Solutions

Q.1 a. What is a pointer? Explain how it is declared and initialized. (4)

Answer:
Different from other normal variables which can store values, pointers are special variables that can hold the address of a variable. Since they store memory address of a variable, the pointers are very commonly said to "point to variables"

A pointer is declared as :
<pointer type> *<pointer-name>
In the above declaration :
1. pointer-type : It specifies the type of pointer. It can be int,char, float etc. This type specifies the type of variable whose address this pointer can store.
2. pointer-name : It can be any name specified by the user. Professionally, there are some coding styles which every code follows. The pointer names commonly start with 'p' or end with 'ptr'
An example of a pointer declaration can be :
char *chptr;
In the above declaration, 'char' signifies the pointer type, chptr is the name of the pointer while the asterisk '*' signifies that 'chptr' is a pointer variable. A pointer is initialized in the following way :
<pointer declaration(except semicolon)> = <address of a variable>
OR
<pointer declaration>
<name-of-pointer> = <address of a variable>

b. What is dynamic memory allocation? Write and explain the different dynamic memory allocation functions in C. (4)

Answer:
The exact size of array is unknown untill the compile time, i.e., time when a compiler compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

c. State the difference between arrays and linked lists. (4)

Answer:

Arrays

Size of an array is fixed

It is necessary to specify the number of elements during declaration

Insertions and deletions are somewhat difficult

It occupies less memory than a linked list for the same number of elements

Linked Lists

Size of a list is variable

It is not necessary to specify the number of elements during declaration

Insertions and deletions are carried out easily

It occupies more memory

d. It is generally said that searching a node in a binary search tree is more efficient than that of a simple binary tree. Why? (4)

Answer:

In binary search tree, the nodes are arranged in such a way that the left node is having less data value than root node value and the right nodes are having larger value than that of root. Because of this while searching any node the value of the target node will be compared with the parent node and accordingly either left sub branch or right sub branch will be searched. So, one has to compare only particular branches. Thus searching becomes efficient.

e. What is the need for Priority queue? (4)

Answer:

In a multiuser environment, the operating system scheduler must decide which of the several processes to run only for a fixed period of time. One algorithm uses queue. Jobs are initially placed at the end of the queue. The scheduler will repeatedly take the first job on the queue, run it until either it finishes or its time limit is up and place it at the end of the queue if it does not finish. This strategy is not appropriate, because very short jobs will soon to take a long time because of the wait involved in the run. Generally, it is important that short jobs finish as fast as possible, so these jobs should have precedence over jobs that have already been running. Further more, some jobs that are not short are still very important and should have precedence. This particular application seems to require a special kind of queue, known as priority queue. Priority queue is also called as Heap or Binary Heap

f. What is a spanning Tree? Does the minimum spanning tree of a graph give the shortest distance between any two specified nodes? (4)

Answer:

A spanning tree is a tree associated with a network. All the nodes of the graph appear on the tree once. A minimum spanning tree is a spanning tree organized so that the total edge weight between nodes is minimized.

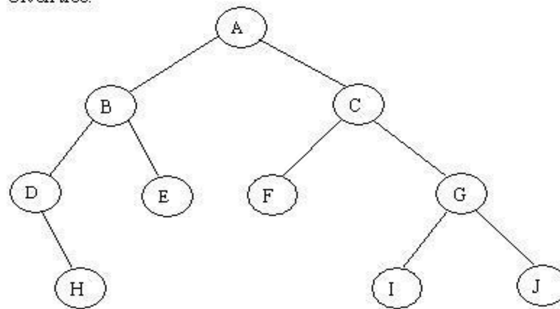
No. Minimal spanning tree assures that the total weight of the tree is kept at its minimum. But it doesn't mean that the distance between any two nodes involved in the minimum-spanning tree is

minimum.

g. What do you mean by balanced trees? (4)

Answer:
Balanced trees have the structure of binary trees and obey binary searchtreeproperties. Apart from these properties, they have some special constraints,which differfrom one data structure to another. However, these constraints areaimed only at reducingthe height of the tree, because this factor determines thetime complexity.Eg: AVL trees, Splay trees

Q.2 a. Traverse the given tree using Inorder, Preorder and Postorder traversals.(6)



Answer:
•Inorder : D H B E A F C I G J

•Preorder: A B D H E C F G I J

•Postorder: H D E B F I J G C A

b. Write a C Program to add a new node to the ascending order singly linked list. (8)

Answer:
```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
/* structure containing a data part and link part */
struct node
{
int data ;
struct node *link ;
} ;
void add ( struct node **, int ) ;
void display ( struct node * ) ;
int count ( struct node * ) ;
```

```
void delete ( struct node **, int ) ;
void main( )
{
struct node *p ;
p = NULL ; /* empty linked list */
add ( &p, 5 ) ;
add ( &p, 1 ) ;
add ( &p, 6 ) ;
add ( &p, 4 ) ;
add ( &p, 7 ) ;
clrscr( ) ;
display ( p ) ;
printf ( "\nNo. of elements in Linked List = %d", count ( p ) ) ;
}
/* adds node to an ascending order linked list */
void add ( struct node **q, int num )
{
struct node *r, *temp = *q ;
r = malloc ( sizeof ( struct node ) ) ;
r -> data = num ;
/* if list is empty or if new node is to be inserted before the first node */
if ( *q == NULL || ( *q ) -> data > num )
{
*q = r ;
( *q ) -> link = temp ;
}
else
{
/* traverse the entire linked list to search the position to insert the
new node */
while ( temp != NULL )
```

```
{
if ( temp -> data <= num && ( temp -> link -> data > num ||
temp -> link ==
NULL ))
{
r -> link = temp -> link ;
temp -> link = r ;
return ;
}
temp = temp -> link ; /* go to the next node */
}
}
}
/* displays the contents of the linked list */
void display ( struct node *q )
{
printf ( "\n" ) ;
/* traverse the entire linked list */
while ( q != NULL )
{
printf ( "%d ", q -> data ) ;
q = q -> link ;
}
}
/* counts the number of nodes present in the linked list */
int count ( struct node *q )
{
int c = 0 ;
/* traverse the entire linked list */
while ( q != NULL )
{
```

```
q = q -> link ;
c++ ;
}
return c ;
}
```

c. What is the difference between storing data on the heap vs. on the stack? (4)

Answer:
The stack is smaller, but quicker for creating variables, while the heap is limited in size only by how much memory can be allocated. Stack would include most compile time variables, while heap would include anything created with malloc or new.

Q.3 a. Sort the given values using Quick Sort. 65 70 75 80 85 60 55 50 45 (5)

Answer:
Sorting takes place from the pivot value, which is the first value of the given elements, this is marked bold. The values at the left pointer and right pointer are indicated using L and R respectively.

65 70L 75 80 85 60 55 50 45R

Since pivot is not yet changed the same process is continued after interchanging the values at L and R positions

65 45 75L 80 85 60 55 50R 70
65 45 50 80L 85 60 55R 75 70
65 45 50 55 85L 60R 80 75 70
65 45 50 55 60R 85L 80 75 70

When the L and R pointers cross each other the pivot value is interchanged with the value at right pointer. If the pivot is changed it means that the pivot has occupied its original position in the sorted order (shown in bold italics) and hence two different arrays are formed, one from start of the original array to the pivot position-1 and the other from pivot position+1 to end.

60L 45 50 55R 65 85L 80 75 70R
55L5 45 50R 60 65 70R 80L 75 85
50L 45R 55 60 65 70 80L 75R 85
In the next pass we get the sorted form of the array.
45 50 55 60 65 70 75 80 85

b. Write a program to find the minimum cost of a spanning tree. (8)

Answer:

```
#include <stdio.h>

#include <conio.h>
```

```
#include <alloc.h>
struct lledge
{
int v1, v2 ;
float cost ;
struct lledge *next ;
} ;
int stree[5] ;
int count[5] ;
int mincost ;
struct lledge * kminstree ( struct lledge *, int ) ;
int getrval ( int ) ;
void combine ( int, int ) ;
void del ( struct lledge * ) ;
void main( )
{
struct lledge *temp, *root ;
int i ;
clrscr( ) ;
root = ( struct lledge * ) malloc ( sizeof ( struct lledge ) ) ;
root -> v1 = 4 ;
root -> v2 = 3 ;
root -> cost = 1 ;
temp = root -> next = ( struct lledge * ) malloc ( sizeof ( struct lledge ) ) ;
temp -> v1 = 4 ;
temp -> v2 = 2 ;
temp -> cost = 2 ;
temp -> next = ( struct lledge * ) malloc ( sizeof ( struct lledge ) ) ;
temp = temp -> next ;
temp -> v1 = 3 ;
temp -> v2 = 2 ;
```

```
temp -> cost = 3 ;
temp -> next = ( struct lledge * ) malloc ( sizeof ( struct lledge ) ) ;
temp = temp -> next ;
temp -> v1 = 4 ;
temp -> v2 = 1 ;
temp -> cost = 4 ;
temp -> next = NULL ;
root = kminstree ( root, 5 ) ;
for ( i = 1 ; i <= 4 ; i++ )
printf ( "\nstree[%d] -> %d", i, stree[i] ) ;
printf ( "\nThe minimum cost of spanning tree is %d", mincost ) ;
del ( root ) ;
getch( ) ;
}
struct lledge * kminstree ( struct lledge *root, int n )
{
struct lledge *temp = NULL ;
struct lledge *p, *q ;
int noofedges = 0 ;
int i, p1, p2 ;
for ( i = 0 ; i < n ; i++ )
stree[i] = i ;
for ( i = 0 ; i < n ; i++ )
count[i] = 0 ;
while ( ( noofedges < ( n - 1 ) ) && ( root != NULL ) )
{
p = root ;
root = root -> next ;
p1 = getrval ( p -> v1 ) ;
p2 = getrval ( p -> v2 ) ;
if ( p1 != p2 )
```

```
{
combine ( p -> v1, p -> v2 ) ;
noofedges++ ;
mincost += p -> cost ;
if ( temp == NULL )
{
temp = p ;
q = temp ;
}
else
{
q -> next = p ;
q = q -> next ;
}
q -> next = NULL ;
}
}
return temp ;
}
int getrval ( int i )
{
int j, k, temp ;
k = i ;
while ( stree[k] != k )
k = stree[k] ;
j = i ;
while ( j != k )
{
temp = stree[j] ;
stree[j] = k ;
j = temp ;
```

```
}
return k ;
}
void combine ( int i, int j )
{
if ( count[i] < count[j] )
stree[i] = j ;
else
{
stree[j] = i ;
if ( count[i] == count[j] )
count[j]++ ;
}
}
void del ( struct lledge *root )
{
struct lledge *temp ;
while ( root != NULL )
{
temp = root -> next ;
free ( root ) ;
root = temp ;
}
}
```

c. Write a brief note on External variables. (5)

Answer:
This algorithm creates a *forest* of trees. Initially the forest consists of **n** single node trees (and no edges). At each step, we add one (the cheapest one) edge so that it joins two trees together. If it were to form a cycle, it would simply link two nodes that were already part of a single connected tree, so that this edge would not be needed.

Q.4 a. Explain how to check the validity of an expression containing nested parentheses? (5)

© IETE

Answer:

One of the applications of stack is checking validity of an expression containing nested parenthesis. An expression will be valid if it satisfies the two conditions.

1.The total number of left parenthesis should be equal to the total number of right parenthesis in the expression.

2. For every right parenthesis there should be a left parenthesis of the same time.

The procedure for checking validity of an expression containing nested parenthesis:

1. First take an empty stack
2. Scan the symbols of expression from left to right.
3. If the symbol is a left parenthesis then push it on the stack.
4. If the symbol is right parenthesis then If the stack is empty then the expression is invalid because Right parentheses are more than left parenthesis. else Pop an element from stack. If popped parenthesis does not match the parenthesis being scanned then it is invalid because of mismatched parenthesis.
5. After scanning all the symbols of expression, if stack is empty then expression is valid else it is invalid because left parenthesis is more than right parenthesis.

b. Explain the following with an example: i) forest ii) graph iii) winner tree (9)

Answer:

i) The default interdomain trust relationships are created by the system during domain controller creation. The number of trust relationships that are required to connect *n* domains is *n* –1, whether the domains are linked in a single, contiguous parent-child hierarchy or they constitute two or more separate contiguous parent-child hierarchies.

When it is necessary for domains in the same organization to have different namespaces, create a separate tree for each namespace. In Windows 2000, the roots of trees are linked automatically by two-way, transitive trust relationships. Trees linked by trust relationships form a forest A single tree that is related

to no other trees constitutes a forest of one tree.

ii) A graph, *G*, consists of two sets *V* and *E*. *V* is a finite non-empty set of *vertices. E* is a set of pairs of vertices, these pairs are called *edges*. *V*(*G*) and *E*(*G*) will represent the sets of vertices and edges of graph *G*.

We will also write *G* = (*V,E*) to represent a graph.

In an *undirected graph* the pair of vertices representing any edge is unordered . Thus, the pairs (*v*1, *v*2) and (*v*2, *v*1) represent the same edge.

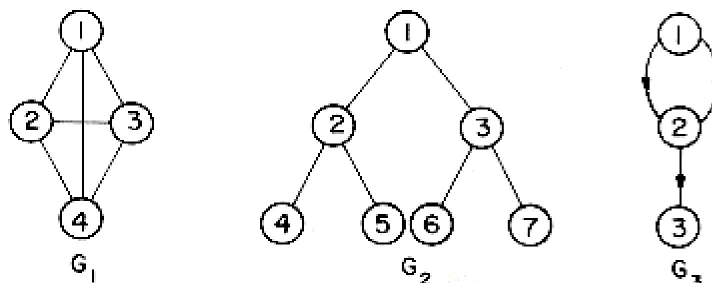
In a *directed graph* each edge is represented by a directed pair (*v*1, *v*2). *v*1 is the *tail* and *v*2 the *head* of the edge. Therefore <*v*2, *v*1> and <*v*1, *v*2> represent two different edges. The graphs *G*1 and *G*2 are undirected. *G*3 is a directed graph.

$V(G1) = \{1,2,3,4\}$; $E(G1) = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$

$V(G2) = \{1,2,3,4,5,6,7\}$; $E(G2) = \{(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)\}$

$V(G3) = \{1,2,3\}$; $E(G3) = \{\langle 1,2\rangle, \langle 2,1\rangle, \langle 2,3\rangle\}$.

Example:



iii) Complete binary tree with k external nodes and k - 1 internal nodes.

External nodes represent tournament players.

Each internal node represents a match played between its two children;

the winner of the match is stored at the internal node.

Root has overall winner

c. Explain and derive the complexity of insertion sort. (4)

Answer:

The worst-case performance occurs when the elements of the input array are in descending order. In the worst-case, the first pass will require one comparison, the second pass will require 2 comparisons, and so on until the last pass which will require (n-1) comparisons. In general, the kth pass will require k-1 comparisons.

Therefore the total number of comparisons is:

F(n)=1+2+3+…..+(n-k)+….+(n-3)+(n-2)+(n-1)

=n(n-1)/2

=O(n2)

Q.5 a. Suppose that a linked list is provided that is either circular or not circular. Write a function that takes as an input a pointer to the head of a linked list and determines whether the list is circular or if the list has an ending node. If the linked list is circular then the function should return true, otherwise should return false if the linked list is not circular. **(8)**

Answer:

```
bool findCircular(Node *head)
{
  Node *slower, * faster;
```

```
  slower = head;
  faster = head->next; //start faster one node ahead
  while(true) {

    // if the faster pointer encounters a NULL element
    if( !faster || !faster->next)
      return false;
   //if faster pointer ever equals slower or faster's next
   //pointer is ever equal to slow then it's a circular list
    else if (faster == slower || faster->next == slower)
       return true;
    else{
      // advance the pointers
       slower = slower->next;
       faster = faster->next->next;
     }
  }
}
```

b. Given a singly linked list, delete all occurrences of a given key in it. For example, consider the following list.

Input: 2 -> 2 -> 1 -> 8 -> 2 -> 3 -> 2 -> 7
Key to delete = 2
Output: 1 -> 8 -> 3 -> 7 **(10)**

Answer:
Following is C implementation for the same.

```
// C Program to delete all occurrences of a given key in linked list
#include <stdio.h>
#include <stdlib.h>
 // A linked list node
struct node
{
```

```
    int data;
    struct node *next;
};
 /* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->data  = new_data;
    new_node->next = (*head_ref);
    (*head_ref)    = new_node;
}
 /* Given a reference (pointer to pointer) to the head of a list and
   a key, deletes all occurrence of the given key in linked list */
void deleteKey(struct node **head_ref, int key)
{
    // Store head node
    struct node* temp = *head_ref, *prev;
     // If head node itself holds the key or multiple occurrences of key
    while (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next;   // Changed head
        free(temp);               // free old head
        temp = *head_ref;         // Change Temp
    }
     // Delete occurrences other than head
    while (temp != NULL)
    {
        // Search for the key to be deleted, keep track of the
        // previous node as we need to change 'prev->next'
        while (temp != NULL && temp->data != key)
```

```
        {
           prev = temp;
           temp = temp->next;
        }
         // If key was not present in linked list
        if (temp == NULL) return;
         // Unlink the node from linked list
        prev->next = temp->next;
         free(temp);  // Free memory
         //Update Temp for next iteration of outer loop
        temp = prev->next;
    }
}
 // This function prints contents of linked list starting from
// the given node
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}
 /* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
     push(&head, 7);
    push(&head, 2);
    push(&head, 3);
```

```
   push(&head, 2);
   push(&head, 8);
   push(&head, 1);
   push(&head, 2);
   push(&head, 2);
    int key = 2; // key to delete
    puts("Created Linked List: ");
   printList(head);


   deleteKey(&head, key);
   puts("\nLinked List after Deletion of 1: ");
    printList(head);
   return 0;
}
}
```

Q.6 a. Define AVL Tree. Explain various rotations of AVL Trees maintaining balance factor while insertion and deletion takes place. (12)

Answer:

According to Knuth, the AVL version of the binary search tree represents a nice compromise between the "optimum" binary tree, whose height is minimal but for which maintenance is seemingly difficult, and the "arbitrary" binary tree, which requires no special maintenance but whose height could possibly grow to be much greater than minimal.

AVL Property: A tree is said to be height-balanced tree, iff all of its nodes, the difference in height between the left and right subtrees is the same or, if not, which of the two subtrees has height one unit larger (balance factor of 0, 1 or -1).

AVL Node Structure: To maintain the balance in a height balanced binary search tree, each node will have to keep an additional piece of information (balance factor) that is needed to efficiently maintain balance in the tree after every insert and delete operation has been performed.

BALANCE FACTOR

KEY

LEFT RIGHT

b. Write a brief note on m-way search tree. (6)

Answer:

An m-way search tree

a. is empty or

b. consists of a root containing **j** (1<=**j**<**m**) keys, **kj**, and a set of sub-trees, **Ti**, (**i** = 0..**j**), such that

i. if **k** is a key in **T0**, then **k** <= **k1**

ii. if **k** is a key in **Ti** (0<**i**<**j**), then **ki** <= **k** <= **ki+1**

iii. if **k** is a key in **Tj**, then **k** > **kj** and

iv. all **Ti** are nonempty m-way search trees or all **Ti** are empty

Q.7 Write short notes on (6×3=18)

(i) Stable marriage problem

(ii) The Max-Flow Min-Cut Theorem

(iii) Boundary Tag method

Answer:

(i) Stable marriage problem

Given N men and N women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are "stable".

Consider the following example.

Let there be two men m1 and m2 and two women w1 and w2.

Let m1's list of preferences be {w1, w2}

Let m2's list of preferences be {w1, w2}

Let w1's list of preferences be {m1, m2}

Let w2's list of preferences be {m1, m2}

The matching { {m1, w2}, {w1, m2} } is not stable because m1 and w1 would prefer each other over their assigned partners. The matching {m1, w1} and {m2, w2} is stable because there are no two people of opposite sex that would prefer each other over their assigned partners.

It is always possible to form stable marriages from lists of preferences Following is algorithm to find a stable matching:

The idea is to iterate through all free men while there is any free man available. Every free man goes to all women in his preference list according to the order. For every woman he goes to, he checks if the woman is free, if yes, they both become engaged. If the woman is not free, then the woman chooses either says no to him or dumps her current engagement according to her preference list. So an engagement done once can be broken if a woman gets better option

(ii) The Max-Flow Min-Cut Theorem

In addition to edge capacity, consider there is capacity at each vertex, that is, a mapping $c : V \rightarrow \mathbf{R}^+$, denoted by $c(v)$, such that the flow f has to satisfy not only the capacity constraint and the conservation of flows, but also the vertex capacity constraint

$$\forall v \in V \setminus \{s, t\} : \qquad \sum_{i \in V} f_{iv} \leq c(v).$$

In other words, the amount of *flow* passing through a vertex cannot exceed its capacity. Define

an *s-t cut* to be the set of vertices and edges such that for any path from *s* to *t*, the path contains a member of the cut. In this case, the *capacity of the cut* is the sum the capacity of each edge and vertex in it.
In this new definition, the **generalized max-flow min-cut theorem** states that the maximum value of an s-t flow is equal to the minimum capacity of an s-t cut in the new sense.

(iii) Boundary Tag method
In multiprogramming systems, processes share a common store. Processes need space for:
code (instructions)
static data (compiler initialized variables, strings, etc.)
global data (global variables)
stack (local variables)
heap (new, malloc)
Memory management is complicated by: unbounded demand | processes may require more than the total physical storage of the machine protection vs. sharing
1. some parts of memory allocated to one process must be protected from another
2. some memory can be shared among processes for better resource utilization (space and time) dynamic memory requirements | stacks and heap grow and shrink dynamically. The operating system may need to take memory from one process to give to another boundary-tag | keep doubly linked list of all pieces of memory. Reserved fields associated with each piece indicate: status of piece (used or unused) start and end of piece
Getmem searches the list for a piece of adequate size. If an exact size match cannot be found, a large piece is split into two smaller pieces, one returned to the user, the other returned to the free list.
Freemem combines adjacent pieces into a single larger piece and when servicing a getmem request, which piece should be chosen?

TEXT BOOKS

I. Kernighan and Ritchie, "The C Programming Language", Prentice Hall of India, 1989
II. Tenenbaum, Augestein and Langsam "Data Structures using C", Prentice Hall, 1992
III. Horowitz and Sahani, "Fundamentals of Data Structures ", Galgotia Book Source (GBS) Publication, reprint, 1994

© IETE