Name: _____

This is a **50-minute** exam. Notes, calculators, or electronic devices are not allowed. Please put away all phones and smart watches. Follow code style writing guidelines for the course (excluding requirements for code documentation). The back of each page may be used if you run out of room. Code will be partially graded on efficiency.

(30 pts) A *palindrome* is a text (word or phrase) that reads the same forwards as it does backwards. Complete the method stub below to see if the given string is a palindrome by using a **stack** and a **StringBuilder**. The use of *arrays* or string addition (+) is **not** allowed. If the text is a palindrome, the method should return **true**, otherwise **false**. You may assume that the string is *non-empty*.

```
private bool IsPalindrome(string text)
{
      Stack<char> stack = new Stack<char>();
       foreach(char c in text)
       {
            stack.Push(c);
       }

       StringBuilder sb = new StringBuilder();

       while(stack.Count > 0)
       {
            sb.Append(stack.Pop());
       }

       return sb.ToString() == text;
}
```

1. (40 pts) There is a holiday party at work. Every person in attendance must bring one gift for a gift exchanging game. As each guest arrives, they draw a random number from a hat. Once the game starts, the guest with the smallest number (the highest priority) gets to go first by picking a gift that someone brought. This is an example of a **priority queue**. This priority queue has the following fields:

   - `private int _count – The number of elements in the queue.`
   - `private int[] _elements – A non-circular` **`array`** `that stores the elements in the queue`

   Implement the priority queue **Dequeue** and **Peek** methods below. When an element is **dequeued**, there should be no gaps left in the *_elements* array except for any empty indices at the end. Dequeue should throw an invalid operation exception if there are no elements left.

```
public int Dequeue()
 {
     int item = Peek();
     _count--;

     for (int i = 0; i < _count; i++)
     {
         if (_elements[i] != item && i+1 < _elements.Length)
         {
             _elements[i] = _elements[i+1];
         }
         else
         {
             _elements[i] = default(int);
         }
     }

     return item;
 }

 public int Peek()
 {
     if (_count == 0)
     {
         throw new InvalidOperationException();
     }

     int smallest = _elements[0];

     for (int i = 1; i < _count; i++)
     {
         if(_elements[i] < smallest)
         {
             smallest = _elements[i];
         }
     }

     return smallest;
 }
```

2. (30 pts) A linked list contains **LinkedListCells** (see last page) that have a **Data** property that stores information, and a **Next** property that points to the next cell in the linked list. In this linked list, the **Next** property of the back (last cell) of the list always contains a reference to the **front** (first cell) of the list. Complete the **InsertAfterNext** method below that inserts a new item after the cell immediately following the given cell. The front and **back** of the linked list should be updated as needed when an item is inserted. You may assume that the linked list already has at least one cell and has the following properties:
   - `private LinkedListCell<T> _front – The first cell of the linked list`
   - `private LinkedListCell<T> _back – The last cell of the linked list`

```
public void InsertAfterNext(T newData, LinkedListCell<T> location)
{
	LinkedListCell<T> cell = new LinkedListCell<T>();
	cell.Data = newData;
	cell.Next = location.Next.Next;
	location.Next.Next = cell;

	if (location.Next == _back)
	{
		_back = cell;
	}
}
```