



2007-03-01

Lecture #8

(Material on Test 2)

Today's Outline

Recap Course so Far

Stack & Queues

Copy Constructor, More Info

Estimate of the Course so far ...

1. C++ OOP	~80%	
2. Singly Linked Lists	~5%	\
3. Doubly Linked Lists	~5%	~20% Data Structures
4. Multi Lists	~5%	/
5. Stacks	~5%	

Estimate of the Remainder of the course

1. C++ OOP	~5%
2. Other Data Structures	~95%

We want to **deal more with the Data Structures** themselves now.
(Queues, Recursion, Trees, Heaps, Hash Tables, Graphs, etc.)

Recall a Linked List Implementation of a **Stack**
(uses singly linked list that ends in NULL)

(Chapter 4.6-4.14 of your book)

- 1. Constructor** (headptr = NULL)
- 2. Push** (insert before headptr)
- 3. Pop** (remove headptr)
- 4. IsEmpty** (if headptr == NULL)
- 5. Destructor** (loop thru and delete linked list)

LIFO = Last in First Out

Queue

(Chapter 5 of your book)

What is a **Queue**?

- linear list of elements
- inserted at one end (rear)
- deleted from the other end (front)

FIFO – first in first out

Examples of Queues

Ex1:) **Printer Queue**

- sent print jobs to printer
- first one to send the job is the first one to get printed

Ex2:) Checkout **Line** at a Convenience Store

- you get into line at the end (rear)
- you leave the line at the front
- first one in line is the first one out of the line

Ex3:) MP3 Player **Song List** (or WinAmp, or Media Player)

- you add songs to your song list, it puts them at the end (rear)
- songs at the front are done first
- first song in list is first one played

Comparison between the Stack & Queue Data Structures

- Both could be implemented using **Singly Linked Lists** that end in NULL (no size restrictions)
- Both could be implemented using **Arrays** (but then there's a size restriction)

Specs for LL Stack

```
class Stack{  
    private:  
        node * top ;  
    public:  
        void push(data) ;  
        data pop( ) ;  
};
```

Specs for LL Queue

```
class Queue{  
    private:  
        node * front ;  
        node * rear ;  
    public:  
        void enqueue(data) ;  
        data dequeue( ) ;  
};
```

LL Stack vs. LL Queue - Constructor

```
//LL Stack Constructor  
// Todo: draw in memory  
Stack::Stack( ){  
    this->top = NULL ;  
}
```

```
//LL Queue Constructor  
Queue::Queue() {  
    this->front = NULL ;  
    this->end = NULL ;  
}
```

LL Stack vs. LL Queue – Destructor

```
// LL Stack Destructor  
// Todo: draw in memory  
// Discussion: what if empty list? Will this work or seg fault?  
Stack::~~Stack() {  
    node * cur = top ;  
    node * prev = NULL ;  
    while(cur != NULL) {  
        prev = cur ;  
        cur = cur->next ;  
        delete prev ;  
    }  
}
```

LL Stack vs. LL Queue – Destructor

```
// LL Queue Destructor  
// Todo: draw in memory  
// Discussion: what if empty list? Will this work or seg fault?  
Queue::~~Queue(){  
    node * cur = front ;   node * prev = NULL ;  
    while(cur != NULL){  
        prev = cur ;  
        cur = cur->next ;  
        delete prev ;  
    }  
}
```

LL Stack vs. LL Queue – Push vs Enqueue

```
// LL Stack Push
// Todo: draw this in memory
// Discussion: what if empty list? Will this work or seg fault?
void Stack::Push(data x){
    node * newnode = new node( ) ;
    newnode->data = x ;
    newnode->next = this->top ;
    this->top = newnode ;
}
```

LL Stack vs. LL Queue – Push vs Enqueue

```
// LL Queue Enqueue
// Todo: draw in memory
// Discussion: what if empty list? Will this work or seg fault?
void Queue::Enqueue(data x){
    node * newnode = new node( ) ;
    newnode->data = x ;
    newnode->next = NULL ;
    end->next = newnode ;
    end = newnode ;
}
```

LL Stack vs. LL Queue – Pop vs Dequeue

// LL Stack Pop

// Todo: draw in memory

// Discussion: what if empty list? Will this work or seg fault?

```
data Stack::Pop(){  
    data output = top->data ;  
    node * oldtop = top ;  
    top = top->next ;  
    delete oldtop ;  
    return output ;  
}
```

LL Stack vs. LL Queue – Pop vs Dequeue

// LL Queue Dequeue

// Todo: draw in memory

// Discussion: what if empty list? Will this work or seg fault?

```
data Queue::Dequeue( ){  
    data output = front->data ;  
    node * oldfront = front ;  
    front = front->next  
    delete oldfront ;  
    return output ;  
}
```

LL Stack vs. LL Queue – Print

```
// LL Stack Print  
// Todo: draw in memory  
// Discussion: what if empty list? Will this work or seg fault?  
void Stack::Print(){  
    node * cur = top ;  
    while(cur != NULL){  
        cout << cur->data << endl ;  
        cur = cur->next ;  
    }  
}
```

LL Stack vs. LL Queue – Print

```
// LL Queue Print  
// Todo: draw in memory  
// Discussion: what if empty list? Will this work or seg fault?  
void Queue::Print() {  
    node * cur = front ;  
    while(cur != NULL) {  
        cout << cur->data << endl ;  
        cur = cur->next ;  
    }  
}
```

Array implementation of a Stack & Queue

Specs for Array Stack

```
class Stack{  
    private:  
        data * array ;  
        int topIndex ;  
        int size ;  
    public:  
        void push(data) ;  
        data pop( ) ;  
};
```

Specs for Circular Array Queue

```
class Queue{  
    private:  
        data * array ;  
        int frontIndex ;  
        int rearIndex ;  
        int size ;  
    public:  
        void enqueue(data) ;  
        data dequeue( ) ;  
};
```

Array Stack vs. Circular Array Queue - Constructor

```
//Array Stack Constructor  
// Todo: draw in memory  
Stack::Stack( ){  
    this->size = 10 ;  
    this->array = new data(this->size) ;  
    this->topIndex = -1 ;  
}
```

```
//Circular Array Queue Constructor  
Queue::Queue() {  
    this->size = 10 ;  
    this->array = new data(this->size) ;  
    this->frontIndex = -1 ;  
    this->rearIndex = -1 ;  
}
```

Array Stack vs. Circular Array Queue – Destructor

// Array Stack Destructor

// Todo: draw in memory

// Discussion: what if empty list? Will this work?

```
Stack::~~Stack() {  
    delete [] this->array ;  
}
```

// Circular Array Queue Destructor

// Todo: draw in memory

// Discussion: what if empty list? Will this work?

```
Queue::~~Queue() {  
    delete [] array ;  
}
```

Array Stack vs. Circular Array Queue – Push vs Enqueue

```
// Array Stack Push
// Todo: draw this in memory
// Discussion: what if empty list? Will this work?
void Stack::Push(data x){
    if(this->top == this->size -1)
        // full stack - out of space error
    else{
        this->top++;
        this->array[top] = x ;
    }
}
```

Array Stack vs. Circular Array Queue – Push vs Enqueue

// Circular Array Queue Enqueue

// Todo: draw in memory

// Discussion: what if empty list? Will this work?

```
void Queue::Enqueue(data x){
    if(this->frontIndex == this->rearIndex + 1 ||
       this->frontIndex == 0 && this->rearIndex == this->size-1)
        // full queue - out of space error
    else{
        this->rearIndex++;
        if(this->rearIndex == this->size)
            this->rearIndex = 0 ; // wrap
        this->array[rearIndex] = x ;
        if(this->frontIndex == -1)
            this->frontIndex = 0 ; // first time
    }
}
```

Array Stack vs. Circular Array Queue – Pop vs Dequeue

```
// Array Stack Pop
// Todo: draw in memory
// Discussion: what if empty list? Will this work?
data Stack::Pop(){
    if(this->top == -1)
        // empty stack
    else{
        data output = this->array[top] ;
        top-- ;
        return output ;
    }
}
```

Array Stack vs. Array Queue – Pop vs Dequeue

// Circular Array Queue Dequeue

// Todo: draw in memory

// Discussion: what if empty list? Will this work?

```
data Queue::Dequeue( ){  
    if(this->frontIndex == this->rearIndex)  
        // empty queue  
    else{  
        data output = this->array[this->frontIndex] ;  
        this->frontIndex++ ;  
        if(this->frontIndex == this->size)  
            this->frontIndex = 0 ; // wrap  
    }  
}
```

Array Stack vs. Circular Array Queue – Print

```
// Array Stack Print  
// Todo: draw in memory  
// Discussion: what if empty list? Will this work?  
void Stack::Print() {  
    if(this->top == -1)  
        // empty list  
    else {  
        int start = this->top ;  
        int end = this-> 0 ;  
        for(int i = start ; i >= end ; i--)  
            cout << this->array[i] << endl ;  
    }  
}
```

Array Stack vs. Circular Array Queue – Print

```
// Circular Array Queue Print
// Todo: draw in memory
void Queue::Print( ){
    if(this->frontIndex == this->rearIndex)
        // empty list
    else{
        int cur = this->frontIndex ;
        int end = this->rearIndex ;
        while(cur != end){
            cout << this->array[cur] << endl ;
            cur++ ;
            if(cur == this->size)
                cur = 0 ; // wrap
        }
    }
}
```

Alternative Versions of Queues

LL Priority Queue

- **Singly Linked Sorted List (ends in NULL)**
- **Sorted by Priority DESC (highest priority first)**
- **No longer adding at front or back ... enqueue based on priority**
- **dequeue from front (NOTICE we don't need a rear ptr then)**

// Specs

```
class PriorityQueue{  
    private:  
        node * front ;  
    public:  
        PriorityQueue( ) ;  
        PriorityQueue(const PriorityQueue &) ;  
        ~PriorityQueue( ) ;  
        void Enqueue(data) ;  
        data Dequeue( ) ;  
}
```

Constructor

- **create an empty linked list**

Destructor

- **loop thru and destroy the data & nodes in that linked list**

Copy Constructor

- **Creates 2nd Priority Queue with same info**

Enqueue

- **Insert into the sorted Singly Linked list (exactly like homework 2)**

Dequeue

- **Delete headptr from Singly Linked List (exactly like Pop off stack)**

Homework 3

Singly Linked List Implementation of a Priority Queue

- (very similar to the “singly linked list” part of homework 2)
- (very similar to the “stack” we implemented in class)

DESCRIPTION: Printer Queues, Send a Print Job, Print a Print Job

I have written

- the main program (main.cpp)
- the PrintJob class
- the node class
- the Time Class (hours, minutes, AM/PM ... 5:35 pm, 4:20 am)

You have to write

- the PrinterQueue class (it's a Priority Queue full of PrintJobs)
 - constructor, destructor, copy constructor, enqueue, dequeue, overloaded <<, isempty, nexttoprint

Why Copy Constructor?

What we've learned so far ...

```
PriorityQueue a ;  
PriorityQueue b(a) ; // copy constructor called
```

There are actually 2 other scenarios where Copy Constructor is called

Scenario 1 : (**initialization**)

```
PriorityQueue a ;  
PriorityQueue b = a ; // copy constructor called  
// equivalent to PriorityQueue b(a)
```

Scenario 2: (**pass by value**) – draw in memory

```
main() { PriorityQueue a ; f1(a) ; }  
void f1(PriorityQueue p)  
{ // copy made on the stack, copy constructor called  
}
```

In Priority class and Time class, why “friend” ==, <, etc. ???

- member funcs can only overload the right hand side of an operator

Example:

```
bool operator==(const int p) { }
```

```
// Priority p1 ;
```

```
// if(p1 == Priority::LOW) // this is OK
```

```
// if(Priority::LOW == p1) // this will not work
```

Fix:

```
friend bool operator==(const Priority & p, const int p) ;
```

```
// Priority p1 ;
```

```
// if(p1 == Priority::LOW) // this is OK
```

```
// if(Priority::LOW == p1) // this is OK
```