Section Solutions #4

Problem 1: Filling a Region

```
/*
 * Function: IsOffGrid
 * Usage: bool result = IsOffGrid(pt, grid);
 * ----------------------
 * This function returns whether the point is off the bounds of the
 * given Grid.
 */
bool IsOffGrid(pointT pt, Grid<pixelStateT> &grid)
{
      return pt.row < 0 || pt.col < 0 || pt.row >= grid.numRows() ||
             pt.col >= grid.numCols();
}

/*
 * Function: FillRegion
 * Usage: FillRegion(pt, screen);
 * ----------------------
 * This function paints black pixels everywhere inside the
 * region surrounding the point.
 */
void FillRegion(pointT pt, Grid<pixelStateT> &screen)
{
  if (IsOffGrid(pt, screen))
  {
    return;
  }

  if (screen(pt.row, pt.col) == Black)
  {
    return;
  }

  screen(pt.row, pt.col) = Black;

  pointT north = {pt.row, pt.col + 1};
  FillRegion(north, screen);

  pointT south = {pt.row, pt.col - 1};
  FillRegion(south, screen);

  pointT east = {pt.row + 1, pt.col};
  FillRegion(east, screen);

  pointT west = {pt.row - 1, pt.col};
  FillRegion(west, screen);
}
```

Problem 2: Shortest Path through a Maze

```
/*
 * Function: ShortestPathLength
 * Usage: shortestPath = ShortestPathLength(pt);
 * ---------------------------------------
 * This function looks for the length of the shortest path through the
 * maze starting at the specified position pt.  If none exists it
 * returns NoSolution, otherwise it returns the shortest distance.
 *
 * To find the length of the shortest path, the function computes the
 * length of the shortest path from all directions around the
 * point.  The shortest path is then one more than the smallest of
 * these (since it takes one step to move in the given direction).  The
 * base cases occurs when you are outside the maze, in which case the
 * shortest path has zero length, or if you have already been to a
 * location, in which case NoSolution is returned since you've already
 * visited that location and thus don't need to visit it again.
 */

int ShortestPathLength(pointT pt)
{
  int shortest, len;

  if (OutsideMaze(pt))
  {
    return 0;
  }

  if (IsMarked(pt))
  {
    return NoSolution;
  }

  shortest = NoSolution;

  MarkSquare(pt);
  for (directionT dir = North; dir <= West; dir=directionT(dir+1))
  {
    if (!WallExists(pt, dir))
    {
      len = ShortestPathLength(AdjacentPoint(pt, dir));
      if (len < shortest)
      {
        shortest = len;
      }
    }
  }
  UnmarkSquare(pt);
  if (shortest == NoSolution)
  {
    return NoSolution;
  }
  else
  {
    return (shortest + 1);
```

```
    }
}
```

Problem 3: Pointers
The original data structure looks something like this:

By Author	
John Steinbeck	bookT: East of Eden
	bookT: The Grapes of Wrath
Juan Rulfo	bookT: Pedro Paramo

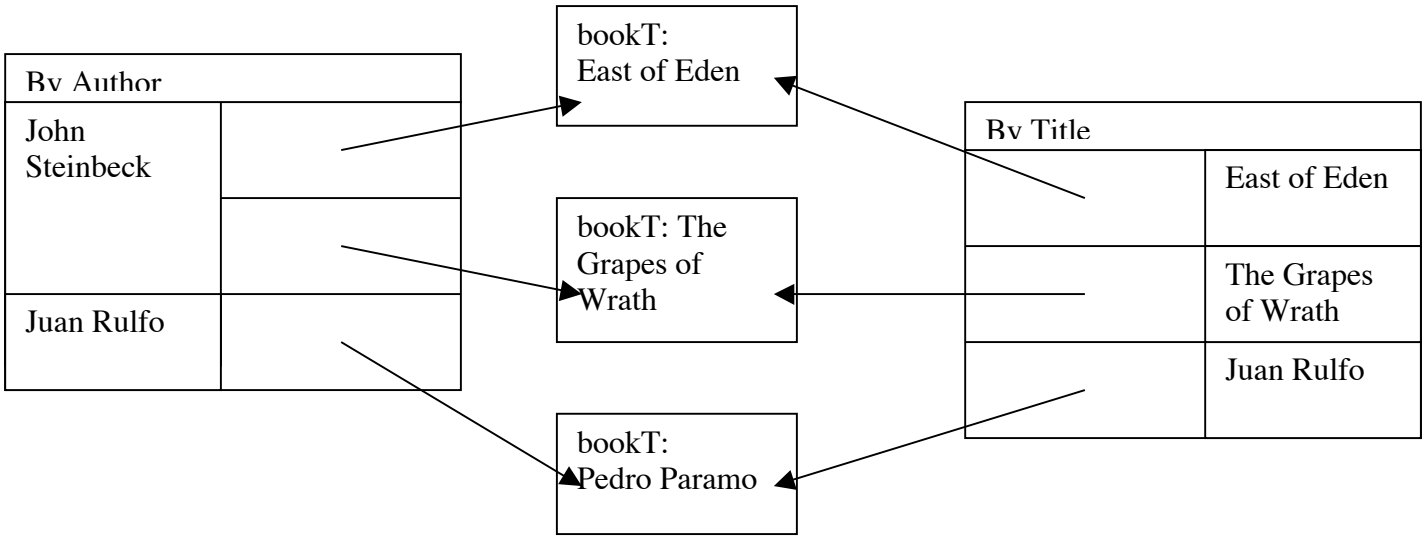
By Title	
East of Eden	bookT: East of Eden
The Grapes of Wrath	bookT: The Grapes of Wrath
Pedro Paramo	bookT: Pedro Paramo

(etc).

If one of the authors is wrong, you must update the appropriate bookT struct as well as each of the Vectors in each of the Maps, since each Vector has its own copy of the struct. If you modified your data structure so that it stored pointers to bookTs, like this:

```
Map<Vector<bookT *> > byAuthor;
Map<Vector<bookT *> > byTitle;
Map<Vector<bookT *> > byPublisher;
...
```

you would only need to modify the original struct since all of the Vectors could point to the same one. You also save space by only storing structs once. This data structure would look like this:



Problem 4: Linked List Warmup

a)
```
Cell * ConvertToListIter(Vector<int> vector)
{
    Cell *head = new Cell;
    head->next = NULL;
    head->value = vector[0];

    Cell *curr = head;
    for(int i = 1; i < vector.size(); i++)
    {
      Cell *newCell = new Cell;
      newCell->next = NULL;
      newCell->value = vector[i];

      curr->next = newCell;
      newCell = curr;
    }
    return head;
}


Cell * ConvertToListRealRecur(Vector<int> &vector, int index)
{
    if(index >= vector.size())
      return NULL;
```

```
    Cell *curr = new Cell;
    curr->value = vector[index];

    curr->next = ConvertToListRealRecur(vector, index + 1);

    return curr;
}

Cell * ConvertToListRecur(Vector<int> vector)
{
    return ConvertToRealRecur(vector, 0);
}
```

b)
```
int SumListIter(Cell *list)
{
    int sum = 0;
    Cell *curr = list;
    while(curr != NULL)
    {
      sum += curr->value;
      curr = curr->next;
    }
    return sum;
}

int SumListRecur(Cell *list)
{
    if(list == NULL)
      return 0;
    else
      return list->value + SumListRecur(list->next);
}
```

Problem 5: Linked List Trace

The function **`PopRocks`** takes the cell in the list that the parameter points to, removes that cell from its current position in the list, and tacks it on the end of the list. Therefore, if the parameter is a pointer to the first element in the list, the lists would end up looking like this:

```
   "30" -> "45" -> "60" -> "15"

   "t" -> "a" -> "r" -> "s"

"hang" -> "a" -> "salami," -> "I'm" -> "a" -> "lasagna" -> "hog!" ->
"Go"
```

Problem 6: Append

```
/**
 * Function: Append
 * Usage: Append(first, second)
 * ----------------------------
 * The following function appends the second linked list on
 * to the end of the first.
```

```
 */
void Append(Cell *& first, Cell* second)
{
  if (first == NULL)
  {
    first = second;
  }
  else
  {
    Append(first->next, second);
  }
}
```

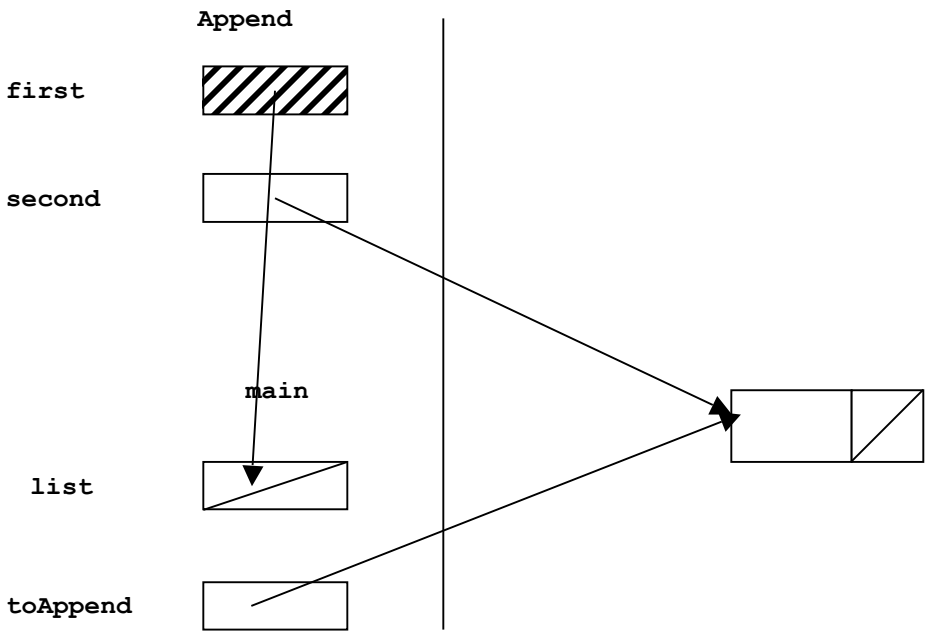
Note that the first parameter to the function is a reference. In the function, we are assigning **`first = second`**. For this change to be seen by the caller, **`first`** needs to be passed by reference. One way that many programmers try to get around this is by changing the function to the following:

```
void IncorrectAppend(Cell *first, Cell *second)
{
  if (first->next == NULL)
  {
    first->next = second;
  }
  else
  {
    IncorrectAppend(first->next, second);
  }
}
```

Because now the assignment is dereferencing a pointer, the change will be seen by the caller. The one problem here is what happens when first is a completely empty list. Because first is NULL, first has no next, so we'll crash if we try to dereference it. This is why the first version is correct and the second is not.

To show why the first one will work, here is a diagram showing what happens when you have an empty list passed to **`Append`**. This first picture is when **`Append`** has first been called but has not executed. Note how **`first`** is a reference to the list in main.

```
int main()
{
  Cell *list = NULL;
  Cell *toAppend = new Cell;
  toAppend->next = NULL;
  Append(list, toAppend);
}
```



Now look at what happens after we make the assignment **`first = second`**. Because **`first`** is a reference to **`list`**, the change is seen in **`main`**!