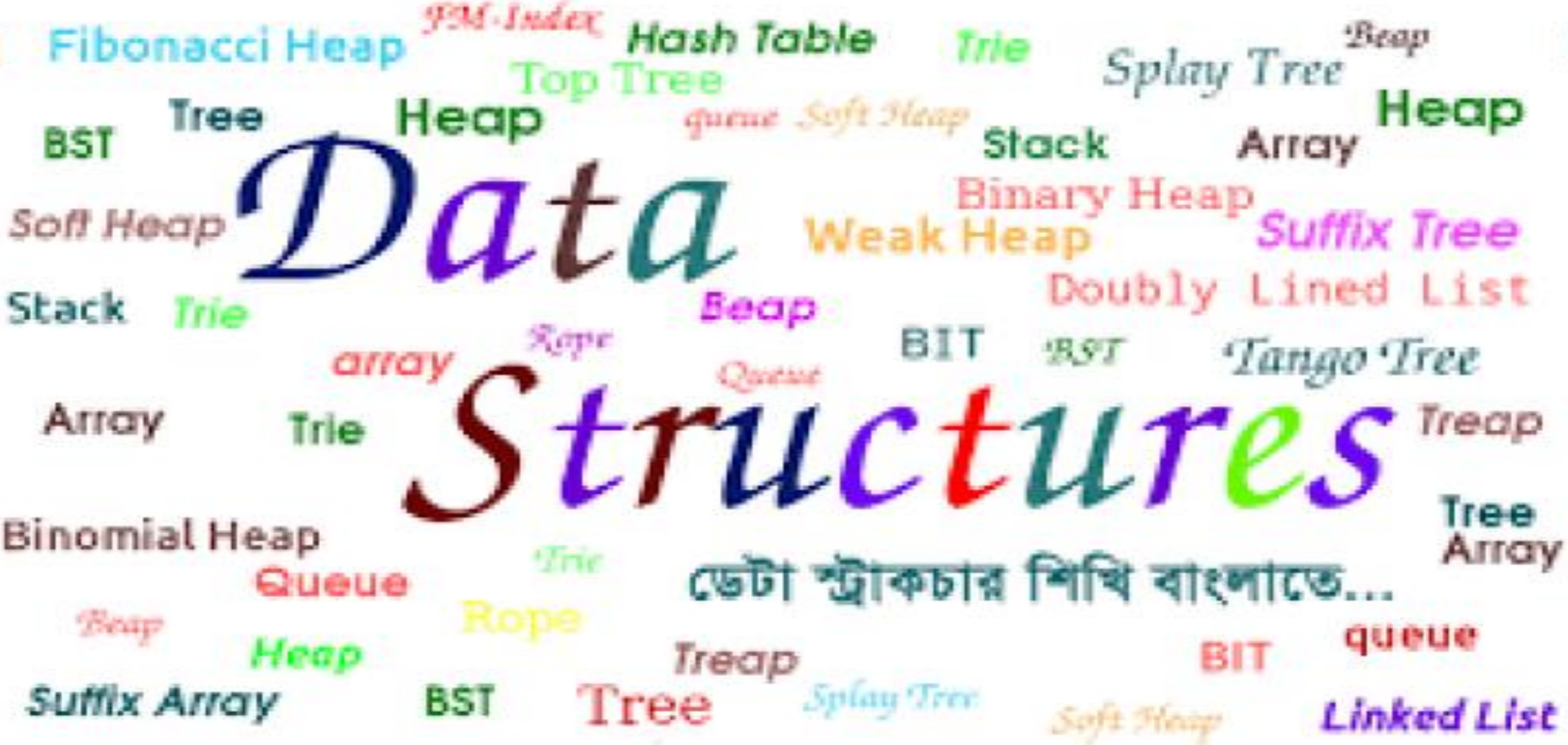

Fibonacci Heap
FM-Index
Hash Table
Trie
Beap
Splay Tree
Top Tree
Tree
Heap
queue
Soft Heap
Heap
BST
Stack
Array
Data
Binary Heap
Soft Heap
Weak Heap
Suffix Tree
Doubly Lined List
Stack
Trie
Beap
Rope
BIT
BST
Tango Tree
array
Queue
Array
Trie
Structures
Treap
Tree
Binomial Heap
Trie
Array
ডেটা স্ট্রাকচার শিখি বাংলাতে...
Queue
Beap
Rope
Heap
Treap
BIT
queue
Suffix Array
BST
Tree
Splay Tree
Soft Heap
Linked List

Introduction to the Stack ADT

Stack: a LIFO (last in, first out) data structure

Examples:

- plates in a cafeteria serving area
- return addresses for function calls

Stack Basics

Stack is usually implemented as a list, with additions and removals taking place at one end of the list

The active end of the list implementing the stack is the top of the stack

Stack types:

- Static – fixed size, often implemented using an array
- Dynamic – size varies as needed, often implemented using a linked list

Stack Operations and Functions

Operations:

- push: add a value at the top of the stack
- pop: remove a value from the top of the stack

Functions:

- isEmpty: true if the stack currently contains no elements
- isFull: true if the stack is full; only useful for static stacks

Static Stack Implementation

Uses an array of a fixed size

Bottom of stack is at index 0. A variable called top tracks the current top of the stack

```
const int STACK_SIZE = 3;
char s[STACK_SIZE];
int top = 0;
```

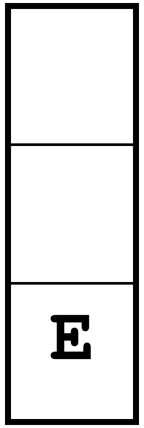
top is where the next item will be added

Array Implementation Example

This stack has max capacity 3, initially top = 0 and stack is empty.

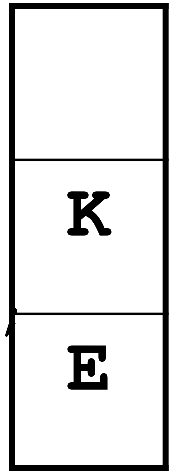
`push('E');`

top is 1



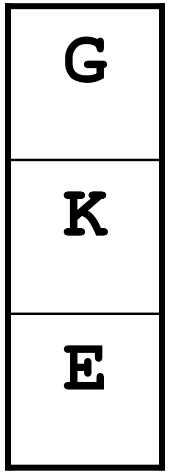
`push('K');`

top is 2



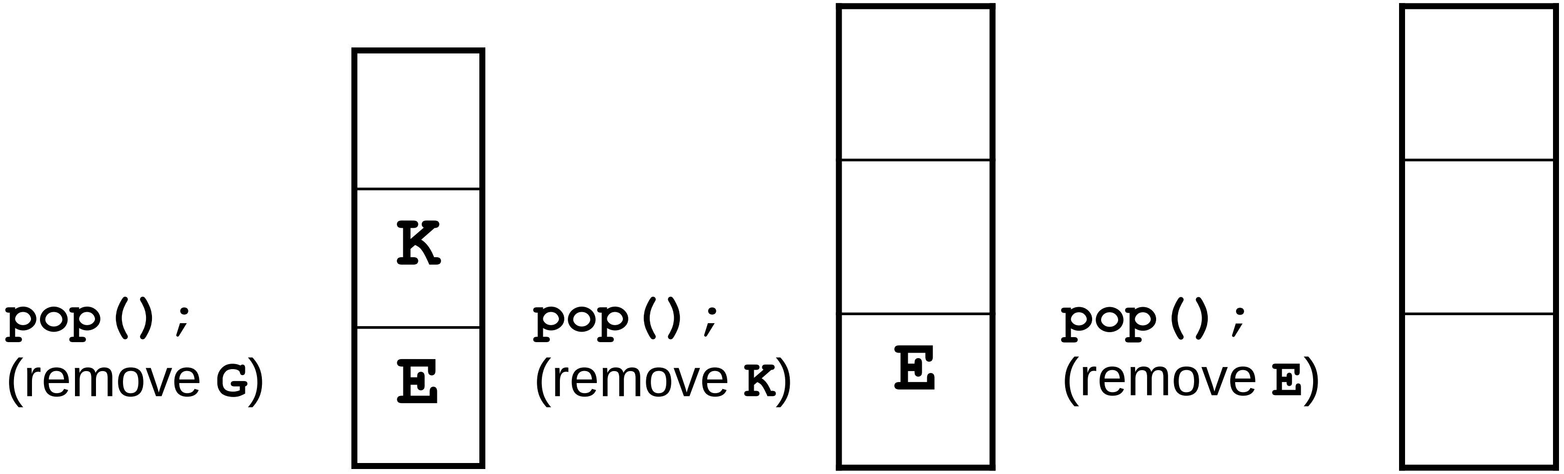
`push('G');`

top is 3



Stack Operations Example

After three pops, `top == 0` and the stack is empty



Array Implementation

```
char s[STACK_SIZE];
int top=0;
```

To check if stack is empty:

```
bool isEmpty()
 {
   if (top == 0)
          return true;
  else return false;
 }
```

Array Implementation

```
char s[STACK_SIZE];
int top=0;
```

To check if stack is full:

```
bool isFull()
{
  if (top == STACK_SIZE)
          return true;
 else return false;
}
```

Array Implementation

To add an item to the stack

```
void push(char x)
 {
  if (isFull())
     {error(); exit(1);}
  // or could throw an exception
  s[top] = x;
  top++;
 }
```

Array Implementation

To remove an item from the stack

```
void pop(char &x)
 {
  if (isEmpty())
     {error(); exit(1);}
  // or could throw an exception
 top--;
  x = s[top];
}
```

Class Implementation

```
class STACK
  {
private:
     char *s;
     int capacity, top;
  public:
     void push(char x);
     void pop(char &x);
     bool isFull(); bool isEmpty();
     STACK(int stackSize);
     ~STACK()
  };
```

Exceptions from Stack Operations

Exception classes can be added to stack object definition to handle cases where an attempt is made to push onto a full stack (overflow) or to pop from an empty stack (underflow)

Program that uses **`push`** and **`pop`** operations should do so from within a **`try`** block.

`catch` block(s) should follow the **`try`** block, interpret what occurred, and inform the user.

Dynamic Stacks

Implemented as a linked list

Can grow and shrink as necessary

Can't ever be full as long as memory is available

Linked List Implementation

Node for the linked list

```
struct LNode
{
  char value;
  LNode *next;
  LNode(char ch, LNode *p = 0)
   { value = ch; next = p;}
 };
```

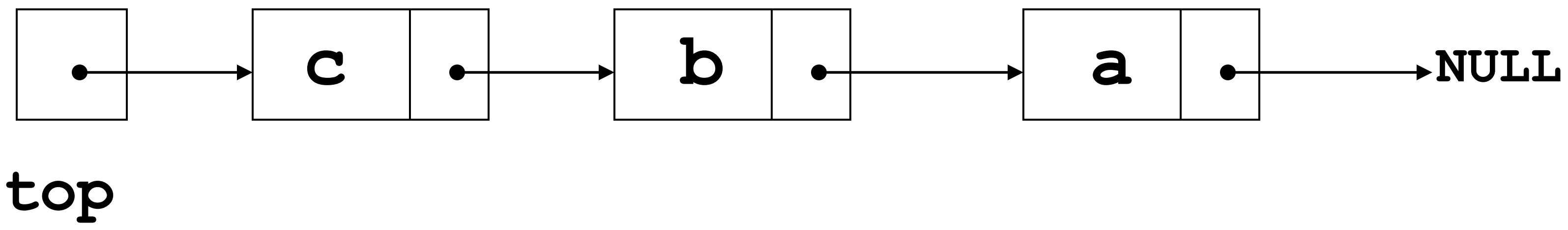
Pointer to beginning of linked list, which will serve as top of stack

```
  LNode *top = NULL;
```

Linked List Implementation

A linked stack after three push operations:

```
push('a'); push('b'); push('c');
```



Operations on a Linked Stack

Check if stack is empty:

```
bool isEmpty()
{
  if (top == NULL)
    return true;
  else
    return false;
}
```

Operations on a Linked Stack

Add a new item to the stack

```
void push(char x)
{
  top = new LNode(x, top);
}
```

Operations on a Linked Stack

Remove an item from the stack

```
void pop(char &x)
{
  if (isEmpty())
  { error(); exit(1);}
  x = top->value;
  LNode *oldTop = top;
  top = top->next;
  delete oldTop;
}
```

The STL **`stack`** Container

Stack template can be implemented as a **`vector`**, **`list`**, or a **`deque`**

Implements **`push`**, **`pop`**, and **`empty`** member functions

Implements other member functions:

- **`size`**: number of elements on the stack
- **`top`**: reference to element on top of the stack (must be used with **`pop`** to remove and retrieve top element)

Defining an STL-based Stack

Defining a stack of **`char`**, named **`cstack`**, implemented using a **`vector`**:

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

Implemented using a list:

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

Implemented using a **`deque`** (default):

```
stack< char > cstack;
```

Spaces are required between consecutive **> >** symbols to distinguish from stream extraction

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

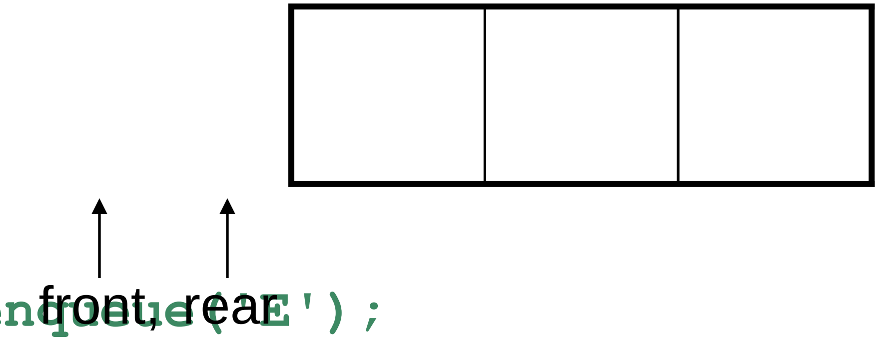
front: position from which elements are removed

enqueue: add an element to the rear of the queue

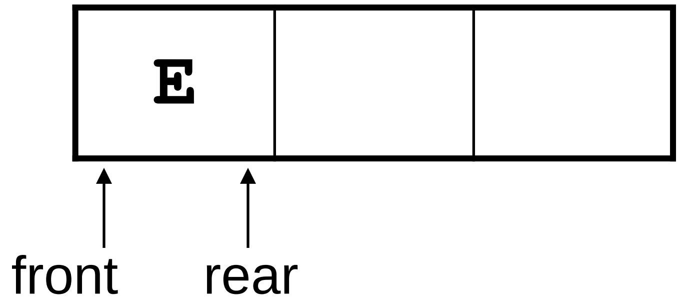
dequeue: remove an element from the front of a queue

Array Implementation of Queue

An empty queue that can hold **`char`** values:

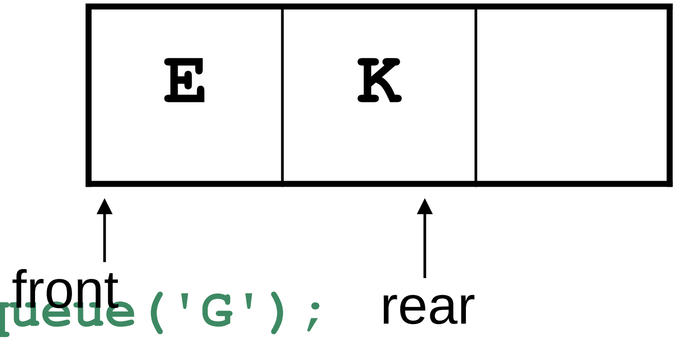


`enqueue('E');`

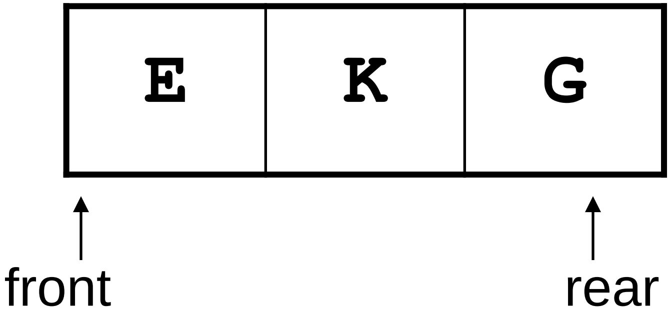


Queue Operations - Example

```
enqueue('K');
```

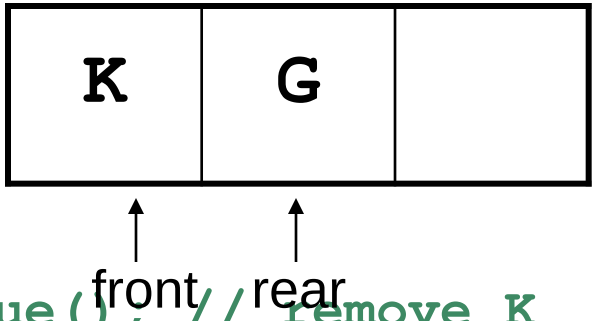



```
enqueue('G');
```

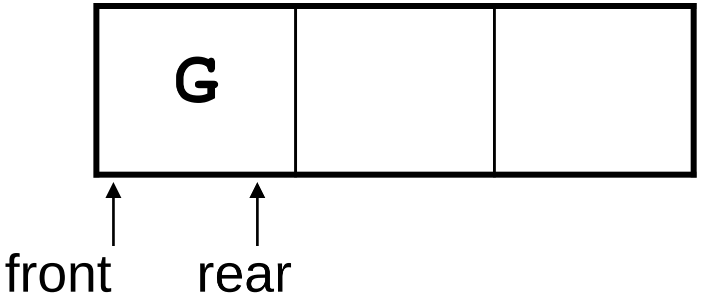


Queue Operations - Example

`dequeue(); // remove E`



`dequeue(); // remove K`



Array Implementation Issues

In the preceding example, Front never moves.

Whenever **`dequeue`** is called, all remaining queue entries move up one position. This takes time.

Alternate approach:

- Circular array: **`front`** and **`rear`** both move when items are added and removed. Both can ‘wrap around’ from the end of the array to the front if warranted.

Other conventions are possible

Array Implementation Issues

Variables needed

- `const int QSIZE = 100;`
- `char q[QSIZE];`
- `int front = -1;`
- `int rear = -1;`
- `int number = 0; //how many in queue`

Could make these members of a queue class, and queue operations would be member functions

`isEmpty` Member Function

Check if queue is empty

```
bool isEmpty()
{
  if (number > 0)
    return false;
  else
    return true;
}
```

`isFull` Member Function

Check if queue is full

```
bool isFull()
{
  if (number < QSIZE)
    return false;
  else
    return true;
}
```

enqueue and **dequeue**

To enqueue, we need to add an item **`x`** to the rear of the queue

Queue convention says **`q[rear]`** is already occupied. Execute

```
if(!isFull)
 { rear = (rear + 1) % QSIZE;
// mod operator for wrap-around
   q[rear] = x;
   number ++;
}
```

enqueue and **dequeue**

To dequeue, we need to remove an item `x` from the front of the queue

Queue convention says `q[front]` has already been removed. Execute

```
     if(!isEmpty)
{   front = (front + 1) % QSIZE;
    x = q[front];
    number--;
}
```

`enqueue` and `dequeue`

`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

enqueue and **dequeue**

Enqueue wraps around by executing

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

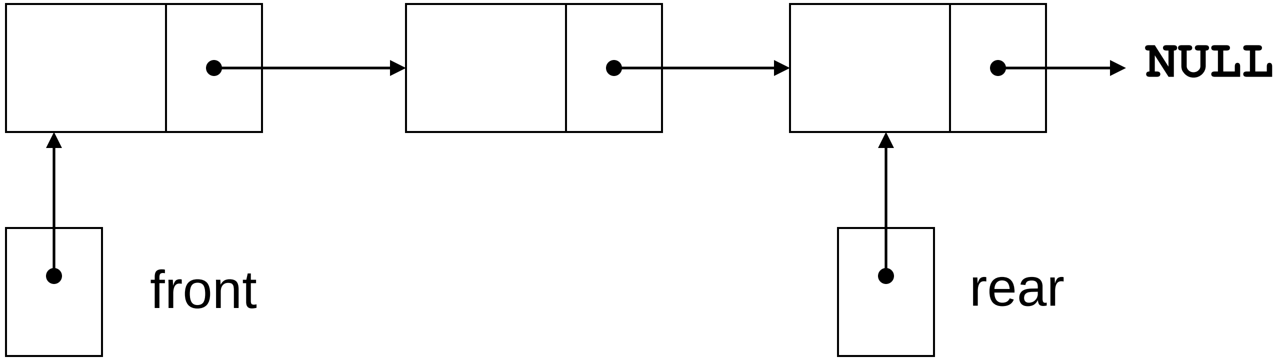
Dequeue wraps around by executing

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

Dynamic Queues

Like a stack, a queue can be implemented using a linked list

Allows dynamic sizing, avoids issue of wrapping indices



Dynamic Queue Implementation Data Structures

```
struct QNode
{
   char value;
   QNode *next;
   QNode(char ch, QNode *p = NULL);
   {value = ch; next = p;}
}
QNode *front = NULL;
QNode *rear = NULL;
```

isEmpty Member Function

To check if queue is empty:

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

enqueue Member Function

To add item at rear of queue

```
void enqueue(char x)
{
  if (isEmpty())
   { rear = new QNode(x);
     front = rear;
     return;
   }
  rear->next = new QNode(x);
  rear = rear->next;
}
```

dequeue Member Function

To remove item from front of queue

```
void dequeue(char &x)
{
  if (isEmpty())
   { error(); exit(1);
   }
  x = front->value;
  QNode *oldfront = front;
  front = front->next;
  delete oldfront;
}
```

The STL **deque** and **queue** Containers

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 a queue based on a **vector**, **list**, or **deque**. Has member functions to enqueue (**push**) and dequeue (**pop**)

Defining a Queue

Defining a queue of **`char`**, named cQueue, based on a **`deque`**:

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

Defining a **`queue`** with the default base container

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

Defining a queue based on a **`list`**:

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

Spaces are required between consecutive **> >** symbols to distinguish from stream extraction

Eliminating Recursion

Recursive solutions to problems are often elegant but inefficient

A solution that does not use recursion is more efficient for larger sizes of inputs

Eliminating the recursion: re-writing a recursive algorithm so that it uses other programming constructs (stacks, loops) rather than recursive calls