July 22, 2010

Nodes Reviewed

Queues

Yesterday we talked about the Stack. A Stack is a data collection structure which gives you fast access to the most recent item passed to it. It is considered a Last-In-First-Out structure in that whatever is the most recent data to be added to the stack is also the most recent data to be removed from the stack. There are many applications for stacks in real world applications and mathematical problems.

A Queue is the opposite of a stack. A Queue is a data collection structure which gives you fast access to the oldest item passed to it. It is considered a First-In-First-Out structure in that whatever is the first piece of data to be added is also the first piece of data removed.

The most common real world example of a queue is a grocery store checkout line. In the UK, the word "checkout line" is "queue". The checkout line is a "First-In-First-Out" system of processing customers' shopping carts. The first customer in line is the first one out the door. The person who arrives to the checkout line always has to start in the back and wait for everyone in front of him to checkout. Once everyone in front of a cart has left, the cart now becomes the front of the line and can be processed.

Queues do not appear as often as stacks in applications and real world problems. The most common computer application of a queue is a networked printer that accepts print jobs. Back in our Adler computer lab, we have a networked printer. If each of us are working on our programs and had to print them, we use the print function in whatever application we are using. If someone on the other side of the lab hits "print" just a few seconds before you hit "print", then their document is added to the queue first, and your document is added to the system in a "spool" for printing later. The printer has an on-board memory for accepting jobs that it is not yet able to print.

Another example of a queue is parallel computing applications. One model of parallel computing is the "worker bee" model. In this model, a series of computers are all networked to act as a cluster computer. In cluster computing, the machines do not work together to solve a problem, but they can communicate with each other and pass messages from machine to machine. One computer will represent a queen bee. The remaining machines will be added to a queue of "waiting" machines. The queen will accept jobs for processing and immediately assign the job to the first machine in the queue of waiting machines. This machine is now in the state of "working" and is removed from the queue. When the worker machine is finished with the task, it alerts the queen that the task is finished and is added back to the end of "waiting" machines queue. In this system is the second machine in the queue could be faster than the first machine. The second machine could finish its job in a faster time than the first machine and return to the queue faster. The advantage to this system is that for the first complete pass of the queue, each machine is used once. Faster machines are assigned jobs and added back to the queue faster than slow machines. If one machine in the queue is extremely slow, it only effects one job, and the rest of the system hums along smoothly without even knowing that one machine has fallen behind.

Circular Queue implementation

In the stack, we required three fields: an array of elements, the top variable, and the MAX_SIZE variable. In the queue, we need four fields: an array of elements, a front variable, a back variable, and a MAX_SIZE variable.

```
class Queue<T>
    int MAX_SIZE = 100
    int front
    int back
    T[] elements
```

```
Queue:
    front = 0
    back  = 0
    elements = Array[MAX_SIZE]

isEmpty:
    return front == back

isFull:
    return front == (back + 1)%MAX_SIZE

size:
    return (back - front + MAX_SIZE) % MAX_SIZE

front:
    if isEmpty:
        print "Error: Queue is empty."
        exit
    return elements[front]

enqueue(data):
    if isFull:
        print "Error: Queue is full."
        exit
    elements[back] = data
    back = (back + 1) % MAX_SIZE

dequeue:
    if isEmpty:
        print "Error: Queue is empty."
        exit
    temp = elements[front]
    front = (front+1) % MAX_SIZE
    return temp
```

Priority Queue

The Priority Queue is a combination of the Queue and the Insertion Sort. All of the operations of the queue are the same except for the "enqueue" operation. The enqueue operation acts as a simple insertion sort, where you have to move around elements to find the proper place for your

next data point to live. There are several algorithms which can be solved with a Priority Queue: most notably is the famous Dijkstra's algorithm for determining the shortest distance between two points when given a network of roads.