Some tvector functions

tvector( ); // default constructor (size==0)
tvector( int size ); // initial size of vector is size
tvector( int size, const itemType & fillValue ); // all entries == fillValue
tvector( const tvector & vec ); // copy constructor

int length( ) const; // support for old programs, deprecated
int capacity( ) const; // use in place of length()
int size( ) const; // # elements constructed/stored
void resize( int newSize ); // change size dynamically;
                   // can result in losing values
void reserve(int size); // reserve size capacity
void push_back(const itemType& t);
void pop_back();
void clear(); // size == 0, capacity unchanged

Some string functions

int length(); // returns the number of characters in the string
string substr(int pos, int len); //returns substring of len characters
                   // beginning at position pos
int find(string s); // returns first position/index at which string s begins
                   // returns string::npos if s does not occur

Some tqueue functions

tqueue( ); // construct empty queue

const Type & front( ) const; // return front (no dequeue)
bool isEmpty( ) const; // return true if empty else false
int length( ) const; // return number of elements in queue
int size( ) const; // return number of elements in queue

void enqueue( const Type & item ); // insert item (at rear)
void dequeue( ); // remove first element
void dequeue( Type & item ); // combine front and dequeue
void makeEmpty( ); // make queue empty
void clear( ); // same as makeEmpty

PROBLEM 1 : (Analysis: 8 points)

Consider the following three program segments. For each one give the worst case running time (Big-Oh) in terms of N and M (some will only need one of these).
int k = M;
while (k > 0)
{
    k = k / 2;
    cout << k << endl;
}

Worst case running time is:

int sum = 0;
for (int k=0; k < N; k += 4)
{
    for (int j=1; j<=N; j *= 2)
    {
        sum += j+k;
    }
}

Worst case running time is:

for (int k=0; k < N; k++)
{
    for (int j=k+1; j < N; j++)
    {
        for(int h=j+1; h < M; h++)
        {
            cout << h << endl;
        }
    }
}

Worst case running time is:

PROBLEM 2 : (Recursion: 8 points)

Consider the following Mystery function.

int Mystery(int num)
{
    if (num > 4)
    {

What is the value returned from the call Mystery(5)?

What is the value returned from the call Mystery(8)?

What is the recurrence relation for Mystery assuming it is called with N? (you do not have to give the base case)

PROBLEM 3:  (All Queued Up: 10 points)

Write the function ReturnAndRemoveMiddle whose header is shown below. ReturnAndRemoveMiddle removes the middle item from the queue and returns it. If there are an even number of elements, the “middle” item removed could be either of the two middle items. The resulting queue should have the original elements in the same order as before, minus the middle element. See page 2 for the tqueue header.

For example, consider the queue shown above. The call ReturnAndRemoveMiddle would return and remove either 23 or 18 from numbers.

```cpp
int ReturnAndRemoveMiddle(tqueue<int> & numbers)
// pre: numbers is not empty
// post: numbers is the same as before except the middle element is missing,
// returns the middle element
{
}
```
PROBLEM 4: (Super Heroes: 24 points)

Consider the following Node definition to define a toy and its quantity.

```cpp
struct Node
{
    string name; // name of toy
    int quantity; // quantity of toy
    Node * next;

    Node(const string & nm, int quant, Node * nxt)
        : name(nm), quantity(quant), next(nxt)
    {
    }
};
```

PART A (6 pts):
Write the function `PrintLargeQuantities` whose header is shown below. `PrintLargeQuantities` prints the names of the toys in `list` whose quantity is greater than or equal to `large`.

```cpp
void PrintLargeQuantities(Node * list, int large)
    // precondition: large > 0
    // postcondition: print the names of the toys with quantity greater than
    // or equal to large
{
}
```

PART B (8 pts):
Write the recursive function `LastNode` whose header is shown below. `LastNode` returns a pointer to the last node in the list, or NULL if the list is empty.

For example, the call `PrintLargeQuantities(toys,15)` applied to the list shown above would output:

```
superman
nightwing
```
For example, in the list shown above, the call \texttt{LastNode(toys)} would return a pointer to the node with the value \texttt{nightwing}.

Your function must use \texttt{recursion} to receive full credit.

\begin{verbatim}
Node * LastNode(Node * list)
// note: recursive function
// postcondition: returns pointer to last node in list, or NULL if list
// is empty
{
}
\end{verbatim}

Give the recurrence relation for this function assuming there are \(N\) nodes in list. (You do not have to give the base case)

What is the worst case running time (Big-Oh)? (You do not have time to solve this recurrence relation, just think about what happens)

\textbf{PART C (10 pts):}

Write the function \textit{MakeSpecial} whose header is shown below. \textit{MakeSpecial} is given the name of a toy in the linked list and creates a new Node in the list with the same name with the word \texttt{special} attached to it. The original quantity is split between the two nodes. If the original quantity is an odd number, the new node’s quantity should be one less than the old node’s quantity.

\begin{verbatim}
void MakeSpecial(Node * list, const string & name)
// precondition: list != NULL, name is in list
// postcondition: The node with name is split into 2 nodes, each node has
//   half as much quantity as the original node did, and one name has
\end{verbatim}

For example, in the figure from PART B, the call \texttt{MakeSpecial(toys,"spiderman")} results in the figure shown above. The toy \texttt{spiderman} has been split into two Nodes and the new Node is at the end of the list with the name \texttt{spidermanspecial}. Note the original quantity has been split between these two Nodes.

In writing \textit{MakeSpecial} you may use the function \textit{LastNode} that you wrote in Part B. Assume the function \textit{LastNode} works correctly regardless of what you wrote for Part B.
// "special" attached to it. The special node is put at the end of the list.
{
}

Assuming there are N nodes in list, what is the worst case running time (Big-Oh) of MakeSpecial?

**PROBLEM 5 : (Toy Store: 20 points)**

This problem builds on the previous problem. Now consider creating an array of linked lists to represent the inventory of toys from a toy store. Each linked list in the array will have a “header node” that contains the type of toys and its quantity field is set to 0.

The figure below shows an inventory for a store that has three types of toys. Note the header nodes which are *games*, *superheroes* and *dolls*.

The figure below shows an inventory for a store that has four types of toys.

**PART A (10 pts):** Write the function *AddQuantity* whose header is shown below. *AddQuantity* is given an array of linked lists, a category, a toy name in that category, and an amount. It then
adds the given amount to the quantity for the toy

For example, using the toys array on the top of the previous page, the call \texttt{AddQuantity(toys,"superheroes","robin",4)} would update the quantity under "robin" from 3 to 7.

\begin{verbatim}
void AddQuantity(const tvector<Node *> store, const string & category,
                  const string & name, int amount)
  // pre: category is in store, name is in category's list, amount > 0
  // post: quantity of name is updated by "amount"
{
}
\end{verbatim}

Assume there are \(N\) categories and at most \(M\) items in any category. What is the worst case running time (Big-Oh) of \texttt{AddQuantity}?

**PART B (10 pts)**: Write the function \texttt{CreateSpecialCategory} whose header is shown below. \texttt{CreateSpecialCategory} removes all the nodes in the array that have the word "special" in their name and creates a new linked list of these nodes in the array.

For example, the call \texttt{CreateSpecialCategory(toys)} applied to the top figure on page 9 results in the bottom figure on page 9. A new linked list with a header node is added to the array with all the special Nodes.

In writing \texttt{CreateSpecialCategory}, you may call the function \texttt{RemoveAndMakeSpecialList} whose description is below. You do not have to write this function.

\begin{verbatim}
Node * RemoveAndMakeSpecialList(Node * list)
  // pre: list has a header node
  // post: removes special nodes from list and returns them in a linked list
  //        that does not have a header node, returns NULL if no special nodes
\end{verbatim}

Complete \texttt{CreateSpecialCategory} below.

\begin{verbatim}
void CreateSpecialCategory(tvector<Node *> & store)
  // pre: there is at least one "special" node in store
  // post: a new linked list is added to store containing all the "special"
  //        nodes, which are removed from the old lists they were in
{
}