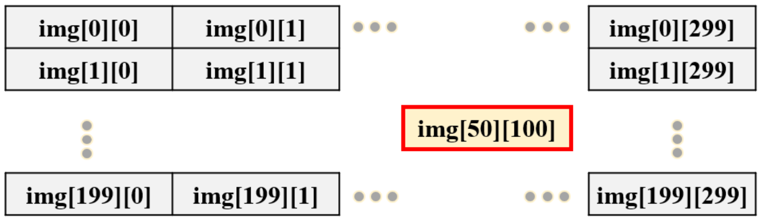
Midterm

1. Please determine whether the following asymptotic upper bounds are correct or not. If they are, provide a pair of the constants, c and n_0, that meet the criteria of the definition. (8%)
   (a) $10n + 600 = O(n^2)$
   correct (2%). c = 1 and $n_0 = 30$
   (b) $5n + 100 = O(n)$
   correct (2%). c = 6 and $n_0 = 100$

2. Please identify the **memory address** of each **array** element in the questions below. Note that memory addresses are in **decimal** notation, and array indexing begins at **0**.

(a)
```
struct Person {
    int age;
    char name[16];
    float data[2];
}
Person team[5];
```

(b)


   (a) Figure (a) shows the struct **Person**. Assume the **team** array begins at memory address **1000**, with sizes of **4** bytes for **int**, **1** byte for **char**, and **4** bytes for **float**. What is the memory address of **team[2].name**? (6%)
   1060
   (b) Figure (b) shows the image data stored in a **row-major** two-dimensional array, **img[row][column]**, beginning at memory address **2000**. The image has 300 columns and 200 rows, with each pixel taking **2 bytes** to store. What is the memory address of **img[50][100]**? (6%)
   2000+50*300*2+100*2 = 32200
   (c) Continuing from (b), what would be the memory address of **img[50][100]** if the 2D array is stored in **column-major** order? (4%)
   2000+100*200*2+50*2 = 42100

3. The figure below shows how the non-zero terms of a sparse matrix are stored in a 1D array, **smArray**. The **"Fast Transpose"** algorithm provides an efficient way to transpose the matrix by constructing two additional arrays, **rowSize** and **rowStart**.
   (a) Please fill in the **rowSize** and **rowStart** arrays (the status before filling in the **smArray** of the transposed matrix) (8%)
   See the figure below.
   (b) Please show how to determine the index to be filled for the first element in the original smArray (value = 4). (4%)
   As shown from the figure below, we first use the column of smArray[0] to index the rowStart array (lookup rowStart[2]). The value of rowStart[2] will be the index to fill in the smArray for A^T. After that, we should increase the rowStart[2] for the subsequent operations.

	rowSize	rowStart
[0]	(a) 1	(e) 0
[1]	(b) 1	(f) 1
[2]	(c) 2	(g) 2
[3]	(d) 1	(h) 4

A	row	col	value
smArray[0]	0	2	4
smArray[1]	1	0	2
smArray[2]	1	2	6
smArray[3]	2	1	3
smArray[4]	3	3	8

	rowSize	rowStart
[0]	(a) 1	(e) 0
[1]	(b) 1	(f) 1
[2]	~~(c) 2~~	(g) 2
[3]	(d) 1	(h) 4

A^T	row	col	value
smArray[0]			
smArray[1]			
smArray[2]			
smArray[3]			
smArray[4]			

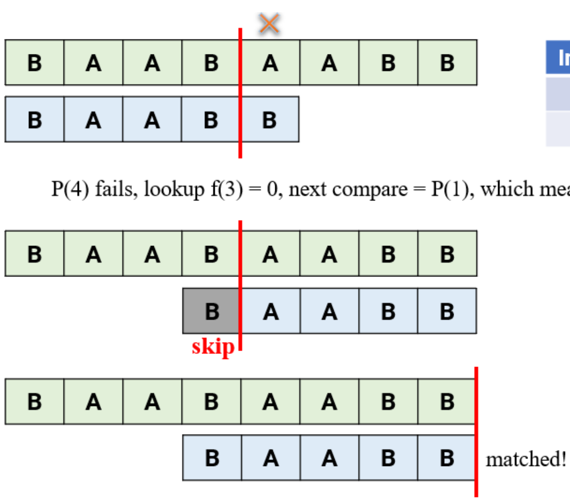
(c) Please analyze the **time complexity** of the **"Fast Transpose"** algorithm with **explanation**, assuming the original matrix has **m** rows, **n** columns, and **t** non-zero terms. (4%)
It takes O(n) to construct the rowSize and the rowStart arrays, and O(t) to fill in the smArray of the transposed matrix. As a result, the total time complexity is O(n+t).

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

(a) Construct the **failure function** for the pattern **"BAABB"**. and explain the meaning of the values in the array. (8%) **The answer is shown below:**

Index (j)	0	1	2	3	4
P	B	A	A	B	B
f	-1	-1	-1	0	0

(b) Describe how **KMP** algorithm uses the **failure function** to match a string **"BAABAABB"** and a pattern **"BAABB"**. (6%)
The most important three steps are shown below:

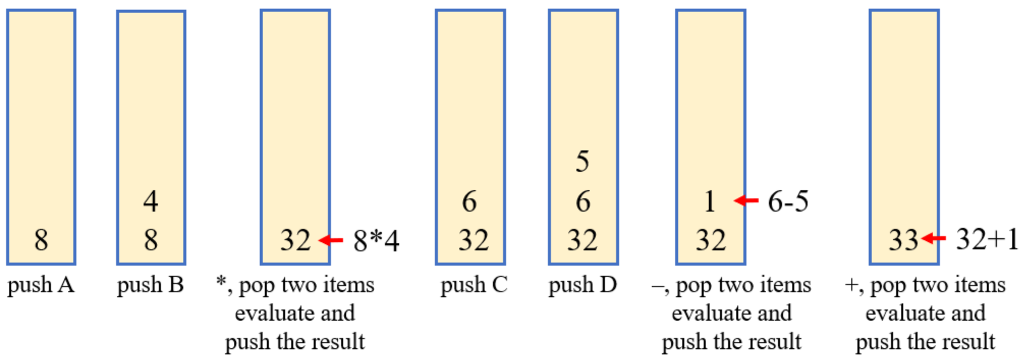


Index (j)	0	1	2	3	4
P	B	A	A	B	B
f	-1	-1	-1	0	0

5. Use a **stack** to convert the infix expression, **"A+B-C*D/E"**, into its postfix form. It is recommended to display your steps using the following table. (10%)
The answer is shown below:

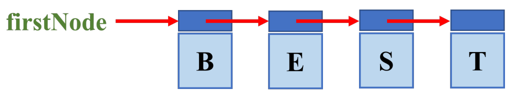
Next Token	Stack	Output
A	∅	A
+	+	A
B	+	AB
−	−	AB+
C	−	AB+C
*	−*	AB+C
D	−*	AB+CD
/	−/	AB+CD*
E	−/	AB+CD*E
		AB+CD*E/−

6. Use a **stack** to evaluate the **postfix** expression, **"AB*CD-+"**, given that A = 8, B = 4, C = 6, and D = 5. Show the stack contents at each step of the process. (8%)
The answer is shown below:



7. When implementing a **stack** using an **array**, should we choose the **stack top** at the beginning or the end of the array? Explain 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%).

8. Please show the steps to perform the following operations on the **singly linked list** shown below:



(a) **Insert** a new node containing data **A** at index 2 (between nodes E and S). (4%)

i. traverse to the second node (the node before the inserted position, E) from the firstNode and store it in a pointer, *beforeNode*.

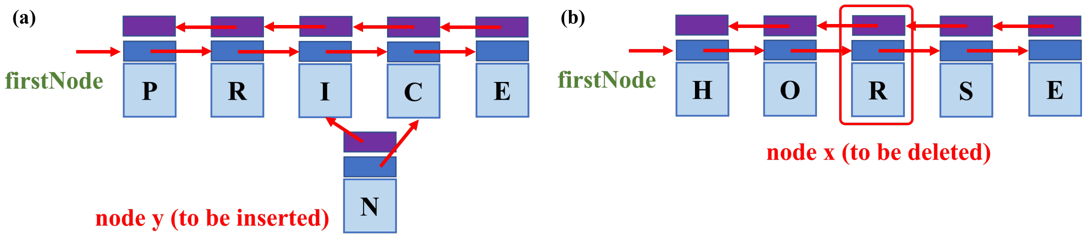
ii. Create a node with data A and set its pointer to the next node of *beforeNode*.

iii. Set the pointer of the *beforeNode* to the newly created node.

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

i. **traverse to the second node (the node before the target index, E) and record it using a pointer, *beforeNode***
ii. **Use another pointer, *deleteNode*, to identify the node to be deleted (the next node of *beforeNode*, S)**
iii. **Change the pointer of *beforeNode* to the next node of *deleteNode* (T)**
iv. **Delete the *deleteNode***

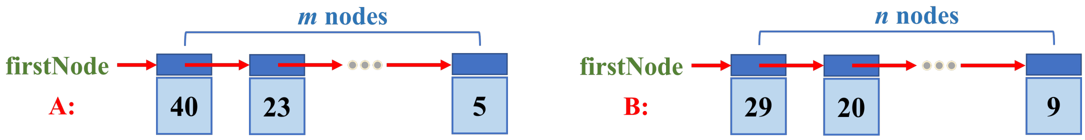
9. Please show the steps to perform the following operations on the **doubly linked list** below:



(a) **Insert** a new node **y** containing data **N** as the fourth node (4%)
   i. **traverse to the third node (the node before the inserted position, I) from the firstNode and store it in a pointer x.**
   ii. **y->left = x**
   iii. **y->right = x->right**
   iv. **x->right->left = y**
   v. **x->right = y**

(b) **Delete** the third node, the memory should be correctly released (4%)
   i. **traverse to the third node (the node to be deleted) from the firstNode and store it in a pointer x.**
   ii. **x->left->right = x->right**
   iii. **x->right->left = x->left**
   iv. **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 **decreasing** order. Please design an algorithm to **merge** the two lists into one singly linked list sorted in **decreasing** order (6%). Also, analyze the time complexity of your algorithm (4%).



(1) We can use a method similar to the polynomial addition by utilizing two pointers, *ptrA* and *ptrB*, to track the first node in *A* and *B*, respectively. If the value in *ptrA* is greater than the one in *ptrB*, we add a node with the value from *A* to the end of the merged list and move *ptrA* to its next node. Otherwise, we add a node with the value from *B* and advance *ptrB* to its next node. This process continues until we reach the end of either *A* or *B*, after which we append any remaining nodes from *A* or *B* to the merged list (6%). The time complexity of this operation is O(m+n) (4%).