ShanghaiTech University

CS101 Algorithms and Data Structures

Fall 2019

Homework 1

Due date: 23:59, September 22, 2019

1. Please write your solutions in English.
2. Submit your solutions to gradescope.com.
3. Set your FULL Name to your Chinese name and your STUDENT ID correctly in Account Settings.
4. If you want to submit a handwritten version, scan it clearly. Camscanner is recommended.
5. When submitting, match your solutions to the according problem numbers correctly.
6. No late submission will be accepted.
7. Violations to any of the above may result in zero grade.

0: Brief review

In the lectures in Week 1, we introduced two basic data structures: **array** and **linked list**. They are both widely used in storing and accessing data. Though their operations look quite similar, the efficiency may vary significantly depending on specific application scenarios.

- **Array**
  When we initialize an array, the compiler will ask for a **continuous** space from the memory. The continuity in memory determines the efficiency of array's operation. Compared to linked list, array outperforms in **accessing** data once the index is given. For example, `A[k] (k is a valid index)` is always in $O(1)$ time complexity. From the view of computer, accessing `A[k]` is encoded as accessing `MEM[addr of A + k * sizeof(element)]`. The access of memory is considered as $O(1)$. Pushing element to the end and replacing current element are also $O(1)$, since they are still implemented based on index access. However, **insertion** and **deletion** of original array are very inefficient $O(n)$. Since we always need to maintain the memory continuity for the array, the extra element memory needed by insertion and the rubbish element memory caused by deletion will call for the replacement of all the elements to the right of the target element. As the array grows in size, it could be the case that the continuous memory occupied by current array runs out. Then the compiler will ask for a larger continuous memory space from the system, and copy all the elements to new array (in $O(n)$), and free the previous memory space.

  According to the properties of array, we should try not to perform insertion, deletion or resize, which means the best application scenario for array is when there is no modifications in array, for example, read data from a **const src** array, compute and write to a **dst** file. The memory continuity is sometimes crucial in optimizing performance. It is friendly for **cache** (cache is about 10x faster than memory). The parallelism in register scale (register is at least 100x faster than memory), for example, `Intel SSE SIMD intrinsics`, also requires the memory continuity of data.

- **Linked List**
  Normally, we use pointer to implement a linked list. In this case, the memory usage of linked list can be incontinuous. It's very interesting that you can tell that the memory **incontinuity** brings the efficiency of **insertion, deletion and resize** for linked list. The detailed implementation for these operations have been demonstrated in lectures and discussions.

  Apparently, linked list is very unfriendly for cache, and `SIMD intrinsics` is also not compatible. In this homework, we want you to practice the operations of linked list and the usage of pointer based on some newly defined operations.

0: Linked list data structure

You should use the defined linked list to implement all the newly defined operations. Please pay attention to the following rules:

1. There is no dummy node as the entry node, i.e. if the linked list is not empty, `head` points to the first node.
2. Use `new` and `delete` to deal with memory issues.
3. Remember to maintain `head` and `tail`.
4. You may not use all of the blank lines. There is at most one statement (ended with `;`) in each blank line.
5. If there is any syntax error, this line will be counted as wrong.

```
class Node {
private:
    int data;
    Node *next;

public:
    Node(int val, Node *nextNode):data(val), next(nextNode)
    {
    }
... ...
};



class List {
private:
        // List: empty
        //       head = tail = NULL;
        // List: Node(1) -> NULL
        //       head = Node(1);
        //       tail = Node(1);
        // List: Node(1) -> Node(2) -> NULL
        //       head = Node(1);
        //       tail = Node(2);
        Node *head;
        Node *tail;
... ...
};
```

1: Insert Before

This operation has been discussed in detail. Given a new data and the target node, create a new node using the new data and insert the new node before the target node. You should implement this function with $O(1)$ time complexity.

- Assume the target node is neither `head` nor `tail`.

```
void InsertBefore (Node *curr, int data)
{
    Node *newNode = new Node(----------------------------);

    ------------------------------------------------------;

    ------------------------------------------------------;
}
```

2: Remove Duplication

Given a linked list in ascending order, you should remove all the nodes with duplicate data. Traverse the list for at most once. For example:

```
Before:
list:  1 -> 2 -> 2 -> 3 -> 3 -> NULL

After:
list:  1 -> 2 -> 3 -> NULL
```

```
void RemoveDuplication (List *list)
{
    Node *curr = list->head;
    if (curr == NULL)
    {
        return;
    }

    while (curr->next != NULL)
    {
        if (curr->data == curr->next->data)
        {
            ----------------------------------------------------;

            if (-------------------------------------)
            {
                ------------------------------------------------;
            }

            ----------------------------------------------------;

            -------------------------=-------------------------;

        }
        else
        {
            ----------------------------------------------------;
        }
    }
}
```

3: Move Head

Given a non-empty linked list `src` and a linked list `dst`. Move the head node of `src` to the tail of `dst`. For example:

```
Before:
src:  1 -> 2 -> 3 -> NULL
dst:  4 -> 5 -> 6 -> NULL

After:
src:  2 -> 3 -> NULL
dst:  4 -> 5 -> 6 -> 1 -> NULL
```

```
void MoveHead (List *src, List *dst)
{
    Node *headNode = src->head;
    assert(headNode);

    ----------------------------------------------------------;


    if (-----------------------------------)
    {
        src->tail = NULL;
    }

    ----------------------------------------------------------;

    if (dst->head == NULL)
    {
        ----------------------------------------------------;
    }
    else
    {
        ----------------------------------------------------;
    }

    ----------------------------------------------------------;

}
```

4: Alternate Split

Given a linked list `src`, split it to two linked list `a` and `b` in alternating order. For example:

```
Before:
src:  1 -> 2 -> 3 -> 4 -> 5 -> NULL
a:  NULL
b:  NULL

After:
src:  NULL
a:  1 -> 3 -> 5 -> NULL
b:  2 -> 4 -> NULL
```

- You may assume `src` has more than 4 elements, i.e. after this function, `head` and `tail` of `a` and`b` point to different nodes.
- `a` and `b` are initially empty.
- Use `MoveHead` to implement this function.

```
void AlternateSplit (List *src, List *a, List *b)
{
    assert (a == NULL && b == NULL);

    while (--------------------------------------)
    {
        ----------------------------------------------------------;

        if (--------------------------------------)
        {
            ----------------------------------------------------------;
        }
    }
}
```

5: Sorted Merge

Given two non-empty sorted (in ascending order) linked list `a` and `b`. Merge them to a single empty sorted (in ascending order) linked list `dst`.

```
Before:
a:  1 -> 4 -> 6 -> 7 -> NULL
b:  2 -> 3 -> 5 -> NULL
dst:  NULL

After:
a:  NULL
b:  NULL
dst:  1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> NULL
```

- You may assume all the elements' data are unique.
- Use `MoveHead` to implement this function.

```
void SortedMerge (List *a, List *b, List *dst)
{
    assert(dst->head == NULL);
    assert(a->head != NULL && b->head != NULL);

    while (true)
    {
        if (a->head == NULL)
        {
            if (------------------------------)
            {
                ------------------------------------------------------;

                ------------------------------------------------------;

                b->head = b->tail = NULL;
            }

            ------------------------------------------------------------;
        }

        if (b->head == NULL)
        {
            // similar to above, ignore it
            ... ...
        }

        if (a->head->data < b->head->data)
        {
            ------------------------------------------------------------;
        }
        else
        {
            ------------------------------------------------------------;
        }
    }
}
```