B.TECH.

TCS - 302

THIRD SEMESTER EXAMINATION, 2007-2008

DATA STRUCTURE USING 'C'

Time: 3 Hours *Total Marks:* 100

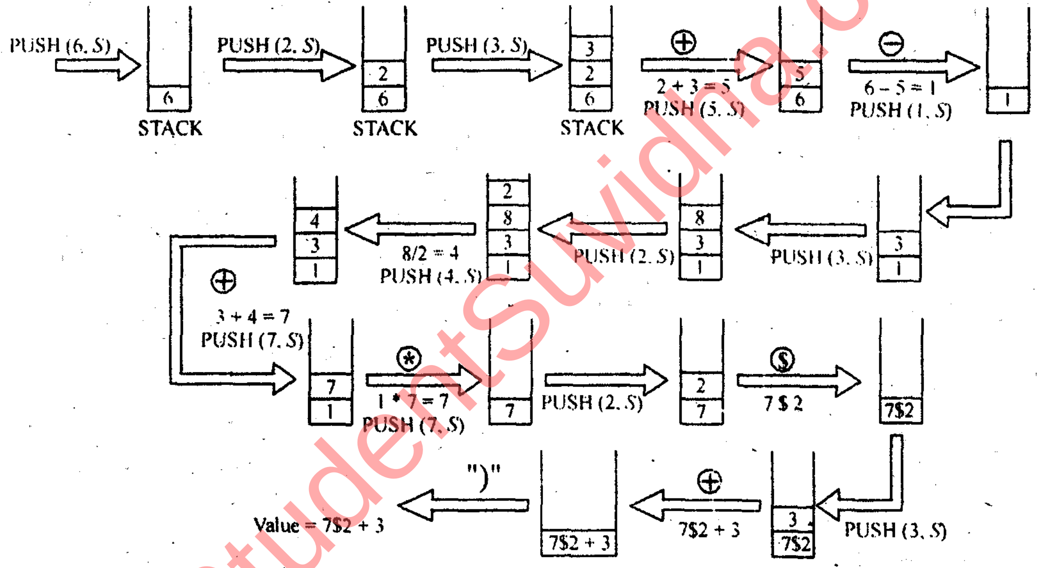
Note: *Attempt all questions.*

Q.1. Attempt any four parts of the following: **(5×4=20)**

1. (*a*) Show the detailed contents of the stack for given postfix expression to evaluate
6 2 3 + – 3 8 2 / + * 2 $ 3 +

Ans. Given expression is

P; 6, 2, 3, +, –, 3, 8, 2; / + * 2 $ 3 +) not right paranthesis at end of equation



Answer:

Value = 7 * 2 + 3

Value = 17 $ → * or ↑

Value = 7 ↑ 2 + 3 = 49 + 3

Value = 52.

1. (*b*) Define stack as a data structure and discuss its applications.

Ans. A stack is an ordered list in which all insertions and deletion are made at one end, called the ToP. Since the last element to be inserted into the stack will be the first to be required. In this way stacks referred to as (LIFO).

```
sruct stack                 void create stack (struct stack *S)
{                           {
      int ToP,                   S → ToP = – 1,
      int A[20];            }
```

```
void PUSH (struct stack *S, int value)
{
        S → ToP = S → ToP + 1,
        S → A[S → ToP] = value
}
int PoP (struct stack *S)
{
        int X;
            X = S → A[S → ToP]
            S → ToP = S → ToP – 1
            return (X),
}
```

Application of stacks:

1. Stacks are used to pass parameters between functions.
2. Stacks are used in recursion.
3. Expression evaluation.
4. Polish notation.
5. Stacks are used when a program uses sub-programs.

1. (c) Write a C function to convert a valid paranthesized infix expression to postfix form.

Ans. Infix expression to postfix expression: Suppose θ is an arithmetic expression written in infix notation.

Step 1. PUSH "(" onto STACK and add ")" to the end of θ.

Step 2. Scan θ from left to right and reject step 3, 4, 5 and 6 for each element of θ.

Step 3. If an operand is encountered, add M to P.

Step 4. If a left parenthesis "(" is encountered, PUSH it onto STACK.

Step 5. If an operator $\otimes$ is encountered them:

(*a*) Repeatedly PoP from STACK and add to P each operator which has the same priority as or higher periority than $\otimes$.

(*b*) Add operator $\otimes$ to STACK.

Step 6. If a right parenthesis is encountered, then

(*a*) Repeatedly PoP from STACK and add to P each operator until a left parenthesis is encountered.

(*b*) Remove the left parenthesis [Do not add the left parenthesis ToP].

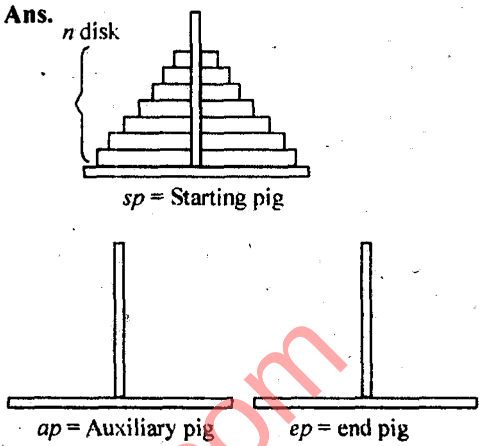
Exit

```
void in FIX_TO_Postfix (char * source, char * target)
{
    char *s, *;
    stack * top;
    create_stack (&top),
    S = souce,
    t = target;
    while (*s)
    {
        if ((*s = = ' ') || (*s = = '\t')) /*stop white
                                                spaces*/
        {
            s++;
            continue;
        }
        else
        if(*s = = '(')
        {
            PUSH (&toy. *s);
            s++;
        }
        else
            if(*s = = ')')
            {
                while ((! Is-empty (toy)) &&
                                    (peek(fog)! ='('))
                {
                    *t = PoP(&toy);
                    t++,
                    *t=' '; /*add one space*/
                    t++
                }
            }
            PoP(&toy);    /*remove left parenthesis
                                         from stage
            s++
            }
            else
                if(is digit (*s) || is alpha (*s))
                {
                    *t = *s;
                    t++;
                    *t = ' ';    /*add one space */
                    t++
                    s++;
                }
                else
                    if(*s = =' + ' || *s = = '–' || *s =
                      =' *' || *s = = ' | ' || *s = ='%')
while ((| ! is empty (toy)) && (peek (toy)! = ')') &&
        (get priority (peek (joy)) >= get priority (*s)))
        {
            *f = PoP(&top),
            t++,
            *t = ' ',                /*add one space */
            t++
        }
        PUSH(&top, *s),
        s++,
    }
}
while((! *s-empty (top)) && (peek (top)! = '('))
    {
        *t = PoP(&top),
        t++,
        *t = ' ',           /*add one space */
        t++;
    }
        *t = '\o'          /*terminate the string*/
}

int get priority (char of)
{
    int priority;
    if (op = == ' ) ' || op = = '*' || op = = '%')
        priority = 1;
    else
        if(op = = '+' || op = = '(' )
            priority = 0,
        return (priority)
```

1. (d) Write a recursive 'C' function for solving towers of Hanoi problem.

Ans.



```
void move (int n, char sp, char ap, char ep)
{
    if(n = = 1)
    {
        printf("\n move FROM %C to %C", sp, ep)
    }
    else
    {
        move(n – 1, ap, ep, ap),
        move(1, sp, ' ', ep),
        move(n – 1, ap, sp, ep),
    }
}
```

1. (e) Write a C function for string matching.

Ans. NaiveSearch(*string* s[1..n], *string* sub[1..m])

```
    for i from 1 to n-m+1
     for j from 1 to m
        if s[i+j-1] ? sub[j]
           jump to next iteration of outer loop
      return i
    return not found
```

1. (f) What do you understand by best, worst and average case analysis of an algorithm? Give proper notations for these complexity measures.

Ans. In computer science, **best, worst** and **average cases** of a given algorithm express what the resource usage is *at least*, *at most* and *on average*, respectively. Usually the resource being considered is running time, but it could also be memory or other resource.

In real-time computing, the worst-case execution time is often of particular concern since it is important to know how much time might be needed *in the worst case* to guarantee that the algorithm would always finish on time.

Average performance and worst-case performance are the most used in algorithm analysis. Less widely found is best-case performance, but it does have uses, for example knowing the best cases of individual tasks can be used to improve accuracy of an overall worst-case analysis. Computer scientists use probabilistic analysis techniques, especially expected value, to determine expected average running times.

Best-case performance: The term *best-case performance* is used in computer science to describe the way an algorithm behaves under optimal conditions. For example, a simple linear search on an array has a worst-case performance O(n) (for the case where the desired element is the last, so the algorithm has to check every element; see Big O notation), and average running time is O(n/2) (the average position of an element is the middle of the array), but in the best case the desired element is the first element in the array and the run time is O(1).

Worst case versus average case performance: Worst-case performance analysis and average case performance analysis have similarities, but usually require different tools and approaches in practice.

Determining what *average input* means is difficult, and often that average input has properties which make it difficult to characterise mathematically (consider, for instance, algorithms that are designed to operate on strings of text). Similarly, even when a sensible description of a particular "average case" (which will probably only be applicable for some uses of the algorithm) is possible, they tend to result in more difficult to analyse equations.

Worst-case analysis has similar problems, typically it is impossible to determine the exact worst-case scenario. Instead, a scenario is considered which is at least as bad as the worst case. For example, when analysing an algorithm, it may be possible to find the longest possible path through the algorithm (by considering maximum number of loops, for instance) even if it is not possible to determine the exact input that could generate this. Indeed, such an input may not exist. This leads to a *safe* analysis (the worst case is never underestimated), but which is *pessimistic*, since no input might require this path.

When analyzing algorithms which often take a small time to complete, but periodically require a much larger time, amortized analysis can be used to determine the worst-case running time over a (possibly infinite) series of operations. This **amortized worst-case** cost can be much closer to the average case cost, while still providing a guaranteed upper limit on the running time.

Q.2. Attempt any four parts of the following: (5×4=20)

2. (*a*) Differentiate between linear and non-linear data structure.

Ans. Linear Data Structures: In this exhibit four data structures are considered. Choose one of the following:

a) Vectors
b) Linked lists
c) Stacks
d) Queues

Non-linear Data Structures: Non-Linear container classes represent trees and graphs. Each node or item may be connected with two or more other nodes or items in a non-linear arrangement. Moreover removing one of the links could divide the data structure into two disjoint pieces. For example a doubly linked list has two pointers from each node, however removing one of these makes the list into a singly linked list. While removing a pointer from a tree destroys the shape of the tree. Not surprisingly iterating non-linear container classes is much more challenging than the linear ones. Consider the following options:

(a) Binary trees - number of descendents at most two
(b) Any other trees
(c) Graphs

2. (*b*) Define priority queue. Write a 'C' function for insertion of an element in a priority queue.

Ans. Priority queue is an abstract data type in computer programming, supporting the following three operations:

- add an element to the queue with an associated priority
- remove the element from the queue that has the highest priority, and return it
- (optionally) peek at the element with highest priority without removing it.

A simple way to implement a priority queue data type is to keep a list of elements, and search through the list for the highest priority element for each "minimum" or "peek" operation. This implementation takes O(1) time to insert an element, and O(n) time for "minimum" or "peek". There are many more efficient implementations available.

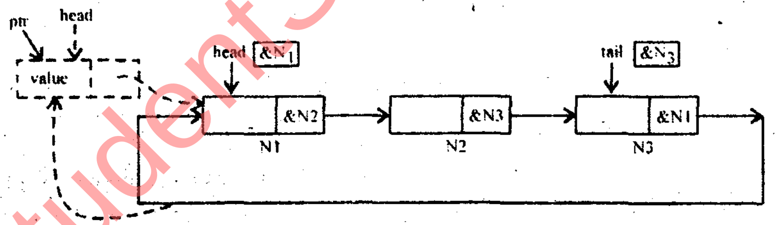
If a self-balancing binary search tree is used, all three operations take O(log *n*) time; this is a popular solution in environments that already provide balanced trees. The van Emde Boas tree, another associative array data structure, can perform all three operations in O(log log *n*) time, but at a space cost for small queues of about $O(2^{m/2})$, where *m* is the number of bits in the priority value, which may be prohibitive.

```
Struct node
{
    int priority;
    int info;
    struct node * link;
}   *front = NULL;
```

```
insert( )
{
    struct node * temp, *q;
    int add_item, item_priority;
    temp = (struct node*) malloc(size of (struct node));
    pf("Input the item value to be added in the queue");
    p.f. ("Enter priority"); scanf —
    temp-->info=added_item;
    temp ->priority = item_priority
    if(front = = NULL || item_priority < front  > priority)
    {
        temp --> link = front;
        front = temp;
    }
    else
    {
        q = front;
        while(q  > link! = NULL && q  > link --> priority
                                            <= item_priority)
            q = q --> link;
            temp --> link = q --> link;
                q --> link = temp;
    }
```

2. (*c*) Write a 'C' function for insertion operation in a circular linked list.

Ans. Insertion of big in a circular linked list:



```
typed of struct list
{
    int *INFO,
    struct list *NEXT,
}node;
node * head = NULL, *tail = NULL;
{
    node  *ptr;
    ptr = malloc (size of (node)),
    ptr --> INFO = value.
    if(head = = NULL)
    {
        ptr --> NEXT = ptr,
        head = ptr,
        tail = ptr,
    }
    else
        {
            ptr --> NEXT = head,
            head = ptr,
            tail = NEXT = ptr,
        }
        repair(head),
}
```

2. (*d*) **What is a singly linked list? Explain with an example how a singly linked list can be used for sorting a set of N numbers.**

Ans. Singly-Linked Lists: The singly-linked list is the most basic of all the linked data structures. A singly-linked list is simply a sequence of dynamically allocated objects, each of which refers to its successor in the list. Despite this obvious simplicity, there are myriad implementation variations. Figure shows several of the most common singly-linked list variants.

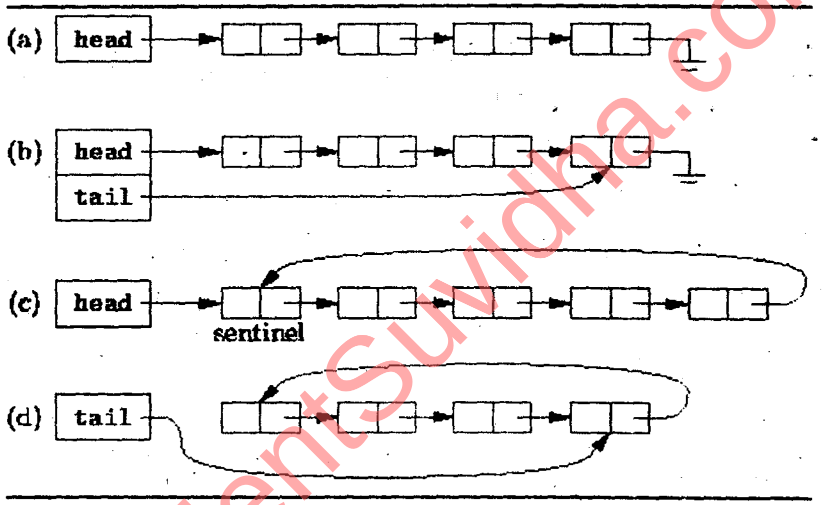


Figure: *Singly-linked list variations.*

2. (*e*) **What are the advantages and disadvantages of doubly linked list? Also give its applications.**

Ans. Advantages/Disadvantages of Doubly Linked: The primary advantage of a doubly linked list is that given a node in the list, one can navigate easily in either direction. This can be very useful. for example. if the list is storing strings, where the strings are lines in a text file (e.g., a text editor). One might store the "current line" that the user is on with a pointer to the appropriate node; if the user moves the cursor to the next or previous line, a single pointer operation can restore the current line to its proper value. Or, if the user moves back 10 lines, for example, one can perform 10 pointer operations (follow the chain) to get to the right line. For either of these operations, if the list is singly linked, one must start at the head of the list and traverse until the proper point is reached. This can be very inefficient for large lists.

The primary disadvantage of doubly linked lists are that (1) each node requires an extra pointer, requiring more space, and (2) the insertion or deletion of a node takes a bit longer (more pointer operations).

The primary advantage of a header node is that insertion and deletion require no special cases. The primary disadvantage of using a header node is that it becomes unclear what the header node stores. Is it a regular list node? Or is it a special node? If it's a regular node, what data should it store? For a list of strings, for example. one often wants to provide a function to return a string from a given node. If the list is empty. what is returned? It can't be `NULL`, because the functions has a return type of String. It should be a string that can't possibly ever actually be used.

2. (*f*) **Write a 'C' function for addition of two polynomials using linked list representation of polynomials.**

Ans. Polynomial Addition:

```
typedef struct POLY
{
    int COEF;
    int EXP,                     | COEF | EXP | NEXT |
    struct POLY * NEXT,
}node
node * PADD(node *A, node *B, node *C)
{
    /*Polynomials A & B regenerated as singly linked
        lists are summed to form the new list named (x)
    node *p, *q;
Step 1. p = A, q = B || Initialize p, q pointers to link list
                                        A & B respectively
Step 2. //Allocate free memory space
            c = create new node
            d = C
Step 3. Repeat stage 4, 5, 6 while p!=NULL and q!=NULL
Step 4. if(r → EXP = = q → EXP)
        {
            x = f → COEF + q → COEF
            if(x! = 0)
            {
                ATTACH(x, p → EXP, d)
            }
                p = p → NEXT
                q = q → NEXT
        }
Step 5. if (r → EXP < q → EXP)
        {
            ATTACH(q → COEF, q → EXP, d)
            q = q → NEXT
        }
Step 6. if (r → EXP > q → EXP)
        {
            ATTACH (p → COEF, p → EXP, d)
            p = p → NEXT
        }
Step 7. while (p! = NULL)
            //copy remaining terms of polynomials A
        {
            ATTACH (p → COEF, p → EXP, d)
            p = p → NEXT
        }
Step 8. while (q! = NULL)
            //copy remaining terms of polynomials B
        {
            ATTACH (q → COEF, q → EXP, d)
                q = q → NEXT
        }
Step 9. d → NEXT = NULL,
        t = C,
        C = C ⇒ NEXT,
        free(f);
}
    ATTACH(C, E, d)
    {
Step 1. I = create memory attach
                                    //mollar (size of (node))
Step 2. I → EXP = E
Step 3. I → COEF = C
Step 4. d → NEXT = I
                    //attach this node to the end of this left
Step 5. d = I   //move pointer d to the new last node
}
```

Q.3. Attempt any two parts of the following:

(10×2=20)

3. (*a*) (*i*) **Write an iterative 'C' function for inorder traversal of a binary tree.**

(*ii*) **Write the applications of tree data structures.**

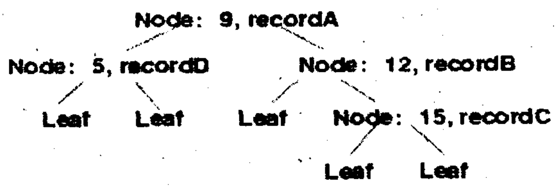
Ans. (*i*) Iteration C function for in order traversal of a Binary tree:

```
void inorder (ree * root)
{
    do
    {
        while (root!=NULL)
        {
            PUSH(&STACK, root)
            root = root → LEFT
        }
        if (!stack_empty(&STACK))
        {
            root = PoP (&STACK).
            printf("%/", root → INFO),
            root = root → RIGHT.
        }
    }while (root! = NULL || !stack_empty (&STACK)),
    geth( ).
}
        typedef struck BT
        {
            inf INFO,
            struct BT *LEFT,
            struct BT &RIGHT,
        };
```

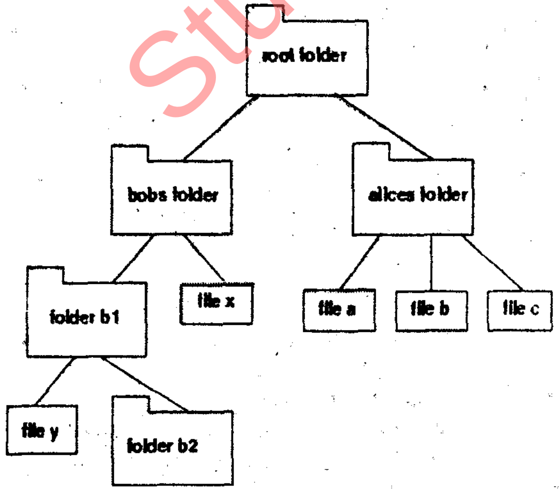
Ans. (*ii*) Applications of Trees:

1. Trees can hold objects that are sorted by their keys. The nodes are ordered so that all keys in

a node's left subtree are less than the key of the object at the node, and all keys in a node's right subtree are greater than the key of the object at the node. Here is an example of a tree of records, where each record is stored with its integer key in a tree node:



2. Here, the leaves are used as "end points" and hold nothing.
   We call such a tree an *ordered tree* or a *search tree*. The tree drawn above is ordered on the integer keys saved in the nodes.
   The advantages of ordered trees over sorted arrays are:
   - both insertions (and retrievals) of objects by key take on the average $\log_2 N$ time, where N is the number of objects stored.
   - the tree naturally grows to hold an arbitrary, unlimited number of objects.
3. Trees can hold objects that are located by keys that are sequences. For example, we might have some books with these Library of Congress catalog numbers:
   Trees are used to represent phrase structure of sentences, which is crucial to language processing programs.
4. An operating system maintains a disk's file system as a tree, where file folders act as tree nodes:



5. The tree structure is useful because it easily accommodates the creation and deletion of folders and files.

3. **(b) (i) Write an algorithm to count the leaf nodes in a binary tree.**
   (ii) Write short notes on height balanced trees and weight balanced trees.

Ans. (i) Total no. of leaf nodes (recursive approach)

```
int leaf node (tree *T)
{
    if((T==NULL)||(T→LEFT==NULL&&T
                          →RIGHT=NULL))
        return(1),
    else
        return(leaf node (T→LEFT)+(leaf
                          node(T→RIGHT))),
}
```

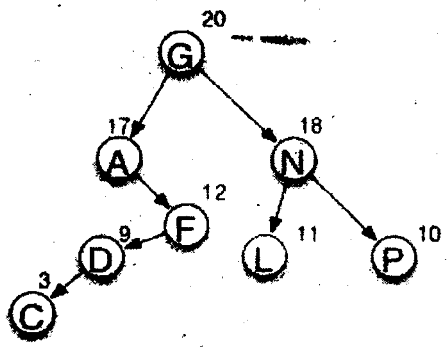
***(ii)* Height -balanced tree:** Height-balancing is important when we wish to use a binary search tree to implement a keyed table. The reason it is important is that in a height-balanced tree, the ratio of the height of the tree to the number of nodes is logarithmic. Maintaining a logarithmic ratio ensures that the big-O for the insert, delete, and find operations will all be logarithmic.

Every binary tree has a height, or depth, that depends upon the height of its subtrees. Observe when interacting with the corresponding visualization in the *I'll Try* mode that you will be asked supply the height of each of the subtrees first. In fact, you will be prompted for the height of each node in a postorder traversal of the tree. Another important observation that you should make is that in this visualization, the nodes of the trees that you will see contain no values. The property of height-balancing depends only upon the shape of the tree, not on the contents of the nodes.

Once you understand the concept of tree height, proceed to balance factors, which depends upon your understanding of height and is necessary for understanding height-balancing. When you are asked to supply the balance factors, all the heights of each subtree is provided to assist you.

Finally, move to height-balanced trees, and balanced trees, which are a special case. You should note that full and complete trees are the most balanced and that every complete tree is a balanced tree. Every balanced tree is a height-balanced-2 tree and so on. As the *n* in height-balance-*n* increases the more unbalanced the trees are allowed to be.

Weight-balanced tree: A weight-balanced binary tree is a binary tree where the most probable item is the root item. The left subtree consists of items less than the root items ranking, not its probability. The right sub tree consists of items greater than the root items ranking.



Example of weight balanced tree

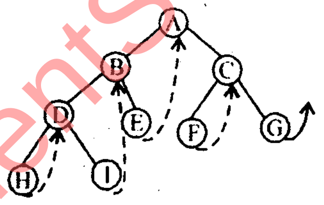
3. (*c*) (*i*) **What is a threaded binary tree? Explain inorder threading.**

(*ii*) **Explain Hash table implementation in detail.**

Ans. (*i*) **Threaded binary tree:** We know that in linked prepresentation of any B.T. there are more NULL links than actional pointers. For efficiently used we will replace certain NULL entries by spec ial pointers which point to node higher in the tree, there special pointers called threads and BT will such pointers are called threaded trees.

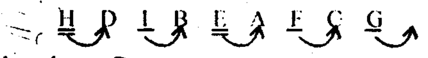
There are many way to thread a BT

1. One way inorder threading
2. Two way inorder threading



One way inorder Threader B.T.

Inorder threading:



E has predecessor thread which point to B.

B has successor thread which points to A.

(*ii*) **A Simple Hash Table Implementation:** Hash tables are pretty cool things, and quite versatile. They have a wide range of uses including associative arrays in perl (I am sure we all love these) and many others. I apologize in advance for the differences in code between the article and the included files. I already wrote this program and have added comments into the article to make it comprehensible, some of the code (the hash algorithm is really nasty to read). I have tried to add the comments back into the original code, but I could have screwed up.

To effectively use a hash table you need to know approximately how many data items you will have. This is because if you under estimate you will get a lot of collisions (these are explained later in the article), and if you over estimate you will waste memory. You can get around this by using an array of pointers to your hash structure as this should greatly reduce the memory used by unoccupied elements. A hash table is an array that has a prime number of elements. (See prime.txt for a rather large list of prime numbers.) When you wish to add an piece of data to the hash table you need to take the key (usually characters) and turn it into a number. This is the hash algorithm, the only really necessary requirement is that the hash algorithm will always produce the same results for the same key. It is also desirable to distribute the results for different keys nicely. You then modulus the number produced by the hash algorithm by the number of elements in the hash table. Then you put the data in that element of the array.

Unsigned int hash (char *s); // Starts on line #9 of hash.c

```
unsigned int hash (char *s) {
  int i, n;                          // Our temporary variables
  unsigned int hashval;
  unsigned int ival;
  char *p;
  p = (char *) &ival;                // p lets us reference the integer value as a
                                     // character array instead of an integer. This
                                     // is actually pretty bad style even though it
                                     // works. You should try a union if you know
                                     // how to use them.
  hashval = ival = 0;                // Initialize our variables
  // Figure out how many characters are in an integer the correct answer is 4
  // (on an i386), but this should be more cross platform.
  n = (((log10((double)(UINT_MAX)) / log10(2.0))) / CHAR_BIT) + 0.5;
  // loop through s four characters at a time
  for(i = 0; i < strlen(s); i += n) {
    // voodoo to put the string in an integer don't try and use strcpy, it
    // is a very bad idea and you will corrupt something.
    strncpy(p, s + i, n);
    // accumulate our values in hashval
    hashval += ival;
  }
  // divide by the number of elements and return our remainder
  return hashval % HASHELEMENTS;
}
```

If we could just shove the data in the array it would be pretty simple, but what if there are two keys that hash to the same value? This is a collision, and slows down our lookup. A good hash algorithm won't have many of these, but the are inevitable. In this case I have implemented a linked list.

I hope this has been useful to you, have fun hashing.

- Makefile
- hash.h
- hash.c
- getch.c
- gettoken.c
- main.c

prime.txt

Q.4. Attempt any two parts of the following: **(10×2=20)**

4. (*a*) (*i*) Write a non-recursive algorithm for quick sort.

(ii) Derive the time complexity of Merge sort in average case.

Ans. (i)

```
#include <stdio.h>
#include <conio.h>
#define MAXELT      100
#define INFINITY    32760
                // numbers in list should not exceed
               // this. Change the value to suit your
                // needs
#define SMALLSIZE    10      // not less than 3
#define STACKSIZE    100
               // should be ceiling(lg(MAXSIZE)+1)
int list[MAXELT+1];
                    // one extra, to hold INFINITY
struct {            // stack element.
    int a,b;
} stack[STACKSIZE];
int top=-1;              // initialise stack
int main()               // overhead!
{
  int i=-1,j,n;
  char t[10];
  void quicksort(int);
  do {
    if (i!=-1)
      list[i++]=n;
    else
      i++;
    printf("Enter the numbers <End by #>: ");
    flush(stdin);
    scanf("%[^\n]",t);
    if (scanf(t,"%d",&n)<1)
    break;
  } while (1);
  quicksort(i-1);
  printf("\nThe list obtained is ");
  for (j=0;j<i;j++)
    printf("\n %d",list[j]);
  printf("\n\nProgram over.");
  getch();
  return 0;    // successful termination.
}
```

```
void interchange(int *x,int *y)    // swap
{
  int temp;
  temp=*x;
  *x=*y;
  *y=temp;
}
void split(int first,int last,int *splitpoint)
{
  int x,i,j,s,g;
  // here, atleast three elements are needed
  if (list[first]<list[(first+last)/2]) { // find median
    s=first;
    g=(first+last)/2;
  }
  else {
    g=first;
    s=(first+last)/2;
  }
  if (list[last]<=list[s])
    x=s;
  else if (list[last]<=list[g])
    x=last;
  else
    x=g;
  interchange(&list[x],&list[first]);   // swap the
split-point element
                                        // with the first
  x=list[first];
  i=first+1;                  // initialise
  j=last+1;
  while (i<j) {
    do {                      // find j
      j—;
    } while (list[j]>x);
    do {
      i++;                    // find i
    } while (list[i]<x);
    interchange(&list[i],&list[j]);    // swap
  }
  interchange(&list[i],&list[j]);      // undo the
extra swap
  interchange(&list[first],&list[j]);
```

```
                          // bring the split-point
                          // element to the first
  *splitpoint=j;
}
void push(int a,int b)            // push
{
  top++;
  stack[top].a=a;
  stack[top].b=b;
}
void pop(int *a,int *b)           // pop
{
  *a=stack[top].a;
  *b=stack[top].b;
  top—;
}
void insertion_sort(int first,int last)
{
  int i,j,c;
  for (i=first;i<=last;i++) {
    j=list[i];
    c=i;
    while ((list[c-1]>j)&&(c>first)) {
      list[c]=list[c-1];
      c—;
    }
    list[c]=j;
  }
}
void quicksort(int n)
{
  int first,last,splitpoint;
  push(0,n);
  while (top!=-1) {
    pop(&first,&last);
    for (;;) {
      if (last-first>SMALLSIZE) {
        // find the larger sub-list
        split(first,last,&splitpoint);
        // push the smaller list
        if (last-splitpoint<splitpoint-first) {
          push(first,splitpoint-1);
          first=splitpoint+1;
        }
        else {
          push(splitpoint+1,last);
          last=splitpoint-1;
        }
      }
      else {  // sort the smaller sub-lists
              // through insertion sort
        insertion_sort(first,last);
        break;
      }
    }
  }                  // iterate for larger list
```

(*ii*) Complexity of Mergesort: Mergesort is a divide and conquer algorithm, as outlined below. Note that the function *mergesort* calls itself *recursively*. Let us try to determine the time complexity of this algorithm.

```
list mergesort (list L, int n);
  {
    if (n = = 1)
      return (L)
    else {
      Split L into two halves L1 and L2 ;
      return (merge (mergesort (L1, n/2).
(mergesort (L2, n/2))
      }
  }
```

Let T(n) be the running time of Mergesort on an input list of size n. Then,

$T(n) \le C_1$ (if $n = 1$) (C_1 is a constant)

$$\le \underbrace{2T\left(\frac{n}{2}\right)}_{\text{two recursive calls}} + \underbrace{C_2 n}_{\text{cost of merging}} \text{ (if } n > 2)$$

If n = 2^k for some k, it can be shown that

$T(n) \le 2^k T(1) + C_2 k 2^k$

That is, $T(n)$ is $O(n \log n)$.

}

4. (*b*) (*i*) **Write a 'C' function for insertion sort. Give the worst case time complexity for insertion sort.**

(*ii*) **Explain two way merge sort.**

Ans (*i*) Insertion sort:

```
void insertion (INT A[ ], int N)
{
  int temp, ptr, k,
  a[o] = – 1000,
  for (k = 1, k<=N, K++)
```

```
                                //k is the press number
    {
        temp = A[k]
        ptr = k – 1;
        while(temp < A[ptr])           //Shifting
        {
            A[ptr + 1] = A[ptr],
            ptr = ptr – 1,
        }
            A[ptr + 1] = temp          //Insertion
    }
}
```

Worst case complexity for insertion sort algorithm:

$$F(n) = \sum_{k=1}^{N} k$$

(max. no. of comparison in each pass)

$$= 1 + 2 + 3 + \ldots N$$

$$= \frac{N}{2}[2 \times 1 + (N-1).1] > \frac{N(N+1)}{2}$$

$$= \frac{N^2}{2} + \frac{N}{2}$$

Therefore

$\approx 0(N^2)$.

(*ii*) Two-way merge sort:

Definition: A *k-way merge sort* that sorts a data stream using repeated merger. It distributes the input into two streams by repeatedly reading a *block* of input that fits in memory, a *run*, sorting it, then writing it to the next stream. It merges runs from the two streams into an output stream. It then repeatedly distributes the runs in the output stream to the two streams and merges them until there is a single sorted output.

4. (*c*) (*i*) **Derive complexity of search operation in an AVL tree.**

(*ii*) **Write a short note on B tree.**

Ans. (*i*) A non-specialized sequence container that performed better than O(N) in both kinds of operation would be a great advance, even if it performed worse than vector on random access, and worse than list on insert/erase. The sequence container avl_array, proposed here, has precisely these properties. It provides O(log N) random access and O(log N) insertion/removal. By using it, the aforementioned algorithm would have its time complexity lowered down to O(N·log N).

As in list in avl_array every element is linked to its two sequence neighbors.

```
     next–> ················ next–>
   A     B  ······   F     G
     <–prev ················ <–prev
```

Additionally, every element is linked to three elements more. These links are employed to build a tree structure:

```
          D
         / \
       B     F
      /\    /\
     A  C  E  G
```

The topmost element is called the root of the tree. Trees are self-similar structures. That is, every node is the root of a subtree that contains the nodes placed under it. In every node, a variable stores the rank of the node in its subtree (starting with 0). These kind of trees are called rank trees or <u>order statistic trees</u>.

```
            D,3
           /   \
       B,1       F,1
       /  \     /  \
    A,0  C,0  E,0  G,0
```

This rank variable is employed for reaching the n'th element by descending from the root and taking every time the left branch (if n<rank) or the right branch (if n>rank). Before taking a right branch, n must be adjusted by subtracting the current rank from it. The time complexity of this random access operation is O(log N), regarded that the tree is balanced. Maintaining balance and ranks make insert/erase operations O(log N) too.

That said, it must be clarified that the current implementation of `avl_array` doesn't store the rank of every node, but the count of nodes in its subtree (including both branches and the node itself):

```
            D,7
           /   \
       B,3       F,3
       /  \     /  \
    A,1  C,1  E,1  G,1
```

The rank of a node is found in the count field of the root of its left subtree (if there's no left subtree, then the rank is 0). The total size of the tree is the root's count. On insertion/removal the count fields in the path to the root must be updated, which requires adding (unconditionally) left and right counts of every node (only the nodes in the way up to the root need to be updated).

The underlying tree structure is never exposed to the user. It is only employed for providing logarithmic time random access, and it doesn't represent any meaningful hierarchy relationship within elements.

***(ii)* B tree:** In computer science, a **B-tree** is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. It is most commonly used in databases and filesystems.

In B-trees, internal nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 B-tree (often simply referred to as a **2-3 tree**), each internal node may have only 2 or 3 child nodes.

A B-tree is kept balanced by requiring that all leaf nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more hop further removed from the root.

B-trees have substantial advantages over alternative implementations when node access times far exceed access times within nodes. This usually occurs when most nodes are in secondary storage such as hard drives. By maximizing the number of child nodes within each internal node, the height of the tree decreases, balancing occurs less often, and efficiency increases. Usually this value is set such that each node takes up a full disk block or an analogous size in secondary storage. While 2-3 B-trees might be useful in main memory, and are certainly easier to explain, if the node sizes are tuned to the size of a disk block, the result might be a 129-513 B-tree.

A B-Tree of order m (the number of keys in each node) is a tree which satisfies the following properties :

1. Every node has <= m children.
2. Every node (except root and leaves) has >= m/2 children.
3. The root has at least 2 children.
4. All leaves appear in the same level, and carry no information.

A non-leaf node with k children contains k – 1 keys

Q. 5. Attempt any two parts of the following: (10×2=20)

5. (*a*) (*i*) Prove the correctness of Kruskal's algorithm for minimum spanning tree.

(*ii*) Define Graph, multigraph and weighted matrix.

Ans. (*i*) Proof of correctness: Let P be a connected, weighted graph and let Y be the subgraph of P produced by the algorithm. Y cannot have a cycle, since the last edge added to that cycle would have been within one subtree and not between two different trees. Y cannot be disconnected, since the first encountered edge that joins two components of Y would have been added by the algorithm. Thus, Y is a spanning tree of P.

It remains to show that the spanning tree Y is minimal:

Let Y_1 be a minimum spanning tree. If $Y = Y_1$ then Y is a minimum spanning tree. Otherwise, let e be the first edge considered by the algorithm that is in Y but not in Y_1. has a cycle, because you cannot add an edge to a spanning tree and still have a tree. This cycle contains another edge f which at the stage of the algorithm where e is added to Y, has not been considered. This is because otherwise e would not connect different trees but two branches of the same tree. Then is also a spanning tree. Its total weight is less than or equal to the total weight of Y_1. This is because the algorithm visits e before f and therefore . If the weights are equal, we consider the next edge e which is in Y but not in Y_1. If there is no edge left, the weight of Y is equal to the weight of Y_1 although they consist of a different edge set and Y is also a minimum spanning tree. In the case where the weight of Y_2 is less than the weight of Y_1 we can conclude that Y_1 is not a minimum spanning tree, and the assumption that there exist edges e, f with $w(e) < w(f)$ is incorrect. And therefore Y is a minimum spanning tree (equal to Y_1 or with a different edge set, but with same weight).

(ii) Graph : A set of items connected by *edges*. Each item is called a *vertex* or *node*. Formally, a graph is a *set* of vertices and a *binary relation* between vertices, adjacency.

Multi graph (MultiGraph)

The Class of multigraphs. A multigraph is a Graph containing at least one pair of GraphNodes that are connected by more than one GraphArc.

Weight Matrix Definition

- A weight matrix for a pattern of length *n* is defined as a matrix of numbers $W_{i,x}$ where *i* is in *{1,2,...,n}* and *x* is in *{A,T,G,C}* for dna.
- The score of the string $x_1...x_n$ is given by:

  $W_{1,x1} + W_{2,x2} + ... + W_{n,xn}$

5. (*b*) (*i*) Write a short note on DFS traversal of a graph.

(*ii*) Explain the Dijkstra's algorithm for shortest path in a graph with suitable example.

Ans. (i) Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. Intuitively, one starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hadn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a LIFO stack for exploration.

Space complexity of DFS is much lower than BFS (breadth-first search). It also lends itself much better to heuristic methods of choosing a likely-looking branch. Time complexity of both algorithms are proportional to the number of vertices plus the number of edges in the graphs they traverse (O(|V| + |E|)).

When searching large graphs that cannot be fully contained in memory, DFS suffers from non-termination when the length of a path in the search tree is infinite. The simple solution of "remember which nodes I have already seen" doesn't always work because there can be insufficient memory. This can be solved by maintaining an increasing limit on the depth of the tree, which is called iterative deepening depth-first search.

A depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously-visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G.

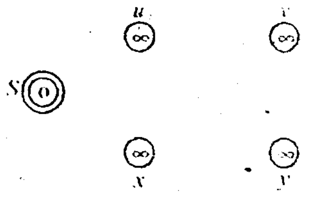
Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

Iterative deepening prevents this loop and will reach the following nodes on the following depths, assuming it proceeds left-to-right as above:

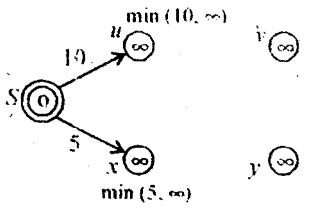
- 0: A
- 1: A (repeated), B, C, E
  (Note that iterative deepening has now seen C, when a conventional depth-first search did not.)
- 2: A, B, D, F, C, G, E, F
  (Note that it still sees C, but that it came later. Also note that it sees E via a different path, and loops back to F twice.)
- 3: A, B, D, F, E, C, G, E, F, B

For this graph, as more depth is added, the two cycles "ABFE" and "AEFB" will simply get longer before the algorithm gives up and tries another branch.

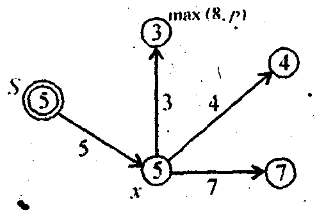
(*ii*) Operation of Dijksha's Algorithm: Graph



Choose the closest node to (S) relax all nodes adjacent to S under predessor for all nodes updates



Choose in closely node X relax all nodes adjaceble in update predessors for *u*, *v* & *y*.



B-Tree: 1. At least N/2 and max. *n* nodes empty children (order in)

2. All leaf node must be at same level.

3. All leaf node contain max *n* – 1 keys.

5. (*c*) (*i*) **Write a 'C' function for traversing a multiway search tree.**

(*ii*) **Write about deletion of a node in a B tree.**

Ans. (*i*) Multi way search: Recordptr *searchkey(node *root, int skey)

```
{
        int i;
        if(root==NULL)
        return NULL;
        else
{       i=0;
        while((i<6) && (skey> root->key[i]))
                        i++;
        if((i<6) && (skey== root->key[i]))
                return root->recptr[i];
else if (i< 6)
        return searchkey(root->nodeptr[i], skey)
        else
return searchkey(root->nodeptr[i], skey)
}}
```

(*ii*) **Deletion From a B-Tree:**

- If the entry to be deleted is not in a leaf, swap it with its successor (or predecessor) under the natural order of the keys. Then delete the entry from the leaf.
- If leaf contains more than the minimum number of entries, then one can be deleted with no further action.
- Otherwise...
- If the node contains the minimum number of entries, consider the two immediate siblings of the node:
- If one of these siblings has more than the minimum number of entries, then redistribute one entry from this sibling to the parent node, and one entry from the parent to the deficient node.

Deletion From a B-Tree (pseudo code)

- if (the entry to remove is not in a leaf) then swap it with its successor;
- currentNode = leaf;
- while (currentNode underflow)

try to redistribute entries from an immediate sibling into currentNode via the parent node;

if (impossible) then merge currentNode with a sibling and one entry from parent;

currentNode := parent of currentNode;