Workshop Exercises 4

Pancake Sorting

In order to practise working with doubly linked lists, we will implement a problem called **Pancake Sorting** (see https://en.wikipedia.org/wiki/Pancake_sorting). The idea is to take a list of numbers and sort them using operations which resemble flipping pancakes.

Example: Suppose the list contains the numbers $1, 5, 3, 6, 2, 4, 7$. Treating this like a stack of pancakes (with the top on the left), we can flip (for example) the top 4 pancakes, i.e. reverse the order of the first 4 elements, getting $\mathbf{6}, \mathbf{3}, \mathbf{5}, \mathbf{1}, 2, 4, 7$.

We need a number of things to complete this task.

1. If you have not yet done so, implement the `DListnode` struct and the `insertAfter` function.

   ### Solution:

   ```
   #include<iostream>

   struct DListNode{
       int data;
       DListNode *next;
       DListNode *previous;
   };

   DListNode *insertAfter(int newData, DListNode *n){
       DListNode *newNode = new DListNode;
       newNode->data = newData;
       if(n != nullptr){
           newNode->prev = n;
           newNode->next = n->next;
           if(n->next != nullptr)
               n->next->previous = newNode;
           n->next = newNode;
       }
       return newNode;
   }
   ```

2. In order for the user to see what they are doing, write an output operator

   ```
   std::ostream &operator<<(const std::ostream &s, DSListNode *n)
   ```

   which outputs the list, separated by spaces, starting with the given node `n` (example output: "3 1 5").

   ### Solution:

   ```
   std::ostream &operator<<(const std::ostream &s, DSListNode *n){
       while(n != nullptr){
           s << n->data << " ";
           n = n->next;
       }
       return s;  // In order to be able to chain outputs
   }
   ```

3. Write a function

```
bool isSorted(DListNode *n)
```

which returns true if the list (starting at `n`) is in increasing order.

Solution:

```
bool isSorted(DListNode *n){
    if(n != nullptr){
        int max = n->data;
        n = n->next;
        while(n != nullptr){
            if(n->data < max)  // out of order
                return false;
            max = n->data;
            n = n->next;
        }
    }
    return true;
}
```

4. Write a function

```
DListNode *initialise(unsigned int k)
```

which creates a list containing the numbers `1,...,k`, arranged like this:

- first all the **even** numbers, in decreasing order.
- then all the **odd** numbers, in increasing order,

For example, using `k=11` should produce a list containing `10,8,6,4,2,1,3,5,7,9,11`

Solution:

```
DListNode *initialise(unsigned int k){
    DListNode *n = nullptr;
    for(unsigned int i = k-(k%2); i>0; i-=2)
        n = insertAfter(i, n);
    for(unsigned int i = 1; i<=k; i+=2)
        n = insertAfter(i, n);
}
```

5. Write a function

```
DListNode* flip(unsigned int k, DListNode *n)
```

which reverses the order of the first `k` nodes in the list starting at `n`. It should check that `n` really is the first node in the list, and do nothing (except complain) if it isn't. The return value should be the new first node of the list.

Solution:

This is the trickiest part of the exercise. First, there are a number of corner cases to keep in mind:

- If `n` is a null pointer, there is nothing to do (empty list)
- If `n` does not point to the first node, we need to complain and not do anything
- If `k` is greater than the length of the list, we might complain, or simply reverse the whole list. In this solution, we pick the second option.

It helps to think about how the flip can be realised by removing and re-inserting individual values. The strategy is:

- Find the last node (we call it `last` below) of the section to be flipped
- For each node before that, remove it and re-insert it after `last`
- For the example $1, 5, 3, 6, 2, 4, 7$, where the first 4 are supposed to be flipped, this gives the steps:
  - `last` is the node containing the 6
  - Removing 1 and re-inserting it after `last` gives $5, 3, 6, \mathbf{1}, 2, 4, 7$
  - Removing 5 and re-inserting it after `last` gives $3, 6, \mathbf{5}, 1, 2, 4, 7$
  - Removing 3 and re-inserting it after `last` gives $6, \mathbf{3}, 5, 1, 2, 4, 7$
  - And we are done. Note how the 5 was pushed in front of the 1, and the 3 in front of the 5.

The code looks like this:

```
DListNode* flip(unsigned int k, DListNode *n){
    if(n == nullptr)  // nothing to do
        return n;
    if(n->previous != nullptr){  // Invalid input, do nothing
        std::cout << "Node is not the first in the list!" << std::endl;
        return n;
    }
    // Find the end of the flipped section: k nodes in, OR the end of the list
    DListNode *last = n;
    for(int i=1; (i<k) && (last->next!=nullptr); i++)
        last = last->next;
    // move everything that is before last
    while(n != last){
        insertAfter(n->data, last); // re-insert data
        n = n->next;                // move the pointer forward
        delete n->previous;         // get rid of the old node
        n->previous = nullptr;
    }
    return n;
}
```

6. Put things together using a main function, which
   - asks the user for the size of list they want,
   - reads the size from standard input,
   - creates the list using `initialise`,
   - as long as the list is not sorted:
     - prints the list using the output operator,
     - asks the user how many numbers to flip,
     - reads the number from standard input,
     - performs the flip;
   - once the list is sorted, prints a success message.

Solution:

```
int main(){
    unsigned int input;
    std::cout << "How many elements would you like?" << std::endl;
    std::cin >> input;
    DListNode *first = initialise(input);
    while(! isSorted(first)){
        std::cout << "Current list: " << first << std::endl;
        std::cout << "Flip how many elements?" << std::endl;
        std::cin >> input;
        first = flip(input, first);
    }
    std::cout << "Done! Final list: " << first << std::endl;
}
```