

Section #5 Solutions

Based on handouts by various current and past CS106B/X instructors and TAs.

1. isSorted (*linked lists*)

```
// iterative solution
bool isSorted(ListNode* front) {
    if (front != nullptr) {
        ListNode* current = front;
        while (current->next != nullptr) {
            if (current->data > current->next->data) {
                return false;
            }
            current = current->next;
        }
    }
    return true;
}

// recursive solution
bool isSortedRecursive(ListNode* front) {
    if (front == nullptr || front->next == nullptr) {
        return true;
    }
    return front->data <= front->next->data && isSortedRecursive(front->next);
}
```

2. removeAllThreshold (*linked lists*)

```
void removeAllThreshold(ListNode*& front, int value, int threshold) {
    while (front != nullptr && front->data >= value - threshold
          && front->data <= value + threshold) {
        ListNode* trash = front;
        front = front->next;
        delete trash;
    }
    if (front != nullptr) {
        ListNode* current = front;
        while (current->next != nullptr) {
            if (current->next->data >= value - threshold
                && current->next->data <= value + threshold) {
                ListNode* trash = current->next;
                current->next = current->next->next;
                delete trash;
            } else {
                current = current->next;
            }
        }
    }
}
```

3. doubleList (*linked lists*)

```
void doubleList(ListNode* front) {
    if (front != nullptr) {
        ListNode* half2 = new ListNode(front->data);
        ListNode* back = half2;
        ListNode* current = front;
        while (current->next != nullptr) {
            current = current->next;
            back->next = new ListNode(current->data);
            back = back->next;
        }
        current->next = half2;
    }
}
```

4. split (*linked lists*)

```
void split(ListNode*& front) {
    if (front != nullptr) {
        ListNode* current = front;
        while (current->next != nullptr) {
            if (current->next->data < 0) {
                ListNode* temp = current->next;
                current->next = current->next->next;
                temp->next = front;
                front = temp;
            } else {
                current = current->next;
            }
        }
    }
}
```

5a. reverse (*linked lists*)

```
// both solutions here avoid creating new nodes

// iterative solution
void reverseIterative(ListNode*& front) {
    ListNode* current = front;
    ListNode* previous = nullptr;
    while (current != nullptr) {
        ListNode* nextNode = current->next;
        current->next = previous;
        previous = current;
        current = nextNode;
    }
    front = previous;
}

// recursive solution on next page
```

```

// recursive solution
void reverseRecursive(ListNode*& front) {
    if (front == nullptr || front->next == nullptr) {
        return;
    }
    ListNode* restOfList = front->next;
    reverseRecursive(restOfList);           // recursive call will update restOfList
    front->next->next = front;           // front now comes AFTER its own next
    front->next = nullptr;                // front is now at the back
    front = restOfList;                  // update the front of the whole list
}

```

5b. braid (*linked lists*)

```

// iterative solution
void braidIterative(ListNode* front) {
    ListNode *reverse = nullptr;
    for (ListNode *curr = front; curr != nullptr; curr = curr->next) {
        ListNode *newNode = new ListNode(curr->data);
        newNode->next = reverse;
        reverse = newNode;
    }
    // reverse now addresses a memory-independent version of the original list,
    // where all of the nodes are in reverse order.
    ListNode *rest = reverse; // rest addresses part that has yet to be braided in
    for (ListNode *curr = front; curr != nullptr; curr = curr->next->next) {
        ListNode *next = rest->next;
        rest->next = curr->next;
        curr->next = rest;
        rest = next;
    }
}

// recursive solution
void braidRecursiveHelper(ListNode* front, Queue<int>& numbers) {
    if (front == nullptr) {
        return;
    }
    numbers.enqueue(front->data);
    braidRecursiveHelper(front->next, numbers);
    ListNode* newNode = new ListNode(numbers.dequeue(), front->next);
    front->next = newNode;
}

void braidRecursive(ListNode* front) {
    Queue<int> numbers;
    braidRecursiveHelper(front, numbers);
}

```

6. drawPolygonalPath (*linked lists*)

```
void drawPolygonalPath(GWindow& window, PointNode* front) {
    if (front == nullptr) {
        return;
    }
    PointNode* current = front;
    window.fillOval(current->data.getX() - 1, current->data.getY() - 1, 2, 2);
    if (current->next == nullptr) {
        return;
    }
    while (current->next != nullptr) {
        window.drawLine(current->data.getX(), current->data.getY(),
                        current->next->data.getX(), current->next->data.getY());
        window.fillOval(current->next->data.getX() - 1, current->next->data.getY() - 1, 2, 2);
        current = current->next;
        if (current == front) {
            break;
        }
    }
}
```