack:private list<T> {
public:
    Stack()=default;
    bool full();
    bool empty();
    void push(T v);
    void pop();
    T top();
    int size();
};
```

```
template<class T>
void Stack<T>::push(T v) {
    list<T>::push_back(v);
}
```

```
template<class T>
void Stack<T>::pop() {
    this->pop_back();
}
```

```
template<class T>
T Stack<T>::top() {
  if (!empty())
     return this->back();
  else
        return T();
}
```

```
template<class T>
bool Stack<T>::empty() {
    return list<T>::empty();
}
```

```
template<class T>
int Stack<T>::size() {
    return this->size();
}
```

```
int main() {
    Stack<int> a, b;
    for (int i=0;i<5;i++) a.push(i*2);
    cout << "Stack a:\n";
    while (!a.empty()) {
        cout << a.top() << ":";
        b.push(a.top());
        a.pop();
    }
    cout << endl;
    cout << "Stack b:\n";
    while (!b.empty()) {
        cout << b.top() << ":";
        b.pop();
    }
    cout << endl;
    return 0;
}
```

```
Stack a:
8:6:4:2:0:
Stack b:
0:2:4:6:8:
```

Dynamic Stacks (Aggregation)

```
template <class T>
class Stack{
    list<T>  stack;
public:
    Stack()=default;
    bool full();
    bool empty();
    void push(T v);
    void pop();
    T top();
    int size();
};
```

```
template<class T>
int Stack<T>::size(){
    return stack.size();
}
```

```
template<class T>
T Stack<T>::top() {
    if (!empty())
        return stack.back();
    else
        return T();
}
```

```
template<class T>
bool Stack<T>::empty(){
    return stack.empty();
}
```

```
template<class T>
void Stack<T>::push(T v) {
    stack.push_back(v);
}
```

```
template<class T>
void Stack<T>::pop() {
    stack.pop_back();
}
```

```
int main() {
    Stack<int> a, b;
    for (int i=0;i<5;i++)  a.push(i*2);
    cout << "Stack a:\n";
    while (!a.empty()){
        cout << a.top() << ":";
        b.push(a.top());
        a.pop();
    }
    cout << endl;
    cout << "Stack b:\n";
    while (!b.empty()) {
        cout << b.top()
        << ":";
        b.pop();
    }
    cout << endl;
    return 0;
}
```

```
Stack a:
8:6:4:2:0:
Stack b:
0:2:4:6:8:
```

Implementing a Stack

- Can also use the implementation of stack available in the STL

```
#include <iostream>
#include <stack>

using namespace std;

int main() {

    stack<int> a;

    for (int i=0;i<5;i++)
       a.push(i);

    while (!a.empty()) {
        cout << a.top() << ":";
        a.pop();
    }
    cout << endl;

    return 0;
}
```

```
int main()
{
    vector<int> v({1,2,3,4,5});
    stack<int,vector<int>>  a(v);

    while (!a.empty()){
        cout << a.top() << ":";
        a.pop();
    }
    cout << endl;

    return 0;
}
```

```
5:4:3:2:1:
```

Defining a Stack

- Defining a stack of chars, named cstack, implemented using a vector:

  stack< char, vector<char> > cstack;

- implemented using a list:

  stack< char, list<char> > cstack;

- implemented using a deque:

  stack<char> cstack;

- When using a compiler that is older than C++ 11, be sure to put spaces between the angled brackets that appear next to each other.

```
stack< char, vector<char> > cstack;
```

Postfix expression evaluation

```
Create a new stack
For each token in expression
    If the token is an operand
        Push the token onto the stack
    Else if the token is an operator
        Pop the top two tokens from the stack (two operands)
        Use the current token to evaluate the two operands
just popped
        Push the result onto the stack (an operand)
Return the top item on the stack (the result value)
```

Postfix expression evaluation

```
int eval (int op1, int op2, char operate) {
    switch (operate) {
        case '*': return op2 * op1;
        case '/': return op2 / op1;
        case '+': return op2 + op1;
        case '-': return op2 - op1;
        default : return 0;
    }
}
```

```
int main() {
    char postfix[] = {'5','6','8','+','*','2','/'};
    int size = sizeof(postfix);
    int val = evalPostfix(postfix, size);
    cout<<"\nExpression evaluates to "<<val;
    cout<<endl;
    return 0;
}
```

Expression evaluates to 35

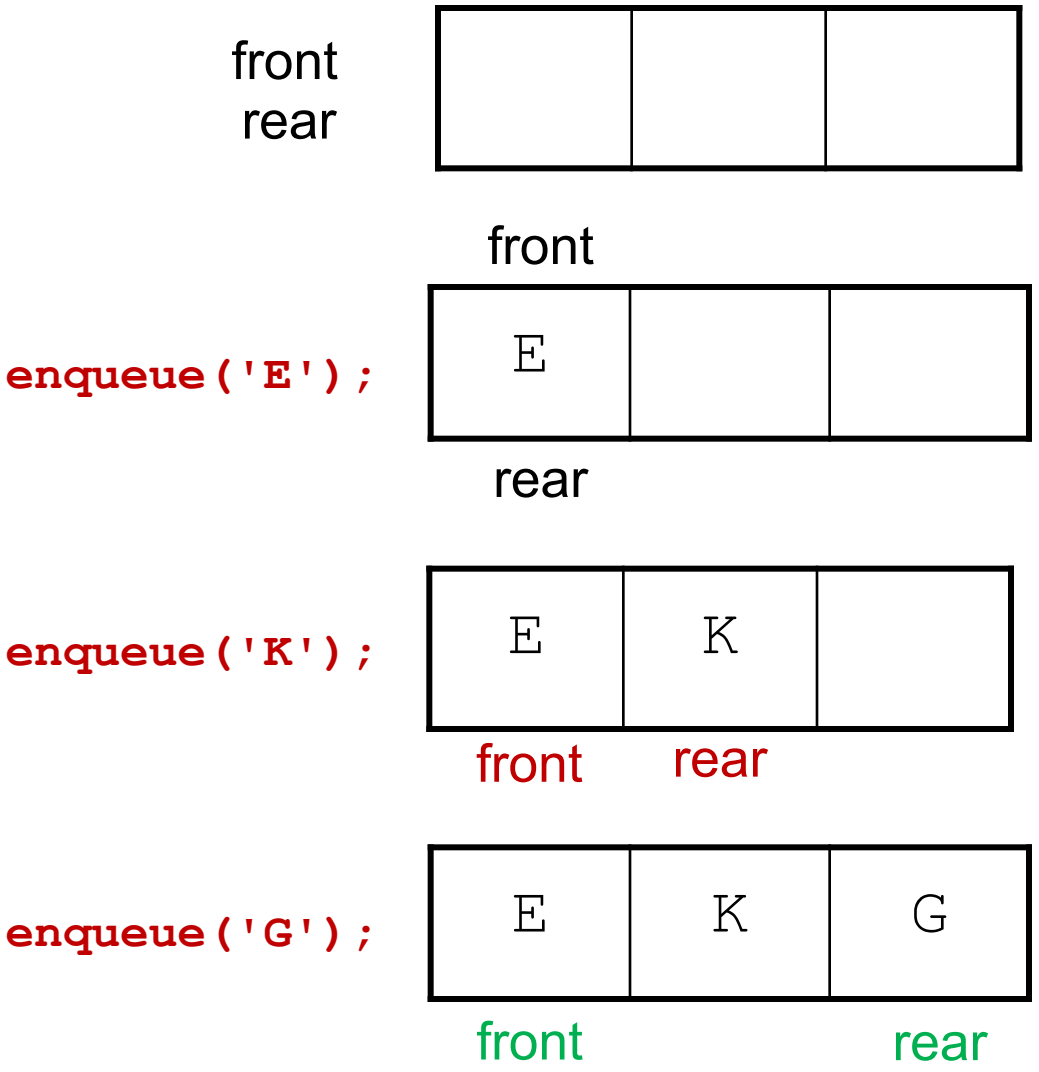
```
int evalPostfix(char postfix[], int size) {
    stack<int> s;
    int i = 0;
    char ch;
    int val;
    while (i < size) {
        ch = postfix[i];
        if (isdigit(ch)) {
            s.push(ch-'0');
        }
        else {
            int op1 = s.top();
            s.pop();
            int op2 = s.top();
            s.pop();
            val = eval(op1, op2, ch);
            s.push(val);
        }
        i++;
    }
    return val;
}
```

Introduction to the Queue ADT

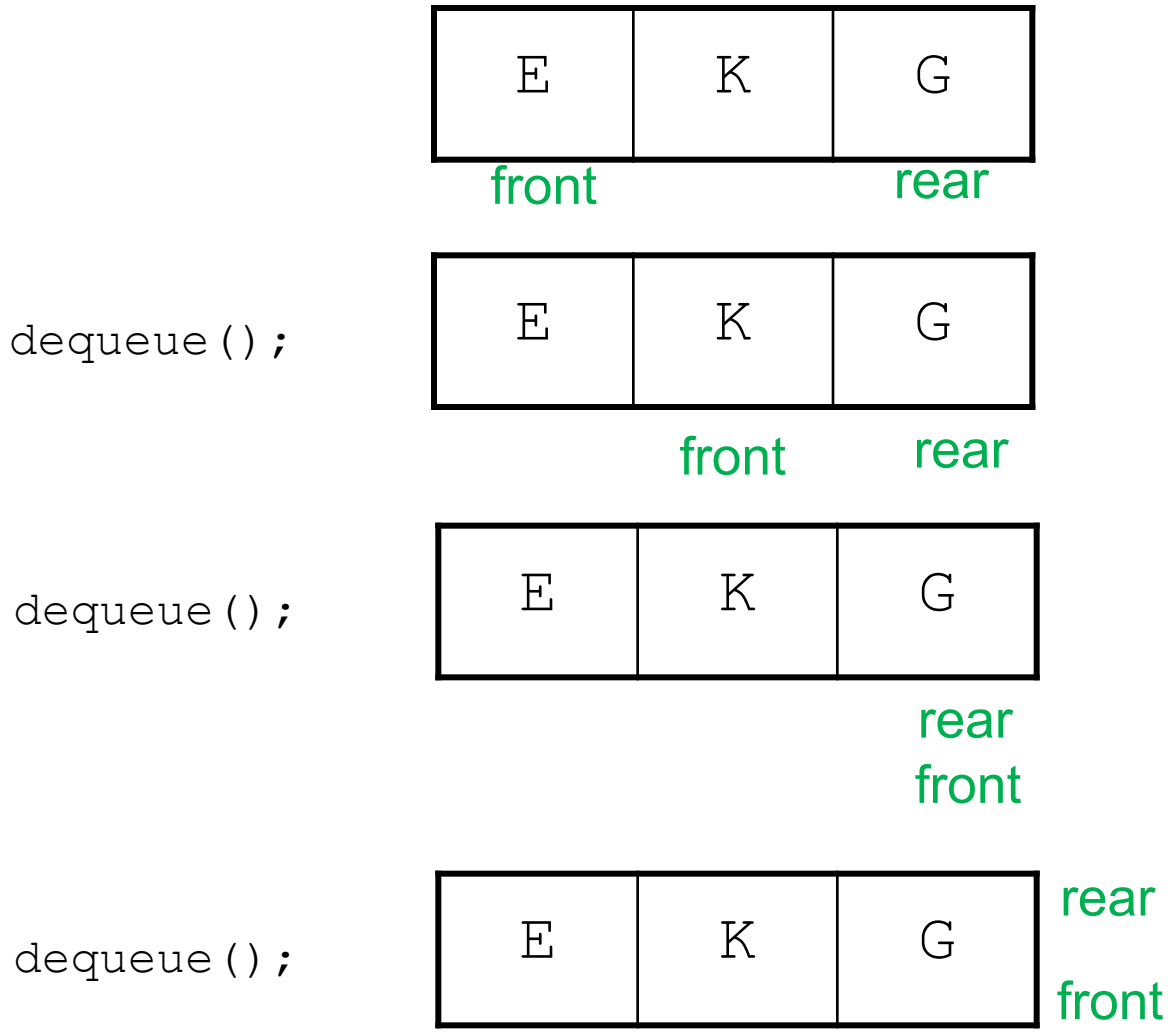
- Queue: a FIFO (first in, first out) data structure.
- Examples:
  - people in line at the theatre box office
  - print jobs sent to a printer
- Implementation:
  - **static**: fixed size, implemented as array
  - **dynamic**: variable size, implemented as linked list

Queue Locations and Operations

- **rear**: position where elements are added
- **front**: position from which elements are removed
- **enqueue**: add an element to the rear of the queue
- **dequeue**: remove an element from the front of a queue



Queue Locations and Operations



Dequeue Issues and Solutions

- When removing an element from a queue, remaining elements must shift to front
- Solutions:
  - Let front index move as elements are removed (works as long as rear index is not at end of array)
  - Use above solution, and also let rear index "wrap around" to front of array, treating array as circular instead of linear (more complex enqueue, dequeue code)

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

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

    - enqueue moves **rear** to the right as it fills positions in the array
    - dequeue moves **front** to the right as it empties positions in the array
    - When enqueue gets to the end, it wraps around to the beginning to use those positions that have been emptied
    - When dequeue gets to the end, it wraps around to the beginning use those positions that have been filled

Queue Locations and Operations

```
template<class T>
class MQueue {
private:
    T *queueArray;
    int queueSize;
    int front;
    int rear;
    int numItems;
public:
    MQueue(int qs);
    MQueue<T>(const MQueue<T>
&qq);
    ~MQueue<T>();
    bool isEmpty()const;
    bool isFull() const;
    void enqueue(int v);
    int dequeue();
    int size();
    void clear();
};
```

```
template<class T>
MQueue<T>::MQueue(int qs) {
    queueSize = qs;
    queueArray = new T[qs];
    front = 0;
    rear = 0;
    numItems = 0;
}
```

```
template<class T>
MQueue<T>::~MQueue(){
    if (queueArray != nullptr)
       delete queueArray;
    front = rear = 0;
    numItems = 0;
    queueSize = 0;
}
```

Copy constructor

```
template<class T>
MQueue<T>::MQueue(const MQueue<T> &qq)
{
    queueSize = qq.queueSize;
    queueArray = new T[queueSize];
    for (int i=0;i<queueSize;i++)
      queueArray[i]=qq.queueArray[i];
    front = qq.front;
    rear = qq.rear;
    numItems = qq.numItems;
}
```

Queue Locations and Operations

```
template<class T>
bool MQueue<T>::isEmpty() const {
    if (numItems == 0)
       return true;
    else return false;
}
```

```
template<class T>
bool MQueue<T>::isFull() const{
    if (numItems >= queueSize)
        return true;
    else return false;
}
```

```
template<class T>
int MQueue<T>::size() {
    return numItems;
}
```

```
template<class T>
void MQueue<T>::clear() {
     front = rear = -1;
     numItems = 0;
}
```

```
template<class T>
void MQueue<T>::enqueue(int v)
{
    if(!isFull())
    {
        rear = (rear + 1) % queueSize;
        queueArray[rear] = v;
        numItems++;
    }
    else
        cout << "Queue is full\n";
}
```

```
template<class T>
int MQueue<T>::dequeue() {
    int v =-1;
    if(!isEmpty()) {
        front = (front + 1) % queueSize;
        v = queueArray[front];
        numItems--;
    }
    else    cout << "Queue is empty..\n";
    return v;
}
```

Queue Locations and Operations

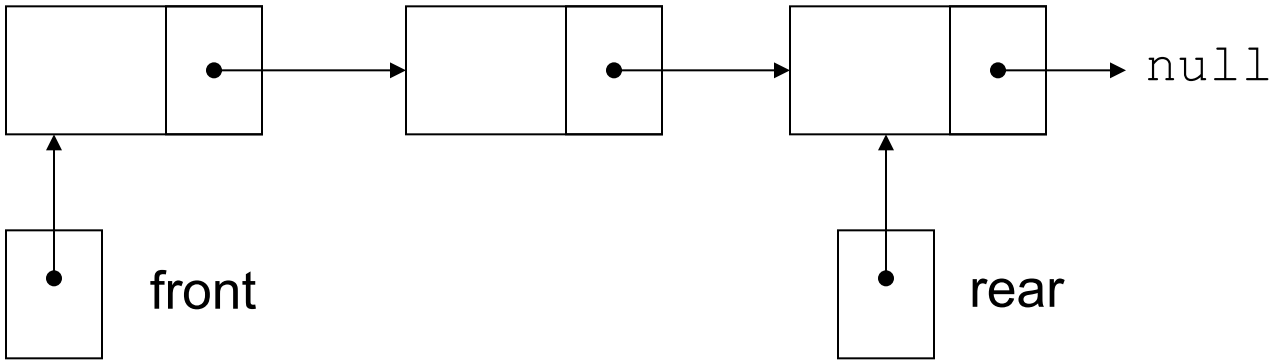
```
template <class T>
void print(MQueue<T> &q){
    MQueue<T> qq(q);
    cout << "Q ("<< qq.size() << ") = ";
    while(!qq.isEmpty())
        cout << qq.dequeue() << ":";
    cout << endl;
}
```

```
int main() {
    MQueue<int> q(5);
    print(q);
    for (int i=0;i<5;i++) {
        int x = rand()%20;
        cout << x << " added to the queue\n";
        q.enqueue(x);
        print(q);
    }
    q.enqueue(10);

    while (!q.isEmpty()){
        int x = q.dequeue();
        cout << x << " is removed from the queue\n";
        print(q);
    }
    q.dequeue();

    return 0;
}
```

```
Q (0) =
7 added to the queue
Q (1) = 7:
9 added to the queue
Q (2) = 7:9:
13 added to the queue
Q (3) = 7:9:13:
18 added to the queue
Q (4) = 7:9:13:18:
10 added to the queue
Q (5) = 7:9:13:18:10:
Queue is full
7 is removed from the queue
Q (4) = 9:13:18:10:
9 is removed from the queue
Q (3) = 13:18:10:
13 is removed from the queue
Q (2) = 18:10:
18 is removed from the queue
Q (1) = 10:
10 is removed from the queue
Q (0) =
Queue is empty..
```

Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- Allows dynamic sizing, avoids issue of shifting elements or wrapping indices
- Define a class for the dynamic queue
- Within the dynamic queue, define a private member class for a dynamic node in the queue
- Define node pointers to the front and rear of the queue



Dynamic Queues

```
template<class T>
struct Node {
    T data;
    Node<T> * next;

    Node(T x) {
        data = x;
        next = nullptr;
    }
};
```

```
template<class T>
class MQueue {
    Node<T> *front;
    Node<T> *rear;
    int sz;

public:
    MQueue() {
        sz = 0;
        front = rear = nullptr;
    }
```

```
 ~MQueue() {
        sz=0;
        while(!isEmpty())
           dequeue();
    }
```

```
MQueue(const MQueue<T>& mq) {
        sz = 0;
        front = rear = nullptr;
        Node<T> *ptr =mq.front;
        while (ptr!=nullptr){
            enqueue(ptr->data);
            ptr=ptr->next;
        }
    }
```

Dynamic Queues

```
int dequeue(){
    if (!isEmpty())
    {
            Node<T> *p = front;
            int x = front->data;
            front = front->next;
            delete p;
            sz--;
            return x;
   }
    else
   {
            cout << "Queue is empty\n";
            return -1;
   }
}
```

```
void enqueue(T x)
{
        if (isEmpty()) {
            front = new Node<T>(x);
            rear = front;
        }
        else
        {
            rear->next = new Node<T>(x);
            rear = rear->next;
        }
        sz++;
    }
```

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

```
bool isEmpty()
{
        if (front == nullptr)
            return true;
        else
            return false;
}
```

Dynamic Queues

```
int main() {

    MQueue<int> q;

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.enqueue(40);
    q.enqueue(50);

    MQueue<int> qq(q);

    while (!qq.isEmpty())
        cout << qq.dequeue() << endl;


    cout << "end...\n";

    return 0;
}
```

```
10
20
30
40
50
end...
```

Implementing a Queue

- Programmers can program their own routines to implement queue operations
- Can also use the implementation of queue and dequeue available in the STL

STL deque and queue

- **deque**: a double-ended queue. Has member functions to enqueue (push_back) and dequeue (pop_front)
- **queue**: container ADT that can be used to provide queue as a vector, list, or deque. Has member functions to enque (push) and dequeue (pop)

- Defining a queue of chars, named cQueue, implemented using a deque:

```
deque<char> cQueue;
```

- implemented using a queue:

```
queue<char> cQueue;
```

- implemented using a list:

```
queue<char, list<char>> cQueue;
```

STL deque

```
void print(deque<int> &c){
    for (auto &x:c)
        cout << x << " ";
    cout << endl;
}

int main()
{
    deque<int> Q;
    Q.push_back(2); print(Q);
    Q.push_back(4); print(Q);
    Q.push_back(6); print(Q);
    Q.front();      cout << Q.front() << endl;
    Q.pop_front();  print(Q);
    Q.push_back(8); print(Q);
}
```

STL queue

```
#include <queue>

void print(queue<int> &c){
    for (int i=0; i<c.size();i++) {
        int x = c.front();
        cout << x << " ";
        c.pop();
        c.push(x);
    }
    cout << endl;
}
```

```
int main()
{
    queue<int> Q;
    Q.push(2); print(Q);
    Q.push(4); print(Q);
    Q.push(6); print(Q);
    Q.front();      cout << Q.front() << endl;
    Q.pop();  print(Q);
    Q.push(8); print(Q);
}
```

```
2
2 4
2 4 6
2
4 6
4 6 8
```