DSA – IAT 1 Answer Key

1

Illustrate KMP algorithm to find a string pattern with an example.
String: abcdabcabcabbabcbcabc
Pattern String:abcabbabc

Answer:

Knuth-Morris-Pratt (KMP) string matching algorithm runs in O(m+n) time to find all occurrences of pattern P in S. **In KMP algorithm, a preprocessing is done in pattern string P and an array of length m is calculated.**
The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.
Unlike Brute-Force/Naïve algorithm, where we slide the pattern by one and compare all characters at each shift, we use a value from lps[] to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.
It uses a preprocessed table called "Prefix Table" or "**Failure** Table" to skip characters comparison while matching.
How to use failure[] to decide next positions (or to know a number of characters to be skipped)?
We start comparison of pat[j] with j = 0 with characters of current window of text.
We keep matching characters str[i] and pat[j] and keep incrementing i and j while pat[j] and str[i] keep **matching**.
When we see a **mismatch**
We know that characters pat[0..j-1] match with str[i-j…i-1] (Note that j starts with 0 and increment it only when there is a match).
We also know (from above definition) that failure[j-1] is count of characters of pat[0…j-1] that are both proper prefix and suffix.
From above two points, we can conclude that we do not need to match these failure[j-1] characters with str[i-j…i-1] because we know that these characters will anyway match.

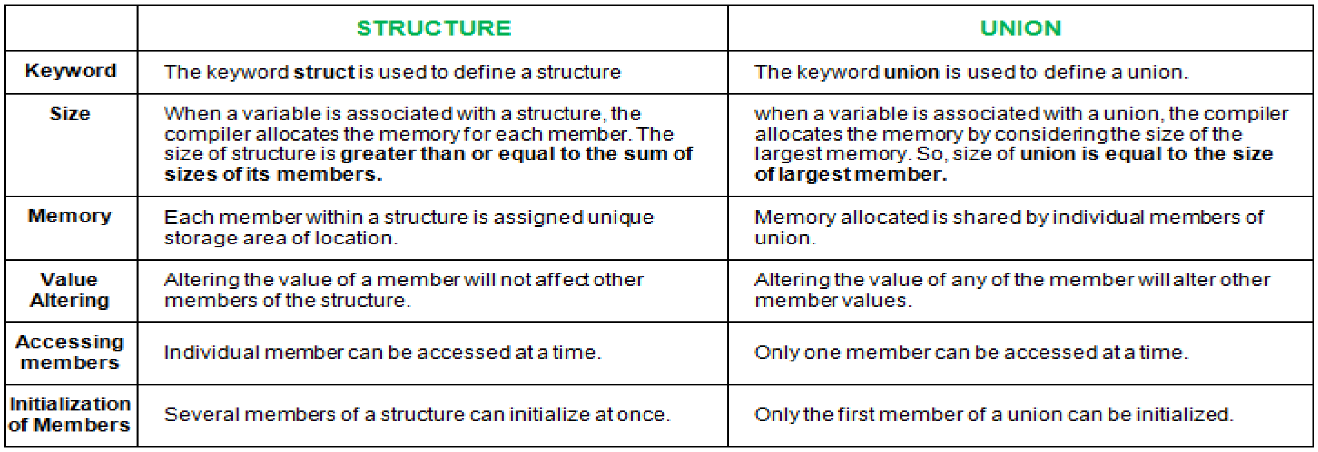
```
void fail(char *pat, int failure[])
{
	int i,j;
	int n = strlen(pat);
	failure[0] = -1;
	for (j=1; j<n; j++)

	{
		i = failure[j-1];
		while (( pat[j] != pat[i+1]) && (i>= 0))
			i= failure[i];
		if (pat[j] == pat[i+1])
			failure[j] = i+1;
		else
			failure[j] = -1;
	}
}
```

a	b	C	a	b	b	a	b	c
-1	-1	-1	0	1	-1	0	1	2

	abcdabcabcabbabcbcabc
	abcabbabc
	        abcabbabc
	            abcabbabc

2	Convert the infix expression a/b – c+ d * e – a * c into postfix expression. Write a function to evaluate that postfix expression and trace that for given data a=6, b=3, c = 1, d = 2, e =4. Answer: Infix to postfix : ab/c-de*+ac*- (With tracing) POSTFIX EVALUATION : 63/1-24*+61*- = 3 (with tracing and function)

3a	What is structure? How it is different from array? Explain different types of structure declaration with examples and give difference between Union and Structure.
	
	Answer:
	
	Data are a collection of facts or simply values or sets of values.
	Data structure is representation of the logical or mathematical model of a particular organization of data.
	We can declare a structure using "struct" keyword. A structure must be declared first before using it just like all other data type. Structure can be declared by two ways.
	Tagged Declaration
	Typedef Declaration
	Typedef:
	typedef struct
	{
	data-type var-name1;
	data-type var-name2;
	:
	data-type var-nameN;
	}*identifier*; // global declaration
	Tagged:
	struct *tag_name*
	{
	data-type var-name1;
	data-type var-name2;
	:
	data-type var-nameN;
	};



	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

3b Explain dynamic memory allocation functions in details

Answer:

Malloc

Calloc

Realloc

Free

With syntax and examples

4 Write the function to perform the following on SLL: Insertion and deletion from front and end.

```
#include <stdio.h>
#include <stdlib.h>

// Definition of a node
struct Node {
    int data;
    struct Node* next;
};

// Global head pointer
struct Node* head = NULL;

// Function to insert at the front
void insertFront(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = head;
    head = newNode;
    printf("Inserted %d at the front.\n", value);
}

// Function to insert at the end
void insertEnd(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (head == NULL) {  // If the list is empty
        head = newNode;
        printf("Inserted %d at the end.\n", value);
        return;
    }

    struct Node* temp = head;
    while (temp->next != NULL) {  // Traverse to the end of the list
        temp = temp->next;
    }
    temp->next = newNode;
    printf("Inserted %d at the end.\n", value);
}

// Function to delete from the front
void deleteFront() {
    if (head == NULL) {
        printf("List is empty. Cannot delete from front.\n");
        return;
    }
    struct Node* temp = head;
    head = head->next;
    printf("Deleted %d from the front.\n", temp->data);
    free(temp);
}

// Function to delete from the end
void deleteEnd() {
    if (head == NULL) {
        printf("List is empty. Cannot delete from end.\n");
        return;
    }
    if (head->next == NULL) {  // If the list has only one node
        printf("Deleted %d from the end.\n", head->data);
        free(head);
        head = NULL;
        return;
    }

    struct Node* temp = head;
    struct Node* prev = NULL;

    while (temp->next != NULL) {  // Traverse to the end of the list
        prev = temp;
        temp = temp->next;
    }
    prev->next = NULL;  // Disconnect the last node
    printf("Deleted %d from the end.\n", temp->data);
    free(temp);
}

// Function to display the list
void displayList() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    printf("List elements: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function to test the linked list operations
int main() {
    insertFront(10);
    insertFront(20);
    insertEnd(30);
    insertEnd(40);

    displayList();

    deleteFront();
    displayList();

    deleteEnd();
    displayList();

    deleteEnd();
    deleteFront();
    displayList();

    return 0;
}
```

5 Explain Sparse matrices. Implement the fast transpose algorithm for it.

$$\begin{bmatrix} 0 & 9 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Answer:

Sparse matrix:

5	8	8
0	1	9
0	5	4
1	2	6
1	6	1
2	3	5
2	6	1
3	6	3
4	2	6

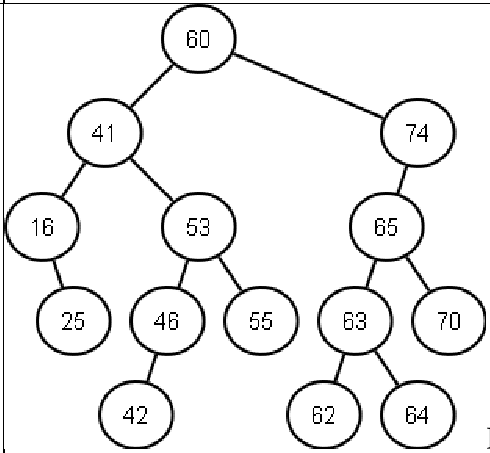
Fast transpose:

5	8	8
1	0	9
2	1	6
2	4	6
3	2	5
5	0	4
6	1	1
6	2	1
6	3	3

```
void transpose(int trip1[][3],int trip2[][3])
{
    int x,y,z,n;
    trip2[0][0] = trip1[0][1];
```

```
   trip2[0][1] = trip1[0][0];
   trip2[0][2] = trip1[0][2];
   z=1;
   n=trip1[0][2];
   for(x=0;x<trip1[0][1];x++)
           for(y=1;y<=n;y++)
              /*if a column number of current triple==x
              then insert the current triple in b2 */
              if(x==trip1[y][1])
              {
                        trip2[z][0]=x;
                        trip2[z][1]=trip1[y][0];
                        trip2[z][2]=trip1[y][2];
                        z++;
            }
}
```

5B



Preorder: 60,41,16,25,53,46,42,55,74,65,63,62,64,70
Inorder: 16,25,41,42,46,53,55,60,62,63,64,65,70,74
Post order: 25,16,42,46,55,53,41,62,64,63,70,65,74,60

6

```
#include <stdio.h>
#define SIZE 5  // Define the size of the circular queue

int queue[SIZE];
int front = -1;
int rear = -1;

// Function to insert an element into the circular queue
void QINSERT(int element) {
   if ((rear + 1) % SIZE == front) {
      printf("Queue Overflow! Cannot insert %d\n", element);
      return;
   } else if (front == -1) { // First insertion
```

```
        front = 0;
        rear = 0;
    } else {
        rear = (rear + 1) % SIZE; // Increment rear circularly
    }
    queue[rear] = element;
    printf("Inserted %d into the queue.\n", element);
}

// Function to delete an element from the circular queue
int QDELETE() {
    if (front == -1) {
        printf("Queue Underflow! Cannot delete.\n");
        return -1;
    }
    int deletedElement = queue[front];
    if (front == rear) { // Queue becomes empty after deletion
        front = -1;
        rear = -1;
    } else {
        front = (front + 1) % SIZE; // Increment front circularly
    }
    printf("Deleted %d from the queue.\n", deletedElement);
    return deletedElement;
}

// Function to display the elements of the circular queue
void Display() {
    if (front == -1) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Queue elements: ");
    int i = front;
    while (1) {
        printf("%d ", queue[i]);
        if (i == rear) break;
        i = (i + 1) % SIZE;
    }
    printf("\n");
}

// Main function to test the circular queue
int main() {
    QINSERT(10);
    QINSERT(20);
    QINSERT(30);
    QINSERT(40);
    QINSERT(50); // Queue is now full

    Display();

    QDELETE();
    QDELETE();
    Display();

    QINSERT(60);
    QINSERT(70);
    Display();

    QDELETE();
    Display();

    return 0;
}
```