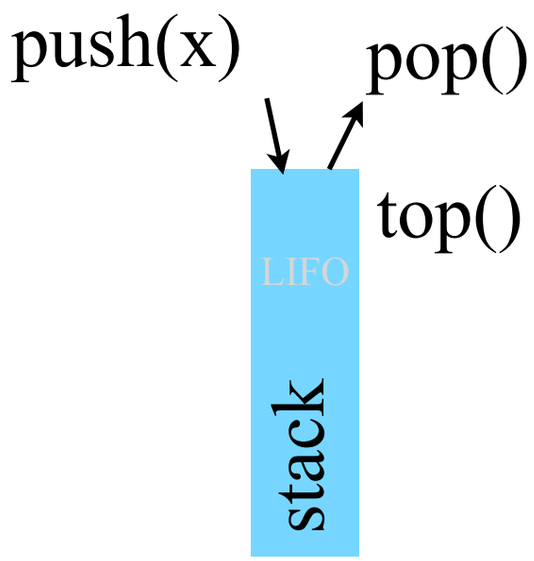
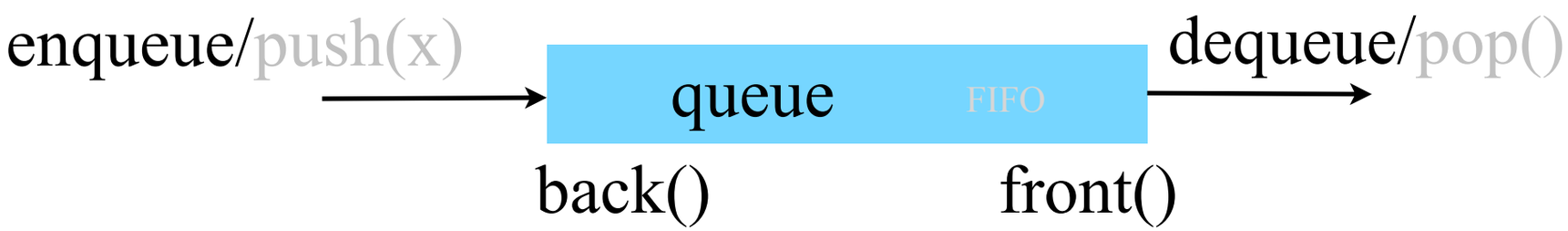
Abstract Data Types

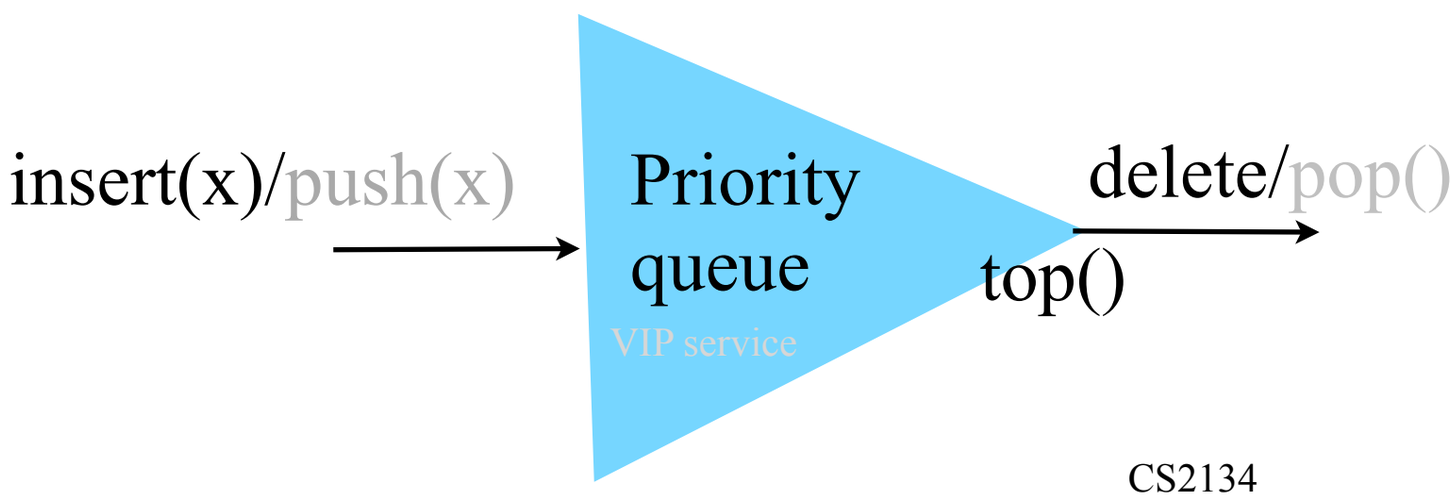
- Abstract Data Types
  - Abstract description of the operations provided and the relationships among them
  - Different implementations are possible for the same ADT
  - Separation of concerns between data type implementation and use
  - Were designed around common algorithmic constructs, rather than physical design
- Classes in Object Oriented languages group data (member variables) with operations to manipulate the data (member functions)
- OO languages developed to support ADTs

ADT's

stack, queue, priority queue



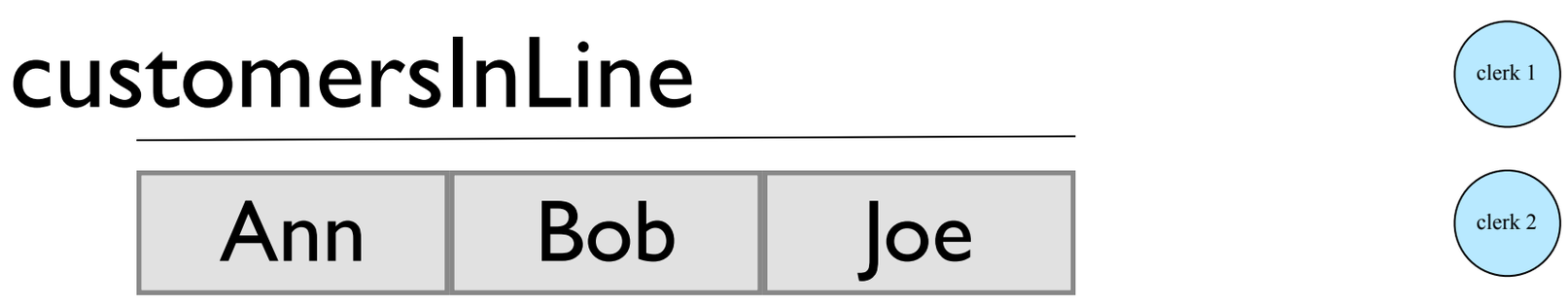




Queue

ADT Queue

- FIFO : first in first out
- useful whenever there is contention for a resource and “jobs” have to wait
  - packets in a router
  - files waiting to be printed
  - calls to a large company are placed in a queue when the operators are busy
- Simulation of “real world” queues:
  - customers in a store, bank, etc
  - airplanes waiting for a runway



Queue

There are many possible ways to implement the ADT queue

```
#include <queue>
int main ()
{
   queue<string> customersInLine;

   customersInLine.push("Joe");
   customersInLine.push("Bob");
   customersInLine.push("Ann");

   cout<< "# customers waiting is "<<    customersInLine.size()<<endl;
   cout<< "Next customer: "<<  customersInLine.front()<<endl;

    customersInLine.pop();

    cout<< "# customers waiting "<<    customersInLine.size()<<endl;
    cout<< "Next customer: "<<  customersInLine.front()<<endl;

    customersInLine.pop();

    customersInLine.push("Adam");

    customersInLine.pop();
    customersInLine.pop();
```

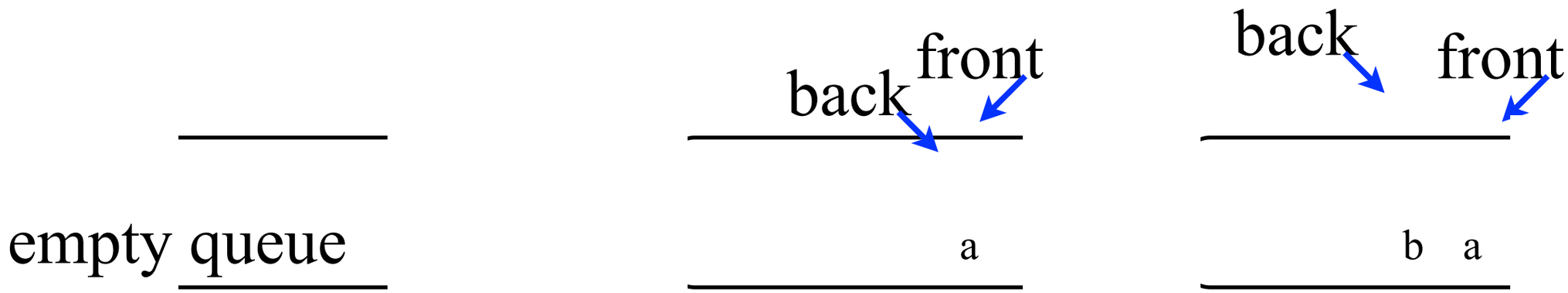
customersInLine

Ann	Bob	Joe

Queue Interface:

```
template< class Object >
class
Queue
{
private:
    ....
public:
    Queue( );

    bool empty( ) const;

    const Object& front( )const;
    const Object& back( )const;

    void push ( const Object& x ); // should be enqueue
    void push ( Object && x); // should be enqueue
    void pop( ); // should be dequeue
};
```

NOT a complete class!



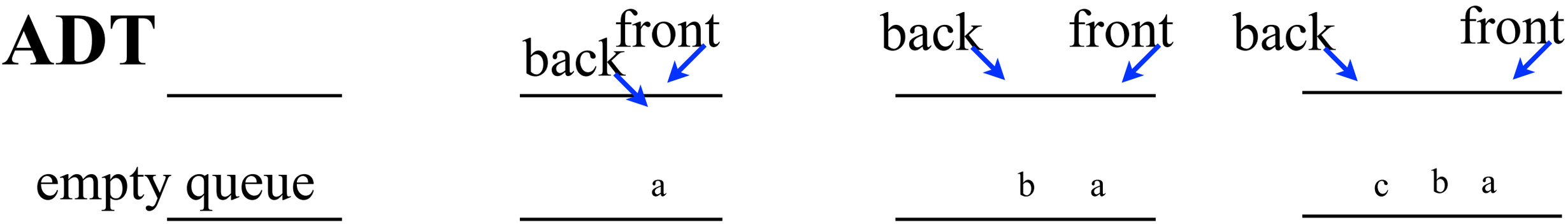
How to Implement a Queue?

Requirement: Constant Time Operations

Thinking about ADT queue

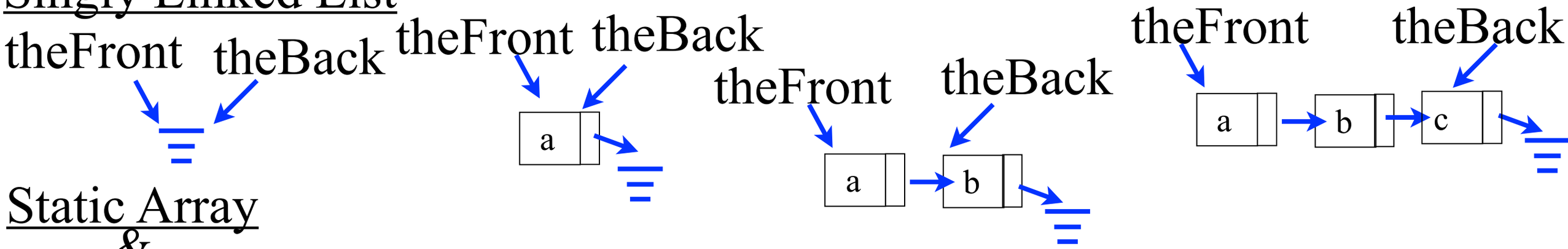
enqueue(a)
enqueue(b)
enqueue(c)
dequeue()
dequeue()
dequeue()

ADT

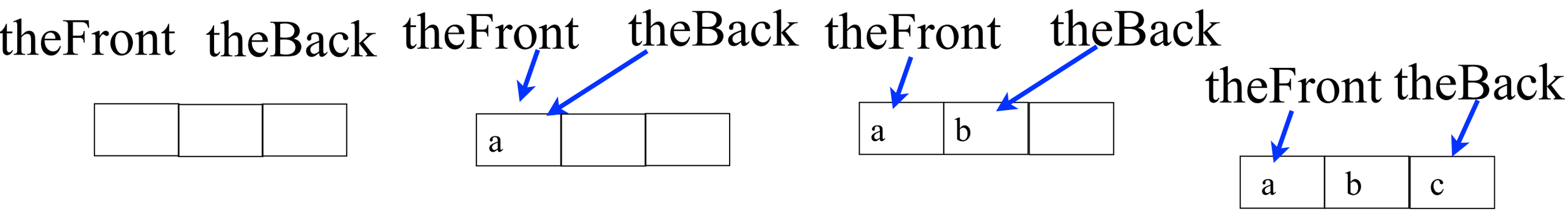


Conceptual Representation

<u>Singly Linked List</u>

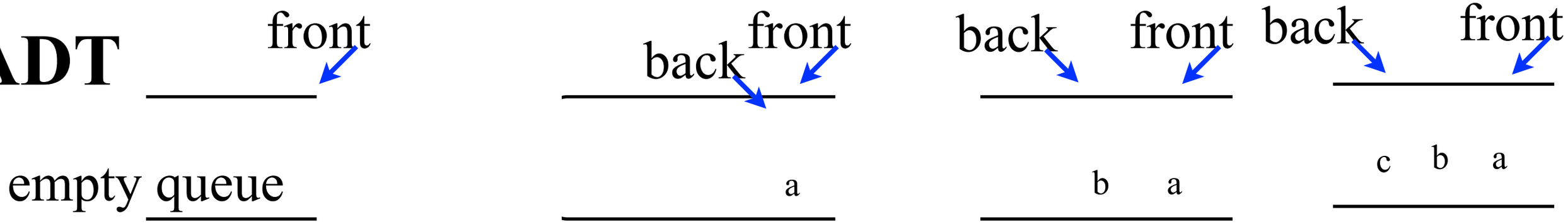


<u>Static Array</u>
&
<u>Dynamic Array</u>



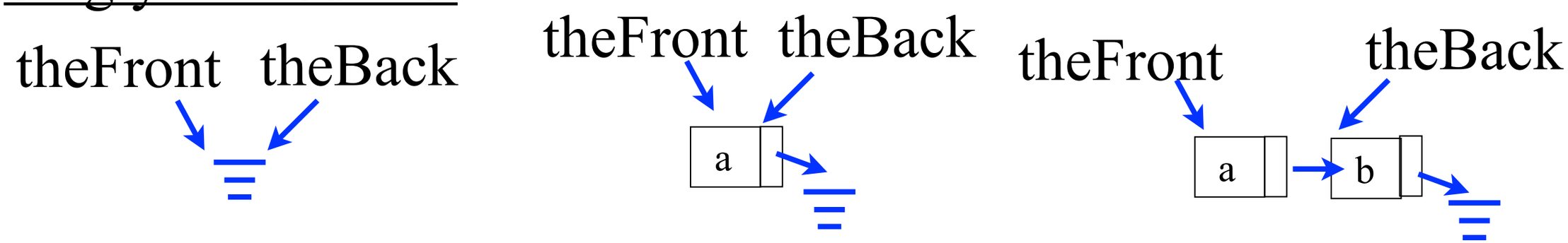
How can we use an array?

ADT

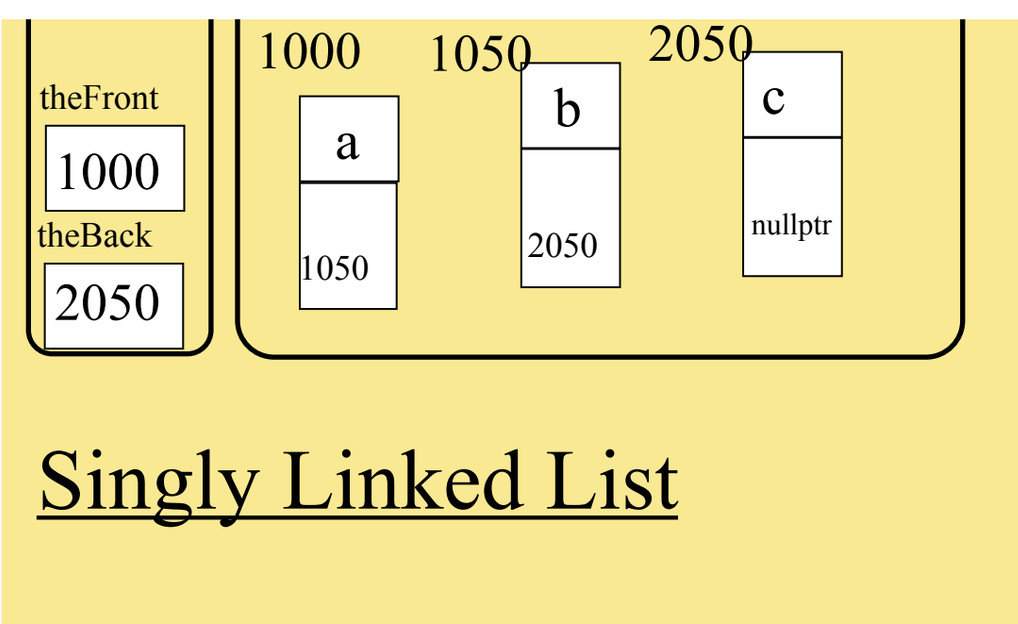


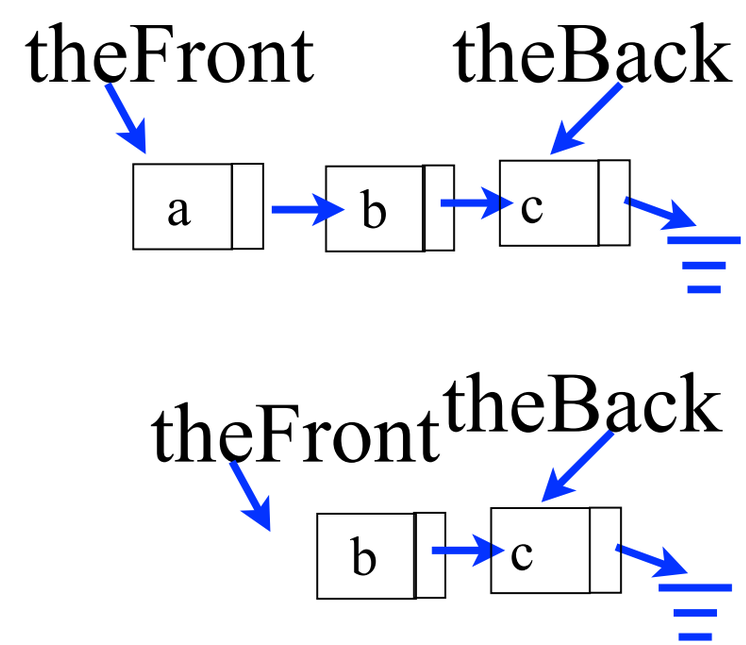
Conceptual Representation

Singly Linked List



Memory Level





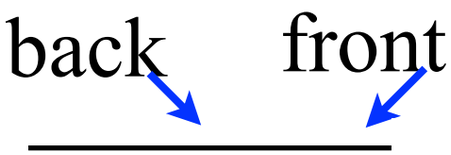
Queue Implementation 1: linked list

- data members:

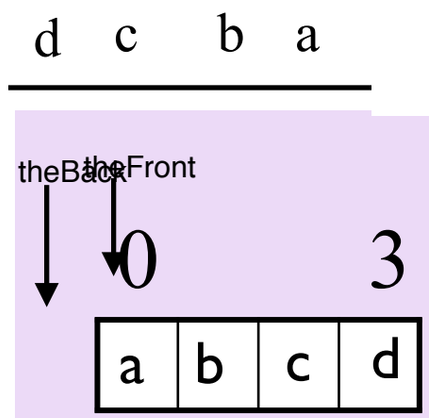
```
ListNode *theFront;
ListNode *theBack;
```

- representation invariant: queue elements are stored in the list with front of queue at front of list and back of queue at end of list
- Operations
  - push(enqueue): insert new node x after back
  - pop(dequeue): delete front element

Queue Implementation 2: Array



Array Implementation



- data members:

  Object theArray[MAX];

  int theFront;

  int theBack;

  int currentSize;

- initial idea
  - theArray[++theBack] = x //to enqueue x
  - theFront++ // to dequeue

- Problem:
  - sequence of enqueues and dequeues shifts “acti
    part of the array to the right. Can hit MAX with few
    elements actually in the queue
- Solution: Circular Array
  - increment mod MAX to “wrap around”

Queue Implementation 3: Dynamic Array

Circular Array Implementation

- data members:

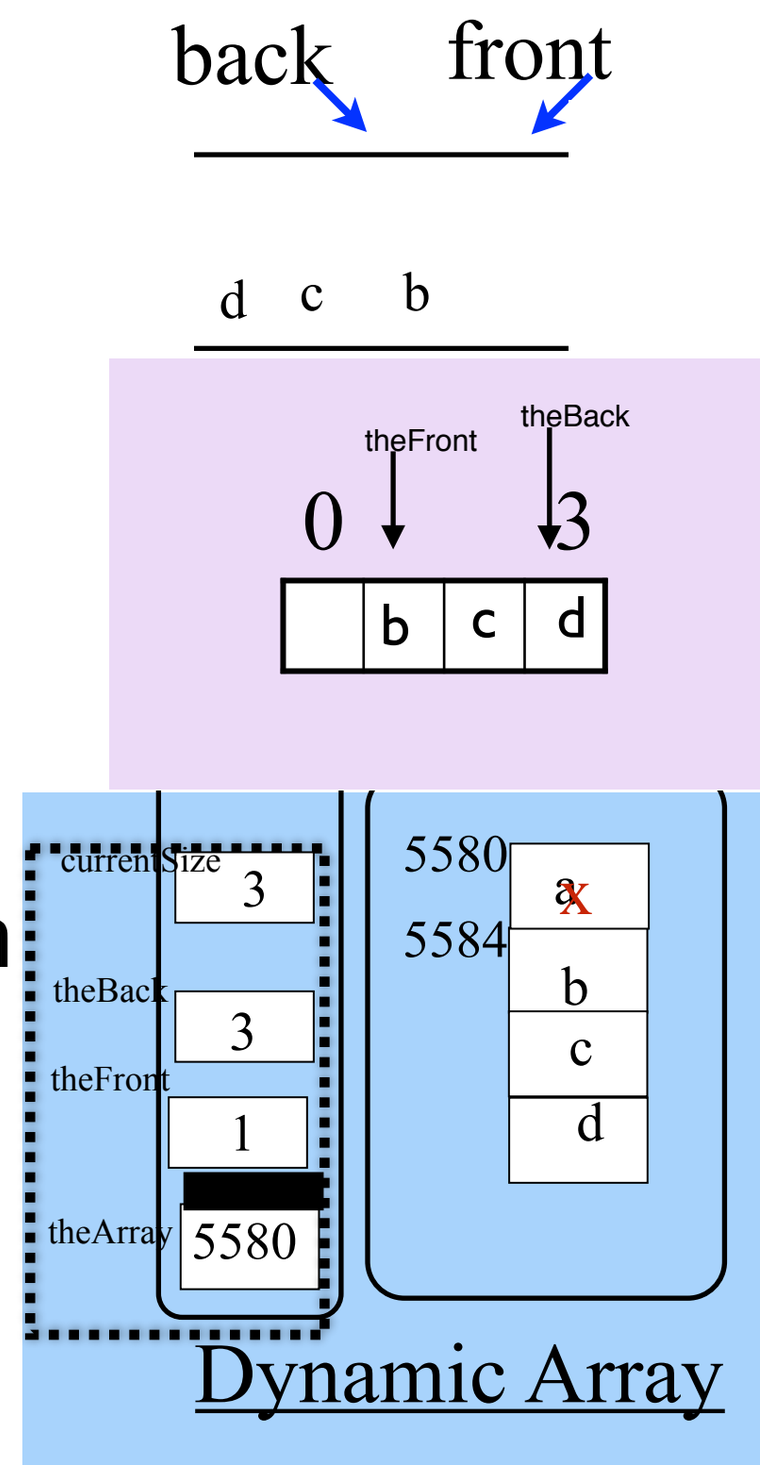
  vector<T> theArray;

  int theFront;

  int theBack;

  int currentSize;

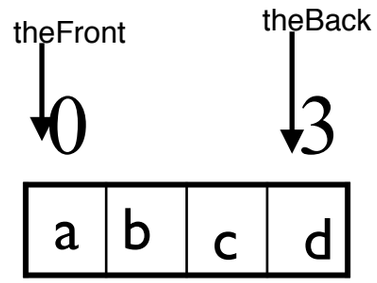
- Operations same as Static Array version, except that the vector is doubled when necessary to prevent overflow.
- Each operation is O(1), expect push() when array doubling is done. This is rare, so "amortized time" is O(1)



The Queue class

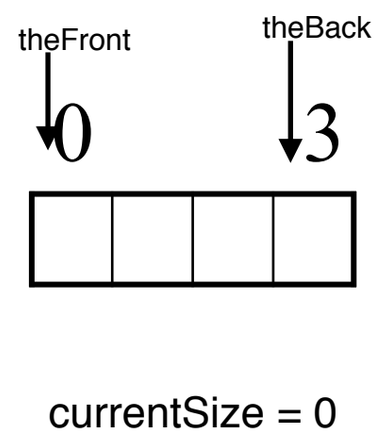
This code has been modified from the book code to be closer to the c++ implementation of the queue class

```
template <class Object>
class Queue
{
   public:
     Queue( );

     bool empty( ) const;
     const Object & front( ) const;
     const Object & back( ) const;

     void pop( );  // dequeue
     void push( const Object & x ); // enqueue
     void push( Object && x );   // enqueue
  private:
     vector<Object> theArray;
     int            currentSize;
     int            theFront;
     int            theBack;

     void makeEmpty( );
     void increment( int & y ) const;
     void doubleQueue( );
};
```



Constructor

```
// Construct the queue.
template <class Object>
Queue<Object>::Queue( ) : theArray( 4 )
{
    makeEmpty( );
}
```

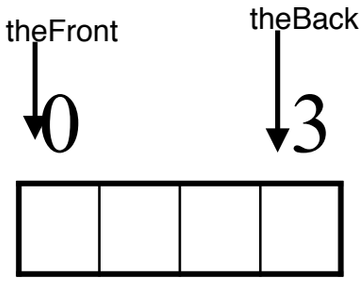



```
// Make the queue logically empty.
template <class Object>
void Queue<Object>::makeEmpty( )
{
    currentSize = 0;
    theFront = 0;
    theBack = theArray.size( ) - 1;
}
```

```
template <class Object>
class Queue
{
  public:
    Queue( );

    bool empty( ) const;
    const Object & front( ) const;
    const Object & back( ) const;

    void pop( );
    void push( const Object & x );
    void push( Object && x );
  private:
    vector<Object> theArray;
    int            currentSize;
    int            theFront;
    int            theBack;

    void makeEmpty( );
    void increment( int & y ) const;
    void doubleQueue( );
};
```

empty

```
// Test if the queue is logically empty.
// Return true if empty, false, otherwise.
template <class Object>
bool Queue<Object>::empty( ) const
{
        return currentSize == 0;
}
```



currentSize = 0

```
template <class Object>
class Queue
{
   public:
     Queue( );

     bool empty( ) const;
     const Object & front( ) const;
     const Object & back( ) const;

     void pop( );
     void push( const Object & x );
     void push( Object && x );
  private:
     vector<Object> theArray;
     int            currentSize;
     int            theFront;
     int            theBack;

     void makeEmpty( );
     void increment( int & y ) const;
     void doubleQueue( );
};
```

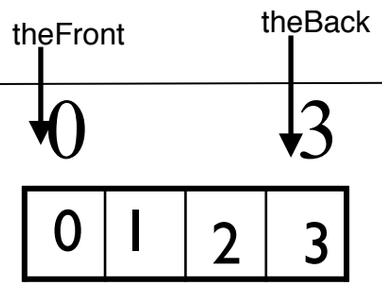
How would we write push(Object && x)

enqueue/push

```
// Insert x into the queue.
template <class Object>
void Queue<Object>::push( const Object & x )    // enqueue
{
   if( currentSize == theArray.size( ) )
      doubleQueue( );
   increment( theBack );
   theArray[ theBack ] = x;
   currentSize++;
}

// Internal method to increment x with wraparound.
template <class Object>
void Queue<Object>::increment( int & y ) const
{
   if( ++y == theArray.size( ) )
      y = 0;
}
```

```
template <class Object>
class Queue
{
   public:
     Queue( );

     bool empty( ) const;
     const Object & front( ) const;
     const Object & back( ) const;

     void pop( );
     void push( const Object & x );
     void push( Object && x );
  private:
     vector<Object> theArray;
     int            currentSize;
     int            theFront;
     int            theBack;

     void makeEmpty( );
     void increment( int & y ) const;
     void doubleQueue( );
};
```

```
Queue<int> q;
for( int i = 0; i < 4; i++ )
   q.push( i );
```

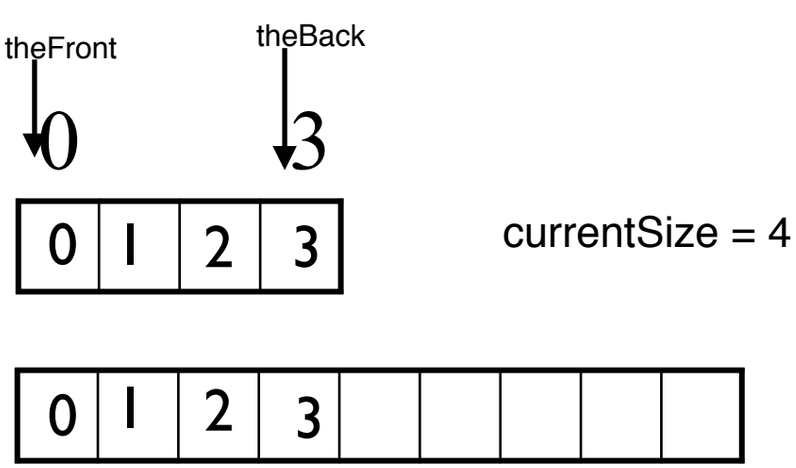


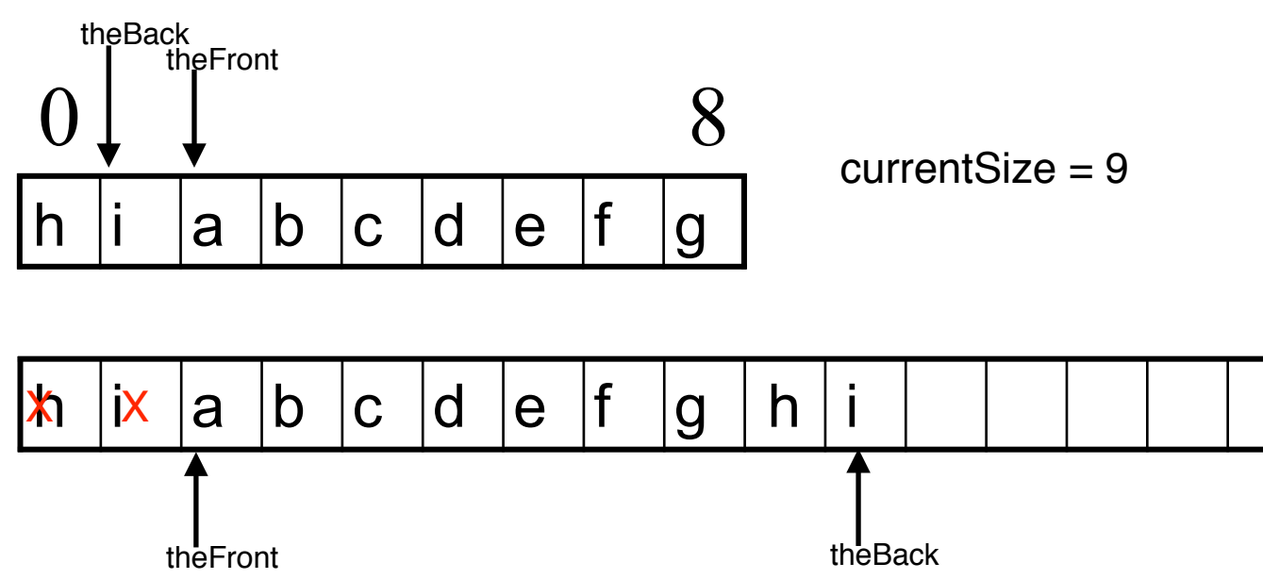
enqueue/push

```
// Insert x into the queue.
template <class Object>
void Queue<Object>::push( Object && x )          // enqueue
{
   if( currentSize == theArray.size( ) )
       doubleQueue( );
   increment( theBack );
   theArray[ theBack ] = std::move( x );
   currentSize++;
}

// Internal method to increment x with wraparound.
template <class Object>
void Queue<Object>::increment( int & y ) const
{
    if( ++y == theArray.size( ) )
        y = 0;
}
```

doubleQueue: Internal method to double capacity

```
// Internal method to double capacity.
template <class Object>
void Queue<Object>::doubleQueue( )
{
    theArray.resize( theArray.size( ) * 2 + 1 );
    if( theFront != 0 )
    {
        for( int i = 0; i < theFront; i++ )
            theArray[ i + currentSize ] = std::move( theArray[ i ] );
        theBack += currentSize;
    }
}
```



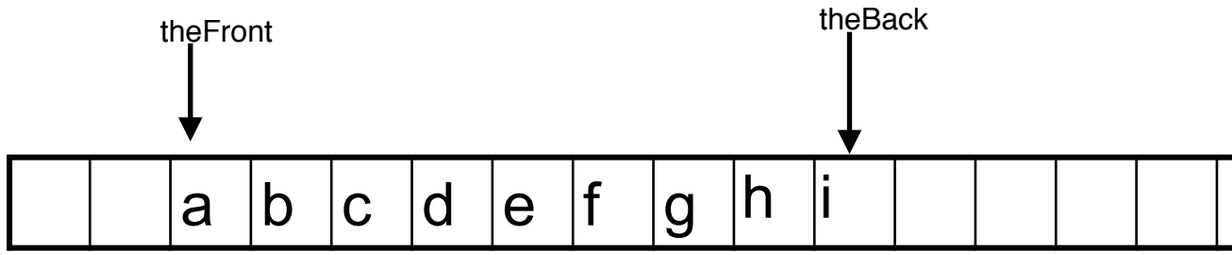


front

```
// Return the least recently inserted item in the queue
// or throw UnderflowException if empty.
template <class Object>
const Object & Queue<Object>::front( ) const
{

    if( empty( ) )
        throw underflow_exception("Stack is empty" );
    return theArray[ theFront ];
}
```

---

q.front( );

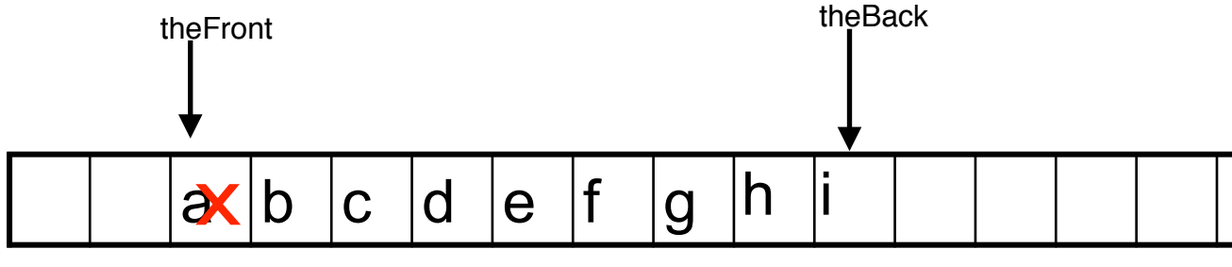


dequeue/pop

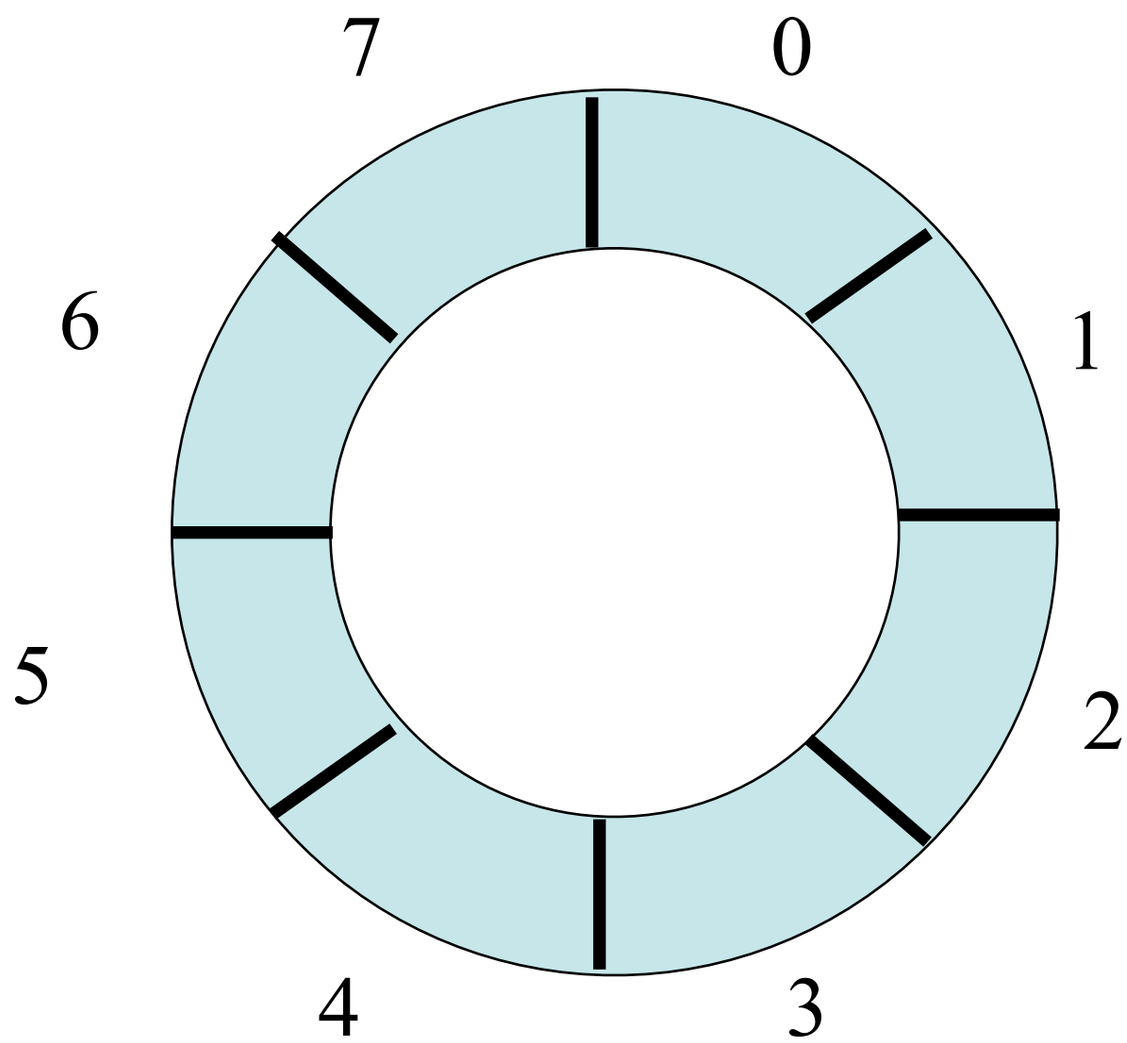
```
// Return and remove the least recently inserted item from the queue.
// Throws UnderflowException if queue is already empty.
template <class Object>
void Queue<Object>::pop( )    // dequeue
{
    if( empty( ) )
        throw underflow_exception("Stack is empty" );

    currentSize--;
    increment( theFront );
    return;
}
```

---

q.pop();



Abstract Data Type: Queue



<u>Operations</u>

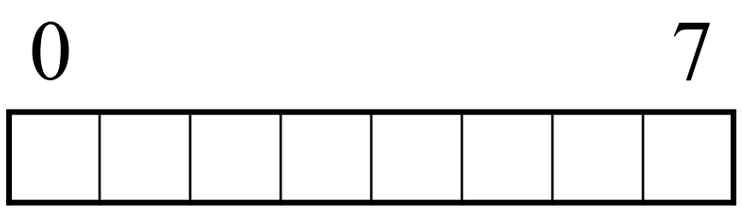
queue<char> s;

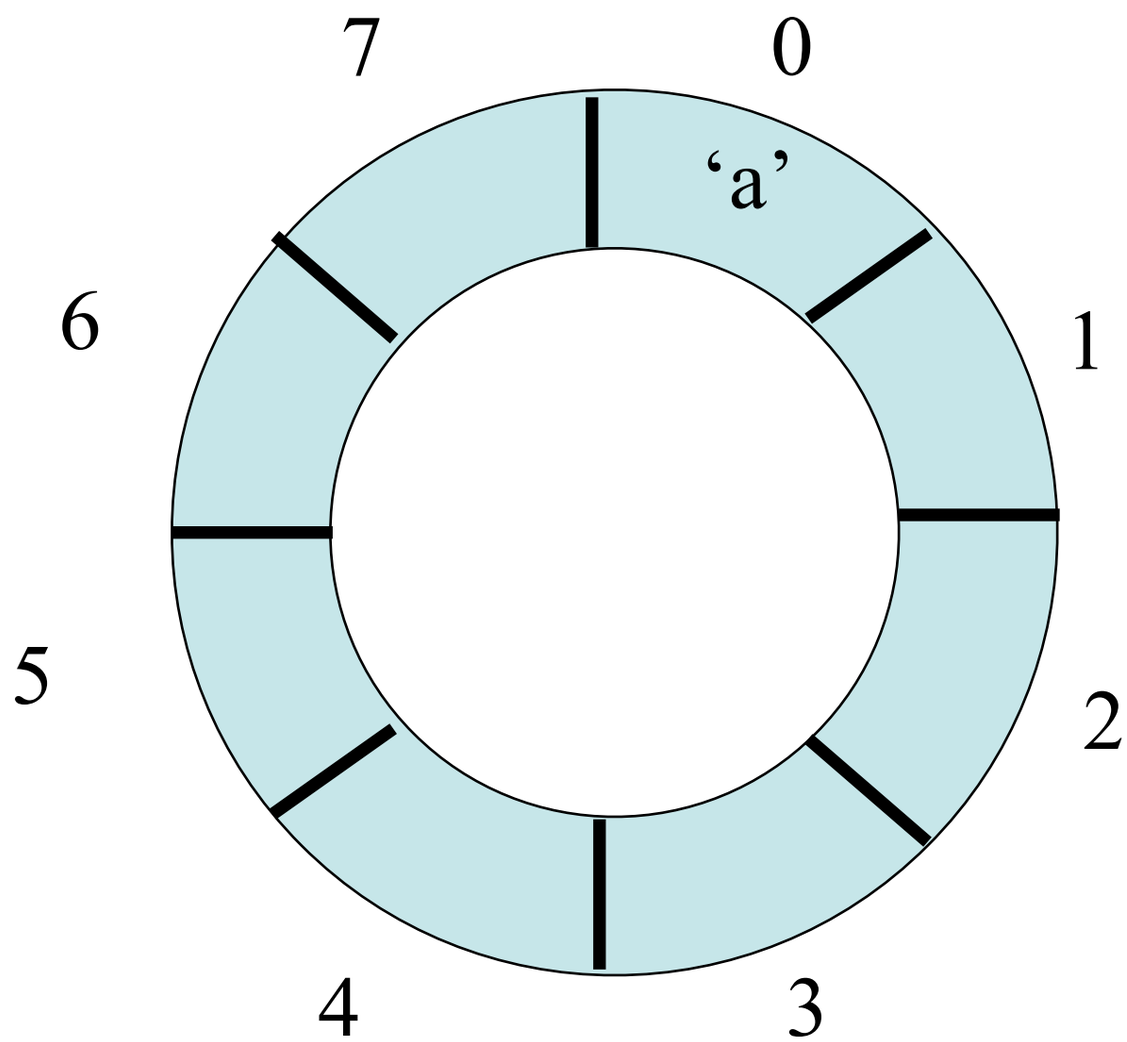
<u>The values</u>

theFront = 0

theBack = 7

current size = 0;





<u>Operations</u>

queue<char> s;

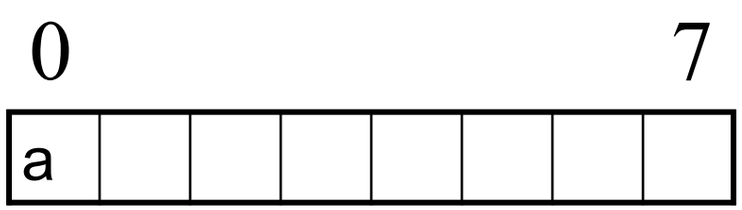
s.push('a');

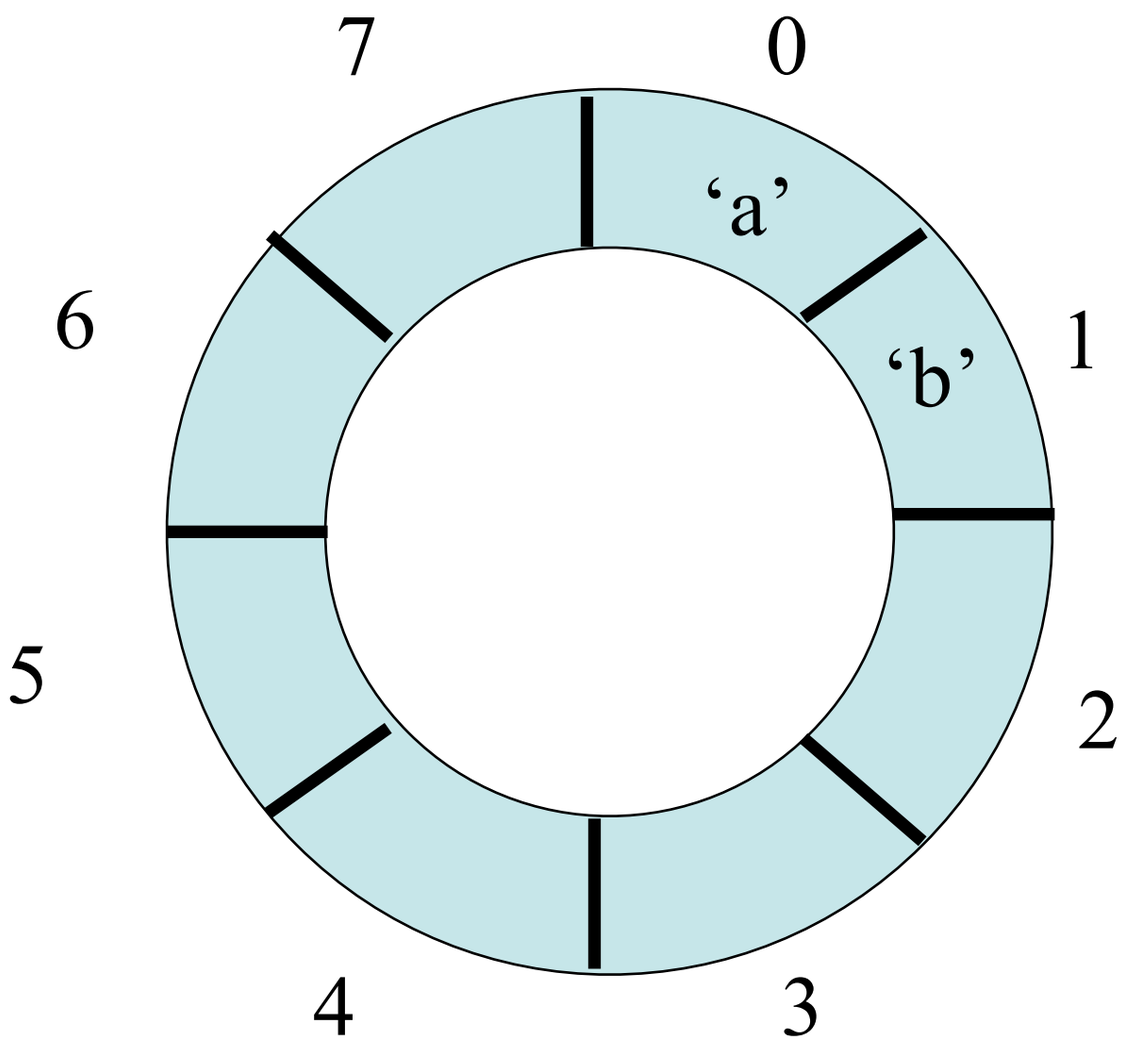
<u>The values</u>

theFront = 0

theBack = 0

current size = 1;



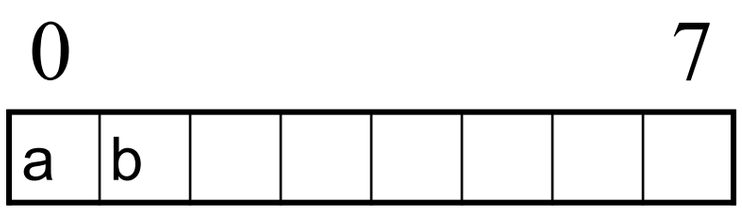


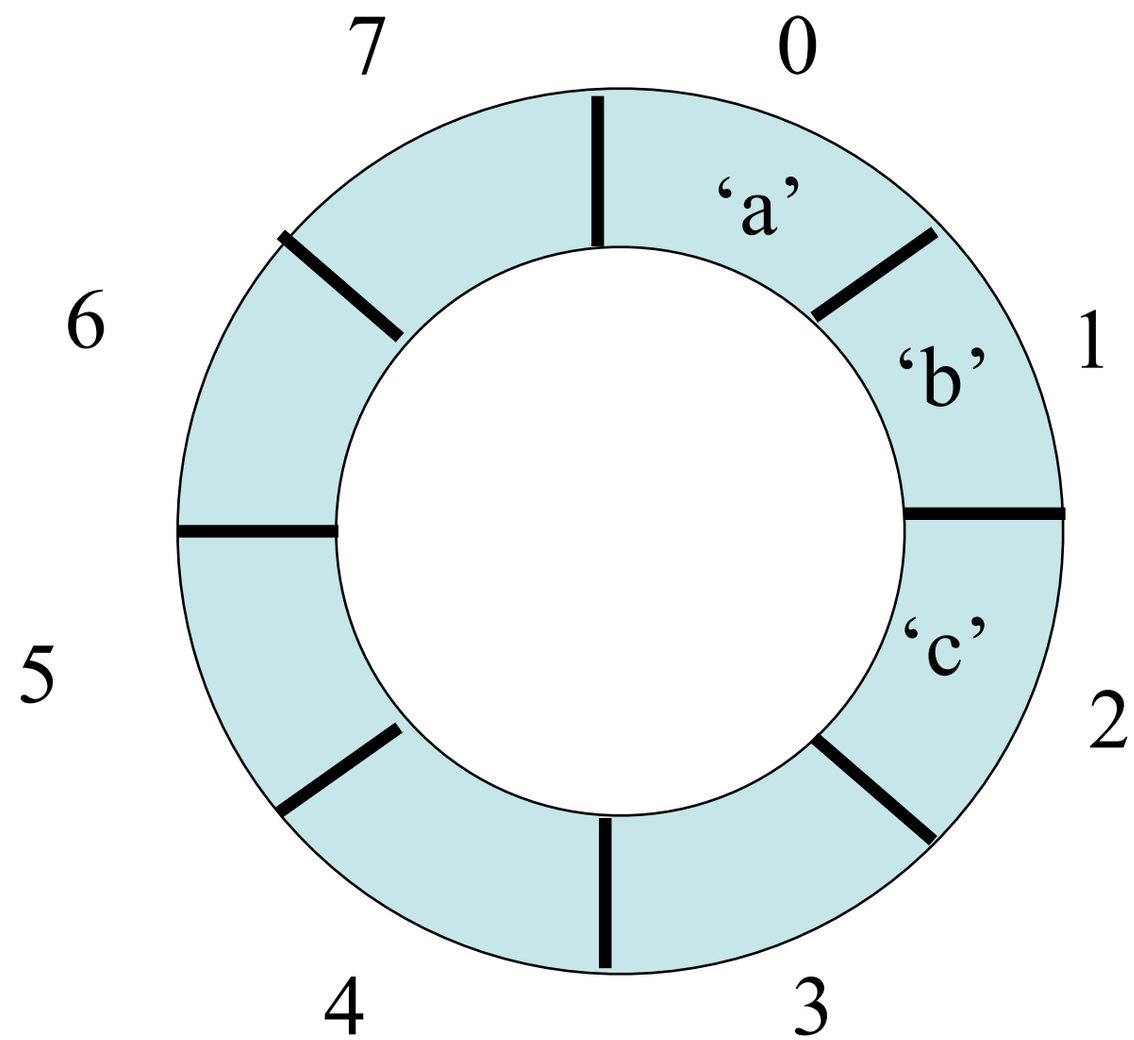
<u>Operations</u>

```
queue<char>  s;
s.push('a');
s.push('b');
```

<u>The values</u>

theFront = 0
theBack = 1
current size = 2;





<u>Operations</u>

queue<char> s;

s.push('a');

s.push('b');

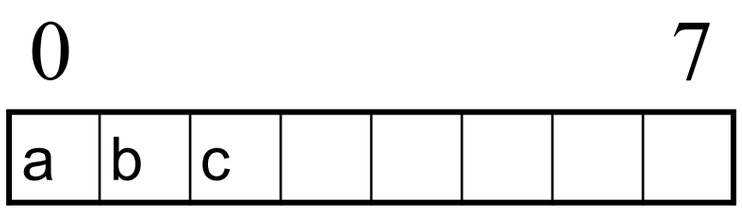
s.push('c');

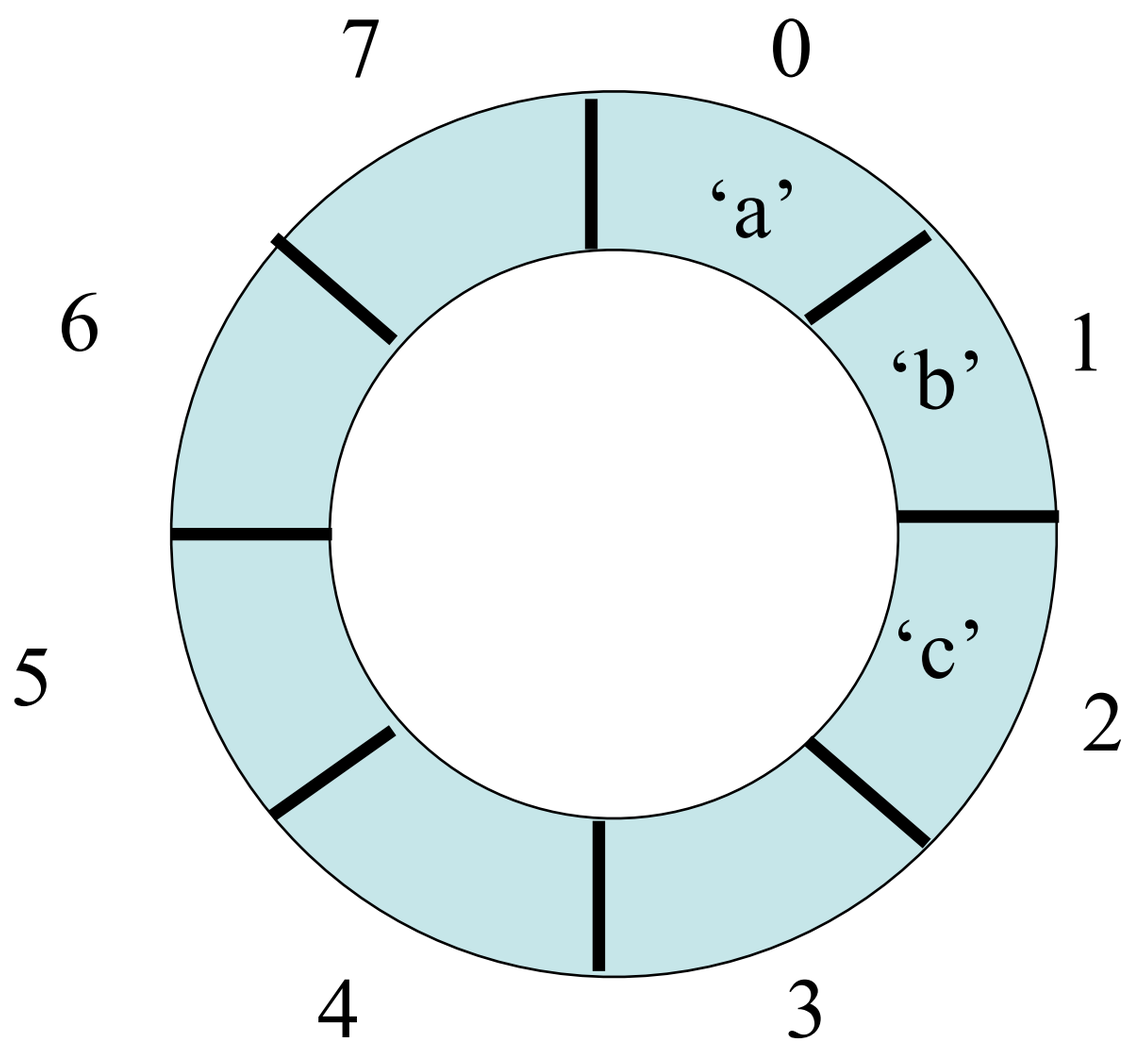
<u>The values</u>

theFront = 0

theBack = 2

current size = 3;





<u>Operations</u>

queue<char> s;

s.push('a');

s.push('b');

s.push('c');

char x = s.front( );

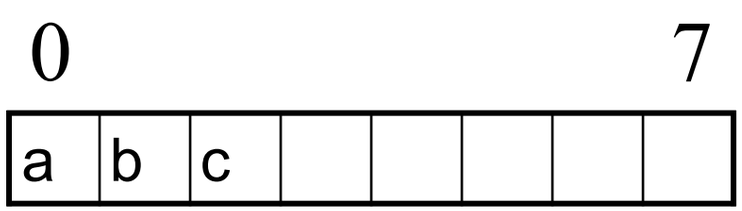
<u>The values</u>

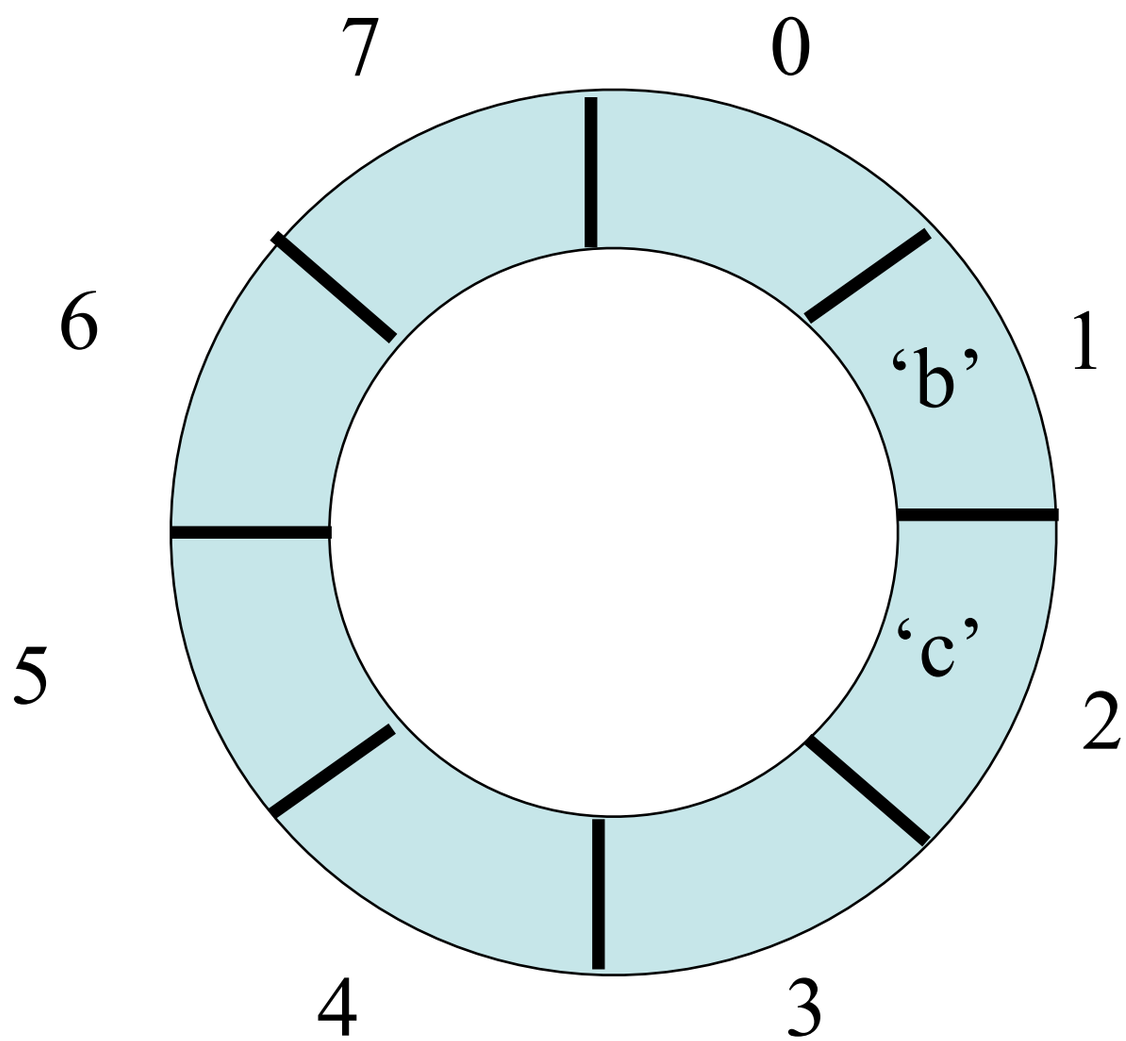
x = 'a'

front = 0

back = 2

current size = 3;





<u>Operations</u>

queue<char> s;

s.push('a');

s.push('b');

s.push('c');

char x = s.front( );

s.pop( );

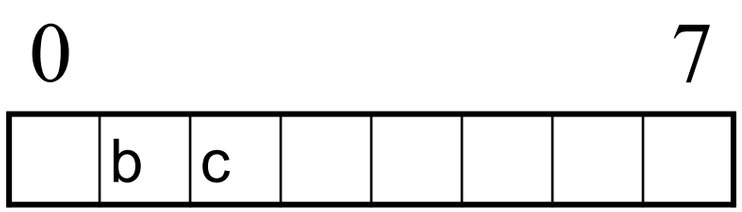
<u>The values</u>

x = 'a'

theFront = 1

theBack = 2

current size = 2;



STL Queue

elements pushed onto back and pop off front

#include<queue>

- bool empty(); O(1)
- size_t size(); O(1)
- T& front(); O(1)
- void push(const T& x); O(1)*
- void pop(); O(1)
- T& back(); O(1)

A queue is used in multiprocessing systems.
e.g. choosing which job to process next for a printer
A queue is used in simulations

*depending on the implementation, might be amortized O(1)