AST20105 Data Structures & Algorithms

CHAPTER 5 – STACKS AND QUEUES

Instructed by Garret Lai

Stacks

Stacks

- A stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data.

- New trays are put on the top of the stack and taken off the top.

- The last tray put on the stack is the first tray remove from the stack.
  - For this reason, a stack is called an LIFO structure:
    - Last in/first out

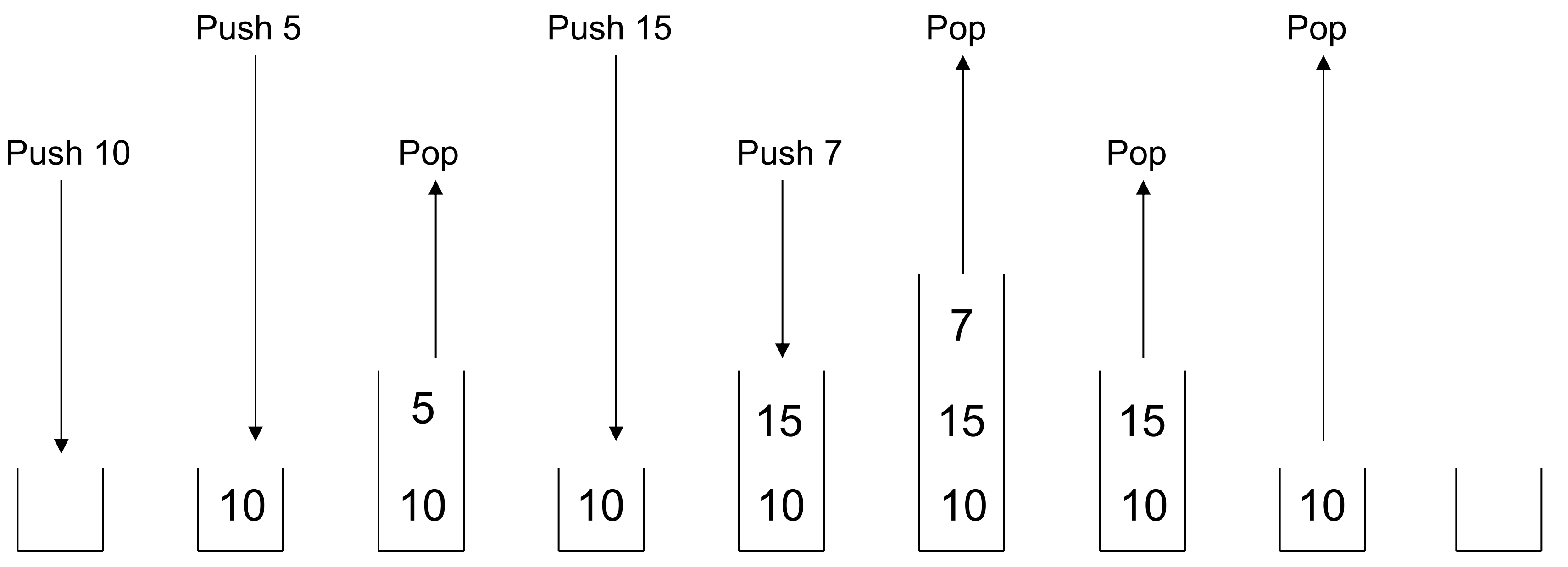
Stacks

- A tray can be taken only if there are trays on the stack
- A tray can be added to the stack only if there is enough room;
  - That is, if the stack is not too high.

Fundamental Operations of Stack

- Fundamental operations:
  - **Push**:
    Insert an element to the top of the stack
  - **Pop**:
    Delete an element from the top of the stack
    (i.e. delete the mostly recently inserted element from the stack)
  - **Top**:
    Examine the element at the top of the stack
    (i.e. examine the most recently inserted element of the stack)

Stacks


Push 10
Push 5
10
Pop
5
10
Push 15
10
Push 7
15
10
Pop
7
15
10
Pop
15
10
Pop
10

Stacks

- The stack is very useful in situations when data have to be stored and then retrieved in reverse order.

Stacks

- One application of the stack is matching delimiters in a program.
  - This is an important example because delimiter matching is part of any compiler:
    - No program is considered correct if the delimiters are mismatched.

Stacks

- In C++ programs, we have the following delimiters:
  - Parentheses “(” and “)”
  - Square brackets “[” and “]”
  - Curly brackets “{” and “}”
  - Comment delimiters “/*” and “*/”

Stacks

- The following examples are statements in which mismatching occurs:
  - a = b + (c - d) * (e - f));
  - g[10] = h[i[9]] + j + k) * l;
  - while (m < (n[8) + o]) { p = 7; /* initialize p */ r = 6; }

Stacks

- A particular delimiter can be separated from its match by other delimiters; that is delimiters can be nested.
  - E.g.
    - while (m < (n[8] + o))

Stacks

- The delimiter matching algorithm:

  - Reads a character from a C++ program and stores it on a stack if it is an opening delimiter.

  - If a closing delimiter is found, the delimiter is compared to a delimiter popped off the stack.

  - If they match, processing continues;

  - If not, processing discontinues by signaling an error.

Stacks

- The delimiter matching algorithm:

  - The processing of the C++ program ends successfully after the end of the program is reached and
    the stack is empty.

Stacks

- As another example of stack application, consider adding very large numbers.
  - The largest magnitude of integers is limited.
  - So we are not able to add 18274364583929273748459595684373 and 8129498165026350236
  - Because integer variables cannot hold such large values, let alone their sum.

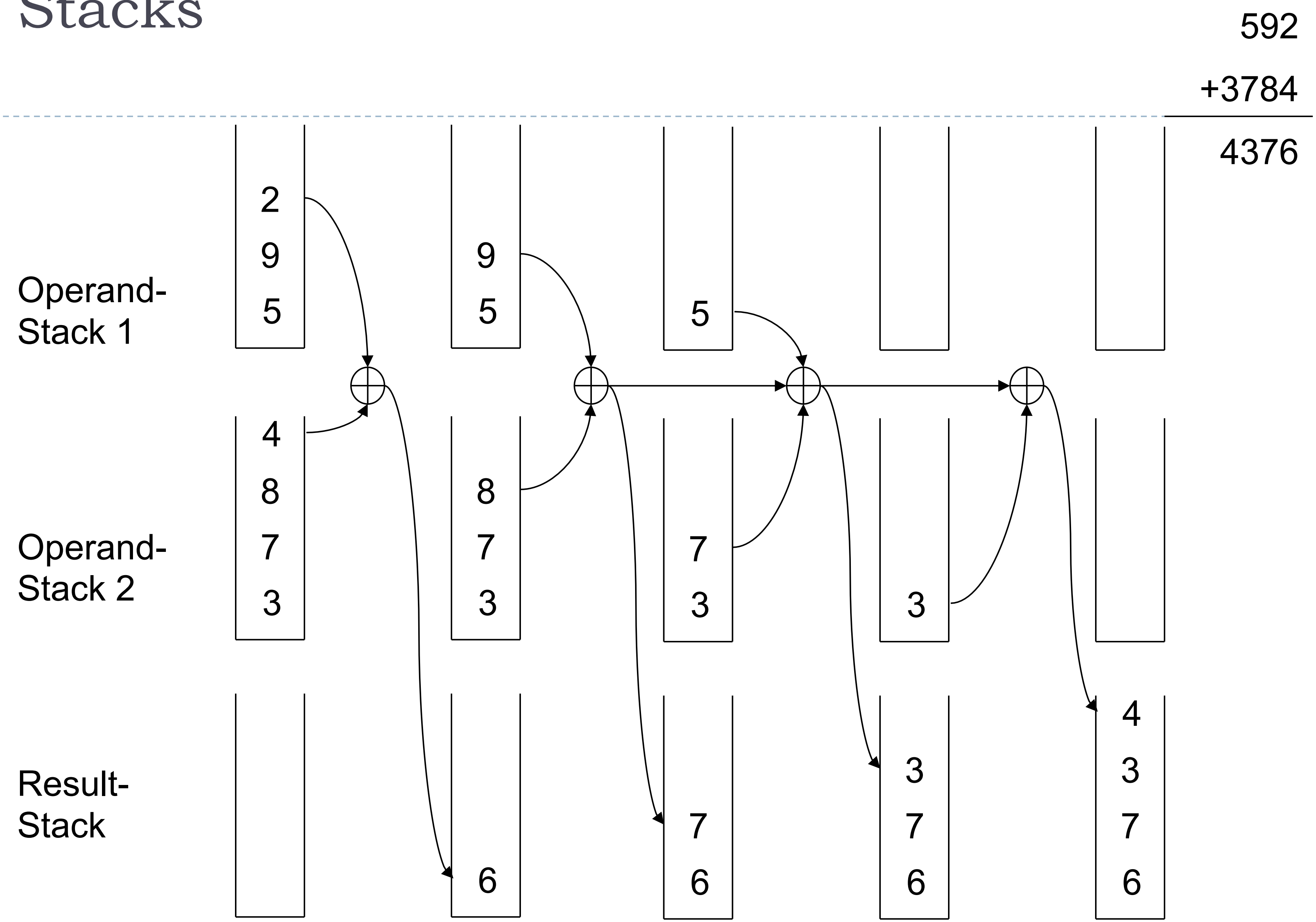
Stacks

- The problem can be solved if we treat these numbers as strings of numerals.

  - Store the numbers corresponding to these numerals on two stacks.

  - And then perform addition by popping numbers from the stacks.

Stacks



592
+3784
4376
Operand-
Stack 1
Operand-
Stack 2
Result-
Stack
2
9
5
9
5
5
4
8
7
3
8
7
3
7
3
3
6
7
6
3
7
6
4
3
7
6

Stack Implementation

- Stack can be implemented with array or linked list
  - Array:
    The size of the stack is fixed
  - Linked List:
    The size is flexible and will never be full

Stack Class using Array Implementation

StackArr.h

```
class StackArr
{
  private:
    int maxTop;
    int stackTop;
    double* values;
  public:
     StackArr(int size = 10);
     ~StackArr();
     bool isEmpty() const;
     bool isFull() const;
     double top() const;
     void push(const double& x);
     double pop();
     void displayStack() const;
};
```

```
StackArr::StackArr(int size)
{
   maxTop = size - 1;
   values = new double[size];
   stackTop = -1;
}

StackArr::~StackArr()
{
   delete [] values;
}

bool StackArr::isEmpty() const
{
   return stackTop == -1;
}

bool StackArr::isFull() const
{
   return stackTop == maxTop;
}

// ...
```

StackArr.cpp

Stack Operation: push

- void push(const double x)
  - Push an element onto the stack
  - If the stack is full, output error message
  - top is used to represent the index of the top element. After an element is pushed, increment top by 1

```
void StackArr::push(const double& x)
{
  if(isFull())
    cout << "Error! The stack is full." << endl;
  else
    values[++stackTop] = x;
}

// ...
```

StackArr.cpp

Stack Operation: pop

- double pop()
  - Pop and return the element at the top of the stack
  - If the stack is empty, output error message
  - After an element is popped, decrement top by 1

```
double StackArr::pop()
{
  if(isEmpty())
  {
    cout << "Error! The stack is empty." << endl;
    return -1;
  }
  else
    return values[stackTop--];
}

// ...
```

StackArr.cpp

Stack Operation: top

- double top()
  - Return the top element of the stack
  - Note: This function DOES NOT DELETE the top element

```
double StackArr::top() const
{
  if(isEmpty())
  {
    cout << "Error! The stack is empty." << endl;
    return -1;
  }
  else
    return values[stackTop];
}

// ...
```

StackArr.cpp

Stack Operation: displayStack

- void displayStack()
  - Print all the elements on screen

```
void StackArr::displayStack() const
{
   cout << "Top -->";
   for(int i=stackTop; i>=0; i--)
      cout << "\t|\t" << values[i] << "\t|" << endl;
   cout << "\t|---------------|"<< endl;
}
```

StackArr.cpp

Using Stack class using Array Implementation

```
#include <iostream>
#include "StackArr.h"
using namespace std;

int main()
{
   StackArr stack(5);
   stack.push(1.0);
   stack.push(2.1);
   stack.push(-2.5);
   stack.push(-9.0);
   stack.displayStack();
   cout << "Top: " << stack.top() << endl;
   stack.pop();
   cout << "Top: " << stack.top() << endl;
   while(!stack.isEmpty())
     stack.pop();
   return 0;
}
```

mainStackArr.cpp

Stack Class using Linked List Implementation

SinglyList.h

```
class SinglyList
{
   private:
      Node* head; // a pointer to the first node in the list
      friend class StackLL;
   public:
      SinglyList();      // constructor
      ~SinglyList();     // destructor
      // isEmpty determines whether the list is empty or not
      bool isEmpty();
      // insertNode inserts a new node at position "index"
      Node* insertNode(int index, double x);
      // findNode finds the position of the node with a given value
      int findNode(double x);
      // deleteNode deletes a node with a given value
      int deleteNode(double x);
      // displayList prints all the nodes in the list
      void displayList() const;
};
```

Stack Class using Linked List Implementation

```
class StackLL : public SinglyList
{
   public:
      StackLL();
      ~StackLL();
      double top() const;
      void push(const double& x);
      double pop();
      void displayStack() const;
};
```

StackLL.h

```
StackLL::StackLL()
{
}

StackLL::~StackLL()
{
}

// ...
```

StackLL.cpp

Stack Operations: top and push

```
double StackLL::top() const
{
    if(head == NULL)
    {
        cout << "Error: The stack is empty." << endl;
        return -1;
    }
    else
        return head->data;
}

void StackLL::push(const double& x)
{
    insertNode(0,x);
}
```

StackLL.cpp

Stack Operations: pop and displayStack

```
double StackLL::pop()
{
    if(head == NULL) {
        cout << "Error: The stack is empty." << endl;
        return -1;
    }
    else
    {
        double val = head->data;
        deleteNode(val);
        return val;
    }
}

void StackLL::displayStack() const
{
    displayList();
}
```

StackLL.cpp

Queues

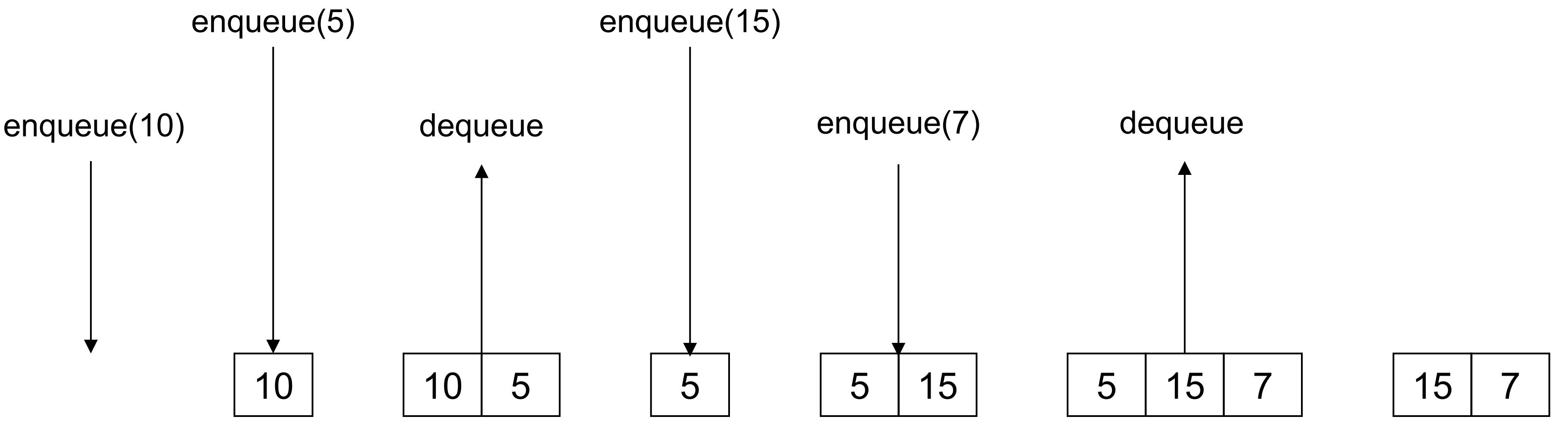
Queues

- A queue is simply a waiting line that grows by adding elements to its end and shrinks by taking elements from its front.
- Unlike a stack, a queue is a structure in which both ends are used:
  - One for adding new elements and
  - One for removing them.
- Therefore, the last element has to wait until all elements preceding it on the queue are removed.
- A queue is an FIFO structure: first in/first out.

Fundamental Operations of Queue

- Fundamental operations:
  - **enqueue**:
    Insert an element to the back of the list
  - **dequeue**:
    Delete an element at the front of the list

Queues


enqueue(5)
enqueue(15)
enqueue(10)
dequeue
enqueue(7)
dequeue
10
10
5
5
5
15
5
15
7
15
7

Queue Implementation

- Queue can also be implemented with array or linked list
  - Array:
    The size of the queue is fixed
  - Linked List:
    The size is flexible and will never be full

Queue Implementation using Circular Array

- Idea:
  - When an item is inserted using enqueue, make the back index move forward
  - When an item is deleted using dequeue, the front index moves by one element towards the back of the queue
  - When an element moves past the end of a circular array, it wraps around to the beginning

Initial

							6	9	12
							Front		Back

Enqueue(2)

2							6	9	12
Back							Front		

Dequeue

2							6	9	12
Back								Front	

Queue Class using Array Implementation

QueueArr.h

```
class QueueArr
{
  private:
    int front;
    int back;
    int counter;
    int maxSize;
    double* values;
  public:
     QueueArr(int size = 10);
     ~QueueArr();
     bool isEmpty() const;
     bool isFull() const;
     bool enqueue(double x);
     bool dequeue(double& x);
     void displayQueue() const;
};
```

```
QueueArr::QueueArr(int size) {
   values = new double[size];
   maxSize = size;
   front = 0;
   back = -1;
   counter = 0;
}

QueueArr::~QueueArr() {
   delete [] values;
}

bool QueueArr::isEmpty() const {
   if(counter) return false;
   else return true;
}

bool QueueArr::isFull() const {
   if(counter < maxSize) return false;
   else return true;
}

// ...
```

QueueArr.cpp

Queue Operation: enqueue

QueueArr.cpp

```
bool QueueArr::enqueue(double x)
{
  if(isFull()) {
    cout << "Error! The queue is full." << endl;
    return false;
  }
  else {
    back = (back + 1) % maxSize;
    values[back] = x;
    counter++;
    return true;
  }
}

// ...
```

Queue Operation: dequeue

QueueArr.cpp

```
bool QueueArr::dequeue(double& x)
{
  if(isEmpty()) {
    cout << "Error! The queue is empty." << endl;
    return false;
  }
  else {
    x = values[front];
    front = (front + 1) % maxSize;
    counter--;
    return true;
  }
}

// ...
```

Queue Operation: displayQueue

```cpp
void QueueArr::displayQueue()
{
   cout << "Front -->";
   for(int i=0; i<counter; i++)
   {
      if(i == 0)
         cout << "\t";
      else
         cout << "\t\t";
      cout << values[(front + i) % maxSize];
      if(i != counter - 1)
         cout << endl;
      else
         cout << "\t<--Back" << endl;
   }
}
```

QueueArr.cpp

Using Queue class using Array Implementation

```cpp
#include <iostream>
#include "QueueArr.h"
using namespace std;

int main()
{
    QueueArr queue(5);
    cout << "Enqueue 5 elements" << endl;
    for(int i=0; i<5; i++)
        queue.enqueue(i);
    queue.enqueue(5);
    queue.displayQueue();
    double value;
    queue.dequeue(value);
    cout << "Obtained element = " << value << endl;
    queue.displayQueue();
    queue.enqueue(7);
    queue.displayQueue();
    return 0;
}
```

mainQueueArr.cpp

Queue Class using Linked List Implementation

QueueLL.h

```
class QueueLL
{
   private:
      Node* front;
      Node* back;
      int counter;
   public:
     QueueLL();
     ~QueueLL();
     bool isEmpty() const;
     void enqueue(double x);
     bool dequeue(double& x);
     void displayQueue() const;
};
```

```
QueueLL::QueueLL()
{
   front = back = NULL;
   counter = 0;
}

QueueLL::~QueueLL()
{
   double value;
   while(!isEmpty())
      dequeue(value);
}

bool QueueLL::isEmpty() const
{
   if(counter) return false;
   else return true;
}

// ...
```

QueueLL.cpp

Queue Operation: enqueue

```
void QueueLL::enqueue(double x)
{
   Node* newNode = new Node;
   newNode->data = x;
   newNode->next = NULL;
   if(isEmpty())
   {
      front = newNode;
      back = newNode;
   }
   else
   {
      back->next = newNode;
      back = newNode;
   }
   counter++;
}

// ...
```

QueueLL.cpp

Queue Operation: dequeue

```
bool QueueLL::dequeue(double& x)
{
    if(isEmpty())
    {
        cout << "Error: The queue is empty." << endl;
        return false;
    }
    else
    {
        x = front->data;
        Node* nextNode = front->next;
        delete front;
        front = nextNode;
        counter--;
    }
}

// ...
```

QueueLL.cpp

Queue Operation: displayQueue

```
void QueueLL::displayQueue() const
{
   cout << "Front -->";
   Node* currNode = front;
   for(int i=0; i<counter; i++)
   {
      if(i == 0)
         cout << "\t";
      else
         cout << "\t\t";
      cout << currNode->data;
      if(i != counter - 1)
         cout << endl;
      else
         cout << "\t<--Back" << endl;
      currNode = currNode->next;
   }
}
```

QueueLL.cpp

Priority Queues

Priority Queues

- In many situations, simple queues are inadequate.
- Because first in/first out scheduling has to be overruled using some priority criteria.

Priority Queues

- In a post office example, a handicapped person may have priority over others.
  - Therefore, when a clerk is available, a handicapped person is served instead of someone from the front of the queue.

Priority Queues

- On roads with tollbooths, some vehicles may be put through immediately, even without paying
  - police cars,
  - ambulances,
  - fire engines,
  - and the like.

Priority Queues

- In a sequence of processes,
  - process P2 may need to be executed before process P1 for the proper functioning of a system,
    - even though P1 was put on the queue of waiting processes before P2.

Priority Queues

- In situations like these, a modified queue, or priority queue, is needed.
  - In priority queues, elements are dequeued according to their priority and their current queue position.

Priority Queues

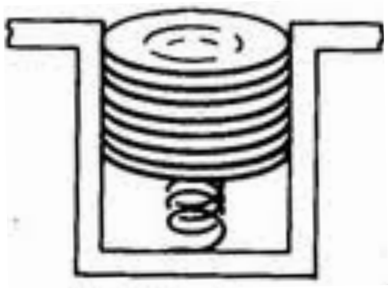
- The problem with a priority queue is in finding an efficient implementation that allows relatively fast enqueuing and dequeuing.
  - Because elements may arrive randomly to the queue,
  - There is no guarantee that
    - the front elements will be the most likely to be dequeued and
    - the elements put at the end will be the last candidates for dequeuing.

Priority Queues

- The situation is complicated because a wide spectrum of possible priority criteria can be used in different cases such as
  - Frequency of use,
  - Birthday,
  - Salary,
  - Position,
  - Status and
  - Others.

Stack and Queue Summary

- Stacks and queues are list that can handle a collection of elements, but with certain restrictions
  - **Stacks:**
    Element can only be inserted and deleted at one end (i.e. the top of the list)
    - Last inserted element will be the first to be examined or deleted
    - First inserted element will be the last to be examined or deleted)
  - **Queues:**
    Element can only be inserted at one end and be deleted at the other end
    - First element will be the first to be examined or deleted
    - Last element will be the last to be examined or deleted



Last in, First Out (LIFO)

First in, First Out (FIFO)

CHAPTER 5 END