**LECTURE NOTES**

**ON**

**DATA STUCTURES**

**III B-Tech I Semester(JNTUH-R13)**



**COMPUTER SCIENCE & ENGINEERING**

**CMR ENGINEERING COLLEGE**

**KANDLAKOYA (V), MEDCHAL (M), R.R.DIST.**

**Syllabus**

**JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY H YDERABAD**

**II Year B.Tech. CSE-I Sem L T/P/D C**

**4 -/-/- 4**

**DATA STRUCTURES**

**UNIT- I**

Basic concepts- Algorithm Specification-Introduction, Recursive algorithms, Data Abstraction Performance

analysis- time complexity and space complexity, Asymptotic Notation-Big O, Omega and Theta notations,

Introduction to Linear and Non Linear data structures.

Singly Linked Lists-Operations-Insertion, Deletion, Concatenating singly linked lists, Circularly linked lists-

Operations for Circularly linked lists, Doubly Linked Lists- Operations- Insertion, Deletion.

Representation of single, two dimensional arrays, sparse matrices-array and linked representations.

**UNIT- II**

Stack ADT, definition, operations, array and linked implementations in C, applications-infix to postfix conversion,Postfix expression evaluation, recursion implementation, Queue ADT, definition and operations ,array and linked Implementations in C, Circular queues-Insertion and deletion operations, Deque (Double ended queue)ADT, array and linked implementations in C.

**UNIT- III**

Trees – Terminology, Representation of Trees, Binary tree ADT, Properties of Binary Trees, Binary Tree Representations-array and linked representations, Binary Tree traversals, Threaded binary trees, Max Priority Queue ADT-implementation-Max Heap-Definition, Insertion into a Max Heap, Deletion from a Max Heap. Graphs – Introduction, Definition, Terminology, Graph ADT, Graph Representations- Adjacency matrix, Adjacency lists, Graph traversals- DFS and BFS.

**UNIT- IV**

Searching- Linear Search, Binary Search, Static Hashing-Introduction, hash tables, hash functions, Overflow Handling. Sorting-Insertion Sort, Selection Sort, Radix Sort, Quick sort, Heap Sort, Comparison of Sorting methods.

**UNIT- V**

Search Trees-Binary Search Trees, Definition, Operations- Searching, Insertion and Deletion, AVL Trees- Definition and Examples, Insertion into an AVL Tree ,B-Trees, Definition, B-Tree of order m, operations-Insertion and Searching, Introduction to Red-Black and Splay Trees(Elementary treatment-only Definitions and Examples),Comparison of Search Trees.

Pattern matching algorithm- The Knuth-Morris-Pratt algorithm, Tries (examples only).

**TEXT BOOKS:**

1. Fundamentals of Data structures in C, 2nd Edition, E.Horowitz, S.Sahni and Susan

Anderson-Freed, Universities Press.

2. Data structures A Programming Approach with C, D.S.Kushwaha and A.K.Misra, PHI.

**REFERENCE BOOKS:**

1. Data structures: A Pseudocode Approach with C, 2nd edition, R.F.Gilberg And B.A.Forouzan, Cengage

Learning.

2. Data structures and Algorithm Analysis in C, 2nd edition, M.A.Weiss, Pearson.

3. Data Structures using C, A.M.Tanenbaum,Y. Langsam, M.J.Augenstein, Pearson.

4. Data structures and Program Design in C, 2nd edition, R.Kruse, C.L.Tondo and B.Leung,Pearson.

5. Data Structures and Algorithms made easy in JAVA, 2nd Edition, Narsimha Karumanchi, CareerMonk

Publications.

6. Data Structures using C, R.Thareja, Oxford University Press.

7. Data Structures, S.Lipscutz,Schaum’s Outlines, TMH.

8. Data structures using C, A.K.Sharma, 2nd edition, Pearson..

9. Data Structures using C &C++, R.Shukla, Wiley India.

10. Classic Data Structures, D.Samanta, 2nd edition, PHI.

11. Advanced Data structures, Peter Brass, Cambridge.

**13 Lecture Schedule with methodology**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| SNo | Period No | Unit No | Date | Topic to be covered in One lecture | Reg/Additional | Teaching aids used LCD/OHP/  BB | Remarks |
| 1 | 1 | **I** |  | **Introduction to subject** | Reg | BB |  |
| 2 | 2 |  |  | **Basic Concepts Algorithm specification-Introduction** | Reg | BB |  |
| 3 | 3 |  |  | **Recursive Algorithms** | Reg | BB |  |
| 4 | 4 |  |  | **Data Abstraction** | Reg | BB |  |
| 5 | 5 |  |  | **Performance analysis -space complexity** | Reg | BB,OHP |  |
| 6 | 6 |  |  | Time comlexity using -variable count | Reg | BB,OHP |  |
| 7 | 7 |  |  | Time complexity using- step count table | Reg | BB OHP |  |
| 8 | 8 |  |  | Asymptotic notations | Reg | BB OHP |  |
| 9 | 9 |  |  | **Linear and nonlinear data structures** | Reg | BB,OHP |  |
| 10 | 10 |  |  | Singly linked list operations- insertion, deletion | Reg | BB OHP |  |
| 11 | 11 |  |  | Concatenation of lists | Reg | BB OHP |  |
| 12 | 12 |  |  | Circular linked list all operations | Reg | BB OHP |  |
| 13 | 13 |  |  | Doubly linked list operations- insertion,deletion | Reg | BB OHP |  |
| 14 | 14 |  |  | Representation of Single, two dimensional arrays | Reg | BB OHP |  |
| 15 | 15 |  |  | Sparse matrices array representation | Reg& Add | BB,OHP |  |
| 16 | 16 |  |  | Sparse matrices linked representation | Reg | BB OHP |  |
| 17 | 1 | **II** |  | Stack Definition , ADT | Reg | BB,OHP |  |
| 18 | 2 |  |  | Stack- Array implementation | Reg& Add | BB,OHP |  |
| 19 | 3 |  |  | Stack Linked representation | Reg | BB,OHP |  |
| 20 | 4 |  |  | Applications- infix to postfix conversion | Reg | BB,OHP |  |
| 21 | 5 |  |  | Postfix Expression Evaluation | Reg | BB,OHP |  |
| 22 | 6 |  |  | Queue ADT- Definition and operations | Reg | BB,OHP |  |
| 23 | 7 |  |  | Queue implementation -Array and linked lists | Reg | BB,OHP |  |
| 24 | 8 |  |  | Circular Queues -all operations using arrays | Reg | BB,OHP |  |
| 25 | 9 |  |  | Circular Queues- all operations using linked list | Reg | BB,OHP |  |
| 26 | 10 |  |  | Double Ended Queue(Deque)- all operations using arrays | Reg | BB,OHP |  |
| 27 | 11 |  |  | Deque all operations using linked list | Reg | BB,OHP |  |
| 28 | 1 | **III** |  | Trees- Treminology, Representatin of Trees, Binary Tree ADT | Reg | BB,OHP |  |
| 29 | 2 |  |  | Properties of Binary Trees, Array representation | Reg | BB,OHP |  |
| 30 | 3 |  |  | Linked List Representatin of Binary Tree | Reg | BB,OHP |  |
| 31 | 4 |  |  | Binary Tree traversals- Recursive | Reg | BB,OHP |  |
| 32 | 5 |  |  | Non- Recursive Binary Tree Traversals | Reg | BB,OHP |  |
| 33 | 6 |  |  | Threaded Binary Trees- Threads, Inorder Traversaland insertion into threaded binary tree | Reg | BB,OHP |  |
| 34 | 7 |  |  | Max priority Queue- ADT, Definition, insertion into a Max Heap | Reg | BB,OHP |  |
| 35 | 8 |  |  | Deletion from a Max Heap | Reg | BB,OHP |  |
| 36 | 9 |  |  | Graphs- Introduction, Definition, Terminology | Reg | BB,OHP |  |
| 37 | 10 |  |  | Graph Representaions- Adjacency Matrix  Adjacency Lists | Reg | BB,OHP |  |
| 38 | 11 |  |  | Graph Traversals-DFS and BFS | Reg | BB,OHP |  |
| 39 | 1 | **IV** |  | Searching- Linear Search & Binary Search | Reg | BB,OHP |  |
| 40 | 2 |  |  | Static Hashing- Introduction, Hash Tables | Reg | BB,OHP |  |
| 41 | 3 |  |  | Hash functions- Division, Mid-Square, Folding, Digit Analysis, converting Keys to Integers | Reg | BB,OHP |  |
| 42 | 4 |  |  | Overflow Handling- open Addressing, Chaining | Reg | BB,OHP |  |
| 43 | 5 |  |  | Sorting-Motivation, Insertion Sort | Reg | BB,OHP |  |
| 44 | 6 |  |  | Selection Sort | Reg | BB,OHP |  |
| 45 | 7 |  |  | Radix Sort | Reg | BB,OHP |  |
| 46 | 8 |  |  | Quick sort | Reg | BB,OHP |  |
| 47 | 9 |  |  | Heap sort | Reg | BB,OHP |  |
| 48 | 10 |  |  | Comparison of sorting techniques | Reg | BB,OHP |  |
| 49 | 1 | **V** |  | Search Trees- Binary Search Trees( search and insert) | Reg | BB,OHP |  |
| 50 | 2 |  |  | Deletion from a BST | Reg | BB,OHP |  |
| 51 | 3 |  |  | AVL Trees- Definition and Examples | Reg | BB,OHP |  |
| 52 | 4 |  |  | Insertion and Deletion from an AVL Tree | Reg | BB,OHP |  |
| 53 | 5 |  |  | B- Trees- Definition, B-Tree of order m | Reg | BB,OHP |  |
| 54 | 6 |  |  | B-Trees- Searching, Insertion and Deletion | Reg | BB,OHP |  |
| 55 | 7 |  |  | Red- Black Trees- Definition and Examples | Reg | BB,OHP |  |
| 56 | 8 |  |  | Splay Trees- Definition and Examples | Reg | BB,OHP |  |
| 57 | 9 |  |  | Pattern matching algorithms- Brute Force Approach | Reg | BB,OHP |  |
| 58 | 10 |  |  | Knuth- Morris- Pratt Algortihm | Reg | BB,OHP |  |
| 59 | 11 |  |  | Example on KMP Pattern Match | Reg | BB,OHP |  |
| 60 | 12 |  |  | Tries- Examples and definitions | Reg | BB,OHP |  |

14. Lecture Notes

**UNIT I**

**Definition**  
An *algorithm* is a finite set of instructions that accomplishes a particular task.

Criteria

* + input
  + output
  + definiteness: clear and unambiguous
  + finiteness: terminate after a finite number of steps
  + effectiveness: instruction is basic enough to be carried out

**Data Type**  
A *data type* is a collection of *objects* and a set of *operations* that act on those objects.

**Abstract Data Type**  
An *abstract data type(ADT)* is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations

Specificatin vs Implementation

Operation specification

* + function name
  + the types of arguments
  + the type of the results

Implementation independent

**\*Structure 1.1:**Abstract data type *Natural\_Number*   
**structure** Natural\_Number is  
 **objects**: an ordered subrange of the integers starting at zero and ending   
 at the maximum integer (*INT\_MAX*) on the computer  
 **functions**:  
 for all x, y ∈ *Nat\_Number*; *TRUE, FALSE ∈ Boolean* and where +, -, <, and == are the usual integer operations.  
 *Nat\_No* Zero ( ) ::= 0  
 *Boolean* Is\_Zero(x) ::= **if** (x) **return** *FALSE*  
 **else return** *TRUE* *Nat\_No* Add(x, y) ::= **if** ((x+y) <= *INT\_MAX*) **return** x+y   
 **else return** *INT\_MAX*  
 *Boolean* Equal(x,y) ::= **if** (x== y) **return** *TRUE* **else return** *FALSE*  
 *Nat\_No* Successor(x) ::= **if** (x == *INT\_MAX*) **return** x  
 **else return** x+1  
 *Nat\_No* Subtract(x,y) ::= **if** (x<y) **return** 0  
 **else return** x-y  
 **end** *Natural\_Number*

**Space Complexity**  
S(P)=C+SP(I)

Fixed Space Requirements (C)  
Independent of the characteristics of the inputs and outputs

* + instruction space
  + space for simple variables, fixed-size structured variable, constants

Variable Space Requirements (SP(I))  
depend on the instance characteristic I

* + number, size, values of inputs and outputs associated with I
  + recursive stack space, formal parameters, local variables, return address

**\*Program 1.1:** Simple arithmetic function   
float abc(float a, float b, float c)  
{  
 return a + b + b \* c + (a + b - c) / (a + b) + 4.00;  
 }  
Sabc(I) = 0   
  
**\*Program 1.2:** Iterative function for summing a list of numbers   
float sum(float list[ ], int n)  
{  
 float tempsum = 0;  
 int i;  
 for (i = 0; i<n; i++)  
 tempsum += list [i];  
 return tempsum;  
}

Ssum(I) = 0

Recall: pass the address of the first element of the array & pass by value

**\*Program 1.3:** Recursive function for summing a list of numbers   
float rsum(float list[ ], int n)  
{  
 if (n) return rsum(list, n-1) + list[n-1];  
 return 0;  
 }  
Ssum(I)=Ssum(n)=6n

\***Figure 1.1:** Space needed for one recursive call of Program 1.3



**Time Complexity**

T(P)=C+TP(I)

Compile time (C)  
independent of instance characteristics

*TP*(*n*)=*caADD*(*n*)+*csSUB*(*n*)+*clLDA*(*n*)+*cstSTA*(*n*)

run (execution) time TP

Definition  
A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Example

* + abc = a + b + b \* c + (a + b - c) / (a + b) + 4.0
  + abc = a + b + c

Regard as the same unit machine independent

**Methods to compute the step count**

1.Introduce variable count into programs

2.Tabular method

* + Determine the total number of steps contributed by each statement  
    step per execution × frequency
  + add up the contribution of all statements

Iterative summing of a list of numbers

**\*Program 1.4:** Program 1.2 with count statements   
  
float sum(float list[ ], int n)  
{  
 float tempsum = 0; **count++**; /\* for assignment \*/  
 int i;  
 for (i = 0; i < n; i++) {  
 **count++**; /\*for the for loop \*/  
 tempsum += list[i]; **count++**; /\* for assignment \*/  
 }  
 **count++**; /\* last execution of for \*/  
 return tempsum;   
 **count++**; /\* for return \*/   
}

2n + 3 steps

**\*Program 1.5:** Simplified version of Program 1.4  
  
float sum(float list[ ], int n)  
{  
 float tempsum = 0;  
 int i;   
 for (i = 0; i < n; i++)  
 **count += 2**;  
 **count += 3**;  
 return 0;  
}

2n + 3 steps

Recursive summing of a list of numbers

**\*Program 1.6:** Program 1.4 with count statements added   
  
float rsum(float list[ ], int n)  
{  
 **count++**; /\*for if conditional \*/  
 if (n) {  
 **count++**; /\* for return and rsum invocation \*/  
 return rsum(list, n-1) + list[n-1];  
 }  
 **count++**;  
 return list[0];  
} 2n+2

**Matrix addition**

**\*Program 1.7:** Matrix addition   
  
void add( int a[ ] [MAX\_SIZE], int b[ ] [MAX\_SIZE],  
 int c [ ] [MAX\_SIZE], int rows, int cols)  
{  
 int i, j;  
 for (i = 0; i < rows; i++)  
 for (j= 0; j < cols; j++)  
 c[i][j] = a[i][j] +b[i][j];  
 }

**\*Program 1.8:** Matrix addition with count statements

void add(int a[ ][MAX\_SIZE], int b[ ][MAX\_SIZE],  
 int c[ ][MAX\_SIZE], int row, int cols )  
{  
 int i, j;  
 for (i = 0; i < rows; i++){  
 **count++**; /\* for i for loop \*/  
 for (j = 0; j < cols; j++) {  
 **count++**; /\* for j for loop \*/  
 c[i][j] = a[i][j] + b[i][j];  
 **count++**; /\* for assignment statement \*/  
 }  
 **count++**; /\* last time of j for loop \*/  
 }  
 **count++**; /\* last time of i for loop \*/  
}

**\*Program 1.9:** Simplification of Program 1.7  
  
void add(int a[ ][MAX\_SIZE], int b [ ][MAX\_SIZE],  
 int c[ ][MAX\_SIZE], int rows, int cols)  
{  
 int i, j;  
 for( i = 0; i < rows; i++) {  
 for (j = 0; j < cols; j++)  
 **count += 2**;  
 **count += 2**;   
 }  
 **count++;**   
}

2rows × cols + 2rows +1

Tabular Method

**Figure 1.2:** Step count table for Program 1.2

Iterative function to sum a list of numbers



Recursive Function to sum of a list of numbers

**\*Figure 1.3:** Step count table for recursive summing function



Matrix Addition

Step count table for matrix addition





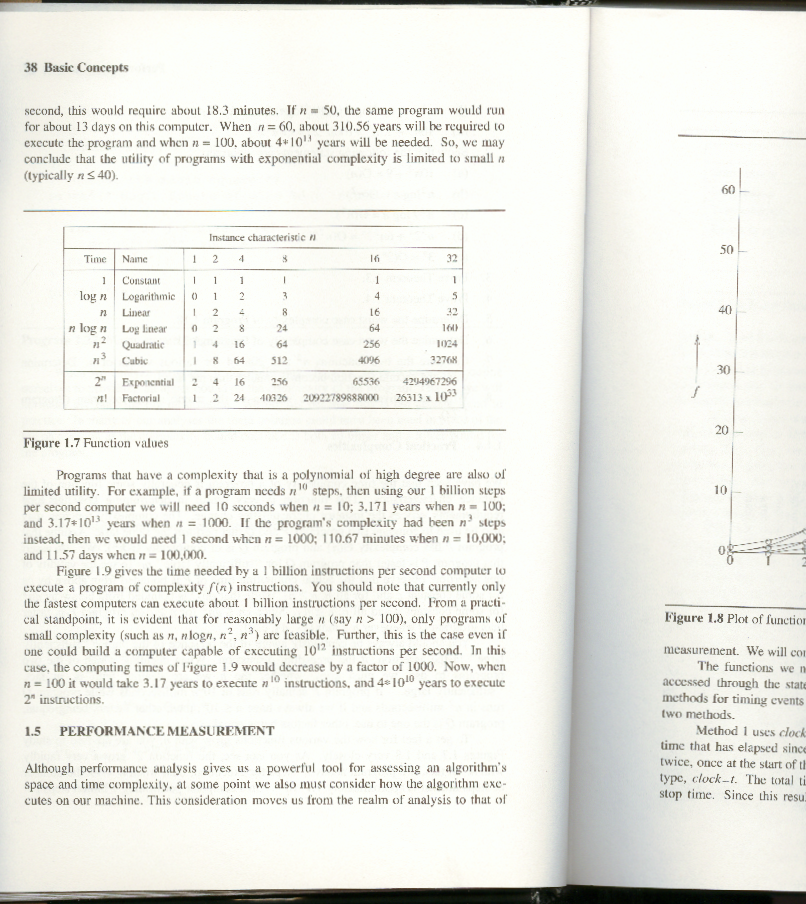
Definition  
f(n) = O(g(n)) iff there exist positive constants c and n0 such that f(n) ≤ cg(n) for all n, n ≥ n0.

Examples

* + 3n+2=O(n) /\* 3n+2≤4n for n≥2 \*/
  + 3n+3=O(n) /\* 3n+3≤4n for n≥3 \*/
  + 100n+6=O(n) /\* 100n+6≤101n for n≥10 \*/
  + 10n2+4n+2=O(n2) /\* 10n2+4n+2≤11n2 for n≥5 \*/
  + 6\*2n+n2=O(2n) /\* 6\*2n+n2 ≤7\*2n for n≥4 \*/



* Complexity of c1n2+c2n and c3n
  + for sufficiently large of value, c3n is faster than c1n2+c2n
  + for small values of n, either could be faster
    - c1=1, c2=2, c3=100 --> c1n2+c2n ≤ c3n for n ≤ 98
    - c1=1, c2=2, c3=1000 --> c1n2+c2n ≤ c3n for n ≤ 998
  + break even point
    - no matter what the values of c1, c2, and c3, the n beyond which c3n is always faster than c1n2+c2n

**Figure 1.4:**Function values 

**SINGLY LINKED LISTS**

**Linked list**

An ordered sequence of nodes with links

The nodes do not reside in sequential locations

The locations of the nodes may change on different runs

**Usual way to draw a linked list**

**create a linked list of words**

typedef struct list\_node \*list\_pointer;  
typedef struct list\_node {  
 char data [4];  
 list\_pointer link;  
 };  
Creation  
list\_pointer ptr =NULL;   
Testing  
#define IS\_EMPTY(ptr) (!(ptr))  
Allocation  
ptr=(list\_pointer) malloc (sizeof(list\_node));

e -> name -> (\*e).name

strcpy(ptr -> data, “bat”);

ptr -> link = NULL;

**Referencing the fields of a node**

typedef struct list\_node \*list\_pointer;  
typedef struct list\_node {  
 int data;  
 list\_pointer link;  
 };  
list\_pointer ptr =NULL

list\_pointer create2( )  
{  
/\* create a linked list with two nodes \*/  
 list\_pointer first, second;  
 first = (list\_pointer) malloc(sizeof(list\_node));  
 second = ( list\_pointer) malloc(sizeof(list\_node));  
 second -> link = NULL;  
 second -> data = 20;  
 first -> data = 10;  
 first ->link = second;  
 return first;  
  
  
  
}

**Insert mat after cat**

1. Get a node that is currently unused ; let its address be paddr.
2. Set the data field of this node to mat.
3. Set paddr’s link field to point to the address found in the link field of the node containing cat.
4. Set the link field of the node containing cat to point to paddr.

**Insert a node after a specific node**

#define IS\_FULL(p) (!(p))

void insert(list\_pointer \*ptr, list\_pointer node)  
{  
/\* insert a new node with data = 50 into the list ptr after node \*/  
   
 list\_pointer temp;  
 temp = (list\_pointer) malloc(sizeof(list\_node));  
 if (IS\_FULL(temp)) //if, temp==NULL   
 {  
 fprintf(stderr, “The memory is full\n”);  
 exit (1);  
 }

temp->data = 50;  
 if (\*ptr) { // nonempty list  
 temp->link =node ->link;   
 node->link = temp;  
 }  
 else { // empty list  
 temp->link = NULL;  
 \*ptr =temp;  
 }  
}

**List Deletion**

1. **Before b) After deletion**

Delete other than first node

void delete(list\_pointer \*ptr, list\_pointer trail,   
 list\_pointer node)  
{  
/\* delete node from the list, trail is the preceding node  
 ptr is the head of the list \*/  
 if (trail)  
 trail->link = node->link;  
 else  
 \*ptr = (\*ptr) ->link;  
 free(node);  
}

**Print out a list (traverse a list)**

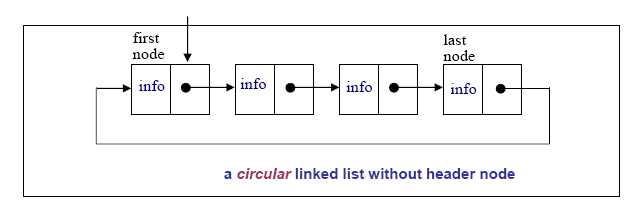
void print\_list(list\_pointer ptr)  
{  
 printf(“The list ocntains: “);  
 for ( ; ptr; ptr = ptr->link)  
 printf(“%4d”, ptr->data);  
 printf(“\n”);  
 }

**Circular Linked Lists**

A Circular Linked List is a special type of Linked List

It supports traversing from the end of the list to the beginning by making the last node point back to the head of the list

A Rear pointer is often used instead of a Head pointer



**MOTIVATION**

Circular linked lists are usually sorted

Circular linked lists are useful for playing video and sound files in “looping” mode

They are also a stepping stone to implementing graphs, an important topic in comp171.

**Definition**

#include <stdio.h>

using namespace std;

struct Node{

int data;

Node\* next;

};

typedef Node\* NodePtr;

Circular Linked List Operations

insertNode(NodePtr& Rear, int item)

//add new node to ordered circular linked list

deleteNode(NodePtr& Rear, int item)

//remove a node from circular linked list

print(NodePtr Rear)

//print the Circular Linked List once

**Traverse The List**

void print(NodePtr Rear){

NodePtr Cur;

if(Rear != NULL){

Cur = Rear->next;

do{

cout << Cur->data << " ";

Cur = Cur->next;

}while(Cur != Rear->next);

cout << endl;

}

}

**Insert Node**

* **Insert into an empty list**

NotePtr New = new Node;

New->data = 10;

Rear = New;

Rear->next = Rear;

* **Insert to head of a Circular Linked List**

New->next = Cur; // same as: New->next = Rear->next;

Prev->next = New; // same as: Rear->next = New;

* **Insert to middle of a Circular Linked List between Pre and Cur**

New->next = Cur;

Prev->next = New;

**Insert to end of a Circular Linked List**

New->next = Cur; // same as: New->next = Rear->next;

Prev->next = New; // same as: Rear->next = New;

Rear = New;

void insertNode(NodePtr& Rear, int item){

NodePtr New, Cur, Prev;

New = new Node;

New->data = item;

if(Rear == NULL){ // insert into empty list

Rear = New;

Rear->next = Rear;

return;

}

Prev = Rear;

Cur = Rear->next;

do{ // find Prev and Cur

if(item <= Cur->data)

break;

Prev = Cur;

Cur = Cur->next;

}while(Cur != Rear->next);

New->next = Cur; // revise pointers

Prev->next = New;

if(item > Rear->data) //revise Rear pointer if adding to end

Rear = New;

}

* **Delete a node from a single-node Circular Linked List**

Rear = NULL;

delete Cur;

* **Delete the head node from a Circular Linked List**

Prev->next = Cur->next; // same as: Rear->next = Cur->next

* **Delete a middle node Cur from a Circular Linked List**

Prev->next = Cur->next;

delete Cur;

DelPrev->next = Cur->next; // same as: Rear->next;

delete Cur;

Rear = Prev;

void deleteNode(NodePtr& Rear, int item){

NodePtr Cur, Prev;

if(Rear == NULL){

cout << "Trying to delete empty list" << endl;

return;

}

Prev = Rear;

Cur = Rear->next;

do{ // find Prev and Cur

if(item <= Cur->data) break;

Prev = Cur;

Cur = Cur->next;

}while(Cur != Rear->next);

if(Cur->data != item){ // data does not exist

cout << "Data Not Found" << endl;

return;

}

if(Cur == Prev){ // delete single-node list

Rear = NULL;

delete Cur;

return;

}

if(Cur == Rear) // revise Rear pointer if deleting end

Rear = Prev;

Prev->next = Cur->next; // revise pointers

delete Cur;

}

void main(){

NodePtr Rear = NULL;

insertNode(Rear, 3);

insertNode(Rear, 1);

insertNode(Rear, 7);

insertNode(Rear, 5);

insertNode(Rear, 8);

print(Rear);

deleteNode(Rear, 1);

deleteNode(Rear, 3);

deleteNode(Rear, 8);

print(Rear);

insertNode(Rear, 1);

insertNode(Rear, 8);

print(Rear);

}

**Result is:**

**1 3 5 7 8**

**5 7**

**1 5 7 8**

**Doubly Linked List**

Move in forward and backward direction.

Singly linked list (in one direction only)

How to get the preceding node during deletion or insertion?

Using 2 pointers

Node in doubly linked list

left link field (llink)

data field (item)

right link field (rlink)

typedef struct node \*node\_pointer;

typedef struct node {

node\_pointer llink;

element item;

node\_pointer rlink;

}

**Empty doubly linked circular list with head node**

**Insertion into an empty doubly linked circular list**

**INSERT**

void dinsert(node\_pointer node, node\_pointer newnode)

{

(1) newnode->llink = node;

(2) newnode->rlink = node->rlink;

(3) node->rlink->llink = newnode;

(4) node->rlink = newnode;

**}**

**DELETE**

void ddelete(node\_pointer node, node\_pointer deleted)

{

if (node==deleted)

printf(“Deletion of head node not permitted.\n”);

else {

(1) deleted->llink->rlink= deleted->rlink;

(2) deleted->rlink->llink= deleted->llink;

free(deleted);

}

}

**ARRAYS**

**Array**: a set of index and value

a collection of data of the same type data structure

For each index, there is a value associated with that index

representation (possible)

implemented by using consecutive memory.

Array A(i)=ai i∈Integers

**Structure** *Array* is   
 **objects:** A set of pairs <*index, value*> where for each value of *index*   
 there is a value from the set *item*. *Index* is a finite ordered set of one or   
 more dimensions, for example, {0, … , n-1} for one dimension,   
 {(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)} for two dimensions,   
 etc.  
 **Functions:** for all A ∈ Array, *i* ∈ *index*, x ∈ *item, j, size* ∈ integer  
 Array Create(j, list) ::= **return** an array of  *j* dimensions where list is a   
 j-tuple whose ith element is the size of the   
 ith dimension. *Items* are undefined.   
 *Item* Retrieve(A, *i*) ::= **if** (*i* ∈ *index*) **return** the item associated with   
 index value *i* in array A  
 **else return** error  
 *Array Store*(A, *i, x*) ::= **if (***i* in *index*) **return** an array that is identical to array A except the new pair <*i, x*> has been inserted **else return** error  **end A**rray  
 **Abstract Data Type *Array***

**Arrays in C**

int list[5], \*plist[5];

list[5]: five integers

list[0], list[1], list[2], list[3], list[4]

\*plist[5]: five pointers to integers

plist[0], plist[1], plist[2], plist[3], plist[4]

implementation of 1-D array

list[0] base address = α

list[1] α + 1\*sizeof(int)

list[2] α + 2\*sizeof(int)

list[3] α + 3\*sizeof(int)

list[4] α + 4\*size(int)

Compare int \*list1 and int list2[5] in C.

Same: list1 and list2 are pointers.

Difference: list2 reserves five locations.

Notations:

list2 - a pointer to list2[0]

(list2 + i) - a pointer to list2[i] (&list2[i])

\*(list2 + i) - list2[i]

**Example: 1-dimension array addressing**

int one[] = {0, 1, 2, 3, 4};

Goal: print out address and value

void print1(int \*ptr, int rows)

{

/\* print out a one-dimensional array using a pointer \*/

int i;

printf(“Address Contents\n”);

for (i=0; i < rows; i++)

printf(“%8u%5d\n”, ptr+i, \*(ptr+i));

printf(“\n”);

}



1. A[-3..2, -1..6, 2..7, 0..5] each of its elements occupies three memory spaces starting from 123

Find the address of the element A[0,1,2,3]

1. In row major order (2) In column major order

The Sparse Matrix Abstract Data Type

Matrix

* + Examples of matrix
* Sparse matrix
  + Many zero items
* Representation of matrix
  + A[][], standard representation
  + Sparse matrix, **store non-zero item only**

**Structure** *Sparse\_Matrix* is  
 **objects:** a set of triples, <*row, column, value*>, where *row*   
 and *column* are integers and form a unique combination, and  
 *value* comes from the set *item.*  
 **functions**:  
 for all *a, b* ∈ *Sparse\_Matrix*, *x* *item, i, j, max\_col*,   
 *max\_row* *index* *Sparse\_Marix* Create(*max\_row, max\_col*) ::=  
 **return** a *Sparse\_matrix* that can hold up to  
 *max\_items* = *max \_row* *max\_col* and   
 whose maximum row size is *max\_row* and   
 whose maximum column size is *max\_col.*

*Sparse\_Matrix* Transpose(*a*) ::=  
 **return** the matrix produced by interchanging  
 the row and column value of every triple.  
*Sparse\_Matrix* Add(*a, b*) ::=  
 **if** the dimensions of a and b are the same   
 **return** the matrix produced by adding   
 corresponding items, namely those with   
 identical *row* and *column* values.  
 **else return** error  
*Sparse\_Matrix* Multiply(*a, b*) ::=  
 **if** number of columns in a equals number of   
 rows in **b  
 return** the matrix *d* produced by multiplying  
 a by *b* according to the formula: *d* [*i*] [*j*] =  
 (a[i][k]*•*b[k][j]) where *d (i, j)* is the *(i,j)*th  
 element  
 **else return** error.

**Abstract data type Sparse-Matrix**

(1) Represented by a two-dimensional array.

Sparse matrix wastes space.

(2) Each element is characterized by <row, col, value>.

**Sparse matrix and its transpose stored as triples**

Sparse\_matrix Create(max\_row, max\_col) ::=  
   
#define MAX\_TERMS 101 /\* maximum number of terms +1\*/  
 typedef struct {  
 int col;  
 int row;  
 int value;  
 } term;  
 term a[MAX\_TERMS]

**Transpose a Matrix**

(1) for each row i   
 take element <i, j, value> and store it   
 in element <j, i, value> of the transpose.

difficulty: where to put <j, i, value>

(0, 0, 15) ====> (0, 0, 15)

(0, 3, 22) ====> (3, 0, 22)

(0, 5, -15) ====> (5, 0, -15)

(1, 1, 11) ====> (1, 1, 11)

Move elements down very often.

(2) For all elements in column j,

place element <i, j, value> in element <j, i, value>

void transpose (term a[], term b[])  
/\* b is set to the transpose of a \*/  
{  
 int n, i, j, currentb;  
 n = a[0].value; /\* total number of elements \*/  
 b[0].row = a[0].col; /\* rows in b = columns in a \*/  
 b[0].col = a[0].row; /\*columns in b = rows in a \*/  
 b[0].value = n;  
 if (n > 0) { /\*non zero matrix \*/  
 currentb = 1;  
 for (i = 0; i < a[0].col; i++)  
 /\* transpose by columns in a \*/  
 for( j = 1; j <= n; j++)  
 /\* find elements from the current column \*/  
 if (a[j].col == i) {  
 /\* element is in current column, add it to b \*/

if (n > 0) { /\*non zero matrix \*/  
 currentb = 1;  
 for (i = 0; i < a[0].col; i++)  
 /\* transpose by columns in a \*/  
 for( j = 1; j <= n; j++)  
 /\* find elements from the current column \*/  
 if (a[j].col == i) {  
 /\* element is in current column, add it to b \*/   
  
 b[currentb].row = a[j].col;  
 b[currentb].col = a[j].row;  
 b[currentb].value = a[j].value;  
 currentb++  
 }  
 }  
} O(columns\*elements)

**Discussion: compared with 2-D array representation**

O(columns\*elements) vs. O(columns\*rows)

elements --> columns \* rows when nonsparse

O(columns\*columns\*rows)

Problem: Scan the array “columns” times.

Solution:

Determine the number of elements in each column of the original matrix.

==>

Determine the starting positions of each row in the transpose matrix.

[0] [1] [2] [3] [4] [5]  
row\_terms = 2 1 2 2 0 1  
starting\_pos = 1 3 4 6 8 8

void fast\_transpose(term a[ ], term b[ ])  
 {  
 /\* the transpose of a is placed in b \*/  
 int row\_terms[MAX\_COL], starting\_pos[MAX\_COL];  
 int i, j, num\_cols = a[0].col, num\_terms = a[0].value;  
 b[0].row = num\_cols; b[0].col = a[0].row;  
 b[0].value = num\_terms;  
 if (num\_terms > 0){ /\*nonzero matrix\*/   
 for (i = 0; i < num\_cols; i++)  
 row\_terms[i] = 0;  
 for (i = 1; i <= num\_terms; i++)  
 row\_term [a[i].col]++  
 starting\_pos[0] = 1;  
 for (i =1; i < num\_cols; i++)   
 starting\_pos[i]=starting\_pos[i-1] +row\_terms [i-1];   
 for (i=1; i <= num\_terms, i++) {  
 j = starting\_pos[a[i].col]++;  
 b[j].row = a[i].col;   
 b[j].col = a[i].row;  
 b[j].value = a[i].value;  
 }  
 }  
}

**Fast transpose of a sparse matrix**

Compared with 2-D array representation

O(columns+elements) vs. O(columns\*rows)

elements --> columns \* rows

O(columns+elements) --> O(columns\*rows)

Cost: Additional row\_terms and starting\_pos arrays are required.

Let the two arrays row\_terms and starting\_pos be shared.

|  |  |  |
| --- | --- | --- |
|  | space | time |
| 2D array | **O(rows \* cols)** | **O(rows \* cols)** |
| Transpose | **O(elements)** | **O(cols \* elmnts)** |
| Fast Transpose | **O(elmnts+MAX\_COL)** | **O(cols + elmnts)** |

**Sparse Matrix Multiplication**

Definition: [D]m\*p=[A]m\*n\* [B]n\*p

Procedure: Fix a row of A and find all elements in column j

of B for j=0, 1, …, p-1.

Alternative 1. Scan all of B to find all elements in j.

Alternative 2. Compute the transpose of B.

(Put all column elements consecutively)



void mmult (term a[ ], term b[ ], term d[ ] )  
/\* multiply two sparse matrices \*/  
{  
 int i, j, column, totalb = b[].value, totald = 0;  
 int rows\_a = a[0].row, cols\_a = a[0].col,  
 totala = a[0].value; int cols\_b = b[0].col,  
 int row\_begin = 1, row = a[1].row, sum =0;  
 int new\_b[MAX\_TERMS][3];  
 if (cols\_a != b[0].row){  
 fprintf (stderr, “Incompatible matrices\n”);  
 exit (1);  
 }

fast\_transpose(b, new\_b);  
a[totala+1].row = rows\_a;  
new\_b[totalb+1].row = cols\_b;  
new\_b[totalb+1].col = 0;  
for (i = 1; i <= totala; ) {  
 column = new\_b[1].row;  
 for (j = 1; j <= totalb+1;) {  
 /\* mutiply row of a by column of b \*/  
 if (a[i].row != row) {  
 storesum(d, &totald, row, column, &sum);  
 i = row\_begin;  
 for (; new\_b[j].row == column; j++)  
 ;  
 column =new\_b[j].row   
 }

else switch (COMPARE (a[i].col, new\_b[j].col)) {  
 case -1: /\* go to next term in a \*/  
 i++; break;  
 case 0: /\* add terms, go to next term in a and b \*/  
 sum += (a[i++].value \* new\_b[j++].value);  
 break;  
 case 1: /\* advance to next term in b\*/  
 j++   
 }  
 } /\* end of for j <= totalb+1 \*/  
 for (; a[i].row == row; i++)  
 ;  
 row\_begin = i; row = a[i].row;  
 } /\* end of for i <=totala \*/  
 d[0].row = rows\_a;  
 d[0].col = cols\_b; d[0].value = totald;  
}

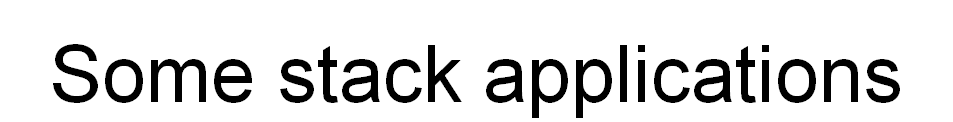
**Program : Sparse matrix multiplication**

**UNIT II**

**STACKS AND QUEUES**

Stack (stack: a Last-In-First-Out (LIFO) list )

* Stack
  + An ordered list
  + Insertions and deletions are made at one end, called top
* Illustration

****

* Implementing recusive call
* Expression evaluation
  + Infix to postfix
  + Postfix evaluation
* Maze problem
* Breadth First Search

**an application of stack: stack frame of function call**

**abstract data type for stack**

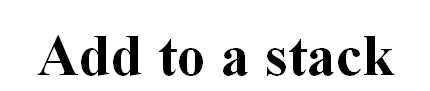
structure *Stack* is  
 objects: a finite ordered list with zero or more elements.  
 functions:  
 for all *stack* ∈ *Stack*, *item* ∈ *element*, *max\_stack\_size*   
 ∈ positive integer  
 *Stack* CreateS(*max\_stack\_size*) ::=  
 create an empty stack whose maximum size is   
 *max\_stack\_size*  *Boolean* IsFull(*stack, max\_stack\_size*) ::=  
 if (number of elements in *stack == max\_stack\_size*)  
 return TRUE  
 else return FALSE  
 *Stack* Add(*stack, item*) ::=  
 if (IsFull(*stack*)) *stack\_full*  
 else insert *item* into top of *stack* and return

*Boolean* IsEmpty(*stack*) ::= **if**(*stack* == CreateS(*max\_stack\_size*))  
  **return** TRUE  
  
 **else return** FALSE  
  
*Element* Delete(*stack*) ::= **if**(IsEmpty(*stack*)) **return**  
  
 **else** remove and return the *item* on the top of the stack

***Structure 2.1:*** *Abstract data type Stack*

**Implementation:** using array

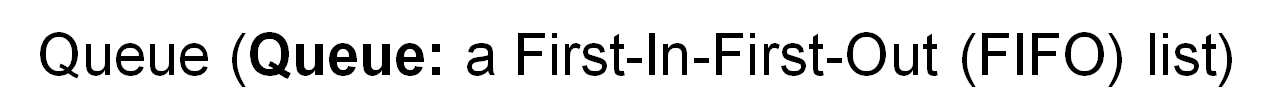
***Stack* CreateS(max\_stack\_size)** ::=  
 #define MAX\_STACK\_SIZE 100 /\* maximum stack size \*/  
 typedef struct {  
 int key;  
 /\* other fields \*/  
 } element;  
 element stack[MAX\_STACK\_SIZE];  
 int top = -1;  
  
 ***Boolean*** **IsEmpty(Stack)** ::= top< 0;  
  
 ***Boolean*** **IsFull(Stack)** ::= top >= MAX\_STACK\_SIZE-1;



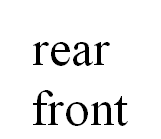
void add(int \*top, element item)  
{  
 if (\*top >= MAX\_STACK\_SIZE-1) {  
 stack\_full( );  
 return;  
 }  
 stack[++\*top] = item;  
}  
  
***\*program 2.1:*** *Add to a stack*

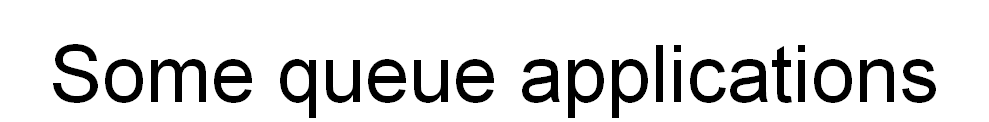
**Delete from a stack**

element delete(int \*top)  
{  
 if (\*top == -1)  
 return stack\_empty( ); /\* returns and error key \*/  
  
 return stack[(\*top)--];  
 }  
  
***\*Program 2.2:*** *Delete from a stack*



* Queue
  + An ordered list
  + All insertions take place at one end, ***rear***
  + All deletions take place at the opposite end, ***front***
* Illustration

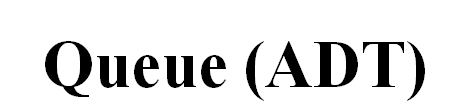


* Job scheduling
* Event list in simulator
* Server and Customs

**Application:** Job scheduling



***Figure 3.2:*** *Insertion and deletion from a sequential queue*

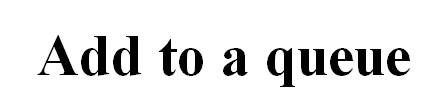


**structure** *Queue* is   
 **objects:** a finite ordered list with zero or more elements.  
 **functions:** for all *queue* ∈ *Queue*, *item* ∈ *element*,   
 *max\_ queue\_ size* ∈ positive integer  
 *Queue* CreateQ(*max\_queue\_size*) ::=  
 create an empty queue whose maximum size is  
 *max\_queue\_size*  
 *Boolean* IsFullQ(*queue, max\_queue\_size*) ::=   
  **if**(number of elements in *queue* == *max\_queue\_size*)  
  **return** *TRUE*  
 **else return** *FALSE*  
 *Queue* AddQ(*queue, item*) ::= **if** (IsFullQ(*queue)) queue\_full* **else** insert *item* at rear of *queue* and return *queue*

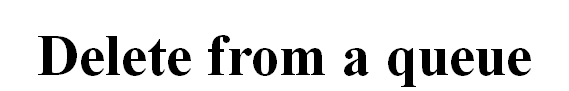
*Boolean* IsEmptyQ(*queue*) ::=  
  **if** (*queue* ==CreateQ(*max\_queue\_size*))  
 **return** *TRUE* **else return** *FALSE  
  
 Element* DeleteQ(*queue*) ::=  
 **if** (IsEmptyQ(*queue*)) **return  
  
 else** remove and return the *item* at front of queue.  
  
  
 ***\*Structure 3.2:*** *Abstract data type Queue*

**Implementation 1:** using array

Queue CreateQ(*max\_queue\_size*) ::=  
# define MAX\_QUEUE\_SIZE 100/\* Maximum queue size \*/  
typedef struct {  
 int key;  
 /\* other fields \*/  
 } element;  
element queue[MAX\_QUEUE\_SIZE];  
int rear = -1;  
int front = -1;  
Boolean IsEmpty(queue) ::= front == rear  
Boolean IsFullQ(queue) ::= rear == MAX\_QUEUE\_SIZE-1



void addq(int \*rear, element item)  
{  
 if (\*rear == MAX\_QUEUE\_SIZE\_1) {  
 queue\_full( );  
 return;  
 }  
 queue [++\*rear] = item;  
}  
  
***\*Program 3.3:*** *Add to a queue*

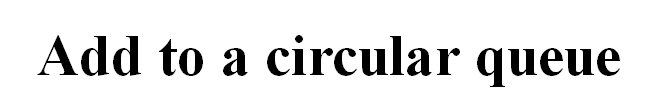


element deleteq(int \*front, int rear)  
{  
 if ( \*front == rear)  
 return queue\_empty( ); /\* return an error key \*/  
  
 return queue [++ \*front];  
}   
  
***\*Program 3.4:*** *Delete from a queue*

**Implementation 2:** regard an array as a circular queue

front: one position counterclockwise from the first element

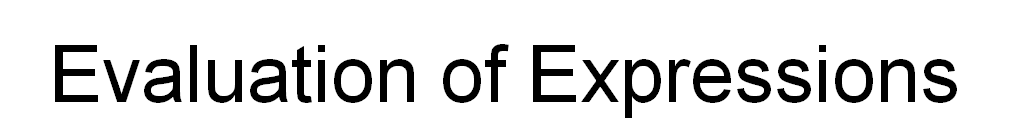
rear: current end



void addq(int front, int \*rear, element item)  
{  
 \*rear = (\*rear +1) % MAX\_QUEUE\_SIZE;  
 if (front == \*rear) /\* reset rear and print error \*/  
 return;  
 }  
 queue[\*rear] = item;   
}  
*\*Program 3.5: Add to a circular queue*

**Delete from a circular queue**

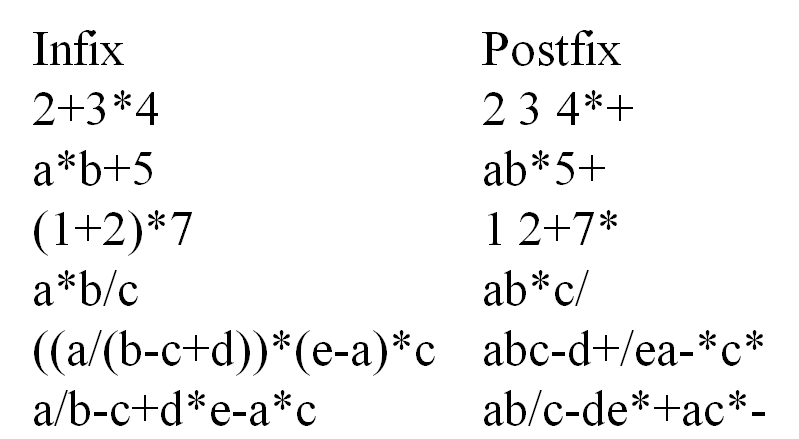
element deleteq(int\* front, int rear)  
{  
 element item;  
 if (\*front == rear)  
 return queue\_empty( );   
 /\* queue\_empty returns an error key \*/  
 \*front = (\*front+1) % MAX\_QUEUE\_SIZE;  
 return queue[\*front];  
}  
  
*\*Program 3.6: Delete from a circular queue*

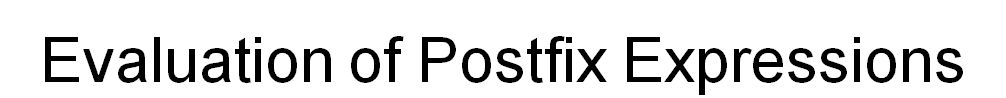


* Evaluating a complex expression in computer
  + ((rear+1==front)||((rear==MaxQueueSize-1)&&!front))
  + x= a/b- c+ d\*e- a\*c
* Figuring out the order of operation within any expression
  + A ***precedence*** hierarchy within any programming language
  + See Figure 3.12

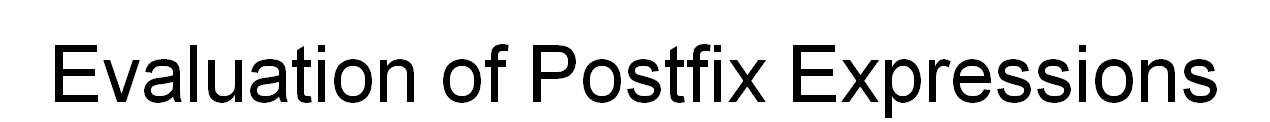
Evaluation of Expressions (Cont.)

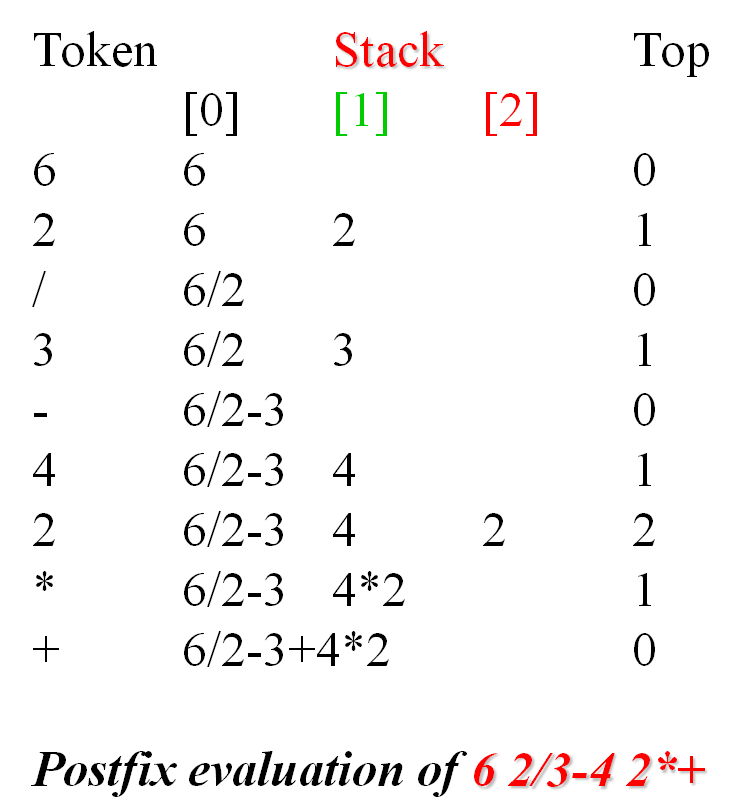
* Ways to write expressions
  + Infix (standard)
  + Prefix
  + Postfix
    - compiler, a parenthesis-free notation





* Left-to-right scan Postfix expression,
  1. Stack operands until find an operator,
  2. Meet operator, remove correct operands for this operator,
  3. Perform the operation,
  4. Stack the result
* Remove the answer from the top of stack





Assumptions:

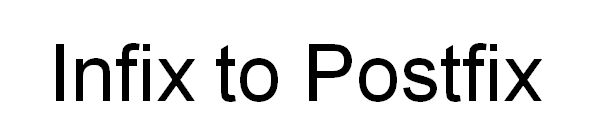
operators: +, -, \*, /, %

operands: single digit integer

#define MAX\_STACK\_SIZE 100   
#define MAX\_EXPR\_SIZE 100 /\* max size of expression \*/  
typedef enum{1paran, rparen, plus, minus, times, divide,   
 mod, eos, operand} precedence;  
int stack[MAX\_STACK\_SIZE]; /\* global stack \*/  
char expr[MAX\_EXPR\_SIZE]; /\* input string

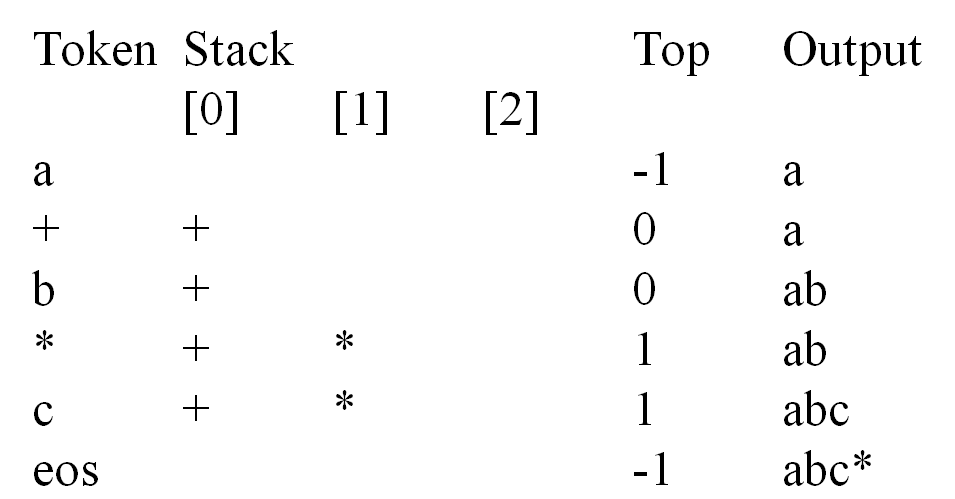
int eval(void)  
{  
 precedence token;  
 char symbol;  
 int op1, op2;  
 int n = 0; /\* counter for the expression string \*/  
 int top = -1;  
 token = get\_token(&symbol, &n);  
 while (token != eos) {  
 if (token == operand)  
 add(&top, symbol-’0’); /\* stack add \*/

else {  
 /\* remove two operands, perform operation, and   
 return result to the stack \*/  
 op2 = delete(&top); /\* stack delete \*/  
 op1 = delete(&top);  
 switch(token) {  
 case plus: add(&top, op1+op2); break;  
 case minus: add(&top, op1-op2); break;   
 case times: add(&top, op1\*op2); break;   
 case divide: add(&top, op1/op2); break;   
 case mod: add(&top, op1%op2);  
 }  
 }  
 token = get\_token (&symbol, &n);  
 }  
 return delete(&top); /\* return result \*/  
}  
 *\*****Program 3.7:*** *Fuprecedence get\_token(char \*symbol, int \*n)  
{  
 \*symbol =expr[(\*n)++];  
 switch (\*symbol) {  
 case ‘(‘ : return lparen;  
 case ’)’ : return rparen;  
 case ‘+’: return plus;  
 case ‘-’ : return minus;  
 case ‘/’ : return divide;  
 case ‘\*’ : return times;  
 case ‘%’ : return mod;  
 case ‘\0‘ : return eos;  
 default : return operand;   
 }  
}****\*Program 3.8:*** *Function to get a token from the input string (p.123)nction to evaluate a postfix expression*

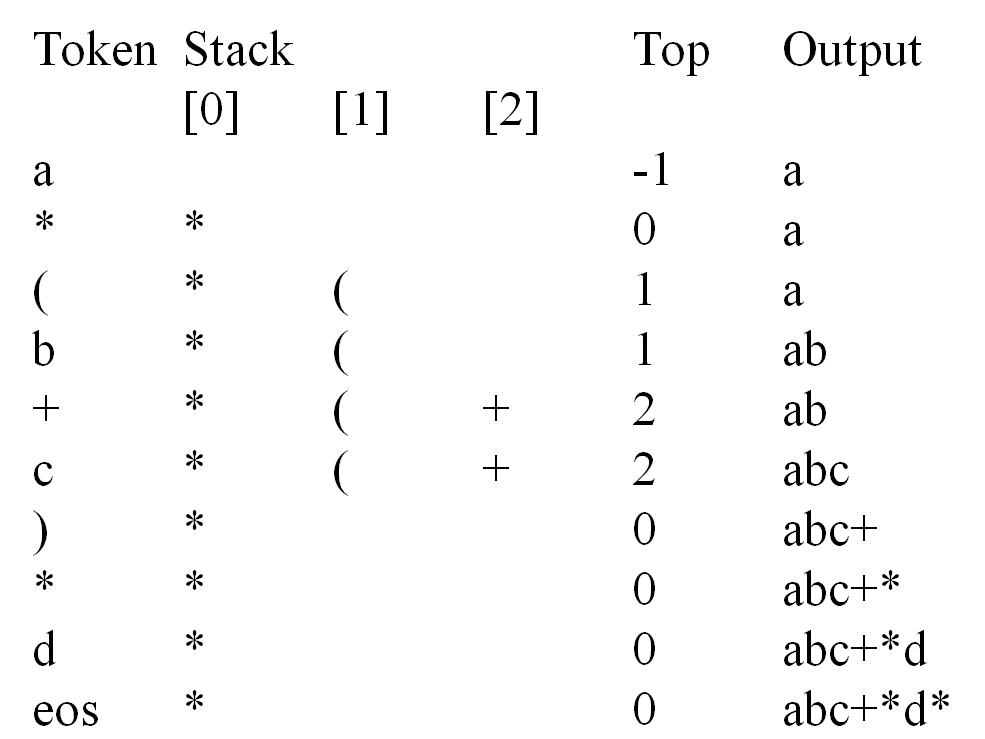


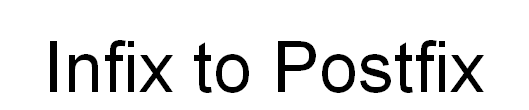
1. Method I
   1. Fully parenthesize the expression
   2. Move all binary operators so that they replace their corresponding right parentheses
   3. Delete all parentheses

* Examples:***a/b-c+d\*e-a\*c***
  + ((((a/b)-c)+(d\*e))-(a\*c)), fully parentheses
  + ab**/**c-de\*+ac\*-, replace right parentheses and delete all parentheses
* Disadvantage
  + inefficient, two passes
* Method II
  + scan the infix expression left-to-right
  + output operand encountered
  + output operators depending on their precedence, i.e., higher precedence operators first
* Example: *a+b\*c*, simple expression



* Example: ***a\*(b+c)\*d*** , parenthesized expression





* Last two examples suggests a precedence-based scheme for stacking and unstacking operators
  + **isp** (in-stack precedence)
  + **icp** (iprecedence stack[MaxStackSize];
  + /\* isp and icp arrays - index is value of precedence
  + lparen, rparen, plus, minus, time divide, mod, eos \*/
  + *static* *int* **isp**[]= { 0, 19, 12, 12, 13, 13, 13, 0};
  + *static* *int* **icp**[]= {20, 19, 12, 12, 13, 13, 13, 0}n-coming precedence)

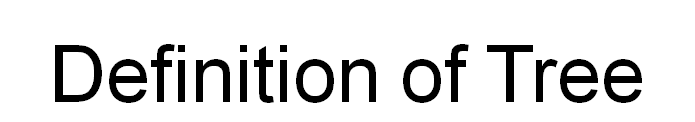
void postfix(void)  
{  
/\* output the postfix of the expression. The expression  
 string, the stack, and top are global \*/  
 char symbol;  
 precedence token;  
 int n = 0;  
 int top = 0; /\* place eos on stack \*/  
 stack[0] = eos;  
 for (token = get \_token(&symbol, &n); token != eos;  
 token = get\_token(&symbol, &n)) {  
 if (token == operand)  
 printf (“%c”, symbol);  
 else if (token == rparen ){

/\*unstack tokens until left parenthesis \*/  
 while (stack[top] != lparen)  
 print\_token(delete(&top));  
 delete(&top); /\*discard the left parenthesis \*/  
 }  
 else{  
 /\* remove and print symbols whose isp is greater  
 than or equal to the current token’s icp \*/  
 while(isp[stack[top]] >= icp[token] )  
 print\_token(delete(&top));  
 add(&top, token);  
 }  
 }  
 while ((token = delete(&top)) != eos)  
 print\_token(token);  
 print(“\n”);  
}  
 ***\*Program 3.9****: Function to convert from infix to postfix*

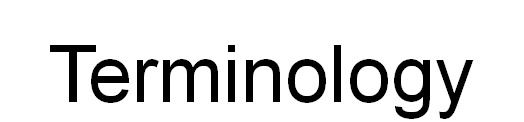
**UNIT III**

**Trees**



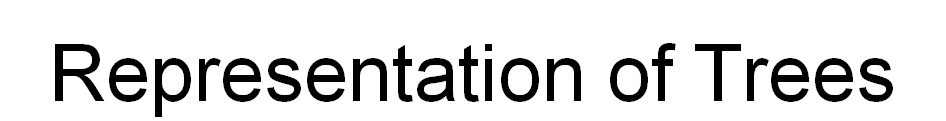


* A tree is a finite set of one or more nodes   
  such that:
* There is a specially designated node called   
  the root.
* The remaining nodes are partitioned into n>=0 disjoint sets T1, ..., Tn, where each of these sets is a tree.
* We call T1, ..., Tn the subtrees of the root.

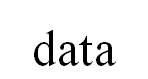


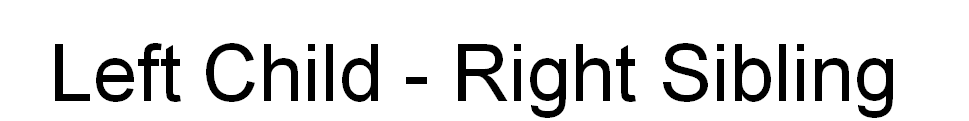
Node, Degree of a node, Leaf (terminal), Nonterminal, Parent, Children, Sibling, Degree of a tree, Ancestor,Level of a node

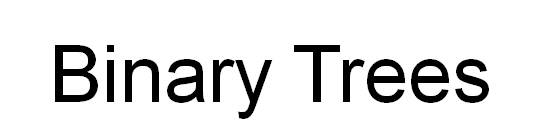
* Height of a tree The degree of a node is the number of subtrees  
  of the node
  + The degree of A is 3; the degree of C is 1.
* The node with degree 0 is a leaf or terminal   
  node.
* A node that has subtrees is the *parent* of the   
  roots of the subtrees.
* The roots of these subtrees are the *children* of   
  the node.
* Children of the same parent are *siblings*.
* The ancestors of a node are all the nodes   
  along the path from the root to the node.



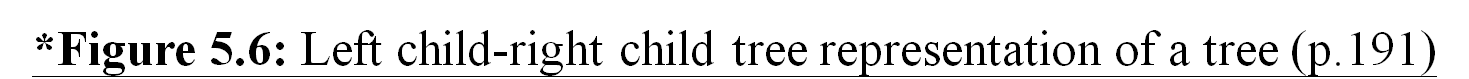
* List Representation
  + ( A ( B ( E ( K, L ), F ), C ( G ), D ( H ( M ), I, J ) ) )
  + The root comes first, followed by a list of sub-trees







* A binary tree is a finite set of nodes that is either empty or consists of a root and two   
  disjoint binary trees called *the left subtree* and *the right subtree*.
* Any tree can be transformed into binary tree.
  + by left child-right sibling representation
* The left subtree and the right subtree are distinguished.



Binary Tree ADT

structure *Binary\_Tree*(abbreviated *BinTree*) is objects: a finite set of nodes either empty or   
consisting of a root node, left *Binary\_Tree*, and right *Binary\_Tree*.

functions:

for all *bt*, *bt1*, *bt2* ∈ *BinTree*, *item* ∈ *element*

*Bintree* Create() ::= creates an empty binary tree

*Boolean* IsEmpty(*bt*) ::= if (*bt*==empty binary tree) return *TRUE* else return *FALSE*

*BinTree* MakeBT(*bt1*, *item*, *bt2*)::= return a binary tree

whose left subtree is *bt1*, whose right subtree is *bt2*,

and whose root node contains the data *item*

*Bintree* Lchild(*bt*)::= if (IsEmpty(*bt*)) return error   
 else return the left subtree of *bt*

*element* Data(*bt*)::= if (IsEmpty(*bt*)) return error  
 else return the data in the root node of *bt*

*Bintree* Rchild(*bt*)::= if (IsEmpty(*bt*)) return error   
 else return the right subtree of *bt*

**Samples of Trees**

**Maximum Number of Nodes in BT**

* The maximum number of nodes on level i of a binary tree is 2i-1, i>=1.
* The maximum nubmer of nodes in a binary tree   
  of depth k is 2k-1, k>=1.

Relations between Number of Leaf Nodes and Nodes of Degree 2

For any nonempty binary tree, T, if n0 is the number of leaf nodes and n2 the number of nodes   
of degree 2, then n0=n2+1

proof:

Let n and B denote the total number of nodes & branches in T.

Let n0, n1, n2 represent the nodes with no children, single child, and two children respectively.

n= n0+n1+n2, B+1=n, B=n1+2n2 ==> n1+2n2+1= n,

n1+2n2+1= n0+n1+n2 ==> n0=n2+1

Full BT VS Complete BT

* A full binary tree of depth k is a binary tree of depth k having 2 -1 nodes, k>=0.
* A binary tree with n nodes and depth k is complete iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.

Binary Tree Representations

* If a complete binary tree with *n* nodes (depth =log *n* + 1) is represented sequentially, then for any node with index *i*, 1<=*i*<=*n*, we have:
  + *parent*(*i*) is at *i*/2 if *i*!=1. If *i*=1, *i* is at the root and has no parent.
  + *left\_child*(*i*) ia at 2*i* if 2*i*<=*n*. If 2*i*>n, then *i* has noleft child.
  + *right\_child*(*i*) ia at 2*i*+1 if 2*i* +1 <=*n*. If 2*i* +1 >n, then *i* has no right child.
* Sequential Representation

Linked Representation

typedef struct node \*tree\_pointer;

typedef struct node {

int data;

tree\_pointer left\_child, right\_child;

};

**Binary Tree Traversals**

Let L, V, and R stand for moving left, visiting the node, and moving right.

There are six possible combinations of traversal

LVR, LRV, VLR, VRL, RVL, RLV

Adopt convention that we traverse left before right, only 3 traversals remain

LVR, LRV, VLR

inorder, postorder, preorder

Inorder Traversal (recursive version)

void inorder(tree\_pointer ptr)

/\* inorder tree traversal \*/

{

if (ptr) {

inorder(ptr->left\_child);

printf(“%d”, ptr->data);

inorder(ptr->right\_child);

}

}

Preorder Traversal (recursive version)

void preorder(tree\_pointer ptr)

/\* preorder tree traversal \*/

{

if (ptr) {

printf(“%d”, ptr->data);

preorder(ptr->left\_child);

preorder(ptr->right\_child);

}

}

Postorder Traversal (recursive version)

void postorder(tree\_pointer ptr)

/\* postorder tree traversal \*/

{

if (ptr) {

postorder(ptr->left\_child);

postorder(ptr->right\_child);

printf(“%d”, ptr->data);

}

}

Iterative Inorder Traversal (using stack)

void iter\_inorder(tree\_pointer node)

{

int top= -1; /\* initialize stack \*/

tree\_pointer stack[MAX\_STACK\_SIZE];

for (;;) {

for (; node; node=node->left\_child)

add(&top, node);/\* add to stack \*/

node= delete(&top);

/\* delete from stack \*/

if (!node) break; /\* empty stack \*/

printf(“%d”, node->data);

node = node->right\_child;

}

}

Trace Operations of Inorder Traversal



Level Order Traversal (using queue)

void level\_order(tree\_pointer ptr)

/\* level order tree traversal \*/

{

int front = rear = 0;

tree\_pointer queue[MAX\_QUEUE\_SIZE];

if (!ptr) return; /\* empty queue \*/

addq(front, &rear, ptr);

for (;;) {

ptr = deleteq(&front, rear);

if (ptr) {

printf(“%d”, ptr->data);

if (ptr->left\_child)

addq(front, &rear,

ptr->left\_child);

if (ptr->right\_child)

addq(front, &rear,

ptr->right\_child);

}

else break;

}

} **Non Recursive Binary Tree Traversals**

**Nonrecursive Inorder Traversal: General Algorithm**

1. current = root; //start traversing the binary tree at

// the root node

1. while(current is not NULL or stack is nonempty)

if(current is not NULL)

{

push current onto stack;

current = current->llink;

}

else

{

pop stack into current;

visit current; //visit the node

current = current->rlink; //move to the

//right child

}

**Nonrecursive Preorder Traversal**

1. current = root; //start the traversal at the root node

2. while(current is not NULL or stack is nonempty)

if(current is not NULL)

{

visit current;

push current onto stack;

current = current->llink;

}

else

{

pop stack into current;

current = current->rlink; //prepare to visit

//the right subtree

}

**Nonrecursive Postorder Traversal**

1. current = root; //start traversal at root node
2. v = 0;
3. if(current is NULL)

the binary tree is empty

1. if(current is not NULL)
   1. push current into stack;
   2. push 1 onto stack;
   3. current = current->llink;
   4. while(stack is not empty)

if(current is not NULL and v is 0)

{

push current and 1 onto stack;

current = current->llink;

}

else

{

pop stack into current and v;

if(v == 1)

{

push current and 2 onto stack;

current = current->rlink;

v = 0;

}

else

visit current;

}

**Threaded Binary Trees**

* Two many null pointers in current representation of binary trees n: number of nodes number of non-null links: n-1 total links: 2n null links: 2n-(n-1)=n+1
* Replace these null pointers with some useful “threads”.
* If ptr->left\_child is null, replace it with a pointer to the node that would be visited *before* ptr in an *inorder traversal*
* If ptr->right\_child is null, replace it with a pointer to the node that would be visited *after* ptr in an *inorder traversal*

Heap

A *max tree* is a tree in which the key value in each node is no smaller than the key values in   
its children. A *max heap* is a complete binary tree that is also a max tree.A *min tree* is a tree in which the key value in each node is no larger than the key values in its children. A *min heap* is a complete binary tree that is also a min tree.

Operations on heaps

* + creation of an empty heap
  + insertion of a new element into the heap;
  + deletion of the largest element from the heap

\

**\*Figure 5.25:** Sample max heaps

**1**

**2**

**3**

**5**

**6**

**1**

**[2]**

**3**

**4**

**[1]**

**2**

Property:

The root of max heap (min heap) contains the largest (smallest).

**\*Figure 5.26:**Sample min heaps

**[1]**

**[2]**

**[3]**

**[5]**

**[6]**

**[1]**

**[2]**

**[3]**

**[4]**

**[1]**

**[2]**

**[4]**

ADT for Max Heap

structure MaxHeap

objects: a complete binary tree of n > 0 elements organized so that   
the value in each node is at least as large as those in its children

functions:

for all *heap* belong to *MaxHeap*, *item* belong to *Element*, *n*,   
*max\_size* belong to integer

MaxHeap Create(max\_size)::= create an empty heap that can   
 hold a maximum of max\_size elements

Boolean HeapFull(heap, n)::= if (n==max\_size) return TRUE  
 else return FALSE

MaxHeap Insert(heap, item, n)::= if (!HeapFull(heap,n)) insert   
 item into heap and return the resulting heap else return error

Boolean HeapEmpty(heap, n)::= if (n>0) return FALSE

else return TRUE

Element Delete(heap,n)::= if (!HeapEmpty(heap,n)) return one  
 instance of the largest element in the heap   
 and remove it from the heap

else return error

Application: priority queue

machine service

* + amount of time (min heap)
  + amount of payment (max heap)

factory

* + time tag

Data Structures

unordered linked list

unordered array

sorted linked list

sorted array

heap

**Figure 5.27:** Priority queue representations



Example of Insertion to Max Heap

initial location of new node

15

2

14

10

15

20

14

10

insert 21 into heap

15

5

14

10

insert 5 into heap

Insertion into a Max Heap

void insert\_max\_heap(element item, int \*n)

{

int i;

if (HEAP\_FULL(\*n)) {

fprintf(stderr, “the heap is full.\n”);

exit(1);

}

i = ++(\*n);

while ((i!=1)&&(item.key>heap[i/2].key)) {

heap[i] = heap[i/2];

i /= 2;

}

heap[i]= item;

}

Example of Deletion from Max Heap

remove

15

2

14

10

15

2

14

14

2

10

(a) Heap structure

(b) 10 inserted at the root

(c) Finial heap

Deletion from a Max Heap

element delete\_max\_heap(int \*n)

{

int parent, child;

element item, temp;

if (HEAP\_EMPTY(\*n)) {

fprintf(stderr, “The heap is empty\n”);

exit(1);

}

/\* save value of the element with the   
 highest key \*/

item = heap[1];

/\* use last element in heap to adjust heap \*/

temp = heap[(\*n)--];

parent = 1;

child = 2;

while (child <= \*n) {

/\* find the larger child of the current

parent \*/

if ((child < \*n)&&

(heap[child].key<heap[child+1].key))

child++;

if (temp.key >= heap[child].key) break;

/\* move to the next lower level \*/

heap[parent] = heap[child];

child \*= 2;

}

heap[parent] = temp;

return item;

}

**GRAPHS**

Definition

A graph, *G=(V, E)*, consists of two sets:

* + a finite set of ***vertices(V)***, and
  + a finite, possibly empty set of edges***(E)***
  + *V(G)* and *E(G)* represent the sets of vertices and edges of *G*, respectively

Undirected graph

* + The pairs of vertices representing any edges is ***unordered***
  + e.g., *(v0, v1)* and *(v1, v0)* represent the same edge

Directed graph

* + Each edge as a directed pair of vertices
  + e.g. *<v0, v1>* represents an edge, *v0* is the tail and *v1* is the head.

Examples for Graph

G1

G2

G3

complete graph

incomplete graph

V(G1)={0,1,2,3} E(G1)={(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)}

V(G2)={0,1,2,3,4,5,6} E(G2)={(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)}

V(G3)={0,1,2} E(G3)={<0,1>,<1,0>,<1,2>}

complete undirected graph: n(n-1)/2 edges

complete directed graph: n(n-1) edges

Complete Graph

A complete graph is a graph that has the   
maximum number of edges

* + for undirected graph with n vertices, the maximum number of edges is n(n-1)/2
  + for directed graph with n vertices, the maximum   
    number of edges is n(n-1)
  + example: G1 is a complete graph

**Adjacent and Incident**

If (v0, v1) is an edge in an undirected graph,

* + v0 and v1 are adjacent
  + The edge (v0, v1) is incident on vertices v0 and v1

If <v0, v1> is an edge in a directed graph

* + v0 is adjacent to v1, and v1 is adjacent from v0
  + The edge <v0, v1> is incident on v0 and v1

Subgraph and Path

* A subgraph of G is a graph G’ such that V(G’)   
  is a subset of V(G) and E(G’) is a subset of E(G)
* A path from vertex vp to vertex vq in a graph G,   
  is a sequence of vertices, vp, vi1, vi2, ..., vin, vq,   
  such that (vp, vi1), (vi1, vi2), ..., (vin, vq) are edges   
  in an undirected graph
* The length of a path is the number of edges on it

**Figure 6.4:** subgraphs of G1 and G3

(i) (ii) (iii) (iv)

(a) Some of the subgraph of G1

(i) (ii) (iii) (iv)

(b) Some of the subgraph of G3

單一

G1

G3

Simple Path and Style

* A simple path is a path in which all vertices, except possibly the first and the last, are distinct
* A cycle is a simple path in which the first and the last vertices are the same
* In an undirected graph G, two vertices, v0 and v1, are connected if there is a path in G from v0 to v1
* An undirected graph is connected if, for every pair of distinct vertices vi, vj, there is a path   
  from vi to vj

**Connected Component**

* A connected component of an undirected graph   
  is a maximal connected subgraph.
* A tree is a graph that is connected and acyclic.
* A directed graph is strongly connected if there   
  is a directed path from vi to vj and also   
  from vj to vi.
* A strongly connected component is a maximal subgraph that is strongly connected

Degree

* The degree of a vertex is the number of edges incident to that vertex
* For directed graph,
  + the in-degree of a vertex *v* is the number of edges  
    that have *v* as the head
  + the out-degree of a vertex *v* is the number of edges  
    that have *v* as the tail
  + if *di* is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is



degree

G1

G2

3

2

3

3

1

1

1

1

directed graph

in-degree

out-degree

in:1, out: 1

in: 1, out: 2

in: 1, out: 0

3

3

3

ADT for Graph

structure Graph is

objects: a nonempty set of vertices and a set of undirected edges, where each   
edge is a pair of vertices

functions: for all *graph* ∈ *Graph*, *v*, *v*1 and *v*2 ∈ *Vertices*

*Graph* Create()::=return an empty graph

*Graph* InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no   
 incident edge.

*Graph* InsertEdge(*graph*, *v1*,*v2*)::= return a graph with new edge   
 between *v1* and *v2*

*Graph* DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges   
 incident to it are removed

*Graph* DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*)   
 is removed

*Boolean* IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE

else return FALSE

*List* Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

**G4**

**G1**

**G3**

**Graph Representations**

**Adjacency Matrix**

* Let G=(V,E) be a graph with n vertices.
* The adjacency matrix of G is a two-dimensional   
  n by n array, say adj\_mat
* If the edge (vi, vj) is in E(G), adj\_mat[i][j]=1
* If there is no such edge in E(G), adj\_mat[i][j]=0
* The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph   
  need not be symmetric

**Examples for Adjacency Matrix**



G1

G2

symmetric

undirected: n2/2

directed: n2

**Merits of Adjacency Matrix**

* From the adjacency matrix, to determine the connection of vertices is easy
* The degree of a vertex is
* For a digraph, the row sum is the out\_degree, while the column sum is the in\_degree



**Adjacency lists**

* linked list

#define MAX\_VERTICES 50

**1**

**2**

**0**

**G3**

typedef struct node \*node\_ptr;

typedef struct node {

int vertex;

node\_ptr link;

**G1**

**3**

**1**

**2**

**2**

**3**

**0**

**1**

**3**

**0**

**2**

**1**

**0**

} node;

node\_ptr graph[MAX\_VERTICES];

int n = 0; /\* number of nodes

Adjacency lists, by array



**G3**

**1**

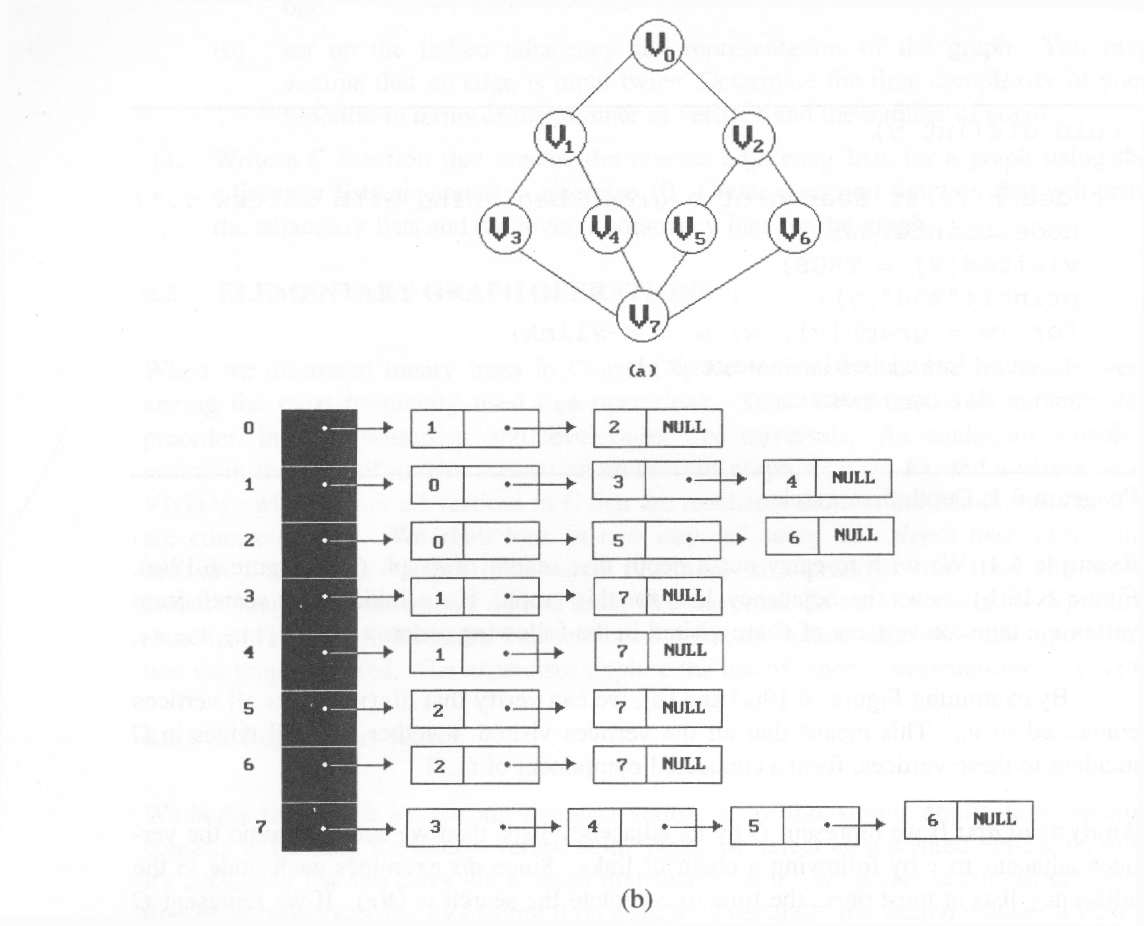
**2**

**0**



**Some Graph Operations**

* Traversal  
  Given G=(V,E) and vertex v, find all w∈V, such that w connects v.
  + Depth First Search (DFS) preorder tree traversal
  + Breadth First Search (BFS) level order tree traversal
* Connected Components
* Spanning Trees



Depth First Search

#define FALSE 0

#define TRUE 1

short int visited[MAX\_VERTICES];

void dfs(int v)

{

node\_pointer w;

visited[v]= TRUE;

printf(“%5d”, v);

for (w=graph[v]; w; w=w->link)

if (!visited[w->vertex])

dfs(w->vertex);

**}**

Data structure: a) adjacency list: O(e) b) adjacency matrix: O(n2)

Breadth-First Search

typedef struct queue \*queue\_ptr;

typedef struct queue {

int vertex;

queue\_ptr link;

};

void addq(queue\_ptr \*, queue\_ptr \*, int);

Int deleteq(queue\_ptr);

void bfs(int v) {

node\_ptr w;

queue\_ptr front, rear;

front=rear=NULL;

printf(“%5d”,v);

visited[v]=TRUE;

addq(&front, &rear, v);

while(front) {

v = deleteq(&front);

for(w=graph[v]; w; w=w->link)

if(!visited[w->vertex]) {

printf(“%5d”, w->vertex);

addq(&front, &rear, w->vertex);

visited[w->vertex] = TRUE;

}

}

}

}

**UNIT-IV**

**Sequential Search(Linear search)**

Example  
44, 55, 12, 42, 94, 18, 06, 67

unsuccessful search

* + n+1

successful search



# define MAX-SIZE 1000/\* maximum size of list plus one \*/  
typedef struct {  
 int key;  
 /\* other fields \*/  
 } element;  
element list[MAX\_SIZE];

**\*Program 4.1:**Sequential search   
  
int seqsearch( int list[ ], int searchnum, int n )  
{  
/\*search an array, list, that has n numbers. Return i, if list[i]=searchnum. Return -1, if searchnum is not in the list \*/  
 int i;  
 list[n]=searchnum; sentinel   
 for (i=0; list[i] != searchnum; i++)  
 ;  
 return (( i<n) ? i : -1);  
}

**Binary Search**

**\*Program 4.2:** Binary search   
  
int binsearch(element list[ ], int searchnum, int n)  
{  
/\* search list [0], ..., list[n-1]\*/  
 int left = 0, right = n-1, middle;  
 while (left <= right) {  
 middle = (left+ right)/2;  
 switch (COMPARE(list[middle].key, searchnum)) {  
 case -1: left = middle +1;  
 break;  
 case 0: return middle;  
 case 1:right = middle - 1;  
 }  
 }  
 return -1;  
}

O(log2n)

**\*Figure 4.1:**Decision tree for binary search

[7]

[2]

[8]

[3]

[0]

[6]

[10]

[1]

[4]

[5]

[9]

[11]

4, 15, 17, 26, 30, 46, 48, 56, 58, 82, 90, 95

* **The Symbol Table  
  Abstract Data Type**

dictionary

- **symbol table** in computer science

- application

1)spelling checker

2)thesarus

3)data dictionary in database

application

4)symbol tables generated by

loader, assembler, and compiler

* **The Symbol Table  
  Abstract Data Type**

operations on symbol table

1)determine if a particular name is

in the table

2)retrieve the attributes of that name

3)modify the attributes of that name

4)insert a new name and its attribute

5)delete a name and its attributes

use hashing

- very good expected performance: O(1)

* **Static Hashing**

hash function



x

f(x) = 

**···**

**···**

hash table

* **Hash Table**

hash tables

- store the identifiers in a fixed

size table called a hash table

0

0

1

2

**···**

s-1

1

b-1

**···**

2

b buckets, and s slots in each bucket

* **Hash Table**

Def)

- identifier density of a hash table:

n/T where

n: number of identifiers in table

T: total number of possible

identifiers

- loading density or loading factor

of a hash table:

a = n/(s·b) where

s: number of slots in each bucket

b: number of bucket

* **Hash Table**

- two identifiers i1 and i2 are *synonyms* with respect to f, if f(i1) = f(i2) where i1 ¹ i2

- an *overflow* occurs when we hash a new identifier, i, into a full bucket

- a *collision* occurs when we hash two nonidentical identifiers into the same bucket

* collisions and overflows occur simultaneously iff bucket size is 1
* **Hash Table**

Example) hash table *ht* with *b*=26, *s*=2, n=10

hash function f

- 1st character of identifier



Identifiers

acos

define

float

exp

char

atan

ceil

floor

**clock**

**ctime**

hash table with 26 bucket and two slots per bucket

**Hash Function**

requirements for a hash function

- easy to compute

- minimizes the number of collision (but, we can not avoid collisions)

uniform hash function

- for randomly chosen x from the identifier space, P[f(x)=i] = 1/b, for all buckets i

- a random x has an equal chance of hashing into any of the b buckets

**mid-square**

- middle of square hash function- frequently used in symbol table applications

hash function fm

1)squaring the identifier

2)obtain the bucket address by using an appropriate number of bits from the middle of the square

3)if we use r bits, 2r buckets are necessary

**division(modular)**

- use the modulus(%) operator

fD(x) = x % M

where M: table size

- range of bucket address: 0 ~ M-1

- the choice of M is critical

- choose M as a prime number such that M does not divide rk±a for small k and a

- choose M such that it has no prime divisors less than 20

**folding**

1)shift folding

ex) identifier x = 12320324111220

123

203

241

112

20

123

203

241

112

20

699

x1

x2

x3

x4

x5

2)folding at the boundaries

→ 123 + 302 + 241 + 211 + 20 = 897

**digit analysis**

- used in case all the identifiers are known in advance

- examine the digits of each identifier

- delete those digits that have skewed distributions

- select the digit positions to be used to calculate the hash address

* **Overflow Handling**

**linear open addressing**

**1) linear probing**

- when overflow occurs, linear search for the empty slot in the hash table using circular rotation

**linear probing**

- represent hash table as a

one-dimensional array

#define MAX\_CHAR 10

/\* max number of characters in an identifier \*/

#define TABLE\_SIZE 13

/\* max table size = prime number\*/

typedef struct {

char key[MAX\_CHAR];

/\* other filed \*/

} element;

element hash\_table[TABLE\_SIZE];

initialize the table

- allow overflows and collisions to

be detected

- all slots to empty(null) string

void init\_table(element ht[]) {

int i;

for (i = 0; i < TABLE\_SIZE; i++) {

ht[i].key[0] = NULL;

}

initialization of a hash table

to insert an element, transform a key

into a number and calculate hash

address

int transform(char \*key) {

/\* simple additive approach to create a natural

number that is within the integer range \*/

int number = 0;

while (\*key)

number += \*key++;

return number;

}

int hash(char \*key) {

/\* calculate hash address \*/

return(transform(key) % TABLE\_SIZE);

}

creation of a hash function

insert element into the hash table

- find another bucket if the new

element is hashed into a full

bucket: linear probing

Example) b = 13, s = 1

[0]

**function**

[1]

[2]

**for**

[3]

**do**

[4]

**while**

[5]

[6]

[7]

[8]

[9]

**else**

[10]

[11]

[12]

**if**

hash table with linear probing

(13 buckets, 1 slot/bucket)

4 cases in insertion process

examine the hash table buckets- ht[(f(x)+j) % TABLE\_SIZE], where

0 £ j £ TABLE\_SIZE1)the bucket contains x

- simply report a duplicate identifier

- update information in the other fields of the element

2)the bucket contains the empty string

- bucket is empty, and insert the new element into it

3)the bucket contains a nonempty string other than x

- proceed to examine the next bucket

4)return to the home bucket ht[f(x)](j = TABLE\_SIZE)

- the home bucket is being examined for the second time and all remaining buckets have been examined

- report an error condition and exit

void linear\_insert(element item, element ht[]) {

/\* insert the key into the table using the linear

probing technique, exit the function if the table

is full \*/

int i, hash\_value;

hash\_value = hash(item.key);

i = hash\_value;

while (strlen(ht[i].key)) {

if (!strcmp(ht[i].key, item.key)) {

fprintf(stderr, ”duplicate entry\n”);

exit(1);

}

i = (i + 1) % TABLE\_SIZE;

if (i == hash\_value) {

fprintf(stderr,”the table is full\n”);

exit(1);

}

}

ht[i] = item;

}

linear insert into a hash table

characteristics of linear probing to

resolve overflow

- identifiers tend to cluster together

- increases the search time

Ex) enter the C built-in functions into

a 26-bucket hash table in the order

“acos, atoi, char, define, exp,

ceil, cos, float, atol, floor,

ctime”

* b = 26, s = 1



hash table with linear probing(26 buckets, 1 slot/bucket)

cluster of identifiers in linear probing

- tend to merge as more identifiers is entered into the table

- bigger cluster

solutions

- quadratic probing

- random probing

- rehashing

**2) quadratic probing**

- examine the hash table buckets

ht[f(x)],

ht[(f(x) + i2) % b],

ht[(f(x) - i2) % b],

for 0 £ i £ (b-1)/2,

where

b: number of buckets in the table

- reduce the average number of probes

**3) rehashing**

- use a series of hashing functions

f1, f2, ··· , fb

- bucket fi(x) is examined for

i = 1, 2, ··· , b

**chaning**

defect of linear probing

- comparison of identifiers with different hash values

maintain list of identifiers

- one list per one bucket

- each list has all the synonyms

- requires a head node for each chain

link

link

data(key)

list(linked list)

Bucket (head n\\noden))node)

#define MAX\_CHAR 10

#define TABLE\_SIZE 13

#define IS\_FULL(ptr) (!(ptr))

typedef struct {

char key[MAX\_CHAR];

/\* other fields \*/

} element;

typedef struct list \*list\_ptr;

typedef struct list {

element item;

list\_ptr link;

}

list\_ptr hash\_table[TABLE\_SIZE];

void chain\_insert(element item, list\_ptr ht[]) {

int hash\_value = hash(item.key);

list\_ptr ptr, trail = NULL;

list\_ptr lead = ht[hash\_value];

for (; lead; trail=lead, lead = lead->link)

if (!strcmp(lead->item.key, item.key)) {

fprintf(stderr,”the key is in the table\n”);

exit(1);

}

}

ptr = (list\_ptr)malloc(sizeof(list));

if (IS\_FULL(ptr)) {

fprintf(stderr,“the memory is full\n”);

exit(1);

}

ptr->item = item;

ptr->link = NULL;

if (trail) trail->link = ptr;

else ht[hash\_value] = ptr;

}

chain insert into a hash table

* hash chains

acos

[0]

[1]

[2]

[3]

[4]

[5]

[6]

**···**

[25]

atoi

atol

char

ceil

cos

define

exp

float

floor

ctime

List Verification

* Compare lists to verify that they are identical or identify the discrepancies.
* example
  + international revenue service (e.g., employee vs. employer)
* complexities
  + random order: O(mn)
  + ordered list:   
    O(tsort(n)+tsort(m)+m+n)

**\*Program 4.3:** verifying using a sequential search  
  
void verify1(element list1[], element list2[ ], int n, int m)  
/\* compare two unordered lists list1 and list2 \*/  
{  
int i, j;  
int marked[MAX\_SIZE];  
  
for(i = 0; i<m; i++)  
 marked[i] = FALSE;  
for (i=0; i<n; i++)  
 if ((j = seqsearch(list2, m, list1[i].key)) < 0)  
 printf(“%d is not in list 2\n “, list1[i].key);  
 else  
 /\* check each of the other fields from list1[i] and list2[j], and print out any discrepancies \*/

marked[j] = TRUE;  
for ( i=0; i<m; i++)  
 if (!marked[i])  
 printf(“%d is not in list1\n”, list2[i]key);  
}

**\*Program 4.4:**Fast verification of two lists  
void verify2(element list1[ ], element list2 [ ], int n, int m)  
/\* Same task as verify1, but list1 and list2 are sorted \*/  
{  
 int i, j;  
 sort(list1, n);  
 sort(list2, m);  
 i = j = 0;  
 while (i < n && j < m)  
 if (list1[i].key < list2[j].key) {  
 printf (“%d is not in list 2 \n”, list1[i].key);  
 i++;  
 }  
 else if (list1[i].key == list2[j].key) {  
 /\* compare list1[i] and list2[j] on each of the other field   
 and report any discrepancies \*/  
 i++; j++;  
 }

else {  
 printf(“%d is not in list 1\n”, list2[j].key);  
 j++;  
 }  
for(; i < n; i++)  
 printf (“%d is not in list 2\n”, list1[i].key);  
for(; j < m; j++)  
 printf(“%d is not in list 1\n”, list2[j].key);  
}

**Sorting Problem**

Definition

* + given (R0, R1, …, Rn-1), where Ri = key + data  
    find a permutation σ, such that Kσ(i-1) ≤ Kσ(i), 0<i<n-1

sorted

* + Kσ(i-1) ≤ Kσ(i), 0<i<n-1

stable

* + if i < j and Ki = Kj then Ri precedes Rj in the sorted list

internal sort vs. external sort

criteria

* + # of key comparisons
  + # of data movements

**Insertion Sort**

void insertion\_sort(element list[], int n)

{

int i, j;

element next;

for (i=1; i<n; i++) {

next= list[i];

for (j=i-1; j>=0&&next.key<list[j].key;  
 j--)

list[j+1] = list[j];

list[j+1] = next;

}

}

O(n)



**worse case**

i 0 1 2 3 4

- 5 4 3 2 1

1 4 5 3 2 1

2 3 4 5 2 1

3 2 3 4 5 1

4 1 2 3 4 5

**best case**

i 0 1 2 3 4

- 2 3 4 5 1

1 2 3 4 5 1

2 2 3 4 5 1

3 2 3 4 5 1

4 1 2 3 4 5

left out of order (LOO)

Ri is LOO if Ri < max{Rj}

0≤j<i

k: # of records LOO

Computing time: O((k+1)n)

44 55 12 42 94 18 06 67

\*

\*

\*

\*

\*

**Radix Sort**

Sort by keys

K0, K1, …, Kr-1

Most significant key

Least significant key

R0, R1, …, Rn-1 are said to be sorted w.r.t. K0, K1, …, Kr-1 iff

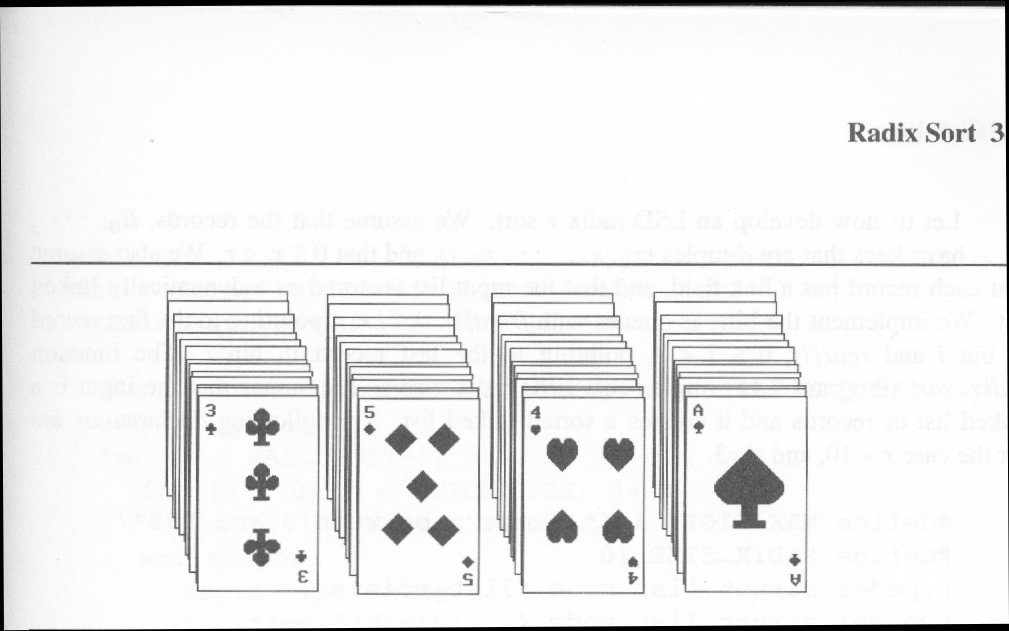


0≤i<n-1

Most significant digit first: sort on K0, then K1….

Least significant digit first: sort on Kr-1, then Kr-2…

**Figure 4.5:** Arrangement of cards after first pass of an MSD sort



Suits: ♣ < ♦ < ♥ < ♠

Face values: 2 < 3 < 4 < … < J < Q < K < A

(1) MSD sort first, e.g., bin sort, four bins ♣ ♦ ♥ ♠

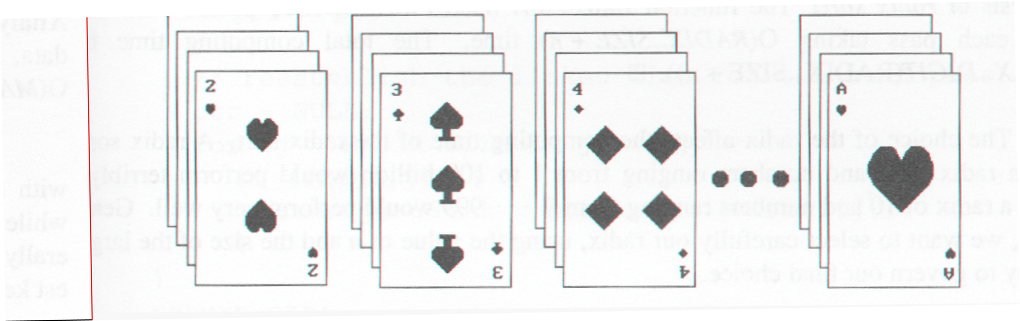
LSD sort second, e.g., insertion sort

(2) LSD sort first, e.g., bin sort, 13 bins

2, 3, 4, …, 10, J, Q, K, A

MSD sort, e.g., bin sort four bins ♣ ♦ ♥ ♠

**Figure :** Arrangement of cards after first pass of LSD sort



**RADIX SORT**

0 ≤ K ≤ 999

(K0, K1, K2)

MSD

LSD

0-9

0-9

0-9

radix 10 sort

radix 2 sort

Example for LSD Radix Sort

**271, 93, 33, 984, 55, 306, 208, 179, 859, 9 After the first pass**

306

208

9

null

null

null

33

null

null

55

859

null

null

271

179

null

984

null

93

null

rear[0]

rear[1]

rear[2]

rear[3]

rear[4]

rear[5]

rear[6]

rear[7]

rear[8]

rear[9]

front[0]

front[1]

front[2]

front[3]

front[4]

front[5]

front[6]

front[7]

front[8]

front[9]

**306, 208, 9, 33, 55, 859, 271, 179, 984, 93** (second pass)

306

null

null

null

859

null

984

null

rear[0]

rear[1]

rear[2]

rear[3]

rear[4]

rear[5]

rear[6]

rear[7]

rear[8]

rear[9]

front[1]

front[2]

front[3]

front[4]

front[5]

front[6]

front[7]

front[8]

front[9]

179

null

208

271

null

null

null

**9, 33, 55, 93, 179, 208, 271, 306, 859, 984** (third pass)

**Data Structures for LSD Radix Sort**

An LSD radix *r* sort,

*R*0, *R*1, ..., *Rn*-1 have the keys that are *d*-tuples  
(*x*0, *x*1, ..., *xd*-1)

#define MAX\_DIGIT 3

#define RADIX\_SIZE 10

typedef struct list\_node \*list\_pointer;

typedef struct list\_node {

int key[MAX\_DIGIT];

list\_pointer link;

}

LSD Radix Sort

list\_pointer radix\_sort(list\_pointer ptr)

{

list\_pointer front[RADIX\_SIZE],  
 rear[RADIX\_SIZE];

int i, j, digit;

for (i=MAX\_DIGIT-1; i>=0; i--) {

for (j=0; j<RADIX\_SIZE; j++)

front[j]=read[j]=NULL;

while (ptr) {

digit=ptr->key[I];

if (!front[digit]) front[digit]=ptr;

else rear[digit]->link=ptr;

rear[digit]=ptr;

ptr=ptr->link;

}

/\* reestablish the linked list for the next pass \*/

ptr= NULL;

for (j=RADIX\_SIZE-1; j>=0; j++)

if (front[j]) {

rear[j]->link=ptr;

ptr=front[j];

}

}

return ptr;

}

Comparison

n < 20: insertion sort

20 ≤ n < 45: quick sort

n ≥ 45: merge sort

hybrid method: merge sort + quick sort

merge sort + insertion sort

**Quick Sort (C.A.R. Hoare)**

Given (R0, R1, …, Rn-1)  
 Ki: pivot key  
if Ki is placed in S(i),  
then Kj ≤ Ks(i) for j < S(i),  
 Kj ≥ Ks(i) for j > S(i).

R0, …, RS(i)-1, RS(i), RS(i)+1, …, RS(n-1)

Example for Quick Sort



void quicksort(element list[], int left, int right)

{

int pivot, i, j;

element temp;

if (left < right) {

i = left; j = right+1;

pivot = list[left].key;

do {

do i++; while (list[i].key < pivot);

do j--; while (list[j].key > pivot);

if (i < j) SWAP(list[i], list[j], temp);

} while (i < j);

SWAP(list[left], list[j], temp);

quicksort(list, left, j-1);

quicksort(list, j+1, right);

}

}

**Analysis for Quick Sort**

Assume that each time a record is positioned, the list is divided into the rough same size of two parts.

Position a list with *n* element needs O(*n*)

*T*(*n*) is the time taken to sort *n* elements

*T*(*n*)<=*cn*+2*T*(*n*/2) for some *c*  
 <=*cn*+2(*cn*/2+2*T*(*n*/4))  
 ...  
 <=*cn*log *n*+*nT*(1)=O(*n*log *n)*

Time and Space for Quick Sort

Space complexity:

Average case and best case: O(log *n*)

Worst case: O(*n*)

Time complexity:

Average case and best case: O(*n* log *n*)

Worst case: O(*n*  )

**Heap Sort**

**Figure 4.2:** Array interpreted as a binary tree

1 2 3 4 5 6 7 8 9 10

26 5 77 1 61 11 59 15 48 19

[1]

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]

[10]

**\*Figure 4.3:** Max heap following first **for** loop of *heapsort*

[1]

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]

[10]

initial heap

**Figure 4.4:** Heap sort example

[1]

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]

[10]

[1]

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]

[10]

(a)

[1]

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]

[10]

[1]

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]

[10]

(c)

(d)

**Heap Sort**

void adjust(element list[], int root, int n)

{

int child, rootkey; element temp;

temp=list[root]; rootkey=list[root].key;

child=2\*root;

while (child <= n) {

if ((child < n) &&

(list[child].key < list[child+1].key))

child++;

if (rootkey > list[child].key) break;

else {

list[child/2] = list[child];

child \*= 2;

}

}

list[child/2] = temp;

} void heapsort(element list[], int n)

{

int i, j;

element temp;

for (i=n/2; i>0; i--) adjust(list, i, n);

for (i=n-1; i>0; i--) {

SWAP(list[1], list[i+1], temp);

adjust(list, 1, i);

}

}

**UNITY –V**

# B-Trees: Balanced Tree Data Structures

## Introduction

Tree structures support various basic dynamic set operations including *Search*, *Predecessor*, *Successor*, *Minimum*, *Maximum*, *Insert*, and *Delete* in time proportional to the height of the tree. Ideally, a tree will be balanced and the height will be *log n* where *n* is the number of nodes in the tree. To ensure that the height of the tree is as small as possible and therefore provide the best running time, a balanced tree structure like a red-black tree, AVL tree, or b-tree must be used.

When working with large sets of data, it is often not possible or desirable to maintain the entire structure in primary storage (RAM). Instead, a relatively small portion of the data structure is maintained in primary storage, and additional data is read from secondary storage as needed. Unfortunately, a magnetic disk, the most common form of secondary storage, is significantly slower than random access memory (RAM). In fact, the system often spends more time retrieving data than actually processing data.

B-trees are balanced trees that are optimized for situations when part or all of the tree must be maintained in secondary storage such as a magnetic disk. Since disk accesses are expensive (time consuming) operations, a b-tree tries to minimize the number of disk accesses. For example, a b-tree with a height of 2 and a branching factor of 1001 can store over one billion keys but requires at most two disk accesses to search for any node (Cormen 384).

## The Structure of B-Trees

Unlike a binary-tree, each node of a b-tree may have a variable number of keys and children. The keys are stored in non-decreasing order. Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key but greater than the preceeding key. A node also has an additional rightmost child that is the root for a subtree containing all keys greater than any keys in the node.

A b-tree has a minumum number of allowable children for each node known as the *minimization factor*. If *t* is this *minimization factor*, every node must have at least *t - 1* keys. Under certain circumstances, the root node is allowed to violate this property by having fewer than *t - 1* keys. Every node may have at most *2t - 1* keys or, equivalently, *2t* children.

Since each node tends to have a large branching factor (a large number of children), it is typically neccessary to traverse relatively few nodes before locating the desired key. If access to each node requires a disk access, then a b-tree will minimize the number of disk accesses required. The minimzation factor is usually chosen so that the total size of each node corresponds to a multiple of the block size of the underlying storage device. This choice simplifies and optimizes disk access. Consequently, a b-tree is an ideal data structure for situations where all data cannot reside in primary storage and accesses to secondary storage are comparatively expensive (or time consuming).

### Height of B-Trees

For *n* greater than or equal to one, the height of an *n*-key b-tree T of height *h* with a minimum degree *t* greater than or equal to 2,

height

The worst case height is *O(log n)*. Since the "branchiness" of a b-tree can be large compared to many other balanced tree structures, the base of the logarithm tends to be large; therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures. Although this does not affect the asymptotic worst case height, b-trees tend to have smaller heights than other trees with the same asymptotic height.

## Operations on B-Trees

The algorithms for the *search*, *create*, and *insert* operations are shown below. Note that these algorithms are single pass; in other words, they do not traverse back up the tree. Since b-trees strive to minimize disk accesses and the nodes are usually stored on disk, this single-pass approach will reduce the number of node visits and thus the number of disk accesses. Simpler double-pass approaches that move back up the tree to fix violations are possible.

Since all nodes are assumed to be stored in secondary storage (disk) rather than primary storage (memory), all references to a given node be be preceeded by a read operation denoted by *Disk-Read*. Similarly, once a node is modified and it is no longer needed, it must be written out to secondary storage with a write operation denoted by *Disk-Write*. The algorithms below assume that all nodes referenced in parameters have already had a corresponding *Disk-Read* operation. New nodes are created and assigned storage with the *Allocate-Node* call. The implementation details of the *Disk-Read*, *Disk-Write*, and *Allocate-Node* functions are operating system and implementation dependent.

### B-Tree-Search(x, k)

i <- 1

while i <= n[x] and k > keyi[x]

do i <- i + 1

if i <= n[x] and k = keyi[x]

then return (x, i)

if leaf[x]

then return NIL

else Disk-Read(ci[x])

return B-Tree-Search(ci[x], k)

The search operation on a b-tree is analogous to a search on a binary tree. Instead of choosing between a left and a right child as in a binary tree, a b-tree search must make an n-way choice. The correct child is chosen by performing a linear search of the values in the node. After finding the value greater than or equal to the desired value, the child pointer to the immediate left of that value is followed. If all values are less than the desired value, the rightmost child pointer is followed. Of course, the search can be terminated as soon as the desired node is found. Since the running time of the search operation depends upon the height of the tree, *B-Tree-Search* is *O(logt n)*.

### B-Tree-Create(T)

x <- Allocate-Node()

leaf[x] <- TRUE

n[x] <- 0

Disk-Write(x)

root[T] <- x

The *B-Tree-Create* operation creates an empty b-tree by allocating a new root node that has no keys and is a leaf node. Only the root node is permitted to have these properties; all other nodes must meet the criteria outlined previously. The *B-Tree-Create* operation runs in time *O(1)*.

### B-Tree-Split-Child(x, i, y)

z <- Allocate-Node()

leaf[z] <- leaf[y]

n[z] <- t - 1

for j <- 1 to t - 1

do keyj[z] <- keyj+t[y]

if not leaf[y]

then for j <- 1 to t

do cj[z] <- cj+t[y]

n[y] <- t - 1

for j <- n[x] + 1 downto i + 1

do cj+1[x] <- cj[x]

ci+1 <- z

for j <- n[x] downto i

do keyj+1[x] <- keyj[x]

keyi[x] <- keyt[y]

n[x] <- n[x] + 1

Disk-Write(y)

Disk-Write(z)

Disk-Write(x)

If is node becomes "too full," it is necessary to perform a split operation. The split operation moves the median key of node *x* into its parent *y* where *x* is the *ith* child of *y*. A new node, *z*, is allocated, and all keys in *x* right of the median key are moved to *z*. The keys left of the median key remain in the original node *x*. The new node, *z*, becomes the child immediately to the right of the median key that was moved to the parent *y*, and the original node, *x*, becomes the child immediately to the left of the median key that was moved into the parent *y*.

The split operation transforms a full node with *2t - 1* keys into two nodes with *t - 1* keys each. Note that one key is moved into the parent node. The *B-Tree-Split-Child* algorithm will run in time *O(t)* where *t* is constant.

### B-Tree-Insert(T, k)

r <- root[T]

if n[r] = 2t - 1

then s <- Allocate-Node()

root[T] <- s

leaf[s] <- FALSE

n[s] <- 0

c1 <- r

B-Tree-Split-Child(s, 1, r)

B-Tree-Insert-Nonfull(s, k)

else B-Tree-Insert-Nonfull(r, k)

### B-Tree-Insert-Nonfull(x, k)

i <- n[x]

if leaf[x]

then while i >= 1 and k < keyi[x]

do keyi+1[x] <- keyi[x]

i <- i - 1

keyi+1[x] <- k

n[x] <- n[x] + 1

Disk-Write(x)

else while i >= and k < keyi[x]

do i <- i - 1

i <- i + 1

Disk-Read(ci[x])

if n[ci[x]] = 2t - 1

then B-Tree-Split-Child(x, i, ci[x])

if k > keyi[x]

then i <- i + 1

B-Tree-Insert-Nonfull(ci[x], k)

To perform an insertion on a b-tree, the appropriate node for the key must be located using an algorithm similiar to *B-Tree-Search*. Next, the key must be inserted into the node. If the node is not full prior to the insertion, no special action is required; however, if the node is full, the node must be split to make room for the new key. Since splitting the node results in moving one key to the parent node, the parent node must not be full or another split operation is required. This process may repeat all the way up to the root and may require splitting the root node. This approach requires two passes. The first pass locates the node where the key should be inserted; the second pass performs any required splits on the ancestor nodes.

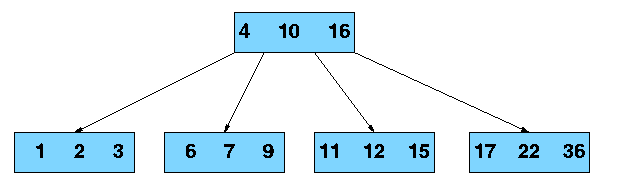
Since each access to a node may correspond to a costly disk access, it is desirable to avoid the second pass by ensuring that the parent node is never full. To accomplish this, the presented algorithm splits any full nodes encountered while descending the tree. Although this approach may result in unecessary split operations, it guarantees that the parent never needs to be split and eliminates the need for a second pass up the tree. Since a split runs in linear time, it has little effect on the *O(t logt n)* running time of *B-Tree-Insert*.

Splitting the root node is handled as a special case since a new root must be created to contain the median key of the old root. Observe that a b-tree will grow from the top.

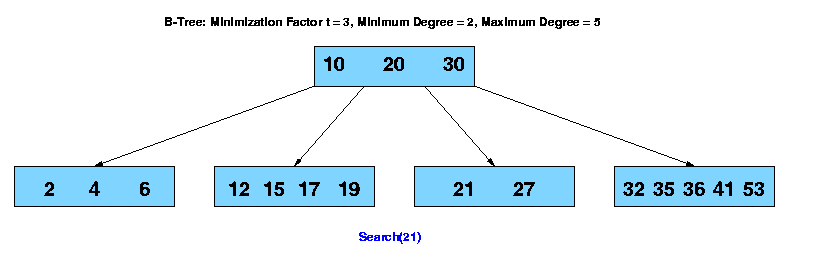
### B-Tree-Delete

Deletion of a key from a b-tree is possible; however, special care must be taken to ensure that the properties of a b-tree are maintained. Several cases must be considered. If the deletion reduces the number of keys in a node below the minimum degree of the tree, this violation must be corrected by combining several nodes and possibly reducing the height of the tree. If the key has children, the children must be rearranged.

**Examples Sample B-Tree**



### Searching a B-Tree for Key 21



**AVL TREES**

**Binary search tree**

time complexity

- average case: O(log2n)

- worst case: O(n)

maintain the binary search tree as a complete binary tree

- minimize the average and maximum search time

- average and worst case: O(log2n)

- a significant increase in the time required to add new element

Jan

Mar

Feb

Apr

May

June

Sept

Oct

Nov

July

Aug

Dec

binary search tree obtained for the months of the year

Jan

Mar

Feb

Apr

May

June

Sept

Oct

Nov

July

Aug

Dec

a balanced tree for the months of the year

Jan

Mar

Feb

Apr

May

June

Sept

Oct

Nov

July

Aug

Dec

degenerate binary search tree

**AVL Trees**

- balanced binary trees

- average and worst case: O(log2n)

Def) *height balanced binary tree*

- an empty binary tree is height balanced

- if T is a nonempty binary tree with TL and TR as its left and right subtrees

- T is height balanced iff

1) TL and TR are height balanced, and

2) |hL - hR| £ 1 where hL and hR are height of TL and TR, respectively

Def) *balance factor*, BF(T), of node T

in a binary tree

- hL - hR where hL and hR are heights of left and right subtree of T

- for any node T in an AVL tree, BF(T) = -1, 0, or 1

**Mar**

0

Mar

-1

**May**

0

**Nov**

0

**May**

-1

**Mar**

-2

**Nov**

0

**May**

0

**Mar**

0

(a) insert March

(b) insert May

(c) insert November

RR

rotation

**Insertion**

**Example**

May

+1

Mar

+1

Nov

0

**Aug**

0

(d) insert August

LL

rotation

**Apr**

0

**Aug**

+1

**Mar**

+2

May

+2

Nov

0

May

+1

Nov

0

**Aug**

0

**Mar**

0

**Apr**

0

(e) insert April

**May**

+2

**Aug**

-1

May

0

**Mar**

+1

Apr

0

**Jan**

0

**Mar**

0

**Aug**

0

**May**

-1

**Jan**

0

Apr

0

Nov

0

Mar

+1

May

-1

Aug

-1

Apr

0

Jan

+1

Nov

0

**Dec**

0

(f) insert January

LR

rotation

Dec

0

**July**

0

Mar

+1

May

-1

Aug

-1

Apr

0

Jan

0

Nov

0

(h) insert July

Mar

+1

May

-1

Apr

0

Nov

0

**Dec**

0

**Jan**

0

July

0

**Aug**

+1

**Feb**

0

RL

rotation

(i) insert February

four kinds of rotations to rebalance

- LL, LR, RR, RL

- LL and RR are symmetric

- LR and RL are symmetric

Let Y: new inserted node, and

A: the nearest ancestor of Y, whose balance factor becomes ±2

**LL**: Y is inserted in the left subtree of the left subtree of A

**LR**: Y is inserted in the right subtree of the left subtree of A

**RR**: Y is inserted in the right subtree

of the right subtree of A**RL**: Y is inserted in the left subtree of the right subtree of A

- height of the subtrees which are not involved in the rotation remain unchanged

**Red-Black Trees**

* Balanced” binary search trees guarantee an O(lgn) running time
* Red-black-tree
  + Binary search tree with an additional attribute for its nodes: color which can be **red** or **black**
  + Constrains the way nodes can be colored on any path from the root to a leaf:

Ensures that no path is more than twice as long as any other path ⇒ the tree is balanced

* For convenience we use a sentinel NIL[T] to represent all the NIL nodes at the leafs
  + NIL[T] has the same fields as an ordinary node
  + Color[NIL[T]] = BLACK
  + The other fields may be set to arbitrary values

Red-Black-Trees Properties

1. Every node is either **red** or **black**
2. The root is **black**
3. Every leaf (NIL) is **black**
4. If a node is **red**, then both its children are **black**
   * No two consecutive red nodes on a simple path from the root to a leaf
5. For each node, all paths from that node to descendant leaves contain the same number of **black** nodes

Black-Height of a Node

* **Height of a node:** the number of edges in the **longest** path to a leaf
* **Black-height** of a node x: bh(x) is the number of black nodes (including NIL) on the path from x to a leaf,

not counting x

Overview: Most important property of   
Red-Black-Trees

A red-black tree with n internal nodes

has height at most 2lg(n + 1)

Need to prove two claims first …Any node x with height h(x) has bh(x) ≥ h(x)/2

Proof

By property 4, at most h/2 red nodes on the path from the node to a leaf

Hence at least h/2 are black

**SPLAY TREES**

* Splay trees are tree structures that:
  + Are not perfectly balanced all the time
  + Data most recently accessed is near the root. (principle of locality; 80-20 “rule”)
* The procedure:
  + After node X is accessed, perform “splaying” operations to bring X to the root of the tree.
  + Do this in a way that leaves the tree more balanced as a whole
* Let X be a non-root node with ≥ 2 ancestors.
  + P is its parent node.
  + G is its grandparent node.

Zig-Zig and Zig-Zag

**Splay Tree Operations**

In this unit pattern matching, is the act of checking some sequence of tokens for the presence of the constituents of some [pattern](http://en.wikipedia.org/wiki/Pattern) .Uses of pattern matching include outputting the locations of a pattern within a token sequence, to output some component of the matched pattern, and to substitute the matching pattern with some other token sequence (I.e., search and replace).

**Contents:**

1. Pattern matching algorithms

2. Standard Tries, Compressed Tries, Suffix tries.

# Brute Force algorithm

**Main features**

* no preprocessing phase;
* constant extra space needed;
* always shifts the window by exactly 1 position to the right;
* comparisons can be done in any order;
* searching phase in ***O***(*mn*) time complexity;
* 2*n* expected text characters comparisons.

**Description**

The brute force algorithm consists in checking, at all positions in the text between 0 and *n*-*m*, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right.

The brute force algorithm requires no preprocessing phase, and a constant extra space in addition to the pattern and the text. During the searching phase the text character comparisons can be done in any order. The time complexity of this searching phase is ***O***(*mn*) (when searching for a*m*-1b in a*n* for instance). The expected number of text character comparisons is 2*n*.

# Boyer-Moore algorithm

**Main features**

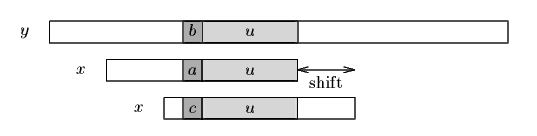
* performs the comparisons from right to left;
* preprocessing phase in ***O***(*m*+sigma) time and space complexity;
* searching phase in ***O***(*mn*) time complexity;
* 3*n* text character comparisons in the worst case when searching for a non periodic pattern;
* ***O***(*n* / *m*) best performance.

**Description**

The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of it or the entire algorithm is often implemented in text editors for the «search» and «substitute» commands.

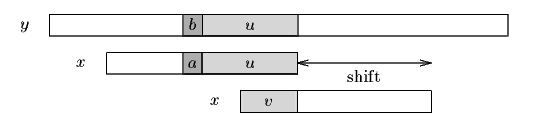
The algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two shift functions are called the **good-suffix shift** (also called matching shift and the **bad-character shift** (also called the occurrence shift).

Assume that a mismatch occurs between the character *x*[*i*]=*a* of the pattern and the character *y*[*i*+*j*]=*b* of the text during an attempt at position *j*.  
Then, *x*[*i*+1 .. *m*-1]=*y*[*i*+*j*+1 .. *j*+*m*-1]=u and *x*[*i*] neq y[*i*+*j*]. The good-suffix shift consists in aligning the segment *y*[*i*+*j*+1 .. *j*+*m*-1]=*x*[*i*+1 .. *m*-1] with its rightmost occurrence in *x* that is preceded by a character different from *x*[*i*] (*see figure* ).



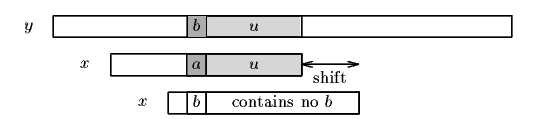
**Figure** . The good-suffix shift, *u* re-occurs preceded by a character *c* different from *a*.

If there exists no such segment, the shift consists in aligning the longest suffix *v* of *y*[*i*+*j*+1 .. *j*+*m*-1] with a matching prefix of *x* (*see figure*).



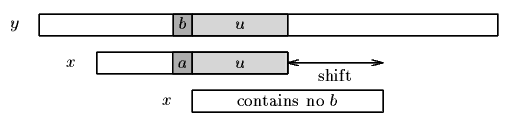
**Figure** . The good-suffix shift, only a suffix of *u* re-occurs in *x*.

The bad-character shift consists in aligning the text character *y*[*i*+*j*] with its rightmost occurrence in *x*[0 .. *m*-2]. (*see figure*)



**Figure** . The bad-character shift, *a* occurs in *x*.

If *y*[*i*+*j*] does not occur in the pattern *x*, no occurrence of *x* in *y* can include *y*[*i*+*j*], and the left end of the window is aligned with the character immediately after *y*[*i*+*j*], namely *y*[*i*+*j*+1] (*see figure*).



**Figure** . The bad-character shift, *b* does not occur in *x*.

Note that the bad-character shift can be negative, thus for shifting the window, the Boyer-Moore algorithm applies the maximum between the the good-suffix shift and bad-character shift. More formally the two shift functions are defined as follows.

The good-suffix shift function is stored in a table bmGs of size *m*+1.

Let us define two conditions:

|  |  |
| --- | --- |
| hand | Cs(*i*, *s*): for each *k* such that *i* < *k* < *m*, *s* geq *k* or *x*[*k*-*s*]=*x*[*k*] and |

|  |  |
| --- | --- |
| hand | Co(*i*, *s*): if *s* <*i* then *x*[*i*-*s*] neq *x*[*i*] |

Then, for 0 leq *i* < *m*: bmGs[*i*+1]=min{s>0 : Cs(*i*, *s*) and Co(*i*, *s*) hold}  
and we define bmGs[0] as the length of the period of *x*. The computation of the table bmGs use a table suff defined as follows: for 1 leq *i* < *m*, suff[*i*]=max{*k* : x[*i*-*k*+1 .. *i*]=x[*m*-*k* .. *m*-1]}

The bad-character shift function is stored in a table bmBc of size sigma. For *c* in Sigma: bmBc[*c*] = min{*i* : 1 leq *i* <*m*-1 and *x*[*m*-1-*i*]=*c*} if *c* occurs in *x*, *m* otherwise.

Tables bmBc and bmGs can be precomputed in time ***O***(*m*+sigma) before the searching phase and require an extra-space in ***O***(*m*+sigma). The searching phase time complexity is quadratic but at most 3*n* text character comparisons are performed when searching for a non periodic pattern. On large alphabets (relatively to the length of the pattern) the algorithm is extremely fast. When searching for a*m*-1b in b*n* the algorithm makes only ***O***(*n* / *m*) comparisons, which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed.

# Knuth-Morris-Pratt string matching

The problem: given a (short) pattern and a (long) text, both strings, determine whether the pattern appears somewhere in the text. [Last time](http://www.ics.uci.edu/~eppstein/161/960222.html) we saw how to do this with finite automata. This time we'll go through the [Knuth](http://www.ics.uci.edu/~eppstein/161/people.html#knuth)-[Morris](http://www.ics.uci.edu/~eppstein/161/people.html#morris)-[Pratt](http://www.ics.uci.edu/~eppstein/161/people.html#pratt) (KMP) algorithm, which can be thought of as an efficient way to build these automata. I also have some [working C++ source code](http://www.ics.uci.edu/~eppstein/161/kmp/) which might help you understand the algorithm better.

First let's look at a naive solution.  
suppose the text is in an array: char T[n]  
and the pattern is in another array: char P[m].

One simple method is just to try each possible position the pattern could appear in the text.

**Naive string matching**:

for (i=0; T[i] != '\0'; i++)

{

for (j=0; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++) ;

if (P[j] == '\0') found a match

}

There are two nested loops; the inner one takes O(m) iterations and the outer one takes O(n) iterations so the total time is the product, O(mn). This is slow; we'd like to speed it up.

In practice this works pretty well -- not usually as bad as this O(mn) worst case analysis. This is because the inner loop usually finds a mismatch quickly and move on to the next position without going through all m steps. But this method still can take O(mn) for some inputs. In one bad example, all characters in T[] are "a"s, and P[] is all "a"'s except for one "b" at the end. Then it takes m comparisons each time to discover that you don't have a match, so mn overall.

Here's a more typical example. Each row represents an iteration of the outer loop, with each character in the row representing the result of a comparison (X if the comparison was unequal). Suppose we're looking for pattern "nano" in text "banananobano".

0 1 2 3 4 5 6 7 8 9 10 11

T: b a n a n a n o b a n o

i=0: X

i=1: X

i=2: n a n X

i=3: X

i=4: n a n o

i=5: X

i=6: n X

i=7: X

i=8: X

i=9: n X

i=10: X

Some of these comparisons are wasted work! For instance, after iteration i=2, we know from the comparisons we've done that T[3]="a", so there is no point comparing it to "n" in iteration i=3. And we also know that T[4]="n", so there is no point making the same comparison in iteration i=4.

## Skipping outer iterations

The Knuth-Morris-Pratt idea is, in this sort of situation, after you've invested a lot of work making comparisons in the inner loop of the code, you know a lot about what's in the text. Specifically, if you've found a partial match of j characters starting at position i, you know what's in positions T[i]...T[i+j-1].

You can use this knowledge to save work in two ways. First, you can skip some iterations for which no match is possible. Try overlapping the partial match you've found with the new match you want to find:

i=2: n a n

i=3: n a n o

Here the two placements of the pattern conflict with each other -- we know from the i=2 iteration that T[3] and T[4] are "a" and "n", so they can't be the "n" and "a" that the i=3 iteration is looking for. We can keep skipping positions until we find one that doesn't conflict:

i=2: n a n

i=4: n a n o

Here the two "n"'s coincide. Define the *overlap* of two strings x and y to be the longest word that's a suffix of x and a prefix of y. Here the overlap of "nan" and "nano" is just "n". (We don't allow the overlap to be all of x or y, so it's not "nan"). In general the value of i we want to skip to is the one corresponding to the largest overlap with the current partial match:

**String matching with skipped iterations**:

i=0;

while (i<n)

{

for (j=0; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++) ;

if (P[j] == '\0') found a match;

i = i + max(1, j-overlap(P[0..j-1],P[0..m]));

}

## Skipping inner iterations

The other optimization that can be done is to skip some iterations in the inner loop. Let's look at the same example, in which we skipped from i=2 to i=4:

i=2: n a n

i=4: n a n o

In this example, the "n" that overlaps has already been tested by the i=2 iteration. There's no need to test it again in the i=4 iteration. In general, if we have a nontrivial overlap with the last partial match, we can avoid testing a number of characters equal to the length of the overlap.

**Standard Tries**

•The standard trie for a set of strings S is an ordered tree such that:

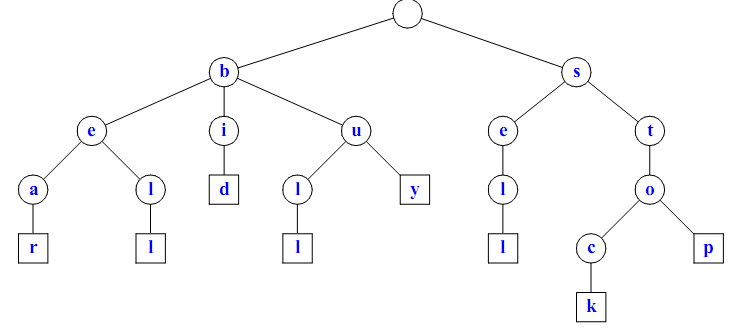
-each node but the root is labeled with a character

-the children of a node are alphabetically ordered

-the paths from the external nodes to the root yield the strings of S

Example: standard trie for the set of strings

S = { bear, bell, bid, bull, buy, sell, stock, stop }



A standard trie uses O(n) space. Operations (ﬁnd, insert, remove) take time O(dm) each, where:

-n = total size of the strings in S,

-m =size of the string parameter of the operation

-d =alphabet size,

**Applications of Tries**

•A standardtrie supports the following operations on a preprocessed text in time O(m), where m = |X|

- wordmatching:ﬁnd the ﬁrst occurrence of word X in the text

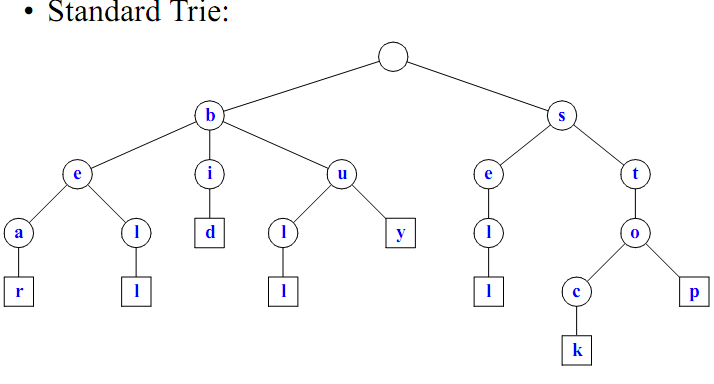
- preﬁx matching: ﬁnd the ﬁrst occurrence of the longest preﬁx of word X in the text

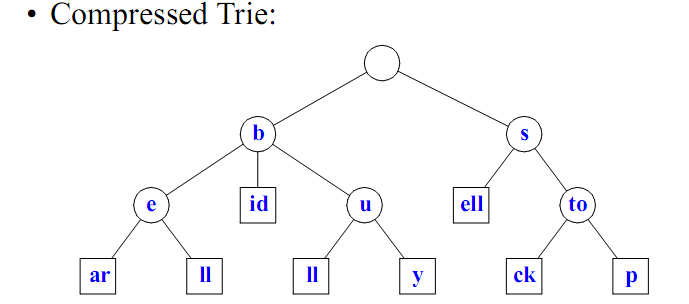
•Each operation is performed by tracing a path in the trie starting at the root

**Compressed Tries**

•Trie with nodes of degree at least 2

•Obtained from standard trie by compressing chains of redundant nodes

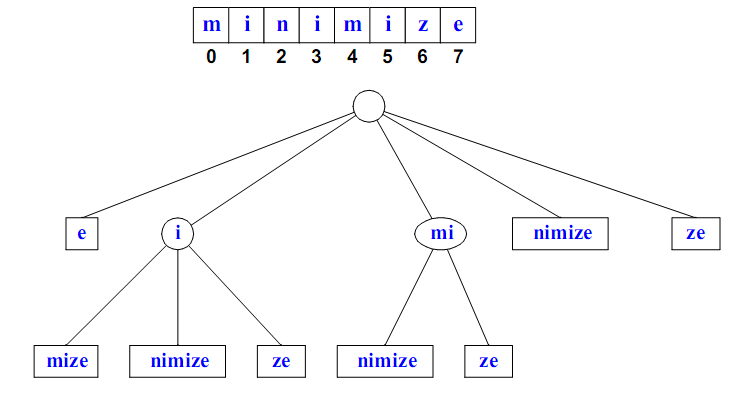


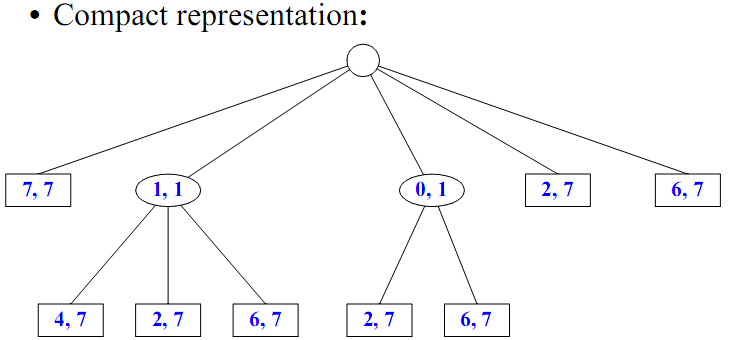


**Sufﬁx Tries**

•A sufﬁx trie is a compressed trie for all the sufﬁxes of a text

Example

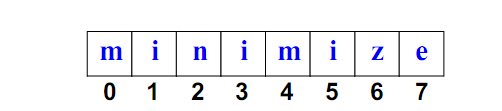


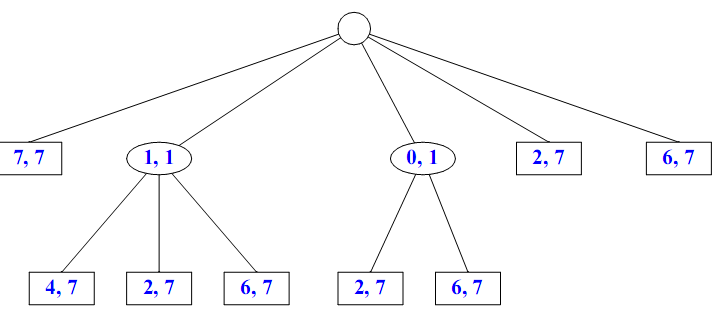


**Properties of Sufﬁx Tries**

•The sufﬁxtrie foratextXofsize n from an alphabet of size d -stores all the n(n−1)/2 sufﬁxes of X in O(n) space

-supports arbitrary patternmatching and preﬁx matching queries in O(dm) time, where m is the length of the pattern -can be constructed in O(dn) time





**15 Additional Topics**

* 1. Addition and Multiplication of Polynomials.
  2. B+ Tree.
  3. Brute –Force Approach in Pattern Matching.

**16University Question papers(attached as separate file)**

**17 Question Bank**

**Unit – 1**

**Objective Questions**

**Q.1** The complexity of multiplying two matrices of order m\*n and n\*p is

**(A)** mnp **(B)** mp

**(C)** mn **(D)** np

**Q.2** Merging 4 sorted files containing 50, 10, 25 and 15 records will take\_\_\_\_time

**(A)** O (100) **(B)** O (200)

**(C)** O (175) **(D)** O (125)

**Q.3** Consider a linked list of n elements. What is the time taken to insert an element after an element pointed by some pointer?

**(A)** O (1) **(B)** O log2 n

**(C)** O (n) **(D)** O n log2 n

**Q.4** The smallest element of an array’s index is called its

**(A)** lower bound. **(B)** upper bound.

**(C)** range. **(D)** extraction.

**Q.5** In a circular linked list

**(A)** components are all linked together in some sequential manner.

**(B)** there is no beginning and no end.

**(C)** components are arranged hierarchically.

**(D)** forward and backward traversal within the list is permitted.

**Q.6** The minimum number of multiplications and additions required to evaluate the polynomial

P = 4x3+3x2-15x+45 is

**(A)** 6 & 3 **(B)** 4 & 2

**(C)** 3 & 3 **(D)** 8 & 3

**Q.7** In a linked list with n nodes, the time taken to insert an element after an element pointed by

some pointer is

**(A)** 0 (1) **(B)** 0 (log n)

**(C)** 0 (n) **(D)** 0 (n 1og n)

**Q.8** What data structure would you mostly likely see in a nonrecursive implementation of a recursive algorithm?

**(A)** Stack **(B)** Linked list

**(C)** Queue **(D)** Trees

**Q.9** Let the following circular queue can accommodate maximum six elements with the

following data

front = 2 rear = 4

queue = \_\_\_\_\_\_\_; L, M, N, \_\_\_, \_\_\_

What will happen after ADD O operation takes place?

**(A)** front = 2 rear = 5

queue = \_\_\_\_\_\_; L, M, N, O, \_\_\_

**(B)** front = 3 rear = 5

queue = L, M, N, O, \_\_\_

**(C)** front = 3 rear = 4

queue = \_\_\_\_\_\_; L, M, N, O, \_\_\_

**(D)** front = 2 rear = 4

queue = L, M, N, O, \_\_\_

**Q.10** A linear collection of data elements where the linear node is given by means of pointer is

called

**(A)** linked list **(B)** node list

**(C)** primitive list **(D)** None of these

**Q.11** Representation of data structure in memory is known as:

**(A)** recursive **(B)** abstract data type

**(C)** storage structure **(D)** file structure

**Q.12** If the address of A[1][1] and A[2][1] are 1000 and 1010 respectively and each element occupies 2 bytes then the array has been stored in \_\_\_\_\_\_\_\_\_ order.

**(A)** row major **(B)** column major

**(C)** matix major **(D)** none of these

**Q.13** An adjacency matrix representation of a graph cannot contain information of :

**(A)** nodes **(B)** edges

**(C)** direction of edges **(D)** parallel edges

**Q.14** Time complexities of three algorithms are given. Which should execute the slowest for large values of N?

**(A)** 1 2 *O N* **(B)** *O**N*

**(C)** *O*log *N***(D)** *None of these*

**Q.15** How does an array differ from an ordinary variable?

**Q.16** Which of the following operations is performed more efficiently by doubly linked list than by singly linked list?

**(A)** Deleting a node whose location in given

**(B)** Searching of an unsorted list for a given item

**(C)** Inverting a node after the node with given location

**(D)** Traversing a list to process each node

**Q.17** The extra key inserted at the end of the array is called a,

**(A)** End key. **(B)** Stop key.

**(C)** Sentinel. **(D)** Transposition.

**Q.18** The time required to delete a node x from a doubly linked list having n nodes is

**(A)** O (n) **(B)** O (log n)

**(C)** O (1) **(D)** O (n log n)

**Part A**

**Q.1** Which sorting algorithm is easily adaptable to singly linked lists? Explain

your answer.

**Q 2.** Determine the frequency counts for all statements in the following program

segment.

for (i=1; i <= n; i ++)

for (j = 1; j <= i; j++)

for (k =1; k <= j; k++)

y ++;

**Q 3 .** Write an algorithm to count number of nodes in the circular linked list.

**Q 4.** Write an algorithm to insert a node in between any two nodes in a linked list

**Q.5** What is the difference between a grounded header link list and a circular header

link list?

**Q 6.** A linear array A is given with lower bound as 1. If address of A[25] is 375 and

A[30] is 390, then find address of A[16].

**Q7.** Write an algorithm to insert a node p at the end of a linked list.

**Q8.** Write an algorithm that counts number of nodes in a linked list.

**Q9.** Write an algorithm to add an element at the end of circular linked list.

**Q10.** Delete a given node from a doubly linked list.

**Part B**

**Q.1** Explain an efficient way of storing a sparse matrix in memory. Write a

module to find the transpose of a sparse matrix stored in this way.

**Q.2** Two linked lists contain information of the same type in ascending order.

Write a module to merge them to a single linked list that is sorted.

**Q.3** An, array, A contains n unique integers from the range x to y (x and y

inclusive where n=y-x). That is, there is one member that is not in A. Design

an O(n) time algorithm for finding that number.

**Q.4** Bubble sort algorithm is inefficient because it continues execution even after

an array is sorted by performing unnecessary comparisons. Therefore, the

number of comparisons in the best and worst cases are the same. Modify the

algorithm in such a fashion that it will not make the next pass when the array

is already sorted.

**Q.5** What do you mean by complexity of an algorithm? Explain the meaning of

worst case analysis and best case analysis with an example.

**Q.6** Explain the method to calculate the address of an element in an array. A

25\*4 matrix array DATA is stored in memory in ‘row-major order’. If base

address is 200 and 4 words per memory cell. Calculate the address of

DATA [12, 3] .

**Q.7** Write an algorithm to insert a node in the beginning of the linked list.

**Q.8** Why do we use asymptotic notation in the study of algorithm? Describe

commonly used asymptotic notations and give their significance.

**Q.9** What is a linear array? Explain how two dimensional arrays are represented in

memory.

**Q.10** Write a complete programme in C to create a single linked list. Write

functions to do the following operations

(i) Insert a new node at the end

(ii) Delete the first node

**Q.11** Define a sparse matrices. Explain the representation of a 4X4 matrix using

linked list.

**Q.12** Write a procedure to reverse a singly linked list.

**Q 13.** Define a sparse matrix. Explain different types of sparse matrices? Show how a

triangular array is stored in memory. Evaluate the method to calculate address of

any element ajk of a matrix stored in memory.

**Q 14.** Show the linked representation of the following two polynomials.



Build a procedure for adding two polynomials stored in linked lists. Verify

steps of your procedure for the above two polynomials.

**Q 15.** What is a sparse matrix? How is it stored in the memory of a computer? Write a

function to find the transpose of a sparse matrix using this representation.

**Q 16.** Write an algorithm for finding solution to the Towers of Hanoi problem. Explain

the working of your algorithm (with 4 disks) with diagrams.

**Q 17.** Suppose we have divided n elements in to m sorted lists. Explain how to

produce a single sorted list of all n elements in time O (n log m )?

**Q.18** Define the term array. How are two-dimensional arrays represented in

memory? Explain how address of an element is calculated in a two

dimensional array.

**Q.19** What is an algorithm? What are the characteristics of a good algorithm?

**Q.20** How do you find the complexity of an algorithm? What is the relation

between the time and space complexities of an algorithm? Justify your answer with an example.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

**Unit 2**

**Objective Questions**

**Q.1** The postfix form of the expression ABCD−EF / G is

**(A)** ABCDE −FG /**(B)** AB CDE −F G /

**(C)** AB CDE −F G / **(D)** AB CDE −F G /

**Q.2** A linear list of elements in which deletion can be done from one end (front) and insertion

can take place only at the other end (rear) is known as a

**(A)** queue. **(B)** stack.

**(C)** tree. **(D)** linked list.

**Q.3** What is the postfix form of the following prefix expression -A/B\*C$DE

**(A)** ABCDE$\*/- **(B)** A-BCDE$\*/-

**(C)** ABC$ED\*/- **(D)** A-BCDE$\*/

**Q.4** The data structure required to evaluate a postfix expression is

**(A)** queue **(B)** stack

**(C)** array **(D)** linked-list

**Q.5** The data structure required to check whether an expression contains balanced parenthesis is

**(A)** Stack **(B)** Queue

**(C)** Tree **(D)** Array

**Q.6** The postfix form of A\*B+C/D is

**(A)** \*AB/CD+ **(B)** AB\*CD/+

**(C)** A\*BC+/D **(D)** ABCD+/\*

**Q.7** What is the postfix form of the following prefix *\*+ab–cd*

**(A)** *ab+cd–\** **(B)** *abc+\*–*

**(C)** *ab+\*cd–* **(D)** *ab+\*cd–*

**Q.8** A stack is to be implemented using an array. The associated declarations are:

int stack [100];

int top = 0;

Give the statement to perform push operation.

**Q.9** Assume that a queue is available for pushing and popping elements. Given an input sequence a, b, c, (c be the first element), give the output sequence of elements if the rightmost element given above is the first to be popped from the queue.

**Q.10** A queue is a,

**(A)** FIFO (First In First Out) list. **(B)** LIFO (Last In First Out) list.

**(C)** Ordered array. **(D)** Linear tree.

**Q.11** Which data structure is needed to convert infix notation to postfix notation?

**(A)** Branch **(B)** Queue

**(C)** Tree **(D)** Stack

**Q.12** The prefix form of A-B/ (C \* D ^ E) is,

**(A)** -/\*^ACBDE **(B)** -ABCD\*^DE

**(C)** -A/B\*C^DE **(D)** -A/BC\*^DE

**Q.13** What is the result of the following operation

Top (Push (S, X))

**(A)** X **(B)** null

**(C)** S **(D)** None of these.

**Q.14** The prefix form of an infix expression p q −r \* t is

**(A)** pq −\*rt . **(B)** −pqr \* t .

**(C)** −pq \* rt . **(D)** −\* pqrt .

**Q.15** Which data structure is used for implementing recursion?

**(A)** Queue. **(B)** Stack.

**(C)** Arrays. **(D)** List.

**Q.16** The equivalent prefix expression for the following infix expression (A+B)-(C+D\*E)/F\*G is

**(A)** -+AB\*/+C\*DEFG **(B)** /-+AB\*+C\*DEFG

**(C)** -/+AB\*+CDE\*FG **(D)** -+AB\*/+CDE\*FG

**Q.17** The equivalent prefix expression for the following infix expression (A+B)-(C+D\*E)/F\*G is

**(A)** -+AB\*/+C\*DEFG **(B)** /-+AB\*+C\*DEFG

**(C)** -/+AB\*+CDE\*FG **(D)** -+AB\*/+CDE\*FG

**Part A**

**Q1.** Write down any four application of a stack.

**Q2.** Convert the following infix expression into a postfix expression (Show steps)

(i)ABD/ E −FG H/ k

(ii) A B D/E −FG

(iii) a bc d/e f g .

**Q.3** What are stacks? How can stacks be used to check whether an expression is

correctly parenthized or not. For eg(()) is well formed but (() or )()( is not.

**Q4.** Convert the following Infix expression to Postfix form using a stack:

*x* + *y* \* *z* + (*p* \* *q* + *r* ) \* *s*, Follow usual precedence rule and assume that the

expression is legal.

**Q5.** Define a stack. Describe ways to implement stack.

**Q6.** Can a Queue be represented by circular linked list with only one pointer

pointing to the tail of the queue? Substantiate your answer using an example.

**Q7.** Convert the following infix expressions to postfix notation

(i) A+((B+C)\*(D+E)+F/G)

(ii) A B CD

**Q8.** Suggest a way of implementing two stacks in one array such that as long as

space is there in an array, you should be able to add an element in either stack.

Using proposed method, write algorithms for push and pop operations for both

the stacks.

**Q9.** Write down any four applications of queues.

**Part B**

**Q.1** Reverse the order of elements on a stack S

(i) using two additional stacks.

(ii) using one additional queue.

**Q.2** Write an algorithm to evaluate a postfix expression. Execute your algorithm

using the following postfix expression as your input : a b + c d +\*f .

**Q.3** What are circular queues? Write down routines for inserting and deleting

elements from a circular queue implemented using arrays.

**Q.4** Implement a Queue using a singly linked list L. The operations INSERT and

DELETE should still take O (1) time.

**Q.5** Explain how to implement two stacks in one array A[1..n] in such a way that

neither stack overflows unless the total number of elements in both stacks

together is n. The PUSH and POP operations should run in O(1) time.

**Q.6** Let P be a pointer to a singly linked list. Show how this list may be used as a

stack. That is, write algorithms to push and pop elements. Specify the value of P

when the stack is empty.

**Q.7** Execute your algorithm to convert an infix expression to a post fix expression

with the following infix expression on your input

mn\*k p/g / ba b / c

**Q.8** A double ended queue is a linear list where additions and deletions can be

performed at either end. Represent a double ended queue using an array to store

elements and write modules for additions and deletions.

**Q.9** Devise a representation for a list where insertions and deletions can be made at

either end. Such a structure is called Deque (Double ended queue). Write

functions for inserting and deleting at either end.

**Q.10** Execute your algorithm to convert an infix expression to a post fix expression

with the following infix expression as input

Q A B/C DE / FG H/ I

**Q11.** Using array to implement the queue structure, write an algorithm/program to

(i) Insert an element in the queue.

(ii) Delete an element from the queue.

**Q12.** Write an algorithm to evaluate an expression given in postfix notation. Show the

execution of your algorithm for the following expression.

AB^CD-EF/GH+/+\*

**Q13.** Write an algorithm to convert an infix expression into postfix expression.

**Q14.** Using stacks, write an algorithm to determine whether the infix expression has

balanced parenthesis or not.

**Q15.** Implement a stack using linked list. Show both the PUSH and POP operations.

**Unit 3**

Q1 Let A be an adjacency matrix of a graph G. The th ij entry in the matrix K A , gives

**(A)** The number of paths of length K from vertex Vi to vertex Vj.

**(B)** Shortest path of K edges from vertex Vi to vertex Vj.

**(C)** Length of a Eulerian path from vertex Vi to vertex Vj.

**(D)** Length of a Hamiltonian cycle from vertex Vi to vertex Vj.

**Q.2** If a node having two children is deleted from a binary tree, it is replaced by its

**(A)** Inorder predecessor **(B)** Inorder successor

**(C)** Preorder predecessor **(D)** None of the above

**Q.3** For an undirected graph with n vertices and e edges, the sum of the degree of each vertex is equal to

**(A)** 2n **(B)** (2n-1)/2

**(C)** 2e **(D)** e2/2

**Q.4** A full binary tree with 2n+1 nodes contain

**(A)** n leaf nodes **(B)** n non-leaf nodes

**(C)** n-1 leaf nodes **(D)** n-1 non-leaf nodes

**Q.5** A full binary tree with n leaves contains

**(A)** n nodes. **(B)** log n 2 nodes.

**(C)** 2n –1 nodes. **(D)** n 2 nodes.

**Q.6** An undirected graph G with n vertices and e edges is represented by adjacency list. What is the time required to generate all the connected components?

**(A)** O (n) **(B)** O (e)

**(C)** O (e+n) **(D)** O 2 e

**Q.7** A graph with n vertices will definitely have a parallel edge or self loop of the total number of edges are

**(A)** more than n **(B)** more than n+1

**(C)** more than (n+1)/2 **(D)** more than n(n-1)/2

**Q.8** The maximum degree of any vertex in a simple graph with *n* vertices is

**(A)** *n–1* **(B)** *n+1*

**(C)** *2n–1* **(D)** *n*

**Q.9** The data structure required for Breadth First Traversal on a graph is

**(A)** queue **(B)** stack

**(C)** array **(D)** tree

**Q.10** The number of different directed trees with 3 nodes are

**(A)** 2 **(B)** 3

**(C)** 4 **(D)** 5

**Q.11** One can convert a binary tree into its mirror image by traversing it in

**(A)** inorder **(B)** preorder

**(C)** postorder **(D)** any order

**Q.12** One can convert a binary tree into its mirror image by traversing it in

**(A)** inorder **(B)** preorder

**(C)** postorder **(D)** any order

**Q.13** The number of leaf nodes in a complete binary tree of depth d is

**(A)** 2d **(B)** 2d–1+1

**(C)** 2d+1+1 **(D)** 2d+1

**Q.14** The pre-order and post order traversal of a Binary Tree generates the same output. The tree can have maximum

**(A)** Three nodes **(B)** Two nodes

**(C)** One node **(D)** Any number of nodes

**Q.15** A graph with n vertices will definitely have a parallel edge or self loop if the total number of edges are

**(A)** greater than n–1 **(B)** less than n(n–1)

**(C)** greater than n(n–1)/2 **(D)** less than n2/2

**Q.16** A binary tree of depth “d” is an almost complete binary tree if

**(A)** Each leaf in the tree is either at level “d” or at level “d–1”

**(B)** For any node “n” in the tree with a right descendent at level “d” all the left

descendents of “n” that are leaves, are also at level “d”

**(C)** Both **(A)** & **(B)**

**(D)** None of the above

**Q.17** In Breadth First Search of Graph, which of the following data structure is used?

**(A)** Stack. **(B)** Queue.

**(C)** Linked List. **(D)** None of the above.

**Q.18** For an undirected graph G with n vertices and e edges, the sum of the degrees of each vertex is

**(A)** ne **(B)** 2n

**(C)** 2e **(D)** en

**Part A**

**Q.1** What are expression trees? Represent the following expression using a tree.

Comment on the result that you get when this tree is traversed in Preorder,

Inorder and postorder. (a-b) / ((c\*d)+e)

**Q.2** Taking a suitable example explains how a general tree can be represented as a

Binary Tree.

**Q.3** What are the different ways of representing a graph? Represent the following

graph using those ways.

****

**Q.4** Give the adjacency matrix for the following graph:

**Q.5** Create a heap with following list of keys:

8, 20, 9, 4, 15, 10, 7, 22, 3, 12

**Q.6** Construct a complete binary tree with depth 3 for this tree which is maintained

in memory using linked representation. Make the adjacency list and adjacency matrix for this tree.

**Q7.** A Binary tree has 9 nodes. The inorder and preorder traversals of the tree

yields the following sequence of nodes:

Inorder : E A C K F H D B G

Preorder: F A E K C D H G B

Draw the tree. Explain your algorithm.

**Q.8** How will you represent a max-heap sequentially? Explain with an example.

**Q9.** Construct the binary tree for the following sequence of nodes in preorder and

inorder respectively.

Preorder : G, B, Q, A, C, K, F, P, D, E, R, H

Inorder: Q, B, K, C, F, A, G, P, E, D, H, R

**Q10.** Give the algorithm to construct a binary tree where the yields of preorder and

post order traversal are given.

**Q11.** Draw a picture of the directed graph specified below:

G = ( V, E)

V(G) = {1, 2, 3, 4, 5, 6}

E(G) = {(1,2), (2, 3), (3, 4), (5,1), (5, 6), (2, 6), (1, 6), (4, 6), (2, 4)}

Obtain the following for the above graph:

(i) Adjacency matrix.

(ii) React ability matrix.

**Q.12** Draw a binary tree from its inorder and preorder traversal sequences given as

follows:

Inorder : d b g e h a c n f

Preorder : a b d e g h c f n

**Part B**

**Q.1** Draw the expression tree of the following infix expression. Convert it in to

Prefix and Postfix expressions.

a bc \* d ef \* g h

**Q.2** Given a set of input representing the nodes of a binary tree, write a non

recursive algorithm that must be able to output the three traversal orders.

**Q.3** How do you rotate a Binary Tree? Explain right and left rotations with the help of

an example.

**Q.4** Show the result of running BFS and DFS on the directed graph given below

using vertex 3 as source. Show the status of the data structure used at each

stage.

****

**Q.5** Explain the representations of graph. Represent the given graph using any two

methods ****

**Q.6** Two Binary Trees are similar if they are both empty or if they are both nonempty

and left and right sub trees are similar. Write an algorithm to determine

if two Binary Trees are similar.

**Q.7** The degree of a node is the number of children it has. Show that in any binary tree, the

number of leaves are one more than the number of nodes of degree 2

**Q.8** Write the non-recursive algorithm to traverse a tree in preorder.

**Q.9** Draw the complete undirected graphs on one, two, three, four and five

vertices. Prove that the number of edges in an n vertex complete graph is

n(n-1)/2.

****

**Q.10** Write an algorithm which does depth first search through an un-weighted

connected graph. In an un-weighted graph, would breadth first search or depth

first search or neither find a shortest path tree from some node? Why?

**Q.11** Write a non recursive algorithm to traverse a binary tree in inorder.

**Q.12** Which are the two standard ways of traversing a graph? Explain them with an

example of each.

**Q.13** Consider the following specification of a graph G

VG1,2,3,4

EG1,2, 1,3, 3,3, 3,4, 4,1

(i) Draw an undirected graph.

(ii) Draw its adjacency matrix.

**Q.14** Write an algorithm to insert an element to a max-heap that is represented

sequentially.

**Q.15** Construct a binary tree whose nodes in inorder and preorder are given as

follows:

Inorder : 10, 15, 17, 18, 20, 25, 30, 35, 38, 40, 50

Preorder: 20, 15, 10, 18, 17, 30, 25, 40, 35, 38, 50

**Q.16** Given the following inorder and preorder traversal reconstruct a binary tree

Inorder sequence D, G, B, H, E, A, F, I, C

Preorder sequence A, B, D, G, E, H, C, F, I

**Q.17** What is a Binary Tree? What is the maximum number of nodes possible in a

Binary Tree of depth d. Explain the following terms with respect to Binary

trees

(i) Strictly Binary Tree (ii) Complete Binary Tree (iii) Almost

Complete Binary Tree

**Q.18** Show the result of running BFS and DFS on a directed graph given below

using vertex 1 as source. Show the status of the data structure used at each

stage.

****

**Q.19** Define adjacency matrix and make the same for the following undirected

graph. **(8)**

****

**Q.20** Show the linked representation of the above graph.

**Q.21** What do you understand by tree traversal? Write a procedure for traversing a

binary tree in preorder and execute it on the following tree.

**Q22.** Sort the following list using Heap Sort technique, displaying each step.

20, 12, 25 6, 10, 15, 13

**Q.23.** Give the adjacency matrix and adjacency list of the following graphs.

****

**Q24.** Sort the following list using Heap Sort

66, 33, 40, 20, 50, 88, 60, 11, 77, 30, 45, 65.

**Q25**. What are the two phases in heap sort algorithm? Sort the following data

using heap sort and show all the intermediate steps.

88, 12, 91, 23, 10, 36, 45, 55, 15, 39, 81

**Q.29** Draw the complete undirected graphs on one, two, three, four and five

vertices. Prove that the number of edges in an n vertex complete graph is

n(n-1)/2.

**Unit 4**

**Q.1** If h is any hashing function and is used to hash n keys in to a table of size m, where n<=m, the expected number of collisions involving a particular key x is :

**(A)** less than 1. **(B)** less than n.

**(C)** less than m. **(D)** less than n/2.

**Q.2** A technique for direct search is

**(A)** Binary Search **(B)** Linear Search

**(C)** Tree Search **(D)** Hashing

**Q.3** You have to sort a list L consisting of a sorted list followed by a few “random” elements. Which of the following sorting methods would be especially suitable for such a task?

**(A)** Bubble sort **(B)** Selection sort

**(C)** Quick sort **(D)** Insertion sort

**Q.4** The searching technique that takes O (1) time to find a data is

**(A)** Linear Search **(B)** Binary Search

**(C)** Hashing **(D)** Tree Search

**Q.5** In worst case Quick Sort has order

**(A)** O (n log n) **(B)** O (n2/2)

**(C)** O (log n) **(D)** O (n2/4)

**Q.6** A sort which relatively passes through a list to exchange the first element with any element less than it and then repeats with a new first element is called

**(A)** insertion sort. **(B)** selection sort.

**(C)** heap sort. **(D)** quick sort.

**Q.7** Which of the following sorting algorithms does not have a worst case running time of 2 O n ?

**(A)** Insertion sort **(B)** Merge sort

**(C)** Quick sort **(D)** Bubble sort

**Q.8** The quick sort algorithm exploit \_\_\_\_\_\_\_\_\_ design technique

**(A)** Greedy **(B)** Dynamic programming

**(C)** Divide and Conquer **(D)** Backtracking

**Q.9** The complexity of searching an element from a set of n elements using Binary search algorithm is

**(A)** O(n) **(B)** O(log n)

**(C)** O(n2) **(D)** O(n log n)

**Q.10** Which of the following sorting methods would be most suitable for sorting a list which is almost sorted

**(A)** Bubble Sort **(B)** Insertion Sort

**(C)** Selection Sort **(D)** Quick Sort

**Q.11** Quick sort is also known as

**(A)** merge sort **(B)** heap sort

**(C)** bubble sort **(D)** none of these

**Q.12** The goal of hashing is to produce a search that takes

**(A)** *O(1)* time **(B)** *O(n2 )* time

**(C)** *O(log n )* time **(D)** *O(n log n )* time

**Q.13** The best average behaviour is shown by

**(A)** Quick Sort **(B)** Merge Sort

**(C)** Insertion Sort **(D)** Heap Sort

**Q.14** Which sorting algorithm is best if the list is already sorted? Why?

**Q.15** What is the time complexity of Merge sort and Heap sort algorithms?

**Q.16** Consider that n elements are to be sorted. What is the worst case time complexity of Bubblesort?

**(A)** O(1) **(B)** O(log2n)

**(C)** O(n) **(D)** O(n2)

**Q.17** A characteristic of the data that binary search uses but the linear search ignores is

the\_\_\_\_\_\_\_\_\_\_\_.

**(A)** Order of the elements of the list.

**(B)** Length of the list.

**(C)** Maximum value in list.

**(D)** Type of elements of the list.

**Q.18** The worst case of quick sort has order

**(A)** O(n2) **(B)** O(n)

**(C)** O (n log2 n) **(D) O** (log2 n)

**Part A**

**Q.1** How many key comparisons and assignments an insertion sort makes in its

worst case?

**Q.2** What is the best case complexity of quick sort and outline why it is so. How

could its worst case behaviour arise?

**Q3.** Write an algorithm to sort a given list using Quick sort method. Describe the

behaviour of Quick sort when input is already sorted.

**Part B**

**Q.1** What is quick sort? Sort the following array using quick sort method.

24 56 47 35 10 90 82 31.

**Q.2** Sort the following sequence of keys using merge sort.

66, 77, 11, 88, 99, 22, 33, 44, 55.

**Q.3** The following values are to be stored in a hash table

25, 42, 96, 101, 102, 162, 197

Describe how the values are hashed by using division method of hashing with

a table size of 7. Use chaining as the method of collision resolution.

**Q.4** Describe insertion sort with a proper algorithm. What is the complexity of

insertion sort in the worst case?

**Q.5** What do you mean by hashing? Explain any five popular hash functions.

**Q.6** Write an algorithm to merge two sorted arrays into a third array. Do not sort

the third array.

**Q.7** Define Hashing. How do collisions happen during hashing? Explain the

different techniques resolving of collision.

**Q.8** What do you mean by hash clash? Explain in detail any one method to resolve

hash collisions.

**Q9.** Execute quick algorithm on the following data till two key values are placed in

their position 12,34,45,15,4,11,7,8,5,14,35,89,43,21.

**Q 10** Sort the following array of elements using quick sort *{3 1 4 1 5 9 2*

*6 5 3 5 8}*

**Q.11** Execute your algorithm for two passes using the following list as input:

66, 33, 40, 20, 50, 88, 60, 11, 77, 30, 45, 65

Describe the behaviour of Quick sort when the input is already sorted.

**Q12.** Write down the algorithm of quick sort.

**Q13.** Draw the 11 item hash table resulting from hashing the keys: 12, 44, 13, 88,

23, 94, 11, 39, 20, 16 and 5 using the hash function h(i) = (2i+5) mod 11.

**Q14.** Write an algorithm for selection sort. Describe the behaviours of selection sort

when the input is already sorted.

**Q15.** Explain Hash Tables, Hash function and Hashing Techniques?

**Q16.** Define hashing. Describe any two commonly used hash functions. Describe one

method of collision resolution.

**Q.17** Compare and contrast various sorting techniques with respect to memory

space and computing time.

**Unit 5**

**Q.1** B Trees are generally

**(A)** very deep and narrow **(B)** very wide and shallow

**(C)** very deep and very wide **(D)** cannot say

**Q.2** If a node in a BST has two children, then its inorder predecessor has

**(A)** no left child **(B)** no right child

**(C)** two children **(D)** no child

**Q.3** A binary tree in which if all its levels except possibly the last, have the maximum number of nodes and all the nodes at the last level appear as far left as possible, is known as

**(A)** full binary tree. **(B)** AVL tree.

**(C)** threaded tree. **(D)** complete binary tree.

**Q.4** A B-tree of minimum degree t can maximum \_\_\_\_\_ pointers in a node.

**(A)** t–1 **(B)** 2t–1

**(C)** 2t **(D)** t

**Q.5** A BST is traversed in the following order recursively: Right, root, left

The output sequence will be in

**(A)** Ascending order **(B)** Descending order

**(C)** Bitomic sequence **(D)** No specific order

**Q.6** One of the major drawback of B-Tree is the difficulty of traversing the keys sequentially.

**Q.8** In order to get the information stored in a Binary Search Tree in the descending order, one should traverse it in which of the following order?

**(A)** left, root, right **(B)** root, left, right

**(C)** right, root, left **(D)** right, left, root

**Part A**

**Q.1** Define a B-Tree.

**Q 2.** What is a height balanced tree? Explain how the height is balanced after

addition/deletion of nodes in it?

**Q 3.** Write an algorithm to test whether a Binary Tree is a Binary Search Tree.

**Q.4** What are B-trees? Draw a B-tree of order 3 for the following sequence of

keys. 3,5,11,10,9,8,2,6,12

**Part B**

**Q.1** What is a Binary Search Tree (BST)? Make a BST for the following sequence

of numbers.

45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48

Traverse the tree in Preorder, Inorder and postorder.

**Q.2** Show the result of inserting the keys.

F, S, Q, K, C, L, H, T, V, W, M, R, N , P, A, B, X, Y, D, Z, E in the order to

an empty B-tree of degree-3.

**Q.3** Make a BST for the following sequence of numbers.

45,32,90,34,68,72,15,24,30,66,11,50,10 Traverse the BST created in Preorder,

Inorder and Postorder.

**Q4.** What are B-trees? Construct a B-Tree of order 3 for the following set of

Input data:

69, 19, 43, 16, 25, 40, 132, 100, 145, 7, 15, 18

**Q.33** Draw a B-tree of order 3 for the following sequence of keys:

2, 4, 9, 8, 7, 6, 3, 1, 5, 10

**Q.5** Explain insertion into a B-tree.

**Q6.** Write an algorithm to delete a particular node from binary search tree. Trace

your algorithm to delete a node (10) from the given tree.

****

**Q7.** What is a Binary Search Tree (BST)? Make a BST for the following sequence

of numbers.

45, 32, 90, 21, 78, 65, 87, 132, 90, 96, 41, 74, 92

**Q8.** Traverse the Binary Search Tree created above in Preorder, Inorder and Postorder.

**Additional Questions**

**Q.1** Write short notes on any **FOUR**:-

(i) B Tree.

(ii) Time Complexity, Big O notation.

(iii) Merge Sort.

(iv) Threaded Binary Tree.

(v) Depth First Traversal.

**Q.2** Write an algorithm INSERT that takes a pointer to a sorted list and a pointer to

a node and inserts the node into its correct position in the list.

**Q. 3**Write short notes on the following:

(i) B-tree.

(ii) Abstract data type.

**Q.4** Define data type and abstract data type. Comment upon the significance of

both.

**Q5** Enumerate various operations possible on ordered lists and arrays. Write

procedures to insert and delete an element in to array.

**Q.6** By taking an example show how multidimensional array can be represented in

one dimensional array.

**Q.7** Show the various passes of bubble sort on an unsorted list 11, 15, 2, 13, 6

**Q.8** Describe the concept of binary search technique? Is it efficient than sequential

search?

**Q.9** Prove the hypothesis that “A tree having ‘m’ nodes has exactly (m–1) edges or

branches”.

****

**Q.10** Write a procedure to insert a node into a linked list at a specific position and

draw the same by taking any example?

**Q.11** List various problem solving techniques.

**Q12.** Explain the concept of primitive data structures.

**Q13.** The system allocates memory for any multidimensional array from a large

single dimensional array. Describe *two* mapping schemes that helps us to store

a *two* dimensional metrics in a *one-dimensional* array.

**Q14.** Write an algorithm for binary search. What are the conditions under which

sequential search of a list is preferred over binary search?

**Q15.** Define the following terms:

(i) Abstract data type.

(ii) Column major ordering for arrays.

(iii) Adjacency multilist.

(iv) Game trees.

**Q16.** Describe various memory allocation strategies.

**Q17.** How memory is freed using Boundary tag method in the context of Dynamic

memory management?

**Q 18.** Define a method for keeping two stacks within a single linear array S in such a

way that neither stack overflows until entire array is used and an entire stack is

never shifted to a different location within the array. Write routines for pushing

and poping elements in two stacks.

**Q19.** Suppose a queue is housed in an array in circular fashion. It is desired to add

new items to the queue. Write down a procedure ENQ to achieve this also

checking whether the queue is full. Write another procedure DQ to delete an

element after checking queue empty status.

**Q20.** Write short notes on the following:

(i) Threaded binary trees.

(ii) Graph traversal.

(iii) Conversion of forest into tree.

(iv) Doubly linked list.

**Q21.** Differentiate between system defined data types and Abstract data types with

suitable examples.

**Q22.** Explain the following:

(i) Complexity of an Algorithm.

**(ii)** The space-time trade off algorithm.

**Q23.** Let a binary tree ‘T’ be in memory. Write a procedure to delete all terminal

nodes of the tree.

**Q24.** Consider the following eight numbers 50, 33, 44, 22, 77, 35, 60 and 40. Display

the construction of the binary by inserting the above numbers in the given order.

**Q25.** Establish the usage of linked lists for polynomial manipulation.

**Q26.** Define a linked-list? How are these stored in the memory? Suppose the linked

list in the memory consisting of numerical values. Write a procedure for each of

the following:

(i) To find the maximum MAX of the values in the list.

(ii) To find the average MEAN of the values in the list.

(iii) To find the product PROD of the values in the list.

**Q27.** Give the binary search algorithm.

**Q28.** What do you understand by structured programming? Explain.

**Q29.** Consider the algebraic expression

E = (5x+z) (3a-b)2

(i) Draw the expression tree corresponding to E

(ii) Find the scope of exponential operator i.e. the subtree rooted at the

exponential operator.

**Q30.** Define an array. How does an array differ from an ordinary variable? How are

arrays represented in the memory?

**Q31.** Consider an array A[20, 10]. Assume 4 words per memory cell and the base

address of array A is 100. Find the address of A[11, 5] assuming row major

storage.

**Q32.** Write a recursive function to count the number of nodes in a binary tree.

**Q33.** Define the following :

(i) AVL tree.

(ii) Thread.

(iii) Heap.

(iv) Binary Search Tree.

**Q34.** Write an algorithm for searching a key from a sorted list using binary search

technique.

**Q35.** Define graph, adjacency matrix, adjacency list, hash function, sparse matrix,

reachability matrix.

**Q36.** Explain various graph traversal schemes and write their merits and demerits.

**Q37.** Write short notes on the following:

(i) Decision and game trees.

(ii) Polynomial representation and manipulation using linked lists.

(iii) Analysis of algorithm.

(iv) Circular queues.

**Q38.** What is the smallest value of n such that an algorithm whose running time is

100n2 runs faster than an algorithm whose running time is 2n on the same

machine.

**Q39.** Let X = (X1, X2, X3,….Xn) and Y= (Y1, Y2, Y3,….Xm) be two linked lists.

Write an algorithm to merge the lists together to obtain the linked list Z such that

Z = (X1, Y1, X2, Y2,….Xm, Ym,Xm+1….Xn) if m<=n or

Z = (X1, Y1,X2,Y2….Xn,Yn,Yn+1….Ym) if m>n.

**Q40.** Devise a representation for a list where insertions and deletions can be made at

either end. Such a structure is called a Deque (Double ended queue). Write

functions for inserting and deleting at either end.

**Q41.** Write binary search algorithm and trace to search element 91 in following list:

13 30 62 73 81 88 91

What are the limitations of Binary Search?

**Q42.** Show the result of running BFS on a complete Binary Tree of depth 3. Show the

status of the data-structure used at each stage.

**Q43.** Define a linked list with a loop as a linked list in which the tail element points

to one of the list’s elements and not to NULL. Assume that you are given a

linked list *L*, and two pointers *P1*, *P2* to the head. Write an algorithm that

decides whether the list has a loop without modifying the original list. The

algorithm should run in time *O(n)* and additional memory *O(1)*, where *n* is the

number of elements in the list.

Q44 Write an algorithm for checking validity of the input, i.e., the program must

know if the input is disjoint, duplicated and has a loop.

**Q.45** Write an algorithm for finding solution to the Tower’s of Hanoi problem.

Explain the working of your algorithm (with 4 disks) with diagrams.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

**18 Assignment Questions**

**Unit -1**

1. Write both a recursive and an iterative c function to compute n!
2. The fibonacci numbers are defined as f0=0 f1=1 fi=f1+fi-2 for i>1 write recursive and non recursive function to compute fi
3. Write a recursive function to compute a binomial coefficient then transform it into an equivalent non recursive function
4. Write a recursive function solving towers Hanoi problem
5. Consider an array A[20, 10]. Assume 4 words per memory cell and the base address of array A is 100. Find the address of A[11, 5] assuming row major

storage.

1. Obtain the indexing formula for lower right and upper-left triangular matrices using row major and column major , consider the cases of
   * 1. Square matrix of order n\*n;
     2. Non square matrix of order m\*n m<>n

**Unit -2**

**1.** Convert the following infix expression into a postfix expression (Show steps)

(i)ABD/ E −FG H/ k

(ii) A B D/E −FG

(iii) a bc d/e f g .

2 After obtaining the postfix expression on the above expressions reverse them into infix expression using corresponding algorithm

**Unit 3**

1. A Binary tree has 9 nodes. The inorder and postorder traversals of the tree

yields the following sequence of nodes:

Inorder : 1 2 3 4 5 6 7 8 9

Postorder: 1 3 5 4 2 8 7 9 6

Draw the tree. Explain your algorithm.

1. How will you represent a max-heap sequentially? Explain with an example.
2. Construct the binary tree for the following sequence of nodes in preorder and

inorder respectively.

Preorder : G, B, Q, A, C, K, F, P, D, E, R, H

Inorder: Q, B, K, C, F, A, G, P, E, D, H, R

1. Give the algorithm to construct a binary tree where the yields of preorder and

post order traversal are given.

1. Draw a picture of the directed graph specified below:

G = ( V, E)

V(G) = {1, 2, 3, 4, 5, 6}

E(G) = {(1,2), (2, 3), (3, 4), (5,1), (5, 6), (2, 6), (1, 6), (4, 6), (2, 4)}

Obtain the following for the above graph:

* 1. Adjacency matrix.
  2. React ability matrix.
  3. **A**djacency List.

**Unit 4**

1. What is quick sort? Sort the following array using quick sort method.

24 56 47 35 10 90 82 31 **(7)**

1. Sort the following sequence of keys using merge sort.

66, 77, 11, 88, 99, 22, 33, 44, 55 **(8)**

1. Apply radix sort ,insertion sort, selection sort techniques on the above data
2. The following values are to be stored in a hash table

25, 42, 96, 101, 102, 162, 197

Describe how the values are hashed by using division method of hashing with

a table size of 7. Use chaining as the method of collision resolution. **(8)**

**Unit 5**

1. What are B-trees? Draw a B-tree of order 3 for the following sequence of

keys. 3,5,11,10,9,8,2,6,12 **(6)**

1. What is a Binary Search Tree (BST)? Make a BST for the following sequence

of numbers.45, 32, 90, 21, 78, 65, 87, 132, 90, 96, 41, 74, 92 **(7)**

1. construct an AVL tree for the following data
   * 1. 30,31,32,23,22,28,24,29,26,27,34,36
     2. 50,55,60,15,20,40,20,45,30,70,80
2. Write short notes Pattern Matching Algorithms
3. Write short notes Tries
4. Explain Red –black trees
5. Explain Splay trees.

**19 Quiz Questions**

**Unit 1**

1) The asymptotic analysis focuses on determining \_\_\_\_\_\_\_terms in the complexity function

2) The data space is needed store\_\_\_\_\_\_\_\_\_

3) Consider a linked list of n elements. What is the time taken to insert an element pointer ? [ ]

1. O(log2n) B. O(n) C.O(1) D.O(n log2n)

4) Data that consists of a single, non decomposable entity are known [ ]

(A) atomic data (B) array new (C) data structure delete (D) standard type

**Unit 2**

1) Which of the following operation is used to add an item in a queue [ ]

(A) write() (B) read() (C) pop() (D) push()

2) Queue can be used to implement [ ]

(A) recursion (B) quick sort (C) radix sort (D) depth first search

**Unit 3**

1. A priority queue can be implemented by [ ]

a) Heap b) BST c) DFS method d) AVL Tree

2. The difference between tree and graph will be [ ]

a) Tree has no cycles, graph can have cycle b) Tree has no parent, graph can have parent

c) Tree has root node, graph has no root node d) Both A and C

3. Which of the following is useful in traversing a given graph by breadth first search [ ]

a) Stack s b) Set c) list d) Queue

4. In a heap\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ element will resides at top position.

5. In a max heap the child element should be \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ than parent element

6. if a heap represented in the form of list, when a parent element available at “ ith “ element in the list then left, right Childs will be available at \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

7. The post order traversal of a binary tree is DEBFCA. find out the pre order traversal [ ]

A)ABFCDE B)ADBFEC C)ABDECF D)ABDCEF

8. The time to initialize the max heap is \_\_\_\_\_\_\_

9) The data structure that is used to keep the vertices whose adjacent vertices are to be visited in the Depth first traversal\_\_\_\_\_\_\_\_\_\_\_ [ ]

a) Queue b) stack c) heap d) dictionary.

10) The number of edges incident from a vertex vi called\_\_\_\_\_\_\_ [ ]

a) In degree b) out degree c) pendent d) degree

11) Graph can be represented by [ ]

a) Adjacency matrix b) adjacency list c) queue d) both a & b

12) \_\_\_\_\_\_\_\_\_ is the application of Priority queue [ ]

a) Scheduling of jobs in operating system b)text editors

c)spell checking programs d)heap

13) Graph is a collection of \_\_\_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_\_\_

14) \_\_\_\_\_\_\_\_\_\_\_ is required when data being sorted do not fit in to main memory.

15) \_\_\_\_\_\_ consists of a set of vertices V and a set of edges E.

16) \_\_\_\_\_\_\_is a complete binary tree in which the value in each node is lesser than those in its children.

17) In an undirected graph, the sum of degrees of all the nodes [ ]

(A) must be even (B) is thrice the number of edges

(C) must be odd (D) need not be even

18) The minimum number of edges in a connected cyclic on n vertices is \_\_\_\_\_\_\_\_\_\_\_\_\_

19) An n vertex undirected graph with exactly n\*(n-1)/2 edges is said to be \_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Unit 4**

1) The order of the binary search algorithm is \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2) In hashing by division the hash function has the form\_\_\_\_\_ [ ]

a)f(k)=K%(d-1) b) f(k)=(K+1)%(d+1) c)f(k)=(k-1)%d d)f(k)=k%d

3) In division hash function , in the hash table of length 11 we can place the value 80 at \_\_\_\_\_position. [ ]

a)5 b)8 c)3 d)10

4)\_\_\_\_\_\_\_\_\_\_ occurs when there isn't room i n the home bucket for the new pair

5) One of the collision handling method \_\_\_\_\_\_\_\_

6) A -------------- sort uses the binary tree concept such that any number is larger than all the numbers in the subtree below it is called [ ]

(A) Quick (B) Bubble (C) Heap t (D) All

7) The number of passes required for sorting M records of length N using simple external sorting algorithm is [ ]

(A) [log(N/M)] (B) [log(M/N)] (C) [log(N\*M)] (D) [log(N+M)]

8) For merging two sorted lists of sizes m and n into a sorted list of size m+n, requires \_ \_\_ \_ \_ \_ \_ \_ no.of comparisons. [ ]

a) O(m) b) O(n) c) O(m+n) d) O(log(m)+log(n))

9) Sorting is not useful for [ ]

a) report generation b) minimizing the storage needed

c) making searching easier and efficient d) responding to queries easily

10) Merge sort uses\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ [ ]

a)divide and conquer b)backtracking c)greedy approach d)heuristic approach

11) The average number of comparisons performed by the merge sort algorithm , in merging two sorted listsof length 2 is [ ]

(A) 8/3 (B) 8/5 (C) 11/7 (D) 1/16

**Unit 5**

1) A binary search tree contains the values - 1,2,3,4,5,6,7,8. The tree is traversed in preorder and the values are printed out. Which of the following sequences is a valid output? [ ]

(A) 5 1 2 3 (B) 1 4 2 6 (C) 1 2 3 4 (D) 5 3 1 2

2) In a binary search tree if the key element is less than the root element then sub tree must be searched in \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ traversal of a binary search tree traverses visits to the nodes in ascending order of key values.

4). In a BST, parent element should be [ ]

a) <left,>right b) >left, <right c) <left, <right d) >left, >right

5) In a red black tree, a root node is \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ leaf node is \_\_\_\_\_\_\_\_\_ [ ]

a) Black, Red b) Red, Black c) Black, Black d) Red, Red

6) Which of the following is search engine [ ]

a) BST b) AVL tree c) Brute Force Alg d) Splay tree

7) A node in a B-tree consists of set of elements, those should be arranged in [ ]

a) Non-increasing order b) Non-decreasing order c) Both, depends on data d) None

8) In an AVL tree, the heights of left, right sub child are differed by [ ]

a) At most one b) At least one c) One d) Depends on data

9) Suffix Trie search time [ ]

a) O (n+1) b) O (n-1) c) O (m) d) None

10) The order of B-tree indicates [ ]

a) Number of element present in the tree b) Number of element present at certain node

c) Total number of leaf nodes d) None

11) In a BST the left child should be \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ than parent element

12) AVL tree is also known as \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

13) A height of BST will be performed by \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ traversal

14) When an element inserted at left sub tree of left child then \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ rotation will be performed.

15) The Knuth-Morris-Pratt (KMP) algorithm looks for the pattern in the text in a \_\_\_\_\_\_\_\_\_\_\_\_order

16) A compressed trie as internal nodes of degree al least \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

17) In a binary tree, certain null entries are replaced by special pointers which point to nodes higher in the tree for efficiency. These special pointers are called \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

18) A binary search tree is constructed with the following keys 20,22,26,21,13,19,18,15,26,28

The above keys are inserted in that order. Then the total keys in the left sub tree and the right sub tree of the tree or respectably. [ ]

a) 5,5 b) 6,4 c) 7,3 d) 4,6

19) The balance factor of a node x in a binary tree is 3. There are 2 nodes in the right sub tree of x. There must be \_ \_ \_ \_ \_ \_ \_ \_ nodes in the left sub tree of x [ ]

a) 2 b) 0 c) 5 d) 3

20) AVL tree is a \_ \_ \_ \_ \_ \_ \_ \_ \_ binary tree [ ]

a) Complete b) Full c) Height balanced d) Skewed

21) In KMP pattern matching algorithm pre processing is done by an auxillary function known as

a) failure function b) prefix function c) postfix function d) insert function

22) In R0 Rotation the node which is imbalanced will be moving towards [ ]

a) Left sub tree b) Right Subtree c) root d) not moving

23) Which traversal of a binary search tree traverses visits to the nodes in ascending order of key values?

a) In Order b) Pre Order c) Post Order d) Past Order

24) AVL tree was not developed by \_ \_ \_ \_ \_ \_ \_ \_ \_ [ ]

a) Velskii b) Anderson c) Landis d) Adelson

25) B-tree of order 3 is also known as \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ [ ]

a) 2-3 tree b) 3-4 tree c) binary tree d) Splay tree

26) In binary search tree, if the element to be inserted is greater than the root node, the element is inserted in \_\_\_\_\_\_\_\_\_\_\_\_

27. The difference between the height of left sub tree & right sub tree is called \_\_\_\_\_\_\_\_

28. All AVL Trees are basically \_\_\_\_\_\_\_\_\_\_

29. \_\_\_\_\_\_\_\_\_ algorithm is recommended for binary strings pattern matching.

30. The permissible balance factors of an AVL trees are \_\_\_\_\_\_\_\_\_

31. In B-tree of order m, the root has child nodes\_\_\_\_\_\_\_\_\_\_\_

32. \_\_\_\_\_\_\_\_\_\_\_ traversal technique the node is processed before the children.

33. \_\_\_\_\_\_\_\_ Tress is designed especially for use on disk.

34. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ algorithm is preferred for pattern matching when the length is of short duration.

35 In a B-tree of order ‘m’ all the leaf nodes except the root node should have minimum of \_\_\_ non empty children. [ ]

a) [m/2] b) [m] c) [m-1] d) [(m/2)-1]

36 \_\_\_\_\_\_\_ algorithm is recommended for the binary strings pattern matching [ ]

a)Brute force b) Boyer Moore c)KMP d)Morris

37 In LR Rotation the node which is imbalanced is replaced by\_\_\_\_\_\_\_\_\_\_\_\_ [ ]

a) root of the left subtree b)root of right subtree

c) left child of right subtree d) right child of left subtree

38 A compressed trie is a kind of standard trie in which internal node has atleast degree of [ ]

a) 3 b) 1 c) 2 d)-1

39) The difference between heights of left subtree and Right subtree is called [ ]

a) Balanced factor b) height difference c) Rank d) Load balance

40) In a AVL Tree LL rotation the node which is imbalanced will move towards\_\_\_\_\_\_\_\_\_\_

41) The search, insert and delete operations on a m-way search tree of height have the complexity as \_\_\_\_\_\_

42) \_\_\_\_\_\_\_\_\_\_algorithm is preferred for pattern matching if the size of string is large compared to the length of the pattern.

43) A node with ‘k’ subtrees will have\_\_\_\_\_\_ elements in B-Tree.

44) \_\_\_\_\_\_\_\_\_ is a technique of finding the substring in text which is equal to pattern.

45) \_\_\_\_\_\_\_\_\_\_ is collection of elements that each element has been added a priority.

46) Find the odd one out [ ]

(A)binary tree (B) AVL tree (C) graph (D) queue

47) Which of the following need not be a binary tree [ ]

(A)Search tree (B) Heap (C) AVL-tree (D) B-tree

48) Which of the following traversal techniques lists the nodes of a binary search tree in ascending order

(A) post-order (B) In-order (C) Pre-order (D) No-order

49) In which trie node allows only one character [ ]

(A) standard (B) compressed (C) suffix (D) none

50) A priority queue is an abstract concept like a \_\_\_\_\_\_\_\_\_\_\_\_

51) Merge sort uses\_\_\_\_\_\_\_\_\_\_\_\_\_ strategy

52**)** The depth of a complete binary tree with n nodes **\_\_\_\_\_\_\_\_\_\_**

53) \_\_\_\_\_\_\_\_\_\_ tree is a self-balancing binary search tree

54) A standard trie uses\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ space

55) A \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ is a tree-based data structure, stores the large text string as a tree for fast pattern matching.

**20 Tutorial problems**

**UNIT-I**

* + Add the following operation to the Natural Number ADT: Predecessor, IsGreater, Multiply, Divide.
  + Create an ADT set: use the standard mathematical definition and include the following operations: Crate, Insert, Remove, IsIn, Union, Intersection and Difference.
  + Show that the following statements are correct
    1. 5n2-6n=ϴ(n2) b) 2n2+nlogn= ϴ(n2)

1. Show that the following statements are incorrect.
   * 1. 10n2+9=O(n) b) n2logn= ϴ(n2)
2. Suppose , an arrayA[-15,…64] is stored in a memory whose starting address is 459. Assume that word size for each element is 2. Then obtain the following

i). How many no. of elements are there in the array A.

ii). If one word of the memory is equal to 2 bytes, then how much memory is required to store the entire array.

iii). What is the location for A[50].

iv). What is the location for 10th element?

v). which element is located at 589.

**UNIT-II**

* + Transform the following infix expressions into their equivalent postfix expressions

i). A\*(B+D)/F-F\*(G+H/K)

ii). A^B\*C+D/A/(E+F)

**UNIT-III**

1. Draw the internal memory representations of the binary tree using

a) Sequential and b) Linked Representation.

****

2. Write the preorder, inorder and post order traversals of the binary tree given in Q.1

3. Draw the binary tree given in Q.1, showing its threaded representation.

4. Suppose that we have the following key values7,16,49,82,5,31,6,2,44

a). Write out the max heap after each value is inserted into the heap.

b). Write the min heap after each value is inserted into the heap.

5. Consider the following specification of a graph G, V(G)={1,2,3,4}

E(G)={(1,2),(1,3),(3,3),(3,4),(4,1)}

a). Draw a picture of the undirected graph.

b). Give its Adjacency Matrix.

c). Give its Adjacency List.

d). Write BFS and DFS Traversals.

**UNIT-IV**

* 1. Consider the given unsorted array. Sort this array in ascending order using insertion sort.

76, 67, 36, 55, 23, 14, 6

* 1. Sort this array using selection sort and show your work in each pass.

7, 23,31,40,56, 78,92

* 1. Sort 07, 10, 99, 02, 80, 14, 25, 63, 88, 33, 11, 72, 68, 39,21, 50 using Radix Sort.
  2. Sort the following numbers using Quick Sort.

3, 1,4,5,9,2,6,10,7,8

1. Given the input{4371, 1323,6173,4199,4344,9699,1889} and hash function as

Key%10. Show the results for the following.

* + - 1. Open addressing using linear probing.
      2. Open addressing using quadratic probing.
      3. Open addressing using double hashing.

**UNIT-V**

* 1. Construct the AVL Tree for the following data.
  2. Explain the steps to build a B-Tree of order 3 for the following data.
  3. Explain how Brute- Force algorithm searches for abdf in pattern abdadefg.
  4. Apply Knuth-Morris –Pratt algorithm to P=bacaaa and

T=bacbacabcbacbbbacabacbabcbbba

5 . Construct a trie for the set of

keywords={inner,input,in,outer,output,put,outing,tint}

**21 Known gaps**

Fortunately, no known gaps as it is extension of C Programming in their I Year.

**22 Discussion Topics**

* Types of Linked Lists
* Sparse matrices representation and manipulation.
* Queues- Circular Queues, Double Ended Queues.
* Non Recursive Binary Tree Traversals.
* Searching Strategies- Linear and Binary Search, Hashing.
* Comparison of Sorting techniques.
* Search Trees.
* Linked list implementation of various Data Structures.
* Pattern Matching Algorithms.

**23 References, Journals, websites and E-links**

**REFERENCE BOOKS:**

1. Data structures: A Pseudocode Approach with C, 2nd edition, R.F.Gilberg And B.A.Forouzan, Cengage

Learning.

2. Data structures and Algorithm Analysis in C, 2nd edition, M.A.Weiss, Pearson.

3. Data Structures using C, A.M.Tanenbaum,Y. Langsam, M.J.Augenstein, Pearson.

4. Data structures and Program Design in C, 2nd edition, R.Kruse, C.L.Tondo and B.Leung,Pearson.

5. Data Structures and Algorithms made easy in JAVA, 2nd Edition, Narsimha Karumanchi, CareerMonk

Publications.

6. Data Structures using C, R.Thareja, Oxford University Press.

7. Data Structures, S.Lipscutz,Schaum’s Outlines, TMH.

8. Data structures using C, A.K.Sharma, 2nd edition, Pearson..

9. Data Structures using C &C++, R.Shukla, Wiley India.

10. Classic Data Structures, D.Samanta, 2nd edition, PHI.

11. Advanced Data structures, Peter Brass, Cambridge.

**Websites and E-Links:**

**1. www.cise.ufl.edu/~sahni/fdsc2ed/**

1. [www.csie.ntu.edu.tw/~hsinmu/courses/media/dsa-13spring/horowitz-28-41.pdf](http://www.csie.ntu.edu.tw/~hsinmu/courses/media/dsa-13spring/horowitz-28-41.pdf)
2. monet.skku.ac.kr/course-materials/graduate/al/lecture
3. iete-elan.ac.in/solQp/soln/DC08-sol.pdf
4. [www.oupinheonline.com](http://www.oupinheonline.com)
5. www2.kenyon.edu/Depts/Math/Aydin/Teach/Sp04/218/ppt

**24 Quality control Sheets**

**25 Students list(attached)**

**26 Group wise students list (attached)**