```
/*
Using the queue ADT
edit from http://www.dreamincode.net/forums/topic/49439-concatenating-queues-in-c/
bin>bcc32 queue.cpp
*/
#include <stdio.h>
#include <stdlib.h>
//      Queue ADT Type Defintions
        typedef struct node
         {
          void*          dataPtr;
          struct node* next;
         } QUEUE_NODE;
        typedef struct
         {
          QUEUE_NODE* front;
          QUEUE_NODE* rear;
          int              count;
         } QUEUE;

//      Prototype Declarations
        QUEUE* createQueue  (void);
        QUEUE* destroyQueue (QUEUE* queue);

        bool  dequeue   (QUEUE* queue, void** itemPtr); // ** keep number in memory
        bool  enqueue   (QUEUE* queue, void*  itemPtr);
        bool  queueFront (QUEUE* queue, void** itemPtr);
        bool  queueRear  (QUEUE* queue, void** itemPtr);
        int   queueCount (QUEUE* queue);
        bool  emptyQueue (QUEUE* queue);
        bool  fullQueue  (QUEUE* queue);
//      End of Queue ADT Definitions

void printQueue          (QUEUE* stack);

int main (void)
{
//      Local Definitions
        QUEUE* one;
        QUEUE* nueng;
        int*   numOne;
        int** itemPtr;

//      Statements
        // Create two queues
        one= createQueue();
        nueng = createQueue();
        for (int Warayut = 1; Warayut  <= 15; Warayut ++)
```

```
        {
                numOne  = (int*)malloc(sizeof(Warayut )); // set pointer to memory
                *numOne = Warayut +10;
                enqueue(one ,numOne);
        }
      for (int Warayut  = 1; Warayut  <= 15; Warayut ++)
       {
                numOne = (int*)malloc(sizeof(Warayut )); // set pointer to memory
                *numOne = 20-Warayut ;
                enqueue(nueng ,numOne);
       }


      printf ("one:\n");
      printQueue (one); // 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
      printf ("nueng:\n");
      printQueue (nueng); // 19  18 17 16 15 14 13 12 11 10 9 8 7 6 5
      return 0;
}

/*    ================= createQueue ================
      Allocates memory for a queue head node from dynamic
      memory and returns its address to the caller.
        Pre   nothing
        Post  head has been allocated and initialized
        Return head if successful; null if overflow
*/
QUEUE* createQueue (void)
{
//    Local Definitions
      QUEUE* queue;
//    Statements
      queue = (QUEUE*) malloc (sizeof (QUEUE));
      if (queue)
        {
                queue->front  = NULL;
                queue->rear   = NULL;
                queue->count  = 0;
        } // if
      return queue;
}     // createQueue

/*    ================= enqueue ================
      This algorithm inserts data into a queue.
        Pre   queue has been created
        Post  data have been inserted
        Return true if successful, false if overflow
*/
```

```
bool enqueue (QUEUE* queue, void* itemPtr)
{
//  Local Definitions
//            QUEUE_NODE* newPtr;
//  Statements
//            if (!(newPtr = (QUEUE_NODE*)malloc(sizeof(QUEUE_NODE)))) return false;
              QUEUE_NODE* newPtr = (QUEUE_NODE*)malloc(sizeof(QUEUE_NODE));
           newPtr->dataPtr = itemPtr;
           newPtr->next    = NULL;
           if (queue->count == 0)
             // Inserting into null queue
             queue->front  = newPtr;
           else
             queue->rear->next = newPtr;
           (queue->count)++;
           queue->rear = newPtr;
           return true;
}          // enqueue

/*         ================= dequeue ================
           This algorithm deletes a node from the queue.
             Pre     queue has been created
             Post   Data pointer to queue front returned and
                              front element deleted and recycled.
             Return true if successful; false if underflow
*/
bool dequeue (QUEUE* queue, void** itemPtr)
{
//         Local Definitions
           QUEUE_NODE* deleteLoc;
//         Statements
           if (!queue->count)
                    return false;
           *itemPtr  = queue->front->dataPtr;
           deleteLoc = queue->front;
           if (queue->count == 1)
             // Deleting only item in queue
             queue->rear  = queue->front = NULL;
           else
             queue->front = queue->front->next;
           (queue->count)--;
           free (deleteLoc);
           return true;
}          // dequeue

/*         ================== queueFront =================
           This algorithm retrieves data at front of the queue
           queue without changing the queue contents.
```

```
            Pre     queue is pointer to an initialized queue
            Post    itemPtr passed back to caller
            Return true if successful; false if underflow
*/
bool queueFront (QUEUE* queue, void** itemPtr)
{
//      Statements
        if (!queue->count)
                return false;
        else
          {
                *itemPtr = queue->front->dataPtr;
                 return true;
          } // else
}       // queueFront

/*      ================== queueRear =================
        Retrieves data at the rear of the queue
        without changing the queue contents.
          Pre     queue is pointer to initialized queue
          Post    Data passed back to caller
          Return true if successful; false if underflow
*/
bool queueRear (QUEUE* queue, void** itemPtr)
{
//      Statements
        if (!queue->count)
                return true;
        else
          {
                *itemPtr = queue->rear->dataPtr;
                return false;
          } // else
}       // queueRear

/*      ================== emptyQueue =================
        This algorithm checks to see if queue is empty
        Pre       queue is a pointer to a queue head node
        Return true if empty; false if queue has data
*/
bool emptyQueue (QUEUE* queue)
{
//      Statements
        return (queue->count == 0);
}       // emptyQueue

/*      ================== fullQueue =================
        This algorithm checks to see if queue is full. It
```

```
          is full if memory cannot be allocated for next node.
            Pre     queue is a pointer to a queue head node
            Return true if full; false if room for a node
*/
bool fullQueue (QUEUE* queue)
{
//        Check empty
if(emptyQueue(queue)) return false; // Not check in heap
//        Local Definitions *
QUEUE_NODE* temp;
//        Statements
          temp = (QUEUE_NODE*)malloc(sizeof(*(queue->rear)));
          if (temp)
            {
                    free (temp);
                    return false; // Heap not full
            } // if
          return true; // Heap full
}         // fullQueue

/*        ================== queueCount =================
          Returns the number of elements in the queue.
            Pre     queue is pointer to the queue head node
            Return queue count
*/
int queueCount(QUEUE* queue)
{
//        Statements
          return queue->count;
}         // queueCount

/*        ================== destroyQueue =================
          Deletes all data from a queue and recycles its
          memory, then deletes & recycles queue head pointer.
            Pre     Queue is a valid queue
            Post    All data have been deleted and recycled
            Return null pointer
*/
QUEUE* destroyQueue (QUEUE* queue)
{
//        Local Definitions
          QUEUE_NODE* deletePtr;
//        Statements
          if (queue)
            {
                    while (queue->front != NULL)
                      {
                              free (queue->front->dataPtr);
```

```
                        deletePtr       = queue->front;
                        queue->front = queue->front->next;
                        free (deletePtr);
                    } // while
                free (queue);
            } // if
        return NULL;
}       // destroyQueue

/*      ================== printQueue ================
        A non-standard function that prints a queue. It is
        non-standard because it accesses the queue structures.
          Pre  queue is a valid queue
          Post queue data printed, front to rear
*/
void printQueue(QUEUE* queue)
{
//      Local Definitions
        QUEUE_NODE* node = queue->front;
//      Statements
        printf ("Front=>");
        while (node)
          {
                printf ("%3d", *(int*)node->dataPtr);
                node = node->next;
          } // while
        printf(" <=Rear\n");
        return;
}       // printQueue
```