Data Structure 2023
U2287
National Taipei University

Midterm

1. Please write-down the **time complexity** for each code segments in **Big-O** notation: (6%)

(a)

```
…
for (int r = 0 ; r < m ; ++r)
        for (int c = 0 ; c < n ; ++c)
                z[r][c] = x[r][c] + y[r][c];
```

(b)

```
if (i > 0)    i++;
else {
        for (int j = 0 ; j < n ; ++j)
                k++;
}
```

(a) O(n × m) (3%); (b) O(n) (3%)

2. Please determine the **memory address** for the **array** item in the below questions. Note the memory addresses are represented in **decimal** notation and the index of an array starts from **0**.

(a)

```
struct Employee {
        int ID;
        int age;
        int gender;
        int salary;
}
Employee employees[5];
```

(b)

img[0][0]	img[0][1]	img[0][2]	…	img[0][399]
img[1][0]	img[1][1]	img[1][2]		img[1][399]
img[2][0]	img[2][1]	img[2][2]		img[2][399]
⋮				
img[299][0]	img[299][1]	img[299][2]		img[299][399]

- Figure (a) shows the struct **Employee**. Assume the **employees** array starts at memory address **1000** and an integer takes **4 bytes** to store, what is the memory address of **employees[1].age**? (4%)
  1020 (4%)
- Figure (b) shows the image data stored in a **row-major** two-dimensional array (img[row][column]) starting at memory address **3000**, the image has 400 columns and 300 rows, and each pixel takes a **byte** to store, what is the memory address of the **img[100][50]**? (6%)
  3000+100*400*1+50*1 = 3000+40000+50 = 43050 (6%)

3. The following figure shows a representation of a **sparse matrix** using a 1-D array **smArray**:

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

	row	col	value
smArray[0]	0	2	3
smArray[1]	0	4	4
smArray[2]	1	2	5
smArray[3]	1	3	7
smArray[4]	3	1	2
smArray[5]	3	2	6

(a) Provide an algorithm to **transpose** the matrix by traversing the **smArray** only **once**. The transposed matrix should also be in **row-major** (6%)
We can use two additional arrays (3%):

- **rowSize[i]: keep the number of non-zero terms in the i-th row of A^T**
- **rowStart[i]: location of the 1-st non-zero term in i-th row of A^T**

We then go through each item in smArray. For each item, we use its column value to look up the rowStart array and find the correct index in the new array in the transpose matrix (3%).

(b) If your inputs are always very **dense**, should you use the above data structure or just store the matrix in a 2D array? Why? (6%)
It is better to use a 2D array (2%). The above algorithm has a complexity of O(row $\times$ column + column) which is slightly higher than the complexity of using a 2D array which is O(row $\times$ column). Moreover, it takes more memory to store the entries in the matrix (4%).

4. Please answer the following questions about **KMP** algorithm for string pattern matching:

(a) Build the **failure function** for the pattern **"ABABCA"**. You should also describe the meaning of the array values (10%).
the value stored in the i-th failure function array item is the ending index of the prefix belonging to the LPS in the substring $P_0 \cdots P_i$ (4%). If the meaning matches the answer in the failure function, each entry in the function can get 1% (total 6%).

A	B	A	B	C	A
-1	-1	0	1	-1	0

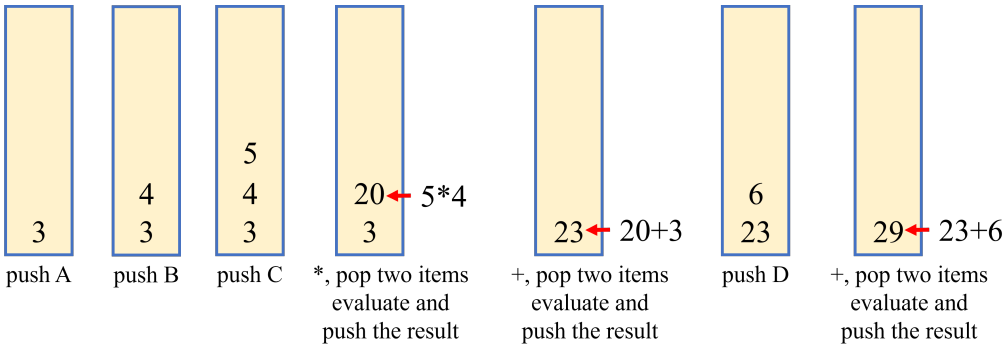
(b) Describe the main ideas of **KMP** algorithm. Your answer should include how the algorithm uses a **failure function** to avoid being a brute-force solution (6%)
The main idea of KMP algorithm is to exploit the information in previous comparisons and pattern to skip unnecessary comparisons, thus making the compared index never go back to achieve a linear time algorithm (3%). To achieve this, we should first build the failure function of the pattern for storing the LPS information. Comparisons are skipped based on the LPS information (3%). Some examples are also welcome.

5. When implementing a **stack** using an **array**, should we choose the **stack top** at the beginning or the end of the array? Justify your answer. (6%)
At the end of the array (2%). If we choose the top at the beginning, we will get an O(n) complexity for both push and pop operations. On the other hand, we can obtain an O(1) complexity for both push and pop operations by choosing the array end as the stack top (4%).

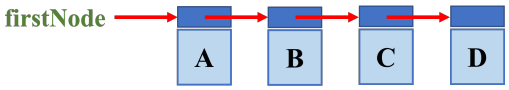
6. A **palindrome** is a string that reads the same from the left and from the right. For example, REDDER, I, ROTATOR, MOM, are palindromes. Please design an algorithm to determine whether a string is a palindrome using a **stack** (6%).
First find the middle of the string. If the length of the string L is even, push the first $L/2$ characters into a stack. Then, for each of the rest of the characters, compare its value with the top data of the stack. If they do not match, return false. If they match, pop the top data and test the next character in the string. Repeat the procedure

until the end of the string and return true if all comparisons succeed (4%). If L is odd, push the first $\lfloor L/2 \rfloor$ characters into a stack, skip the middle one, and repeat the above procedure for the last $\lfloor L/2 \rfloor$ characters (2%).

7. Describe an algorithm to evaluate the **postfix** expression **"ABC*+D+"** with a **stack**, assuming A = 3, B = 4, C = 5, and D = 6. Please draw the stack status step by step. (6%)



8. Please describe how to perform the following operations on the following **singly linked list**:



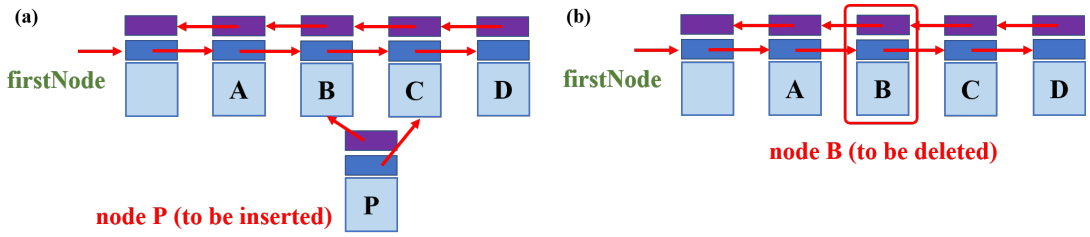
   (a) **Insert** a new node with data **G** at front (4%)
      i. **Create a node with data G**
      ii. **Set the pointer of the new node to the original first node**
      iii. **Update the *firstNode* pointer**

   (b) **Delete** the third node (C), the memory should be correctly released (4%)
      i. **Use a pointer *beforeNode* to record the node before the target index**
      ii. **Use a pointer *deleteNode* to identify the node to be deleted**
      iii. **Change the pointer in *beforeNode* by setting it to *deleteNode->next***
      iv. **Delete the *deleteNode***

   (c) **Reverse** the linked list (A → B → C → D) (4%)
      i. **Use a pointer *current* to record the location of the first node and a pointer *previous* to record the location of the previous node of *current***
      ii. **Use a node *temp* to record the location of the *previous* node**
      iii. **Move *previous* to the location of the *current* node**
      iv. **Update *current* to the location of the next node of *current***
      v. **Update the next node of *previous* to the location of the *temp***
      vi. **Repeat process ii to v until reaching the end of the list**
      vii. **Update *firstNode* to *previous***

9. Please describe how to perform the following operations on the following **doubly linked list**:

   (a) **Insert** a new node with data **P** as the fourth node (between B and C) (4%)
      i. **p->left = x**
      ii. **p->right = x->right**
      iii. **x->right->left = p**



iv. x->right = p

(b) **Delete** the third node (B), the memory should be correctly released (4%)

i. x->left->right = x->right
ii. x->right->left = x->left
iii. delete x

10. The following figure shows 2 **singly linked lists**. The list A has m nodes and the list B has n nodes. The numbers in the nodes are in **non-decreasing** order. Please design an algorithm to **merge** the two lists into one singly linked list sorted in **non-decreasing** order (6%). Also analyze the time complexity of your algorithm (4%).

(1) We can adopt a method similar to the polynomial addition: use two pointers *ptrA* and *ptrB* to locate the first node in *A* and *B*, respectively. If the number in *ptrA* is smaller than the one in *ptrB*, add a node with *A*'s number to the end of the merged list and update *ptrA* to its next node; otherwise, add a node with *B*'s number and update *ptrB* to its next node. The operations proceed until we reach the end of *A* or *B*. Then we copy the remaining nodes in *A* and *B* to the merged list (6%). The time complexity of the above operation is O(m+n) (4%).

11. When storing ordered data, what are the **advantages** and **disadvantages** of using an **array** and a **linked list**, respectively (8%)

Arrays:

- Pros: a specific item can be efficiently found by its index because the data is stored contiguously in the memory (2%)
- Cons: need some relocations after a new data is inserted or an old data is deleted (or the memory needs to be contiguous, thus being more difficult to allocate when the size is large, 2%)

Linked Lists:

- Pros: data can be stored discontinuously in the memory, thus inserting or deleting new data will be much easier (2%)
- Cons: need additional overhead to store the pointers, and it is impossible for random access (2%)