

EE2204 DATA STRUCTURES AND ALGORITHM
(Common to EEE, EIE & ICE)

UNIT I LINEAR STRUCTURES

Abstract Data Types (ADT) – List ADT – array-based implementation – linked list implementation – cursor-based linked lists – doubly-linked lists – applications of lists – Stack ADT – Queue ADT – circular queue implementation – Applications of stacks and queues

UNIT II TREE STRUCTURES

Need for non-linear structures – Tree ADT – tree traversals – left child right sibling data structures for general trees – Binary Tree ADT – expression trees – applications of trees – binary search tree ADT

UNIT III BALANCED SEARCH TREES AND INDEXING

AVL trees – Binary Heaps – B-Tree – Hashing – Separate chaining – open addressing – Linear probing

UNIT IV GRAPHS

Definitions – Topological sort – breadth-first traversal - shortest-path algorithms – minimum spanning tree – Prim's and Kruskal's algorithms – Depth-first traversal – biconnectivity – Euler circuits – applications of graphs

UNIT V ALGORITHM DESIGN AND ANALYSIS

Greedy algorithms – Divide and conquer – Dynamic programming – backtracking – branch and bound – Randomized algorithms – algorithm analysis – asymptotic notations – recurrences – NP-complete problems

TEXT BOOKS

1. M. A. Weiss, "Data Structures and Algorithm Analysis in C", Pearson Education Asia, 2002.
2. ISRD Group, "Data Structures using C", Tata McGraw-Hill Publishing Company Ltd., 2006.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "Data Structures and Algorithms", Pearson Education, 1983.
2. R. F. Gilberg, B. A. Forouzan, "Data Structures: A Pseudocode approach with C", Second Edition, Thomson India Edition, 2005.
3. Sara Baase and A. Van Gelder, "Computer Algorithms", Third Edition, Pearson Education, 2000.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms", Second Edition, Prentice Hall of India Ltd, 2001.

UNIT I LINEAR STRUCTURES

Abstract Data Types (ADT) – List ADT – array-based implementation – linked list Implementation – cursor-based linked lists – doubly-linked lists – applications of lists – Stack ADT – Queue ADT – circular queue implementation – Applications of stacks and queues

ABSTRACT DATA TYPE

In programming each program is breakdown into modules, so that no routine should ever exceed a page. Each module is a logical unit and does a specific job modules which in turn will call another module.

Modularity has several advantages

1. Modules can be compiled separately which makes debugging process easier.
2. Several modules can be implemented and executed simultaneously.
3. Modules can be easily enhanced.

Abstract Data type is an extension of modular design.

An abstract data type is a set of operations such as Union, Intersection, Complement, Find etc.,

The basic idea of implementing ADT is that the operations are written once in program and can be called by any part of the program.

THE LIST ADT

List is an ordered set of elements.

The general form of the list is

$A_1, A_2, A_3, \dots, A_N$

A_1 - First element of the list

A_N - Last element of the list

N - Size of the list

If the element at position i is A_i then its successor is A_{i+1} and its predecessor is A_{i-1} .

Various operations performed on List

1. Insert (X, 5) - Insert the element X after the position 5.
2. Delete (X) - The element X is deleted
3. Find (X) - Returns the position of X.
4. Next (i) - Returns the position of its successor element $i+1$.
5. Previous (i) - Returns the position of its predecessor $i-1$.
6. Print list - Contents of the list is displayed.
7. Makeempty - Makes the list empty.

2.1 .1 Implementation of List ADT

1. Array Implementation
2. Linked List Implementation
3. Cursor Implementation.

Array Implementation of List

Array is a collection of specific number of data stored in a consecutive memory locations.

* Insertion and Deletion operation are expensive as it requires more data movement

* Find and Printlist operations takes constant time.

* Even if the array is dynamically allocated, an estimate of the maximum size of the list is required which considerably wastes the memory space.

Linked List Implementation

Linked list consists of series of nodes. Each node contains the element and a pointer to its successor node. The pointer of the last node points to NULL.

Insertion and deletion operations are easily performed using linked list.

Types of Linked List

1. Singly Linked List
2. Doubly Linked List

3. Circular Linked List.

2.1.2 Singly Linked List

A singly linked list is a linked list in which each node contains only one link field pointing to the next node in the list.

DECLARATION FOR LINKED LIST

Struct node ;

typedef struct Node *List ;

typedef struct Node *Position ;

int IsLast (List L) ;

int IsEmpty (List L) ;

position Find(int X, List L) ;

void Delete(int X, List L) ;

position FindPrevious(int X, List L) ;

position FindNext(int X, List L) ;

void Insert(int X, List L, Position P) ;

void DeleteList(List L) ;

Struct Node

{

int element ;

position Next ;

};

ROUTINE TO INSERT AN ELEMENT IN THE LIST

void Insert (int X, List L, Position P)

/* Insert after the position P*/

```

{
position Newnode;

Newnode = malloc (size of (Struct Node));

If (Newnode! = NULL)

{

Newnode ->Element = X;

Newnode ->Next = P->  Next;

P->  Next = Newnode;

}

}

```

INSERT (25, P, L)

ROUTINE TO CHECK WHETHER THE LIST IS EMPTY

```

int IsEmpty (List L) /*Returns 1 if L is empty */

{

if (L ->  Next == NULL)

return (1);

}

```

ROUTINE TO CHECK WHETHER THE CURRENT POSITION IS LAST

```

int IsLast (position P, List L) /* Returns 1 is P is the last position in L */

{

if (P->Next == NULL)

return

}

```

FIND ROUTINE

position Find (int X, List L)

```
{  
/*Returns the position of X in L; NULL if X is not found */  
position P;  
P = L-> Next;  
while (P!= NULL && P Element != X)  
P = P->Next;  
return P;  
}  
}
```

FIND PREVIOUS ROUTINE

position FindPrevious (int X, List L)

```
{  
/* Returns the position of the predecessor */  
position P;  
P = L;  
while (P -> Next != Null && P ->Next Element != X)  
P = P ->Next;  
return P;  
}
```

FINDNEXT ROUTINE

position FindNext (int X, List L)

```

{
/*Returns the position of its successor */

P = L ->Next;

while (P ->Next != NULL && P ->Element != X)

P = P ->Next;

return P ->Next;

}

```

ROUTINE TO DELETE AN ELEMENT FROM THE LIST

```

void Delete(int X, List L)

{
/* Delete the first occurrence of X from the List */

position P, Temp;

P = Findprevious (X,L);

If (!IsLast(P,L))

{

Temp = P ->Next;

P ->Next = Temp ->Next;

Free (Temp);

}

}

```

ROUTINE TO DELETE THE LIST

```

void DeleteList (List L)

{

```

```

position P, Temp;

P = L →Next;

L→Next = NULL;

while (P! = NULL)

{

Temp = P→Next

free (P);

P = Temp;

}

}

```

2.1.3 Doubly Linked List

A Doubly linked list is a linked list in which each node has three fields namely data field, forward link (FLINK) and Backward Link (BLINK). FLINK points to the successor node in the list whereas BLINK points to the predecessor node.

STRUCTURE DECLARATION : -

```

Struct Node

{

int Element;

Struct Node *FLINK;

Struct Node *BLINK

};

```

ROUTINE TO INSERT AN ELEMENT IN A DOUBLY LINKED LIST

```

void Insert (int X, list L, position P)

{

```



```

Struct Node * Newnode;

Newnode = malloc (size of (Struct Node));

If (Newnode != NULL)

{

Newnode →Element = X;

Newnode →Flink = P →Flink;

P →Flink →Blink = Newnode;

P →Flink = Newnode ;

Newnode →Blink = P;

}

}

```

ROUTINE TO DELETE AN ELEMENT

```

void Delete (int X, List L)

{

position P;

P = Find (X, L);

If ( IsLast (P, L))

{

Temp = P;

P →Blink →Flink = NULL;

free (Temp);

}

else

```

```

{
Temp = P;
P →Blink→ Flink = P→Flink;
P →Flink →Blink = P→Blink;
free (Temp);
}
}

```

Advantage

- * Deletion operation is easier.
- * Finding the predecessor & Successor of a node is easier.

Disadvantage

- * More Memory Space is required since it has two pointers.

2.1.4 Circular Linked List

In circular linked list the pointer of the last node points to the first node. Circular linked list can be implemented as Singly linked list and Doubly linked list with or without headers.

Singly Linked Circular List

A singly linked circular list is a linked list in which the last node of the list points to the first node.

Doubly Linked Circular List

A doubly linked circular list is a Doubly linked list in which the forward link of the last node points to the first node and backward link of the first node points to the last node of the list.

Advantages of Circular Linked List

- It allows to traverse the list starting at any point.
- It allows quick access to the first and last records.

2.1.5 Applications of Linked List

1. Polynomial ADT
2. Radix Sort
3. Multilist

Polynomial ADT

We can perform the polynomial manipulations such as addition, subtraction and differentiation etc.

DECLARATION FOR LINKED LIST IMPLEMENTATION OF POLYNOMIAL ADT

Struct poly

```
{  
int coeff;  
int power;  
Struct poly *Next;  
}*list 1, *list 2, *list 3;
```

CREATION OF THE POLYNOMIAL

poly create (poly *head1, poly *newnode1)

```
{  
poly *ptr;  
if (head1 == NULL)  
{  
head1 = newnode1;  
return (head1);  
}
```

```

else
{
ptr = head1;
while (ptr → next != NULL)
ptr = ptr → next;
ptr → next = newnode1;
}
return (head1);
}

```

ADDITION OF TWO POLYNOMIALS

```

void add ( )
{
poly *ptr1, *ptr2, *newnode;
ptr1 = list1;
ptr2 = list2;
while (ptr1 != NULL && ptr2 != NULL)
{
newnode = malloc (sizeof (Struct poly));
if (ptr1 → power == ptr2 → power)
{
newnode → coeff = ptr1 → coeff + ptr2 → coeff;
newnode → power = ptr1 → power;
newnode → next = NULL;

```

```

list3 = create (list3, newnode);

ptr1 = ptr1→next;

ptr2 = ptr2→next;

}

else

{

if (ptr1→power > ptr2→ power)

{

newnode→coeff = ptr1→coeff;

newnode→power = ptr1→power;

newnode→next = NULL;

list3 = create (list3, newnode);

ptr1 = ptr1→next;

}

else

{

newnode→coeff = ptr2→coeff;

newnode→power = ptr2→power;

newnode→next = NULL;

list3 = create (list3, newnode);

ptr2 = ptr2→ next;

}

}

```

```
}
```

SUBTRACTION OF TWO POLYNOMIAL

```
void sub ( )
```

```
{
```

```
poly *ptr1, *ptr2, *newnode;
```

```
ptr1 = list1 ;
```

```
ptr2 = list 2;
```

```
while (ptr1!= NULL && ptr2!= NULL)
```

```
{
```

```
newnode = malloc (sizeof (Struct poly));
```

```
if (ptr1->power == ptr2->power)
```

```
{
```

```
newnode->coeff = (ptr1->coeff) - (ptr2->coeff);
```

```
newnode->power = ptr1->power;
```

```
newnode->next = NULL;
```

```
list3 = create (list 3, newnode);
```

```
ptr1 = ptr1->next;
```

```
ptr2 = ptr2->next;
```

```
}
```

```
else
```

```
{
```

```
if (ptr1->power > ptr2->power)
```

```
{
```

```

newnode→coeff = ptr1→coeff;
newnode→power = ptr1→power;
newnode→next = NULL;
list 3 = create (list 3, newnode);
ptr1 = ptr1→next;
}
else
{
newnode→coeff = - (ptr2→coeff);
newnode→power = ptr2→power;
newnode→next = NULL;
list 3 = create (list 3, newnode);
ptr2 = ptr2→next;
}
}
}
}
}

```

POLYNOMIAL DIFFERENTIATION

```

void diff ( )
{
poly *ptr1, *newnode;
ptr1 = list 1;
while (ptr1 != NULL)

```

```

{
newnode = malloc (sizeof (Struct poly));

newnode  coeff = ptr1  coeff * ptr1  power;

newnode  power = ptr1  power - 1;

newnode  next = NULL;

list 3 = create (list 3, newnode);

ptr1 = ptr1 → next;

}

}

```

Radix Sort : - (Or) Card Sort

Radix Sort is the generalised form of Bucket sort. It can be performed using buckets from 0 to 9.

In First Pass, all the elements are sorted according to the least significant bit.

In second pass, the numbers are arranged according to the next least significant bit and so on this process is repeated until it reaches the most significant bits of all numbers.

The numbers of passes in a Radix Sort depends upon the number of digits in the numbers given.

PASS 1 :

INPUT : 25, 256, 80, 10, 8, 15, 174, 187

Buckets

After Pass 1 : 80, 10, 174, 25, 15, 256, 187, 8

PASS 2 :

INPUT : 80, 10, 174, 25, 15, 256, 187, 8

After Pass 2 : 8, 10, 15, 25, 256, 174, 80, 187

PASS 3 :

INPUT : 8, 10, 15, 25, 256, 174, 80, 187

After pass 3 : 8, 10, 15, 25, 80, 175, 187, 256

Maximum number of digits in the given list is 3. Therefore the number of passes required to sort the list of elements is 3.

2.2 THE STACK ADT

2.2.1 Stack Model :

A stack is a linear data structure which follows Last In First Out (LIFO) principle, in which both insertion and deletion occur at only one end of the list called the Top.

Example : -

Pile of coins., a stack of trays in cafeteria.

2.2.2 Operations On Stack

The fundamental operations performed on a stack are

1. Push
2. Pop

PUSH :

The process of inserting a new element to the top of the stack. For every push operation the top is incremented by 1.

POP :

The process of deleting an element from the top of stack is called pop operation. After every pop operation the top pointer is decremented by 1.

EXCEPTIONAL CONDITIONS

OverFlow

Attempt to insert an element when the stack is full is said to be overflow.

UnderFlow

Attempt to delete an element, when the stack is empty is said to be underflow.

2.2.3 Implementation of Stack

Stack can be implemented using arrays and pointers.

Array Implementation

In this implementation each stack is associated with a top pointer, which is -1 for an empty stack.

- To push an element X onto the stack, Top Pointer is incremented and then set Stack [Top] = X.
- To pop an element, the stack [Top] value is returned and the top pointer is decremented.
- pop on an empty stack or push on a full stack will exceed the array bounds.

ROUTINE TO PUSH AN ELEMENT ONTO A STACK

```
void push (int x, Stack S)
```

```
{
```

```
if (IsFull (S))
```

```
Error ("Full Stack");
```

```
else
```

```
{
```

```
Top = Top + 1;
```

```
S[Top] = X;
```

```
}
```

```
}
```

```
int IsFull (Stack S)
```

```
{
```

```
if (Top == Arraysize)
```

```
return (1);
```

```
}
```

ROUTINE TO POP AN ELEMENT FROM THE STACK

```
void pop (Stack S)
```

```
{
```

```
if (IsEmpty (S))
```

```
Error ("Empty Stack");
```

```
else
```

```
{
```

```
X = S [Top];
```

```
Top = Top - 1;
```

```
}
```

```
}
```

```
int IsEmpty (Stack S)
```

```
{
```

```
if (S  Top == -1)
```

```
return (1);
```

```
}
```

ROUTINE TO RETURN TOP ELEMENT OF THE STACK

```
int TopElement (Stack S)
```

```
{
```

```
if (! IsEmpty (s))
```

```
return S[Top];
```

```
else  
  
Error ("Empty Stack");  
  
return 0;  
  
}
```

LINKED LIST IMPLEMENTATION OF STACK

- Push operation is performed by inserting an element at the front of the list.
- Pop operation is performed by deleting at the front of the list.
- Top operation returns the element at the front of the list.

DECLARATION FOR LINKED LIST IMPLEMENTATION

```
Struct Node;  
  
typedef Struct Node *Stack;  
  
int IsEmpty (Stack S);  
  
Stack CreateStack (void);  
  
void MakeEmpty (Stack S);  
  
void push (int X, Stack S);  
  
int Top (Stack S);  
  
void pop (Stack S);  
  
Struct Node  
{  
  
int Element ;  
  
Struct Node *Next;  
  
};
```

ROUTINE TO CHECK WHETHER THE STACK IS EMPTY

```

int IsEmpty (Stack S)
{
if (S→Next == NULL)
return (1);
}

```

ROUTINE TO CREATE AN EMPTY STACK

```

Stack CreateStack ( )
{
Stack S;
S = malloc (Sizeof (Struct Node));
if (S == NULL)
Error (" Outof Space");
MakeEmpty (s);
return S;
}

void MakeEmpty (Stack S)
{
if (S == NULL)
Error (" Create Stack First");
else
while (! IsEmpty (s))
pop (s);
}

```

ROUTINE TO PUSH AN ELEMENT ONTO A STACK

```
void push (int X, Stack S)
{
    Struct Node * Tempcell;
    Tempcell = malloc (sizeof (Struct Node));
    If (Tempcell == NULL)
        Error ("Out of Space");
    else
    {
        Tempcell → Element = X;
        Tempcell → Next = S → Next;
        S → Next = Tempcell;
    }
}
```

ROUTINE TO RETURN TOP ELEMENT IN A STACK

```
int Top (Stack S)
{
    If (! IsEmpty (s))
        return S → Next → Element;
    Error ("Empty Stack");
    return 0;
}
```

ROUTINE TO POP FROM A STACK

```

void pop (Stack S)
{
    Struct Node *Tempcell;
    If (IsEmpty (S))
        Error ("Empty Stack");
    else
    {
        Tempcell = S→Next;
        S→Next = S→Next→Next;
        Free (Tempcell);
    }
}

```

2.2.4 APPLICATIONS OF STACK

Some of the applications of stack are :

- (i) Evaluating arithmetic expression
- (ii) Balancing the symbols
- (iii) Towers of Hanoi
- (iv) Function Calls.
- (v) 8 Queen Problem.

Different Types of Notations To Represent Arithmetic Expression

There are 3 different ways of representing the algebraic expression.

They are

* INFIX NOTATION

* POSTFIX NOTATION

* PREFIX NOTATION

INFIX

In Infix notation, The arithmetic operator appears between the two operands to which it is being applied.

For example : - $A / B + C$

POSTFIX

The arithmetic operator appears directly after the two operands to which it applies. Also called reverse polish notation. $((A/B) + C)$

For example : - $AB / C +$

PREFIX

The arithmetic operator is placed before the two operands to which it applies. Also called as polish notation. $((A/B) + C)$

For example : - $+ / ABC$

INFIX PREFIX (or) POLISH POSTFIX (or) REVERSE

POLISH

1. $(A + B) / (C - D) / + AB - CD AB + CD - /$
2. $A + B * (C - D) + A * B - CD ABCD - * +$
3. $X * A / B - D - / * XABD X A * B / D -$
4. $X + Y * (A - B) / + X / * Y - AB - CD XYAB - * CD - / +$
 $(C - D)$
5. $A * B / C + D + / * ABCD AB * C / D +$

1. Evaluating Arithmetic Expression

To evaluate an arithmetic expressions, first convert the given infix expression to postfix expression and then evaluate the postfix expression using stack.

Infix to Postfix Conversion

Read the infix expression one character at a time until it encounters the delimiter. "#"

Step 1 : If the character is an operand, place it on to the output.

Step 2 : If the character is an operator, push it onto the stack. If the stack operator has a higher

or equal priority than input operator then pop that operator from the stack and place it onto the output.

Step 3 : If the character is a left parenthesis, push it onto the stack.

Step 4 : If the character is a right parenthesis, pop all the operators from the stack till it encounters left parenthesis, discard both the parenthesis in the output.

Evaluating Postfix Expression

Read the postfix expression one character at a time until it encounters the delimiter '#'.

Step 1 : - If the character is an operand, push its associated value onto the stack.

Step 2 : - If the character is an operator, POP two values from the stack, apply the operator to

them and push the result onto the stack.

Recursive Solution

N - represents the number of disks.

Step 1. If $N = 1$, move the disk from A to C.

Step 2. If $N = 2$, move the 1st disk from A to B.

Then move the 2nd disk from A to C,

The move the 1st disk from B to C.

Step 3. If $N = 3$, Repeat the step (2) to move the first 2 disks from A to B using C as intermediate.

Then the 3rd disk is moved from A to C. Then repeat the step (2) to move 2 disks from B to C using A as intermediate.

In general, to move N disks. Apply the recursive technique to move N - 1 disks from A to B using C as an intermediate. Then move the Nth disk from A to C. Then again apply the recursive technique to move N - 1 disks from B to C using A as an intermediate.

RECURSIVE ROUTINE FOR TOWERS OF HANOI

```
void hanoi (int n, char s, char d, char i)
{
/* n    no. of disks, s    source, d    destination i    intermediate */
if (n == 1)
{
print (s, d);
return;
}
else
{
hanoi (n - 1, s, i, d);
print (s, d)
hanoi (n-1, i, d, s);
return;
}
}
```

Function Calls

When a call is made to a new function all the variables local to the calling routine need to be saved, otherwise the new function will overwrite the calling routine variables. Similarly the current location address in the routine must be saved so that the new function knows where to go after it is completed.

RECURSIVE FUNCTION TO FIND FACTORIAL : -

```
int fact (int n)

{

int s;

if (n == 1)

return (1);

else

s = n * fact (n - 1);

return (s);

}
```

2.3 The Queue ADT

2.3.1 Queue Model

A Queue is a linear data structure which follows First In First Out (FIFO) principle, in which insertion is performed at rear end and deletion is performed at front end.

Example : Waiting Line in Reservation Counter,

2.3.2 Operations on Queue

The fundamental operations performed on queue are

1. Enqueue
2. Dequeue

Enqueue :

The process of inserting an element in the queue.

Dequeue :

The process of deleting an element from the queue.

Exception Conditions

Overflow : Attempt to insert an element, when the queue is full is said to be overflow condition.

Underflow : Attempt to delete an element from the queue, when the queue is empty is said to be

underflow.

2.3.3 Implementation of Queue

Queue can be implemented using arrays and pointers.

Array Implementation

In this implementation queue Q is associated with two pointers namely rear pointer and front pointer.

To insert an element X onto the Queue Q, the rear pointer is incremented by 1 and then set

Queue [Rear] = X

To delete an element, the Queue [Front] is returned and the Front Pointer is incremented by 1.

ROUTINE TO ENQUEUE

```
void Enqueue (int X)
```

```
{
```

```
if (rear >= max _ Arraysize)
```

```
print (" Queue overflow");
```

```
else
```

```
{
```

```
Rear = Rear + 1;
```

```
Queue [Rear] = X;
```

```
}
```

```
}
```

ROUTINE FOR DEQUEUE

```
void delete ( )
```

```
{
```

```
if (Front < 0)
```

```
print (" Queue Underflow");
```

```
else
```

```
{
```

```
X = Queue [Front];
```

```
if (Front == Rear)
```

```
{
```

```
Front = 0;
```

```
Rear = -1;
```

```
}
```

```
else
```

```
Front = Front + 1 ;
```

```
}
```

```
}
```

In Dequeue operation, if Front = Rear, then reset both the pointers to their initial values. (i.e. F = 0, R = -1)

Linked List Implementation of Queue

Enqueue operation is performed at the end of the list.

Dequeue operation is performed at the front of the list.

Queue ADT

DECLARATION FOR LINKED LIST IMPLEMENTATION OF QUEUE ADT

```
Struct Node;
```

```
typedef Struct Node * Queue;
```

```
int IsEmpty (Queue Q);
```

```
Queue CreateQueue (void);
```

```
void MakeEmpty (Queue Q);
```

```
void Enqueue (int X, Queue Q);
```

```
void Dequeue (Queue Q);
```

```
Struct Node
```

```
{
```

```
int Element;
```

```
Struct Node *Next;
```

```
}* Front = NULL, *Rear = NULL;
```

ROUTINE TO CHECK WHETHER THE QUEUE IS EMPTY

```
int IsEmpty (Queue Q) // returns boolean value /
```

```
{ // if Q is empty
```

```
if (Q→Next == NULL) // else returns 0
```

```
return (1);
```

```
}
```

ROUTINE TO CHECK AN EMPTY QUEUE

```
Struct CreateQueue ( )  
  
{  
  
Queue Q;  
  
Q = Malloc (Sizeof (Struct Node));  
  
if (Q == NULL)  
  
Error ("Out of Space");  
  
MakeEmpty (Q);  
  
return Q;  
  
}  
  
void MakeEmpty (Queue Q)  
  
{  
  
if (Q == NULL)  
  
Error ("Create Queue First");  
  
else  
  
while (! IsEmpty (Q)  
  
Dequeue (Q);  
  
}
```

ROUTINE TO ENQUEUE AN ELEMENT IN QUEUE

```
void Enqueue (int X)  
  
{  
  
Struct node *newnode;  
  
newnode = Malloc (sizeof (Struct node));
```

```

if (Rear == NULL)
{
newnode →data = X;
newnode →Next = NULL;
Front = newnode;
Rear = newnode;
}
else
{
newnode → data = X;
newnode → Next = NULL;
Rear →next = newnode;
Rear = newnode;
}
}

```

ROUTINE TO DEQUEUE AN ELEMENT FROM THE QUEUE

```

void Dequeue ( )
{
Struct node *temp;
if (Front == NULL)
Error("Queue is underflow");
else
{

```



```

temp = Front;

if (Front == Rear)

{

Front = NULL;

Rear = NULL;

}

else

Front = Front →Next;

Print (temp→data);

free (temp);

}

}

```

2.3.4 Double Ended Queue (DEQUE)

In Double Ended Queue, insertion and deletion operations are performed at both the ends.

2.3.5 Circular Queue

In Circular Queue, the insertion of a new element is performed at the very first location of the queue if the last location of the queue is full, in which the first element comes just after the last element.

Advantages

It overcomes the problem of unutilized space in linear queues, when it is implemented as arrays.

To perform the insertion of an element to the queue, the position of the element is calculated by the relation as

$\text{Rear} = (\text{Rear} + 1) \% \text{Maxsize}.$

and then set

Queue [Rear] = value.

ROUTINE TO INSERT AN ELEMENT IN CIRCULAR QUEUE

```
void CEnqueue (int X)
{
    if (Front == (rear + 1) % Maxsize)
        print ("Queue is overflow");
    else
    {
        if (front == -1)
            front = rear = 0;
        else
            rear = (rear + 1) % Maxsize;
        CQueue [rear] = X;
    }
}
```

To perform the deletion, the position of the Front pointer is calculated by the relation

Value = CQueue [Front]

Front = (Front + 1) % maxsize.

ROUTINE TO DELETE AN ELEMENT FROM CIRCULAR QUEUE

```
int CDequeue ( )
{
    if (front == -1)
        print ("Queue is underflow");
```

```

else

{

X = CQueue [Front];

if (Front == Rear)

Front = Rear = -1;

else

Front = (Front + 1)% maxsize;

}

return (X);

}

```

2.3.6 Priority Queues

Priority Queue is a Queue in which inserting an item or removing an item can be performed from any position based on some priority.

2.3.7 Applications of Queue

- * Batch processing in an operating system
- * To implement Priority Queues.
- * Priority Queues can be used to sort the elements using Heap Sort.
- * Simulation.
- * Mathematics user Queueing theory.
- * Computer networks where the server takes the jobs of the client as per the queue strategy.

Important Questions

Part - A

1. Define ADT.
2. What are the advantages of linked list over arrays?
3. What are the advantages of doubly linked list over singly linked list?
4. List the applications of List ADT.
5. Write a procedure for polynomial differentiation.
6. What are the operations performed on stack and write its exceptional condition?
7. What do you mean by cursor implementation of list?
8. List the application of stack
9. Convert the infix expression $a + b * c + (d * e + f) * g$ to its equivalent postfix expression and
prefix expression.
10. Convert the infix expression $(a * b) + ((c * g) - (e / f))$ to its equivalent polish and
reverse
polish expression.
11. Write the recursive routine to perform factorial of a given number.
12. Define Queue data structure and give some applications for it.
13. What is Deque?
14. What is Circular Queue?
15. What is Priority Queue?
16. Write a routine to return the top element of stack.
17. Write a routine to check IsEmpty and IsLast for queue.
18. Write a procedure to insert an element in a singly linked list

19. What is the principle of radix sort?

20. What is Multilist?

Part - B

1. Explain the array and linked list implementation of stack.

2. Explain the array and linked list implementation of Queue.

3. What are the various linked list operations? Explain

4. Explain how stack is applied for evaluating an arithmetic expression.

5. Write routines to implement addition, subtraction & differentiation of two polynomials.

6. Write the recursive routine for Towers of Hanoi.

7. Explain Cursor implementation of List?

8. Write the operations performed on singly linked list?

9. Write the insertion and deletion routine for doubly linked list?

10. Write the procedure for polynomial addition and differentiation?

UNIT II TREE STRUCTURES

Need for non-linear structures – Tree ADT – tree traversals – left child right sibling data structures for general trees – Binary Tree ADT – expression trees – applications of trees – binary search tree ADT

TREES

3.1 PRELIMINARIES :

TREE : A tree is a finite set of one or more nodes such that there is a specially designated

node called the Root, and zero or more non empty sub trees T_1, T_2, \dots, T_k , each of whose roots are connected by a directed edge from Root R.

The ADT tree

A tree is a finite set of elements or nodes. If the set is non-empty, one of the nodes is distinguished as the root node, while the remaining (possibly empty) set of nodes are grouped into subsets, each of which is itself a tree. This hierarchical relationship is described by referring to each such subtree as a child of the root, while the root is referred to as the parent of each subtree. If a tree consists of a single node, that node is called a leaf node.

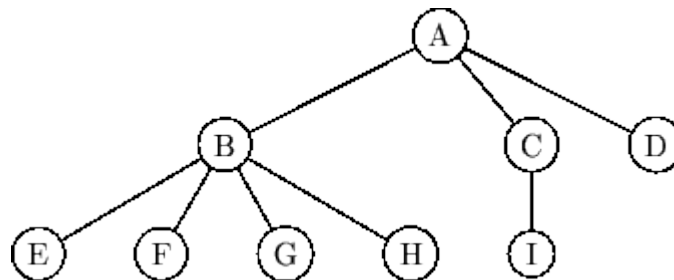


Figure 3.4: A simple tree.

It is a notational convenience to allow an empty tree. It is usual to represent a tree using a picture such as Fig. 3.4, in which the root node is A, and there are three subtrees rooted at B, C and D. The root of the subtree D is a leaf node, as are the remaining nodes, E, F, G, H and I. The node C has a single child I, while each of E, F, G and H have the same parent B. The subtrees rooted at a given node are taken to be *ordered*, so the tree in Fig. 3.4 is different from the one in which nodes E and F are interchanged. Thus it makes sense to say that the *first* subtree at A has 4 leaf nodes.

Example 3.4 Show how to implement the Abstract Data Type tree using lists.

Solution We write $[A\ B\ C]$ for the list containing three elements, and distinguish A from $[A]$. We can represent a tree as a list consisting of the root and a list of the subtrees in order. Thus the list-based representation of the tree in Fig 3.4 is

$[A\ [[B\ [[E]\ [F]\ [G]\ [H]]]\ [C\ [I]]\ [D]]]$.

ROOT : A node which doesn't have a parent. In the above tree.

NODE : Item of Information.

LEAF : A node which doesn't have children is called leaf or Terminal node.

SIBLINGS : Children of the same parents are said to be siblings,. F, G are siblings.

PATH : A path from node n_i to n_k is defined as a sequence of nodes $n_1, n_2, n_3, \dots, n_k$ such that n_i is the parent of n_{i+1} . for . There is exactly only one path from each node to root.

LENGTH : The length is defined as the number of edges on the path.

DEGREE : The number of subtrees of a node is called its degree.

3.2 BINARY TREE

Definition :-

Binary Tree is a tree in which no node can have more than two children.

Maximum number of nodes at level i of a binary tree is 2^{i-1} .

A **binary tree** is a tree which is either empty, or one in which every node:

- has no children; or
- has just a left child; or
- has just a right child; or
- has both a left and a right child.

A **complete** binary tree is a special case of a binary tree, in which all the levels, except perhaps the last, are full; while on the last level, any missing nodes are to the right of all the nodes that are present. An example is shown in Fig. 3.5.

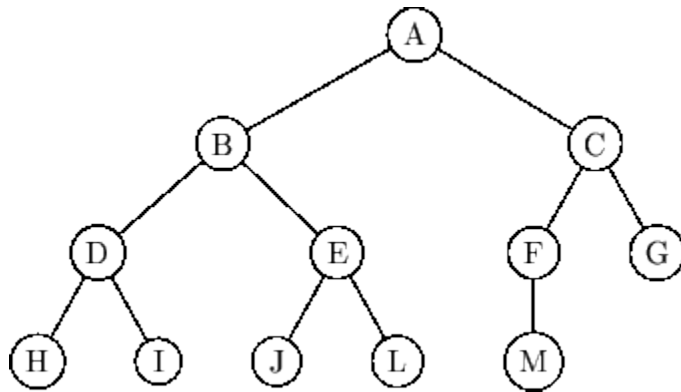


Figure 3.5: A complete binary tree: the only "missing" entries can be on the last row.

Example 3.5 Give a space - efficient implementation of a complete binary tree in terms of an array A. Describe how to pass from a parent to its two children, and vice-versa

Solution An obvious one, in which no space is wasted, stores the root of the tree in $A[1]$; the two children in $A[2]$ and $A[3]$, the next generation at $A[4]$ up to $A[7]$ and so on. An element $A[k]$ has children at $A[2k]$ and $A[2k+1]$, providing they both exist, while the parent of node $A[k]$ is at $A[k \div 2]$. Thus traversing the tree can be done very efficiently.

BINARY TREE NODE DECLARATIONS

Struct TreeNode

{

int Element;

Struct TreeNode *Left ;

Struct TreeNode *Right;

};

COMPARISON BETWEEN

GENERAL TREE & BINARY TREE

General Tree Binary Tree

* General Tree has any * A Binary Tree has not

number of children. more than two children.

FULL BINARY TREE :-

A full binary tree of height h has $2^{h+1} - 1$ nodes.

Here height is 3 No. of nodes in full

binary tree is $= 2^{3+1} - 1$

$= 15$ nodes.

COMPLETE BINARY TREE :

A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes. In the bottom level the elements should be filled from left to right.

3.2.1 REPRESENTATION OF A BINARY TREE

There are two ways for representing binary tree, they are

- * Linear Representation

- * Linked Representation

Linear Representation

The elements are represented using arrays. For any element in position i , the left child is in position $2i$, the right child is in position $(2i + 1)$, and the parent is in position $(i/2)$.

Linked Representation

The elements are represented using pointers. Each node in linked representation has three fields, namely,

- * Pointer to the left subtree

- * Data field

- * Pointer to the right subtree

In leaf nodes, both the pointer fields are assigned as NULL.

3.2.2 EXPRESSION TREE

Expression Tree is a binary tree in which the leaf nodes are operands and the interior nodes are operators. Like binary tree, expression tree can also be traversed by inorder, preorder and postorder traversal.

Constructing an Expression Tree

Let us consider postfix expression given as an input for constructing an expression tree by performing the following steps :

1. Read one symbol at a time from the postfix expression.
2. Check whether the symbol is an operand or operator.
 - (a) If the symbol is an operand, create a one - node tree and push a pointer on to the stack.
 - (b) If the symbol is an operator pop two pointers from the stack namely T_1 and T_2 and form a new tree with root as the operator and T_2 as a left child and T_1 as a right child.

A pointer to this new tree is then pushed onto the stack.

3.3 The Search Tree ADT : - Binary Search Tree

Definition : -

Binary search tree is a binary tree in which for every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X, and the values of all the keys in its right subtree are larger than the key value in X.

Comparison Between Binary Tree & Binary Search Tree

Binary Tree Binary Search Tree

* A tree is said to be a binary tree if it has at most two children. the key values in the left node is less than the root and the key values in the right node is greater than the root.

* It doesn't have any order.

Note : * Every binary search tree is a binary tree.

* All binary trees need not be a binary search tree.

DECLARATION ROUTINE FOR BINARY SEARCH TREE

```
Struct TreeNode;
```

```
typedef struct TreeNode * SearchTree;
```

```
SearchTree Insert (int X, SearchTree T);
```

```
SearchTree Delete (int X, SearchTree T);
```

```
int Find (int X, SearchTree T);
```

```
int FindMin (SearchTree T);
```

```
int FindMax (SearchTree T);
```

```
SearchTree MakeEmpty (SearchTree T);
```

```
Struct TreeNode
```

```
{
```

```
int Element ;
```

```
SearchTree Left;
```

```
SearchTree Right;
```

```
};
```

Make Empty :-

This operation is mainly for initialization when the programmer prefer to initialize the first element as a one - node tree.

ROUTINE TO MAKE AN EMPTY TREE :-

```
SearchTree MakeEmpty (SearchTree T)
```

```
{
```

```
if (T != NULL)
```

```
{
```

```
MakeEmpty (T->Left);
```

```
MakeEmpty (T->Right);
```

```
free (T);
```

```
}
```

```
return NULL ;  
  
}
```

Insert : -

To insert the element X into the tree,

- * Check with the root node T

- * If it is less than the root,

Traverse the left subtree recursively until it reaches

the T → left equals to NULL. Then X is placed in

T → left.

- * If X is greater than the root.

Traverse the right subtree recursively until it reaches

the T → right equals to NULL. Then x is placed in

T → Right.

ROUTINE TO INSERT INTO A BINARY SEARCH TREE

SearchTree Insert (int X, searchTree T)

```
{  
if (T == NULL)  
{  
T = malloc (size of (Struct TreeNode));  
if (T != NULL) // First element is placed in the root.  
{  
T → Element = X;  
T → left = NULL;
```

```

T →Right = NULL;

}

}

else

if (X < T →Element)

T →left = Insert (X, T →left);

else

if (X > T →Element)

T →Right = Insert (X, T →Right);

// Else X is in the tree already.

return T;

}

```

Example : -

To insert 8, 5, 10, 15, 20, 18, 3

* First element 8 is considered as Root.

As $5 < 8$, Traverse towards left

$10 > 8$, Traverse towards Right.

Similarly the rest of the elements are traversed.

Find : -

* Check whether the root is NULL if so then return NULL.

* Otherwise, Check the value X with the root node value (i.e. T →data)

(1) If X is equal to T →data, return T.

(2) If X is less than T →data, Traverse the left of T recursively.

(3) If X is greater than T data, traverse the right of T recursively.

ROUTINE FOR FIND OPERATION

Int Find (int X, SearchTree T)

```
{  
If T == NULL)  
  
Return NULL ;  
  
If (X < T → Element)  
return Find (X, T →left);  
  
else  
  
If (X > T → Element)  
return Find (X, T →Right);  
  
else  
  
return T; // returns the position of the search element.  
}
```

Example : - To Find an element 10 (consider, X = 10)

10 is checked with the Root $10 > 8$, Go to the right child of 8

10 is checked with Root 15 $10 < 15$, Go to the left child of 15.

10 is checked with root 10 (Found)

Find Min :

This operation returns the position of the smallest element in the tree.

To perform FindMin, start at the root and go left as long as there is a left child. The stopping point is the smallest element.

RECURSIVE ROUTINE FOR FINDMIN

int FindMin (SearchTree T)

```

{
if (T == NULL);

return NULL ;

else if (T →left == NULL)

return T;

else

return FindMin (T → left);

```

Example : -

Root T

T

(a) T! = NULL and T→left!=NULL, (b) T! = NULL and T→left!=NULL,

Traverse left Traverse left

Min T

(c) Since T →left is Null, return T as a minimum element.

NON - RECURSIVE ROUTINE FOR FINDMIN

```

int FindMin (SearchTree T)

```

```

{
if (T! = NULL)

while (T →Left != NULL)

T = T →Left ;

return T;

}

```

FindMax

FindMax routine return the position of largest elements in the tree. To perform a FindMax, start at the root and go right as long as there is a right child. The stopping point is the largest element.

RECURSIVE ROUTINE FOR FINDMAX

```
int FindMax (SearchTree T)
{
    if (T == NULL)
        return NULL ;
    else if (T →Right == NULL)
        return T;
    else FindMax (T →Right);
}
```

Example :-

Root T

(a) T! = NULL and T→Right!=NULL, (b) T! = NULL and T→Right!=NULL,

Traverse Right Traverse Right

Max

(c) Since T →Right is NULL, return T as a Maximum element.

NON - RECURSIVE ROUTINE FOR FINDMAX

```
int FindMax (SearchTree T)
{
    if (T! = NULL)
        while (T →Right != NULL)
            T = T →Right ;
}
```



```
return T ;  
  
}
```

Delete :

Deletion operation is the complex operation in the Binary search tree. To delete an element, consider the following three possibilities.

CASE 1 Node to be deleted is a leaf node (ie) No children.

CASE 2 Node with one child.

CASE 3 Node with two children.

CASE 1 Node with no children (Leaf node)

If the node is a leaf node, it can be deleted immediately.

Delete : 8

After the deletion

CASE 2 : - Node with one child

If the node has one child, it can be deleted by adjusting its parent pointer that points to its child node.

To Delete 5

before deletion After deletion

To delete 5, the pointer currently pointing the node 5 is now made to to its child node 6.

Case 3 : Node with two children

It is difficult to delete a node which has two children. The general strategy is to replace the data of the node to be deleted with its smallest data of the right subtree and recursively delete that node.

DELETION ROUTINE FOR BINARY SEARCH TREES

SearchTree Delete (int X, searchTree T)

```
{
```

```

int Tmpcell ;

if (T == NULL)

Error ("Element not found");

else

if (X < T →Element) // Traverse towards left

T →Left = Delete (X, T →Left);

else

if (X > T →Element) // Traverse towards right

T →Right = Delete (X, T →Right);

// Found Element to be deleted

else

// Two children

if (T →Left && T →Right)

{ // Replace with smallest data in right subtree

Tmpcell = FindMin (T →Right);

T →Element = Tmpcell →Element ;

T →Right = Delete (T →Element; T →Right);

}

else // one or zero children

{

Tmpcell = T;

if (T →Left == NULL)

T = T →Right;

```

```
else if (T→ Right == NULL)
```

```
T = T →Left ;
```

```
free (TmpCell);
```

```
}
```

```
return T;
```

```
}
```

UNIT III BALANCED SEARCH TREES AND INDEXING 9

AVL trees – Binary Heaps – B-Tree – Hashing – Separate chaining – open addressing –

Linear probing

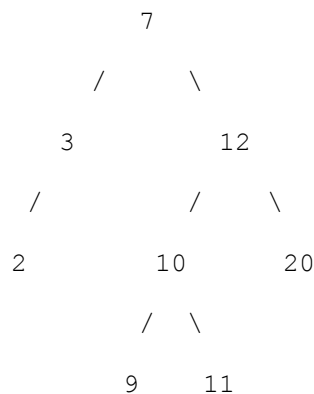
AVL Trees

The Concept

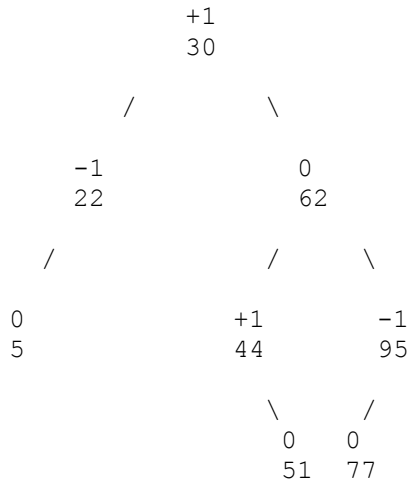
These are self-adjusting, height-balanced binary search trees and are named after the inventors: Adelson-Velskii and Landis. A *balanced* binary search tree has $\Theta(\lg n)$ height and hence $\Theta(\lg n)$ worst case lookup and insertion times. However, ordinary binary search trees have a bad worst case. When sorted data is inserted, the binary search tree is very unbalanced, essentially more of a linear list, with $\Theta(n)$ height and thus $\Theta(n)$ worst case insertion and lookup times. AVL trees overcome this problem.

Definitions

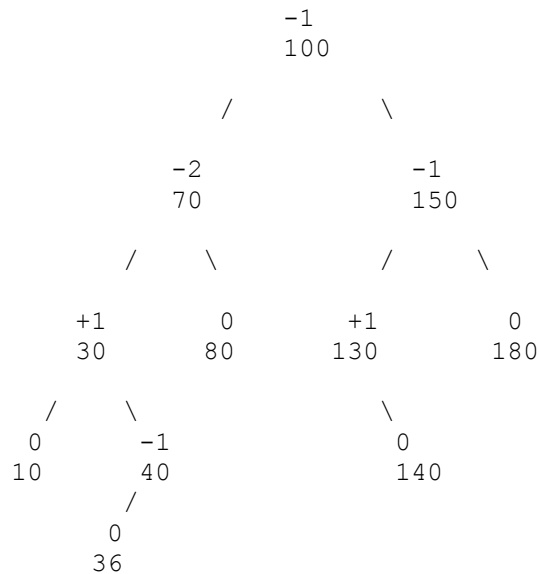
The **height** of a binary tree is the maximum path length from the root to a leaf. A single-node binary tree has height 0, and an empty binary tree has height -1. As another example, the following binary tree has height 3.



An **AVL tree** is a binary search tree in which every node is **height balanced**, that is, the difference in the heights of its two subtrees is at most 1. The **balance factor** of a node is the height of its right subtree minus the height of its left subtree. An equivalent definition, then, for an AVL tree is that it is a binary search tree in which each node has a balance factor of -1, 0, or +1. Note that a balance factor of -1 means that the subtree is left-heavy, and a balance factor of +1 means that the subtree is right-heavy. For example, in the following AVL tree, note that the root node with balance factor +1 has a right subtree of height 1 more than the height of the left subtree. (The balance factors are shown at the top of each node.)



The idea is that an AVL tree is close to being completely balanced. Hence it should have $\Theta(\lg n)$ height (it does - always) and so have $\Theta(\lg n)$ worst case insertion and lookup times. An AVL tree does not have a bad worst case, like a binary search tree which can become very unbalanced and give $\Theta(n)$ worst case lookup and insertion times. The following binary search tree is **not** an AVL tree. Notice the balance factor of -2 at node 70.

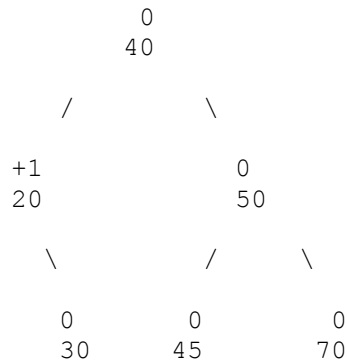


Inserting a New Item

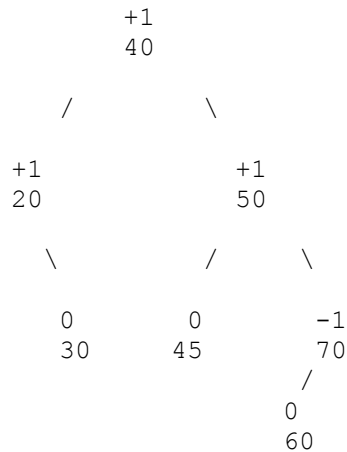
Initially, a new item is inserted just as in a binary search tree. Note that the item always goes into a new leaf. The tree is then readjusted as needed in order to maintain it as an AVL tree. There are three main cases to consider when inserting a new node.

Case 1:

A node with balance factor 0 changes to +1 or -1 when a new node is inserted below it. No change is needed at this node. Consider the following example. Note that after an insertion one only needs to check the balances along the path from the new leaf to the root.

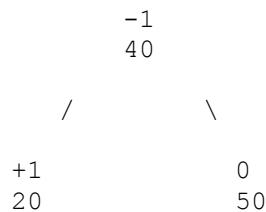


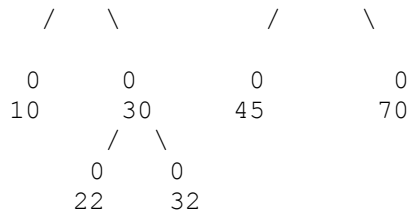
After inserting 60 we get:



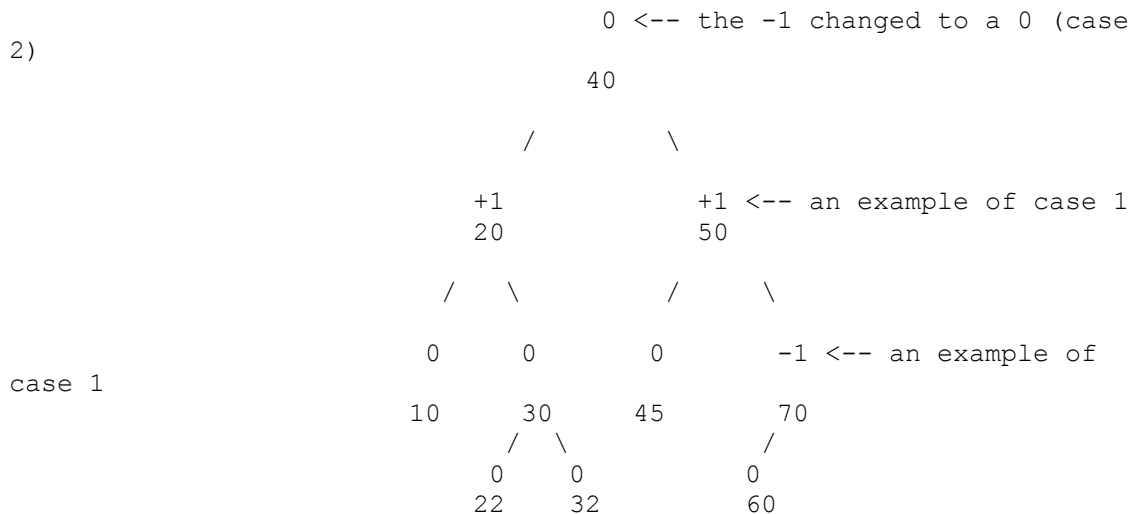
Case 2:

A node with balance factor -1 changes to 0 when a new node is inserted in its right subtree. (Similarly for +1 changing to 0 when inserting in the left subtree.) No change is needed at this node. Consider the following example.





After inserting 60 we get:

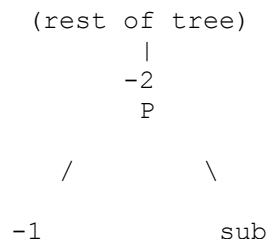


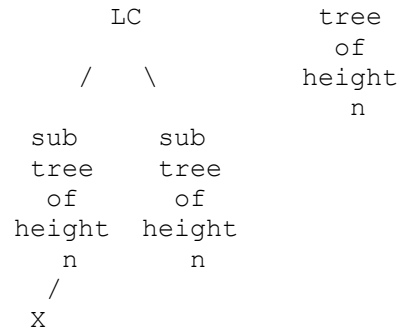
Case 3:

A node with balance factor -1 changes to -2 when a new node is inserted in its left subtree. (Similarly for +1 changing to +2 when inserting in the right subtree.) Change is needed at this node. The tree is restored to an AVL tree by using a rotation.

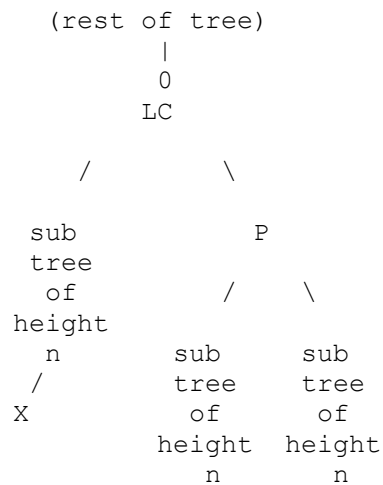
Subcase A:

This consists of the following situation, where P denotes the parent of the subtree being examined, LC is P's left child, and X is the new node added. Note that inserting X makes P have a balance factor of -2 and LC have a balance factor of -1. The -2 must be fixed. This is accomplished by doing a right rotation at P. Note that rotations do not mess up the order of the nodes given in an inorder traversal. This is very important since it means that we still have a legitimate binary search tree. (Note, too, that the mirror image situation is also included under subcase A.)

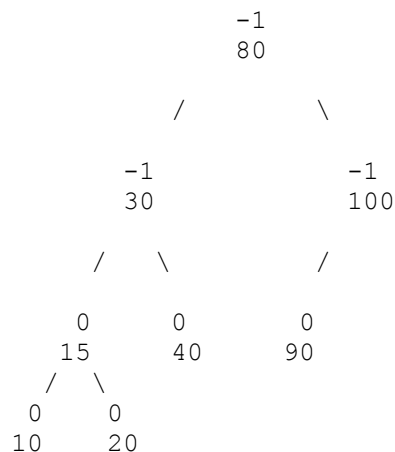




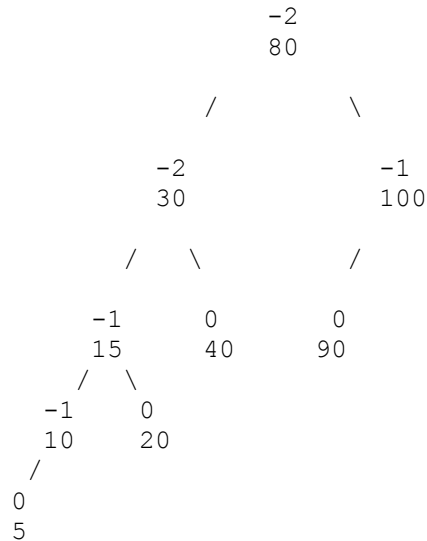
The fix is to use a single right rotation at node P. (In the mirror image case a single left rotation is used at P.) This gives the following picture.



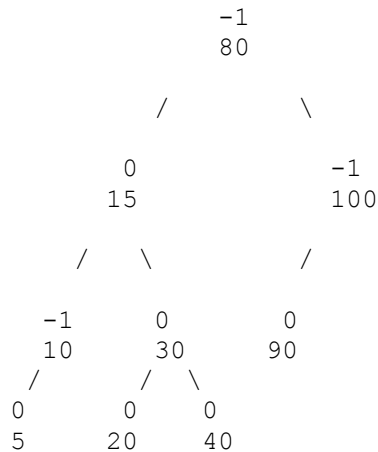
Consider the following more detailed example that illustrates subcase A.



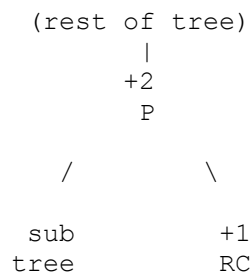
We then insert 5 and then check the balance factors from the new leaf up toward the root. (Always check from the bottom up.)

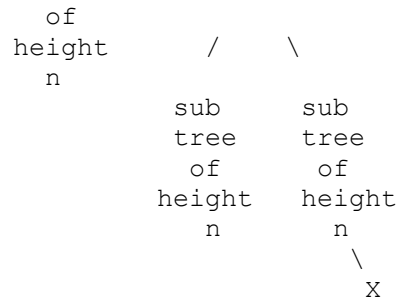


This reveals a balance factor of -2 at node 30 that must be fixed. (Since we work bottom up, we reach the -2 at 30 first. The other -2 problem will go away once we fix the problem at 30.) The fix is accomplished with a right rotation at node 30, leading to the following picture.



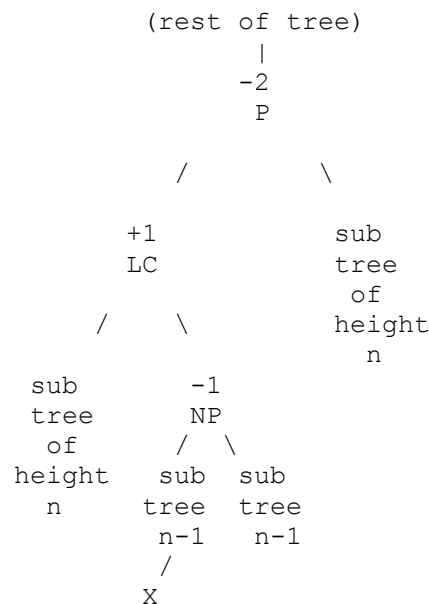
Recall that the mirror image situation is also included under subcase A. The following is a general illustration of this situation. The fix is to use a single left rotation at P. See if you can draw a picture of the following after the left rotation at P. Then draw a picture of a particular example that fits our general picture below and fix it with a left rotation.



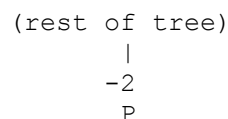


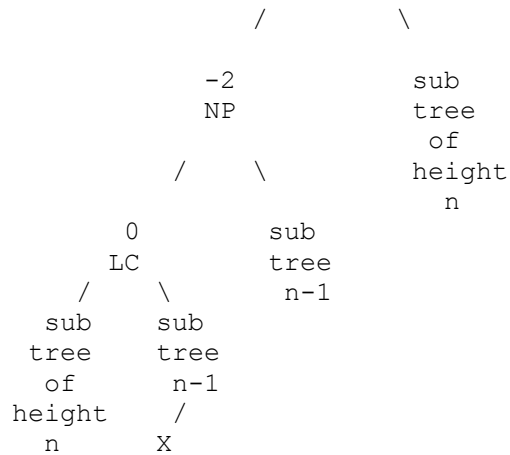
Subcase B:

This consists of the following situation, where P denotes the parent of the subtree being examined, LC is P's left child, NP is the node that will be the new parent, and X is the new node added. X might be added to either of the subtrees of height $n-1$. Note that inserting X makes P have a balance factor of -2 and LC have a balance factor of +1. The -2 must be fixed. This is accomplished by doing a double rotation at P (explained below). (Note that the mirror image situation is also included under subcase B.)

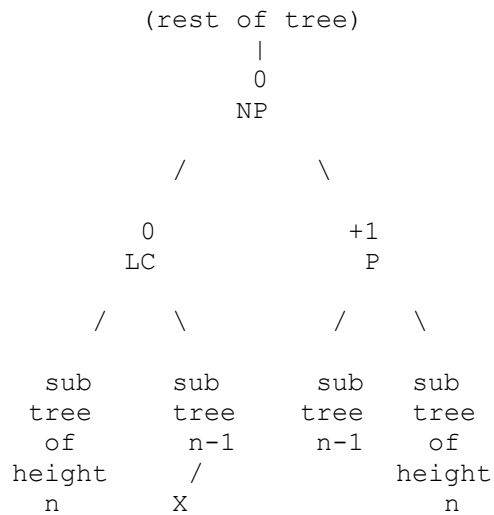


The fix is to use a double right rotation at node P. A double right rotation at P consists of a single **left** rotation at LC followed by a single right rotation at P. (In the mirror image case a double left rotation is used at P. This consists of a single right rotation at the right child RC followed by a single left rotation at P.) In the above picture, the double rotation gives the following (where we first show the result of the left rotation at LC, then a new picture for the result of the right rotation at P).

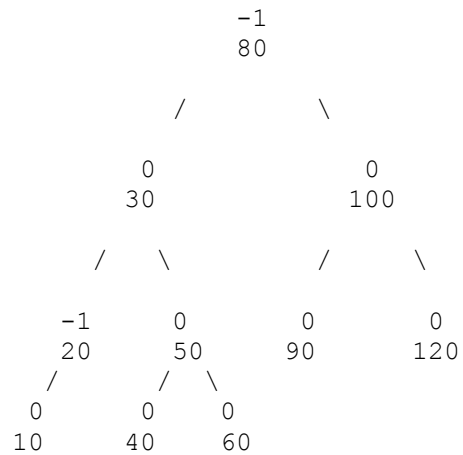




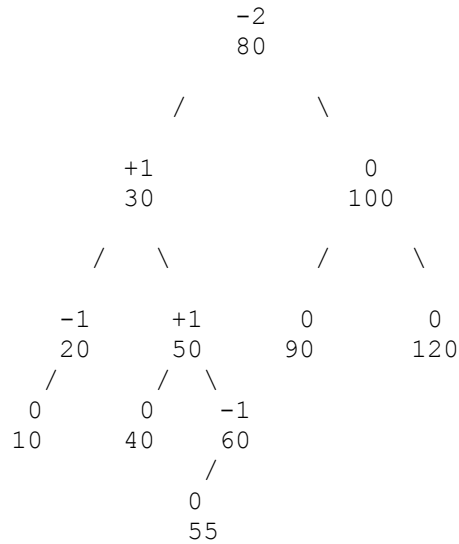
Finally we have the following picture after doing the right rotation at P.



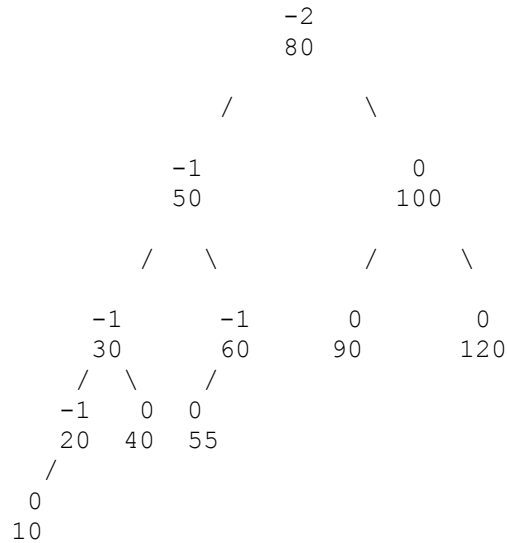
Consider the following concrete example of subcase B.



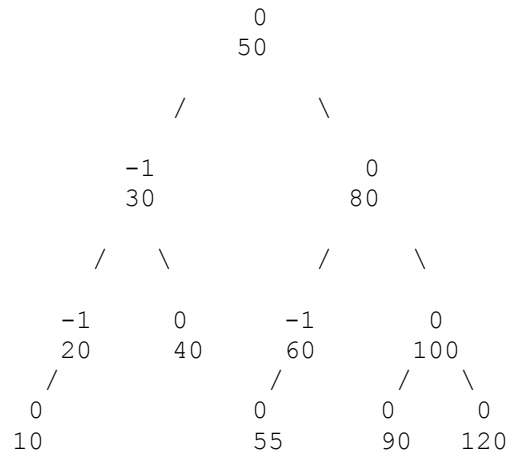
After inserting 55, we get a problem, a balance factor of -2 at the root node, as seen below.



As discussed above, this calls for a double rotation. First we do a single left rotation at 30. This gives the following picture.



Finally, the right rotation at 80 restores the binary search tree to be an AVL tree. The resulting picture is shown below.



BINARY HEAP

The efficient way of implementing priority queue is Binary Heap. Binary heap is merely referred as Heaps, Heap have two properties namely

- * Structure property
- * Heap order property.

Like AVL trees, an operation on a heap can destroy one of the properties, so a heap operation must not terminate until all heap properties are in order. Both the operations require the average running time as $O(\log N)$.

Structure Property

A heap should be complete binary tree, which is a completely filled binary tree with the possible exception of the bottom level, which is filled from left to right.

A complete binary tree of height H has between 2^H and $2^{H+1} - 1$ nodes.

For example if the height is 3. Then the number of nodes will be between 8 and 15. (ie) $(2^3 \text{ and } 2^4 - 1)$.

For any element in array position i , the left child is in position $2i$, the right child is in position $2i + 1$, and the parent is in $i/2$. As it is represented as array it doesn't require

pointers and also the operations required to traverse the tree are extremely simple and fast. But the only disadvantage is to specify the maximum heap size in advance.

Heap Order Property

In a heap, for every node X, the key in the parent of X is smaller than (or equal to) the key in X, with the exception of the root (which has no parent).

This property allows the deleteMin operations to be performed quickly has the minimum element can always be found at the root. Thus, we get the FindMin operation in constant time.

Declaration for priority queue

```
Struct Heapstruct;

typedef struct Heapstruct * priority queue;

PriorityQueue Initialize (int MaxElements);

void insert (int X, PriorityQueue H);

int DeleteMin (PriorityQueue H);

Struct Heapstruct
{
    int capacity;
    int size;
    int *Elements;
};
```

Initialization

```
PriorityQueue Initialize (int MaxElements)
{
    PriorityQueue H;

    H = malloc (sizeof (Struct Heapstruct));
```

```
H → Capacity = MaxElements;
```

```
H → size = 0;
```

```
H → elements [0] = MinData;
```

```
return H;
```

BASIC HEAP OPERATIONS

To perform the insert and DeleteMin operations ensure that the heap order property is maintained.

Insert Operation

To insert an element X into the heap, we create a hole in the next available location, otherwise the tree will not be complete. If X can be placed in the hole without violating heap order, then place the element X there itself. Otherwise, we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. This process continues until X can be placed in the hole. This general strategy is known as Percolate up, in which the new element is percolated up the heap until the correct location is found.

Routine To Insert Into A Binary Heap

```
void insert (int X, PriorityQueue H)
```

```
{
```

```
int i;
```

```
If (Isfull (H))
```

```
{
```

```
Error (" priority queue is full");
```

```
return;
```

```
}
```

```
for (i = ++H → size; H → Elements [i/2] > X; i/=2)
```

```
/* If the parent value is greater than X, then place the element of parent
```

```
node into the hole */.
```

H → Elements [i] = H → Elements [i/2];

H → elements [i] = X; // otherwise, place it in the hole.

}

DeleteMin

DeleteMin Operation is deleting the minimum element from the Heap.

In Binary heap the minimum element is found in the root. When this minimum is removed, a hole is created at the root. Since the heap becomes one smaller, makes the last element X in the heap to move somewhere in the heap.

If X can be placed in hole without violating heaporder property place it.

Otherwise, we slide the smaller of the hole's children into the hole, thus pushing the hole down one level. We repeat until X can be placed in the hole. This general strategy is known as percolate down.

ROUTINE TO PERFORM DELETEMIN IN A BINARY HEAP

```
int Deletemin (PriorityQueue H)
```

```
{
```

```
int i, child;
```

```
int MinElement, LastElement;
```

```
if (IsEmpty (H))
```

```
{
```

```
Error ("Priority queue is Empty");
```

```
return H → Elements [0];
```

```
}
```

```
MinElement = H → Elements [1];
```

```
LastElement = H → Elements [H → size - -];
```

```
for (i = 1; i * 2 <= H → size; i = child)
```



```

{
/* Find Smaller Child */

child = i * 2;

if (child != H → size && H → Elements [child + 1]
< H → Elements [child])

child ++;

// Percolate one level down

if (LastElement > H → Elements [child])

H → Elements [i] = H → Elements [child];

else

break ;

}

H → Elements [i] = LastElement;

return MinElement;

}

```

OTHER HEAP OPERATIONS

The other heap operations are

- (i) Decrease - key
- (ii) Increase - key
- (iii) Delete
- (iv) Build Heap

DECREASE KEY

The Decreasekey (P, Δ, H) operation decreases the value of the key at position P by a positive amount Δ . This may violate the heap order property, which can be fixed by percolate up.

Increase - Key

The increase - key (p, Δ, H) operation increases the value of the key at position p by a positive amount Δ . This may violate heap order property, which can be fixed by percolate down.

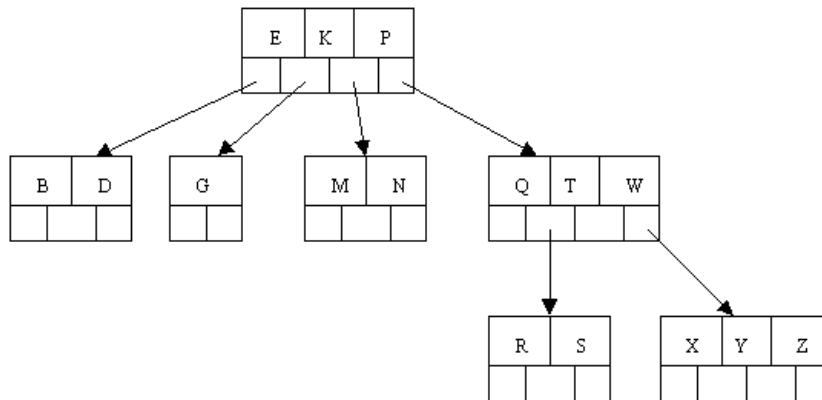
B-Trees

Introduction

A B-tree is a specialized multiway tree designed especially for use on disk. In a B-tree each node may contain a large number of keys. The number of subtrees of each node, then, may also be large. A B-tree is designed to branch out in this large number of directions and to contain a lot of keys in each node so that the height of the tree is relatively small. This means that only a small number of nodes must be read from disk to retrieve an item. The goal is to get fast access to the data, and with disk drives this means reading a very small number of records. Note that a large node size (with lots of keys in the node) also fits with the fact that with a disk drive one can usually read a fair amount of data at once.

Definitions

A *multiway tree of order m* is an ordered tree where each node has at most m children. For each node, if k is the actual number of children in the node, then $k - 1$ is the number of keys in the node. If the keys and subtrees are arranged in the fashion of a search tree, then this is called a *multiway search tree of order m* . For example, the following is a multiway search tree of order 4. Note that the first row in each node shows the keys, while the second row shows the pointers to the child nodes. Of course, in any useful application there would be a record of data associated with each key, so that the first row in each node might be an array of records where each record contains a key and its associated data. Another approach would be to have the first row of each node contain an array of records where each record contains a key and a record number for the associated data record, which is found in another file. This last method is often used when the data records are large. The example software will use the first method.



What does it mean to say that the keys and subtrees are "arranged in the fashion of a search tree"? Suppose that we define our nodes as follows:

```

typedef struct
{
    int Count;           // number of keys stored in the current node
    ItemType Key[3];     // array to hold the 3 keys
    long Branch[4];      // array of fake pointers (record numbers)
} NodeType;
  
```

Then a multiway search tree of order 4 has to fulfill the following conditions related to the ordering of the keys:

- The keys in each node are in ascending order.
- At every given node (call it Node) the following is true:
 - The subtree starting at record Node.Branch[0] has only keys that are less than Node.Key[0].
 - The subtree starting at record Node.Branch[1] has only keys that are greater than Node.Key[0] and at the same time less than Node.Key[1].
 - The subtree starting at record Node.Branch[2] has only keys that are greater than Node.Key[1] and at the same time less than Node.Key[2].
 - The subtree starting at record Node.Branch[3] has only keys that are greater than Node.Key[2].
- Note that if less than the full number of keys are in the Node, these 4 conditions are truncated so that they speak of the appropriate number of keys and branches.

This generalizes in the obvious way to multiway search trees with other orders.

A B-tree of order m is a multiway search tree of order m such that:

- All leaves are on the bottom level.
- All internal nodes (except the root node) have at least $\text{ceil}(m / 2)$ (nonempty) children.

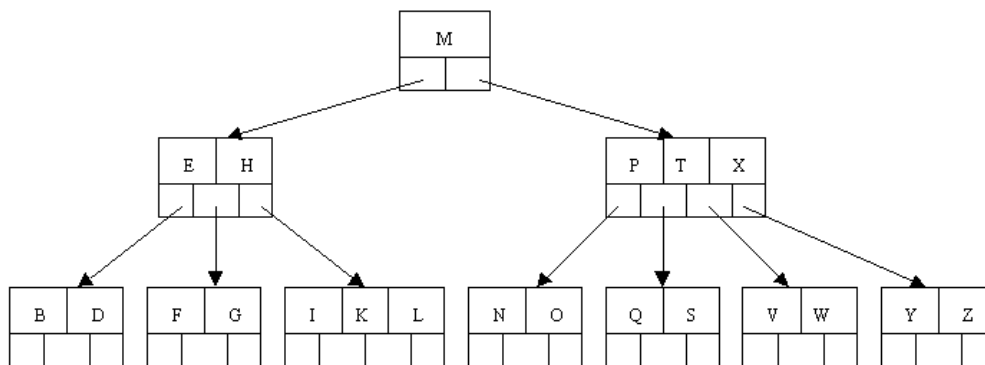
- The root node can have as few as 2 children if it is an internal node, and can obviously have no children if the root node is a leaf (that is, the whole tree consists only of the root node).
- Each leaf node must contain at least $\text{ceil}(m / 2) - 1$ keys.

Note that $\text{ceil}(x)$ is the so-called ceiling function. Its value is the smallest integer that is greater than or equal to x . Thus $\text{ceil}(3) = 3$, $\text{ceil}(3.35) = 4$, $\text{ceil}(1.98) = 2$, $\text{ceil}(5.01) = 6$, $\text{ceil}(7) = 7$, etc.

A B-tree is a fairly well-balanced tree by virtue of the fact that all leaf nodes must be at the bottom. Condition (2) tries to keep the tree fairly bushy by insisting that each node have at least half the maximum number of children. This causes the tree to "fan out" so that the path from root to leaf is very short even in a tree that contains a lot of data.

Example B-Tree

The following is an example of a B-tree of order 5. This means that (other than the root node) all internal nodes have at least $\text{ceil}(5 / 2) = \text{ceil}(2.5) = 3$ children (and hence at least 2 keys). Of course, the maximum number of children that a node can have is 5 (so that 4 is the maximum number of keys). According to condition 4, each leaf node must contain at least 2 keys. In practice B-trees usually have orders a lot bigger than 5.



Operations on a B-Tree

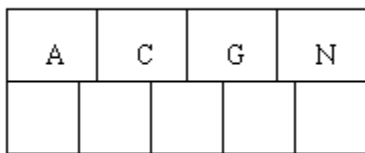
Question: How would you search in the above tree to look up S? How about J? How would you do a sort-of "in-order" traversal, that is, a traversal that would produce the letters in ascending order? (One would only do such a traversal on rare occasion as it would require a large amount of disk activity and thus be very slow!)

Inserting a New Item

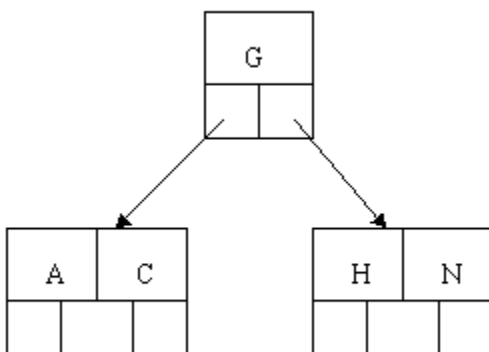
According to Kruse (see reference at the end of this file) the insertion algorithm proceeds as follows: When inserting an item, first do a search for it in the B-tree. If the item is not

already in the B-tree, this unsuccessful search will end at a leaf. If there is room in this leaf, just insert the new item here. Note that this may require that some existing keys be moved one to the right to make room for the new item. If instead this leaf node is full so that there is no room to add the new item, then the node must be "split" with about half of the keys going into a new node to the right of this one. The median (middle) key is moved up into the parent node. (Of course, if that node has no room, then it may have to be split as well.) Note that when adding to an internal node, not only might we have to move some keys one position to the right, but the associated pointers have to be moved right as well. If the root node is ever split, the median key moves up into a new root node, thus causing the tree to increase in height by one.

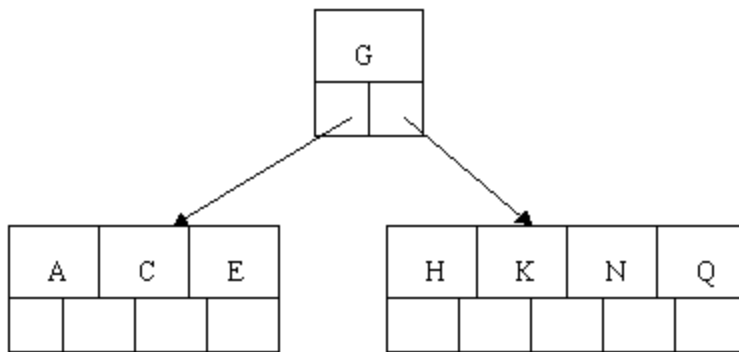
Let's work our way through an example similar to that given by Kruse. Insert the following letters into what is originally an empty B-tree of order 5: C N G A H E K Q M F W L T Z D P R X Y S Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:



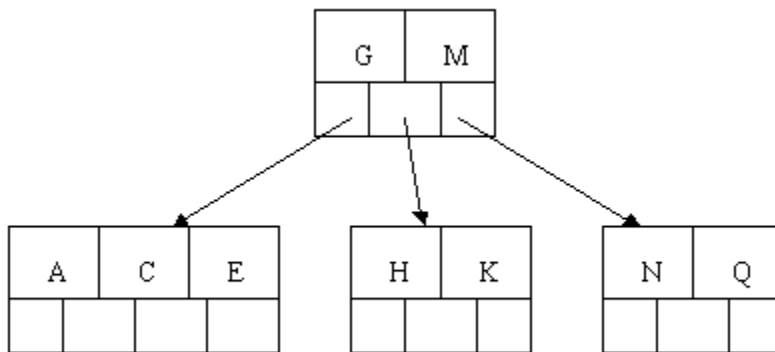
When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node. Note that in practice we just leave the A and C in the current node and place the H and N into a new node to the right of the old one.



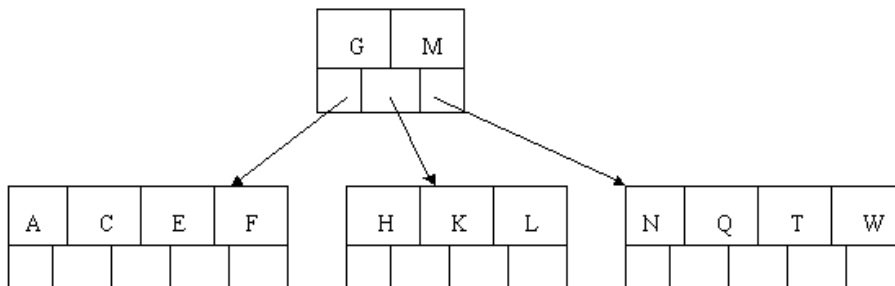
Inserting E, K, and Q proceeds without requiring any splits:



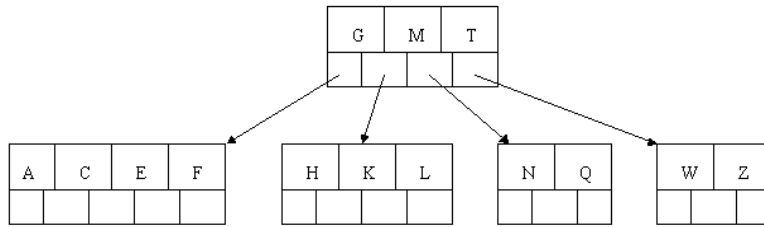
Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.



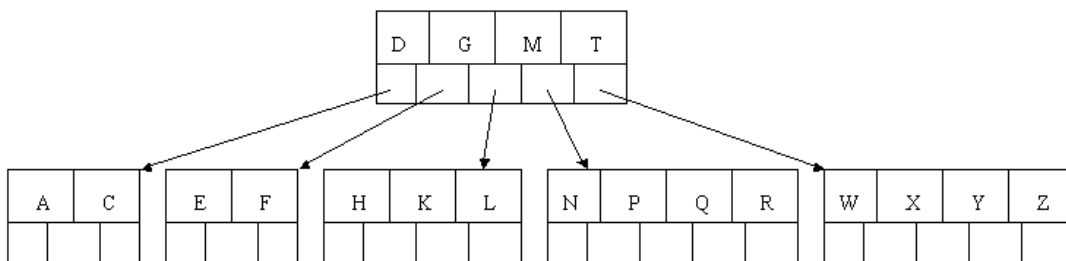
The letters F, W, L, and T are then added without needing any split.



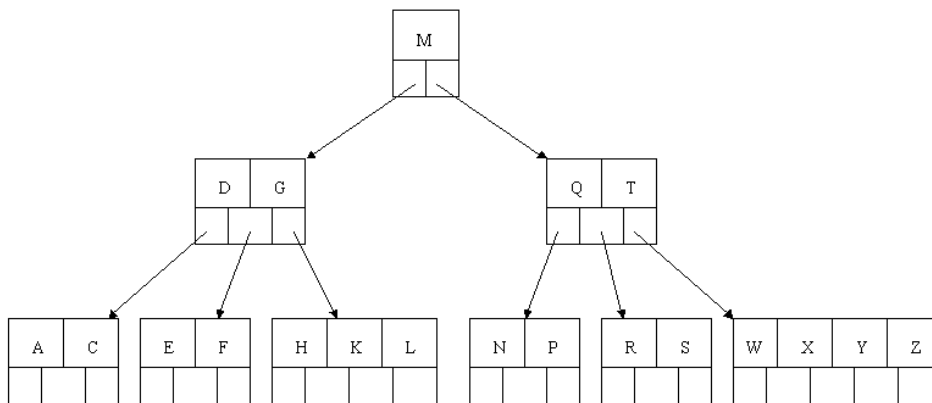
When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.



The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:



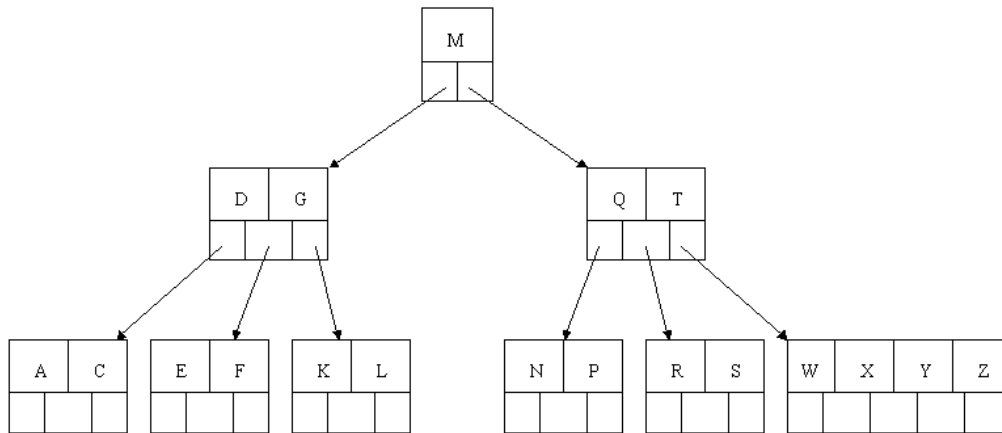
Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits, sending the median M up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains D and G.



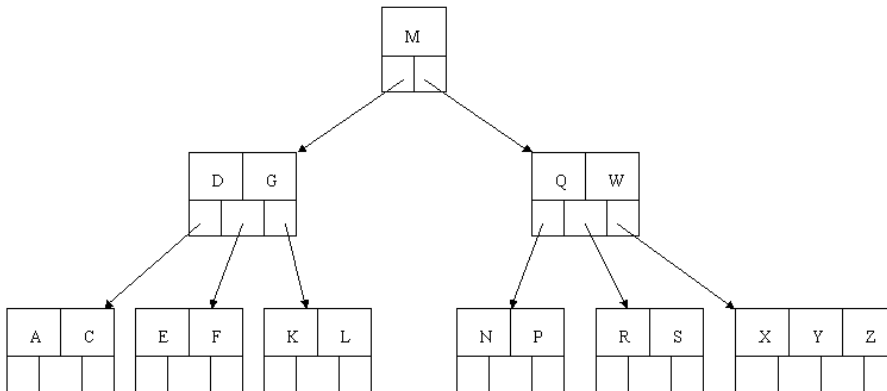
Deleting an Item

In the B-tree as we left it at the end of the last section, delete H. Of course, we first do a lookup to find H. Since H is in a leaf and the leaf has more than the minimum number of

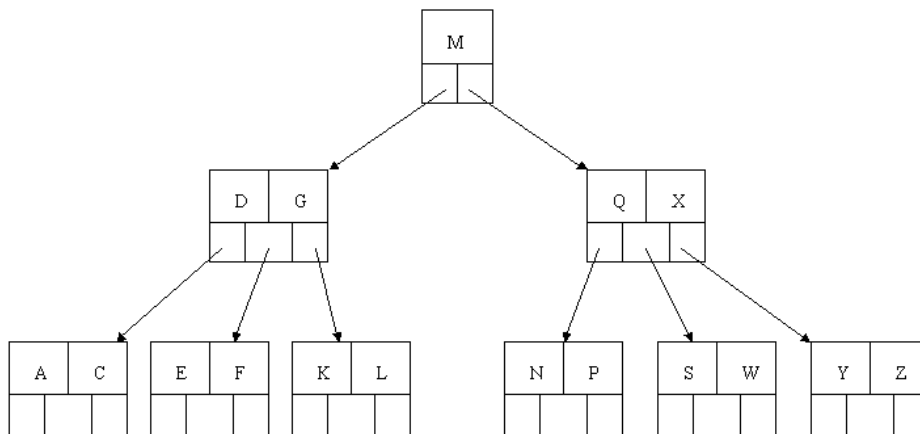
keys, this is easy. We move the K over where the H had been and the L over where the K had been. This gives:



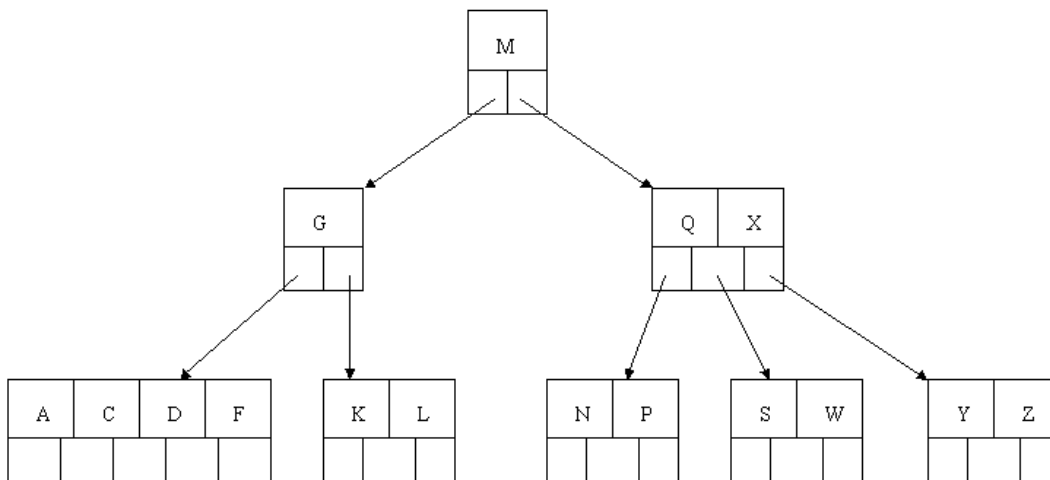
Next, delete the T. Since T is not in a leaf, we find its successor (the next item in ascending order), which happens to be W, and move W up to replace the T. That way, what we really have to do is to delete W from the leaf, which we already know how to do, since this leaf has extra keys. In ALL cases we reduce deletion to a deletion in a leaf, by using this method.



Next, delete R. Although R is in a leaf, this leaf does not have an extra key; the deletion results in a node with only one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we can then borrow a key from the parent and move a key up from this sibling. In our specific case, the sibling to the right has an extra key. So, the successor W of S (the last key in the node where the deletion occurred), is moved down from the parent, and the X is moved up. (Of course, the S is moved over so that the W can be inserted in its proper place.)

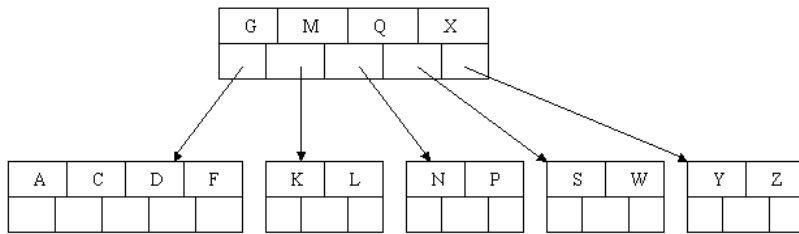


Finally, let's delete E. This one causes lots of problems. Although E is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left. In such a case the leaf has to be combined with one of these two siblings. This includes moving down the parent's key that was between those of these two leaves. In our example, let's combine the leaf containing F with the leaf containing A C. We also move down the D.



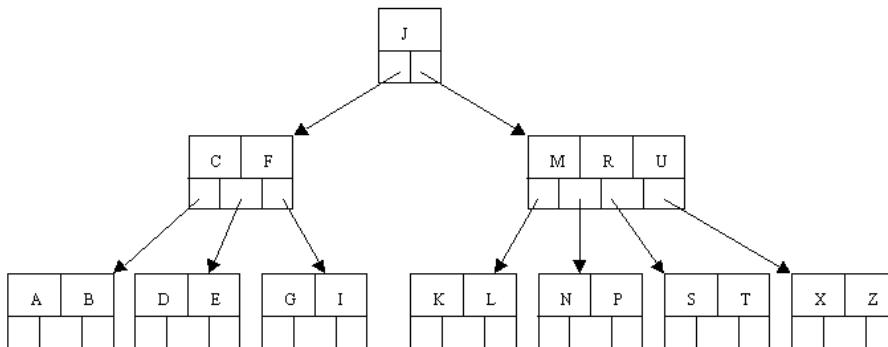
Of course, you immediately see that the parent node now contains only one key, G. This is not acceptable. If this problem node had a sibling to its immediate left or right that had a spare key, then we would again "borrow" a key. Suppose for the moment that the right sibling (the node with Q X) had one more key in it somewhere to the right of Q. We would then move M down to the node with too few keys and move the Q up where the M had been. However, the old left subtree of Q would then have to become the right subtree of M. In other words, the N P node would be attached via the pointer field to the right of M's new location. Since in our example we have no way to borrow a key from a sibling,

we must again combine with the sibling, and move down the M from the parent. In this case, the tree shrinks in height by one.

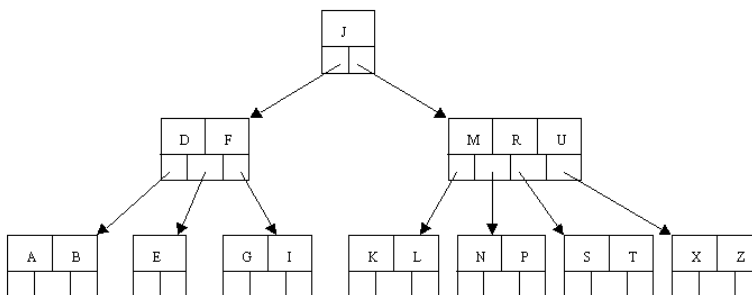


Another Example

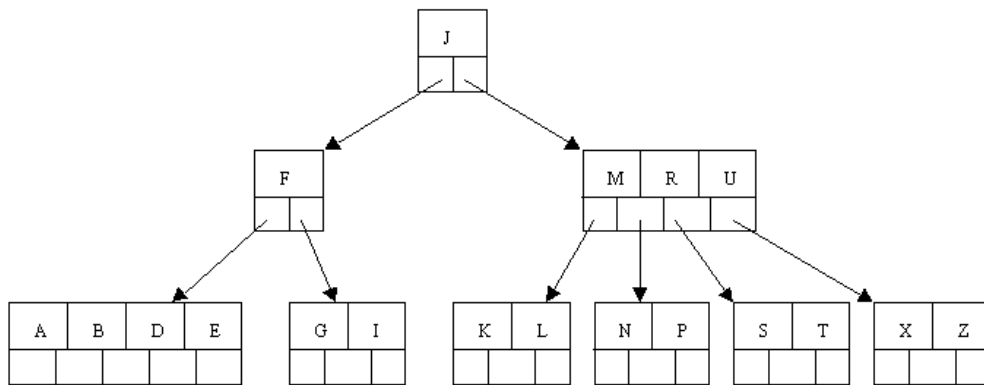
Here is a different B-tree of order 5. Let's try to delete C from it.



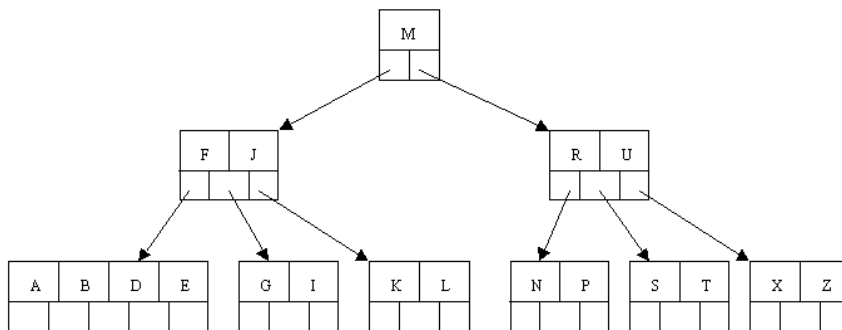
We begin by finding the immediate successor, which would be D, and move the D up to replace the C. However, this leaves us with a node with too few keys.



Since neither the sibling to the left or right of the node containing E has an extra key, we must combine the node with one of these two siblings. Let's consolidate with the A B node.



But now the node containing F does not have enough keys. However, its sibling has an extra key. Thus we borrow the M from the sibling, move it up to the parent, and bring the J down to join the F. Note that the K L node gets reattached to the right of the J.



Hashing

Hash Table

The hash table data structure is an array of some fixed size, containing the keys. A key is a value associated with each record.

Hashing Function

A hashing function is a key - to - address transformation, which acts upon a given key to compute the relative position of the key in an array.

A simple Hash function

$$\text{HASH}(\text{KEYVALUE}) = \text{KEYVALUE} \bmod \text{TABLESIZE}$$

Example : - Hash (92)

Hash (92) = $92 \bmod 10 = 2$

The keyvalue '92' is placed in the relative location '2'.

ROUTINE FOR SIMPLE HASH FUNCTION

Hash (Char *key, int Table Size)

```
{  
    int Hashvalue = 0;  
    while (* key != '\0')  
        Hashval += * key ++;  
    return Hashval % Tablesize;  
}
```

Some of the Methods of Hashing Function

1. Module Division
2. Mid - Square Method
3. Folding Method
4. PSEUDO Random Method
5. Digit or Character Extraction Method
6. Radix Transformation.

Collisions

Collision occurs when a hash value of a record being inserted hashes to an address (i.e. Relative position) that already contain a different record. (ie) When two key values hash to the same position.

Collision Resolution

The process of finding another position for the collide record.

Some of the Collision Resolution Techniques

1. Seperate Chaining

2. Open Addressing

3. Multiple Hashing

Seperate Chaining

Seperate chaining is an open hashing technique. A pointer field is added to each record location. When an overflow occurs this pointer is set to point to overflow blocks making a linked list.

In this method, the table can never overflow, since the linked list are only extended upon the arrival of new keys.

Insertion

To perform the insertion of an element, traverse down the appropriate list to check whether the element is already in place.

If the element turns to be a new one, it is inserted either at the front of the list or at the end of the list.

If it is a duplicate element, an extra field is kept and placed.

INSERT 10 :

Hash (k) = $k \% \text{Tablesize}$

Hash (10) = $10 \% 10$

INSERT 11 :

Hash (11) = $11 \% 10$

Hash (11) = 1

INSERT 81 :

Hash (81) = $81 \% 10$

Hash (81) = 1

The element 81 collides to the same hash value 1. To place the value 81 at this position perform the following.

Traverse the list to check whether it is already present.

Since it is not already present, insert at end of the list. Similarly the rest of the elements are inserted.

ROUTINE TO PERFORM INSERTION

```
void Insert (int key, Hashtable H)
{
    Position Pos, Newcell;

    List L;

    /* Traverse the list to check whether the key is already present */

    Pos = FIND (Key, H);

    If (Pos == NULL) /* Key is not found */
    {
        Newcell = malloc (size of (struct ListNode));

        If (Newcell != NULL)
        (
            L = H    TheLists [Hash (key, H    Tablesize)];

            Newcell → Next = L → Next;

            Newcell → Element = key;

            /* Insert the key at the front of the list */

            L → Next = Newcell;

        }

    }

}
```

FIND ROUTINE

Position Find (int key, Hashtable H)

```
{  
Position P;  
List L;  
L = H → The lists [Hash (key, H → Tablesize)];  
P = L → Next;  
while (P! = NULL && P → Element != key)  
P = P → Next;  
return p;  
}
```

Advantage

More number of elements can be inserted as it uses array of linked lists.

Disadvantage of Seperate Chaining

- * It requires pointers, which occupies more memory space.
- * It takes more effort to perform a search, since it takes time to evaluate the hash function and also to traverse the list.

OPEN ADDRESSING

Open addressing is also called **closed Hashing**, which is an alternative to resolve the collisions with linked lists.

In this hashing system, if a collision occurs, alternative cells are tried until an empty cell is found. (ie) cells $h_0(x)$, $h_1(x)$, $h_2(x)$are tried in succession.

There are three common collision resolution strategies. They are

(i) Linear Probing

(ii) Quadratic probing

(iii) Double Hashing.

LINEAR PROBING

In linear probing, for the i^{th} probe the position to be tried is $(h(k) + i) \bmod \text{tablesize}$, where $F(i) = i$, is the linear function.

In linear probing, the position in which a key can be stored is found by sequentially searching all position starting from the position calculated by the hash function until an empty cell is found.

If the end of the table is reached and no empty cells has been found, then the search is continued from the beginning of the table. It has a tendency to create clusters in the table.

Advantage :

- * It doesn't requires pointers

Disadvantage

- * It forms clusters, which degrades the performance of the hash table for storing and retrieving

UNIT IV GRAPHS

Definitions – Topological sort – breadth-first traversal - shortest-path algorithms – minimum spanning tree – Prim's and Kruskal's algorithms – Depth-first traversal – biconnectivity – Euler circuits – applications of graphs

GRAPH

A graph $G = (V, E)$ consists of a set of vertices, V and set of edges E .

Vertices are referred to as nodes and the arc between the nodes are referred to as Edges. Each edge is a pair (v, w) where $v, w \in V$. (i.e.) $v = V_1, w = V_2$.

BASIC TERMINOLOGIES

Directed Graph (or) Digraph

Directed graph is a graph which consists of directed edges, where each edge in E is unidirectional. It is also referred as Digraph. If (v, w) is a directed edge then $(v, w) \neq (w, v)$

Undirected Graph

An undirected graph is a graph, which consists of undirected edges. If (v, w) is an undirected edge then $(v, w) = (w, v)$

Weighted Graph

A graph is said to be weighted graph if every edge in the graph is assigned a weight or value. It can be directed or undirected graph.

Complete Graph

A complete graph is a graph in which there is an edge between every pair of vertices. A complete graph with n vertices will have $n(n-1)/2$ edges.

Strongly Connected Graph

If there is a path from every vertex to every other vertex in a directed graph then it is said to be strongly connected graph. Otherwise, it is said to be weakly connected graph.

Length

The length of the path is the number of edges on the path, which is equal to $N-1$, where N represents the number of vertices.

The length of the above path V_1 to V_3 is 2. (i.e.) $(V_1, V_2), (V_2, V_3)$.

If there is a path from a vertex to itself, with no edges, then the path length is 0.

Loop

If the graph contains an edge (v, v) from a vertex to itself, then the path is referred to as a loop.

Simple Path

A simple path is a path such that all vertices on the path, except possibly the first and the last are distinct.

A simple cycle is the simple path of length atleast one that begins and ends at the same vertex.

Cycle

A cycle in a graph is a path in which first and last vertex are the same.

Degree

The number of edges incident on a vertex determines its degree. The degree of the vertex V is written as $\text{degree}(V)$.

The indegree of the vertex V , is the number of edges entering into the vertex V .

Similarly the out degree of the vertex V is the number of edges exiting from that vertex V .

Acyclic Graph

A directed graph which has no cycles is referred to as acyclic graph. It is abbreviated as DAG.

DAG - Directed Acyclic Graph.

Representation of Graph

Graph can be represented by Adjacency Matrix and Adjacency list.

One simple way to represents a graph is Adjacency Matrix.

The adjacency Matrix A for a graph $G = (V, E)$ with n vertices is an $n \times n$ matrix, such that

$A_{ij} = 1$, if there is an edge V_i to V_j

$A_{ij} = 0$, if there is no edge.

Advantage

- * Simple to implement.

Disadvantage

- * Takes $O(n^2)$ space to represent the graph
- * It takes $O(n^2)$ time to solve the most of the problems.

Adjacency List Representation

In this representation, we store a graph as a linked structure. We store all vertices in a list and then for each vertex, we have a linked list of its adjacency vertices

Disadvantage

- * It takes $O(n)$ time to determine whether there is an arc from vertex i to vertex j . Since there can be $O(n)$ vertices on the adjacency list for vertex i .

5.3 Topological Sort

A **topological sort** is a linear ordering of vertices in a directed acyclic graph such that if there is a path from V_i to V_j , then V_j appears after V_i in the linear ordering.

Topological ordering is not possible. If the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v .

To implement the topological sort, perform the following steps.

Step 1 : - Find the indegree for every vertex.

Step 2 : - Place the vertices whose indegree is '0' on the empty queue.

Step 3 : - Dequeue the vertex V and decrement the indegree's of all its adjacent vertices.

Step 4 : - Enqueue the vertex on the queue, if its indegree falls to zero.

Step 5 : - Repeat from step 3 until the queue becomes empty.

Step 6 : - The topological ordering is the order in which the vertices dequeued.

Routine to perform Topological Sort

/* Assume that the graph is read into an adjacency matrix and that the indegrees are computed for every vertices and placed in an array (i.e. Indegree []) */

void Topsort (Graph G)

{

Queue Q ;

int counter = 0;

Vertex V, W ;

Q = CreateQueue (NumVertex);

Makeempty (Q);

for each vertex V

if (indegree [V] == 0)

Enqueue (V, Q);

while (! IsEmpty (Q))

{

V = Dequeue (Q);

TopNum [V] = ++ counter;

for each W adjacent to V

if (--Indegree [W] == 0)

Enqueue (W, Q);

}

if (counter != NumVertex)

Error (" Graph has a cycle");

```
DisposeQueue (Q); /* Free the Memory */  
}
```

Note :

Enqueue (V, Q) implies to insert a vertex V into the queue Q.

Dequeue (Q) implies to delete a vertex from the queue Q.

TopNum [V] indicates an array to place the topological numbering.

Dijkstra's Algorithm

The general method to solve the single source shortest path problem is known as Dijkstra's algorithm. This is applied to the weighted graph G.

Dijkstra's algorithm is the prime example of Greedy technique, which generally solve a problem in stages by doing what appears to be the best thing at each stage. This algorithm proceeds in stages, just like the unweighted shortest path algorithm. At each stage, it selects a vertex v, which has the smallest d_v among all the unknown vertices, and declares that as the shortest path from S to V and mark it to be known. We should set $d_w = d_v + C_{vw}$, if the new value for d_w would be an improvement.

ROUTINE FOR ALGORITHM

Void Dijkstra (Graph G, Table T)

```
{  
    int i ;  
    vertex V, W;  
    Read Graph (G, T) /* Read graph from adjacency list */  
    /* Table Initialization */  
    for (i = 0; i < Numvertex; i++)  
    {  
        T [i]. known = False;  
        T [i]. Dist = Infinity;
```

```

T [i]. path = NotA vertex;

}

T [start]. dist = 0;

for ( ; ;)

{

V = Smallest unknown distance vertex;

if (V == Not A vertex)

break ;

T[V]. known = True;

for each W adjacent to V

if ( ! T[W]. known)

{

T [W]. Dist = Min [T[W]. Dist, T[V]. Dist + CVW]

T[W]. path = V;

}

}

}

```

Minimum Spanning Tree

Cost = 5

Spanning Tree with minimum cost is considered as Minimum Spanning Tree

5.5.1 Prim's Algorithm

Prim's algorithm is one of the way to compute a minimum spanning tree which uses a greedy technique. This algorithm begins with a set U initialised to {1}. It this grows a spanning tree, one edge at a time. At each step, it finds a shortest edge (u,v) such that the

cost of (u, v) is the smallest among all edges, where u is in Minimum Spanning Tree and V is not in Minimum Spanning Tree.

SKETCH OF PRIM'S ALGORITHM

```
void Prim (Graph G)
```

```
{
```

```
MSTTREE T;
```

```
Vertex u, v;
```

```
Set of vertices V;
```

```
Set of tree vertices U;
```

```
T = NULL;
```

```
/* Initialization of Tree begins with the vertex '1' */
```

```
U = {1}
```

```
while (U  $\neq$  V)
```

```
{
```

```
Let (u,v) be a lowest cost such that u is in U and v is in V - U;
```

```
T = T  $\cup$  {(u, v)};
```

```
U = U  $\cup$  {V};
```

```
}
```

```
} ROUTINE FOR PRIMS ALGORITHM
```

```
void Prims (Table T)
```

```
{
```

```
vertex V, W;
```

```
/* Table initialization */
```

```
for (i = 0; i < Numvertex ; i++)
```

```

{
T[i]. known = False;
T[i]. Dist = Infinity;
T[i]. path = 0;
}
for (; ;)
{
Let V be the start vertex with the smallest distance
T[V]. dist = 0;
T[V]. known = True;
for each W adjacent to V
If (! T[W] . Known)
{
T[W].Dist = Min
(T[W]. Dist, Cvw);
T[W].path = V;
}
}
}

```

Depth First Search

Depth first works by selecting one vertex V of G as a start vertex ; V is marked visited. Then each unvisited vertex adjacent to V is searched in turn using depth first search recursively. This process continues until a dead end (i.e) a vertex with no adjacent unvisited vertices is encountered. At a deadend, the algorithm backup one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.

The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth first search must be restarted at any one of them.

To implement the Depthfirst Search perform the following Steps :

Step : 1 Choose any node in the graph. Designate it as the search node and mark it as visited.

Step : 2 Using the adjacency matrix of the graph, find a node adjacent to the search node that has not been visited yet. Designate this as the new search node and mark it as visited.

Step : 3 Repeat step 2 using the new search node. If no nodes satisfying (2) can be found, return to the previous search node and continue from there.

Step : 4 When a return to the previous search node in (3) is impossible, the search from the originally chosen search node is complete.

Step : 5 If the graph still contains unvisited nodes, choose any node that has not been visited and repeat step (1) through (4).

ROUTINE FOR DEPTH FIRST SEARCH

Void DFS (Vertex V)

```
{  
visited [V] = True;  
for each W adjacent to V  
if (! visited [W])  
Dfs (W);  
}
```

Example : -

Fig. 5.6

Adjacency Matrix

A B C D

A 0 1 1 1

B 1 0 0 1

C 1 0 0 1

D 1 1 1 0

Implementation

1. Let 'A' be the source vertex. Mark it to be visited.
2. Find the immediate adjacent unvisited vertex 'B' of 'A' Mark it to be visited.
- From 'B' the next adjacent vertex is 'd' Mark it has visited.
4. From 'D' the next unvisited vertex is 'C' Mark it to be visited.

Biconnected Graph

[Basic Concepts]

- **Articulation point:** An Articulation point in a connected graph is a vertex that, if delete, would break the graph into two or more pieces (connected component).
- **Biconnected graph:** A graph with no articulation point called biconnected. In other words, a graph is biconnected if and only if any vertex is deleted, the graph remains connected.
- **Biconnected component:** A biconnected component of a graph is a maximal biconnected subgraph- a biconnected subgraph that is not properly contained in a larger biconnected

The graphs we discuss below are all about loop-free undirected ones.

subgraph.

- A graph that is not biconnected can divide into biconnected components, sets of nodes mutually accessible via two distinct paths.

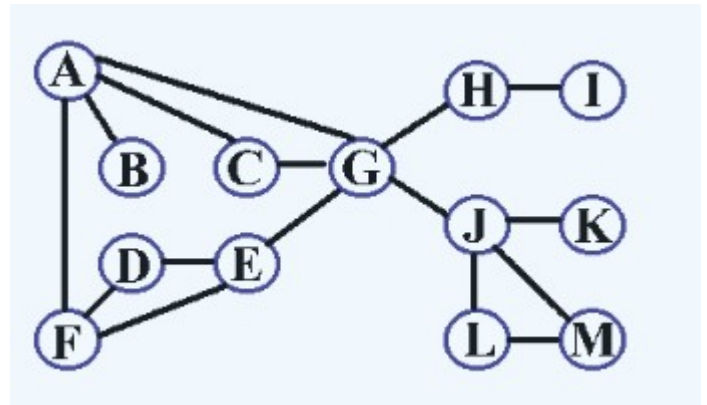
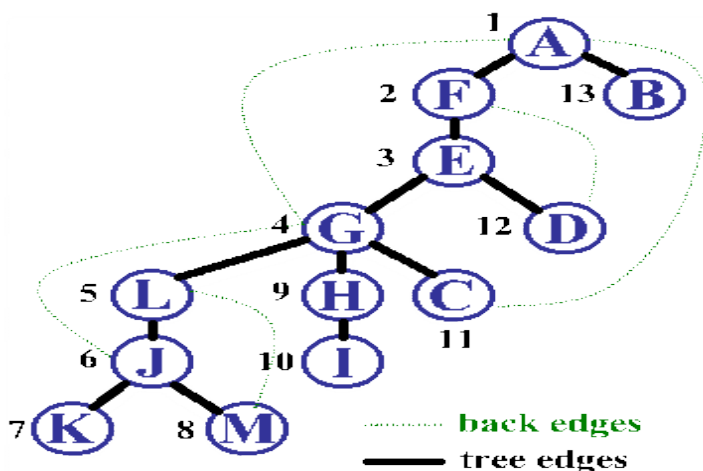


Figure 1. The graph G that is not biconnected

[Example] Graph G in Figure 1:

- Articulation points: **A, H, G, J**
- Biconnected components: **{A, C, G, D, E, F}, {G, J, L, B}, {B, H, I, K}**

□ *How to find articulation points?*



[Step 1.] Find the depth-first spanning tree T for G

[Step 2.] Add back edges in T

[Step 3.] Determine DNF(i) and L(i)

- **DNF(i)**: the visiting sequence of vertices i by depth first search
- **L(i)**: the least DFN reachable from i through a path consisting of zero or more tree edges followed by zero or one back edge

- [Step 4.]** Vertex i is an articulation point of G if and only if either:
- i is the root of T and has at least two children
 - i is not the root and has a child j for which $L(j) \geq \text{DFN}(i)$

Figure 2. Depth-first panning tree of the graph G

[Example] The $\text{DFN}(i)$ and $L(i)$ of Graph G in Figure 1 are:

| | | | | | | | | | | | | | |
|------------------------------------|----------|-----------|-----------|-----------|----------|----------|----------|----------|-----------|----------|----------|----------|----------|
| Vertex i: | A | B | C | D | E | F | G | H | I | J | K | L | M |
| $\text{DFN}(i)$: | 1 | 13 | 11 | 12 | 3 | 2 | 4 | 9 | 10 | 6 | 7 | 5 | 8 |
| $L(i)$: | 1 | 13 | 1 | 2 | 1 | 1 | 1 | 9 | 10 | 4 | 7 | 4 | 5 |

- Vertex G is an articulation point because G is not the root and in depth-first spanning tree in Figure 2, $L(L) \geq \text{DFN}(G)$, that L is one of its children
- Vertex A is an articulation point because A is the root and in depth-first spanning tree in Figure 2, it has more than one child, B and F
- Vertex E is not an articulation point because E is not the root and in depth-first spanning tree in Figure 2, $L(G) < \text{DFN}(E)$ and $L(D) < \text{DFN}(E)$, that G and D are its children

[Program]

```
int visit (int k)
{
    struct node *t;
    int m, min;
    val[k]=++id;
    min=id;
    for (t=adj[k];t!=z;t=t->next){
        if (val[t->v]==0){
            m=visit(t->v);
            if (m<min)
                min=m;;
            if (m>=val[k])
                printf("%C", name(k));
        }else if (val[t->v]<min)
            min=val[t->v];
    }
    return min;
}
```

UNIT V ALGORITHM DESIGN AND ANALYSIS

Greedy algorithms – Divide and conquer – Dynamic programming – backtracking – branch and bound – Randomized algorithms – algorithm analysis – asymptotic notations – recurrences – NP-complete problems

Top-Down Algorithms: Divide-and-Conquer

In this section we discuss a top-down algorithmic paradigm called *divide and conquer*. To solve a given problem, it is subdivided into one or more subproblems each of which is similar to the given problem. Each of the subproblems is solved independently. Finally, the solutions to the subproblems are combined in order to obtain the solution to the original problem.

Divide-and-conquer algorithms are often implemented using recursion. However, not all recursive functions are divide-and-conquer algorithms. Generally, the subproblems solved by a divide-and-conquer algorithm are *non-overlapping*.

Example-Binary Search

Consider the problem of finding the position of an item in a sorted list. I.e., given the sorted sequence $S = \{a_1, a_2, \dots, a_n\}$ and an item x , find i (if it exists) such that $a_i = x$. The usual solution to this problem is *binary search*.

Binary search is a divide-and-conquer strategy. The sequence S is split into two subsequences, $S_L = \{a_1, a_2, \dots, a_{\lfloor n/2 \rfloor}\}$ and $S_R = \{a_{\lfloor n/2 \rfloor + 1}, a_{\lfloor n/2 \rfloor + 2}, \dots, a_n\}$. The original problem is split into two subproblems: Find x in S_L or S_R . Of course, since the original list is sorted, we can quickly determine the list in which x must appear. Therefore, we only need to solve one subproblem.

Program `□` defines the function `BinarySearch` which takes four arguments, `array`, `x`, `i` and `n`. This routine looks for the position in `array` at which item `x` is found. Specifically, it considers the following elements of the array:

`array[i], array[i + 1], array[i + 2], ..., array[i + n - 1].`

```

1  template <class T>
2  unsigned int BinarySearch (
3      Array<T> const& array, T const& target,
4      unsigned int i, unsigned int n)
5  {
6      if (n == 0)
7          throw invalid_argument ("empty array");
8      if (n == 1)
9      {
10         if (array [i] == target)
11             return i;
12         throw domain_error ("target not found");
13     }
14     else
15     {
16         unsigned int const j = i + n / 2;
17         if (array [j] <= target)
18             return BinarySearch (array, target, j, n - n / 2);
19         else
20             return BinarySearch (array, target, i, n / 2);
21     }
22 }

```

Program: Divide-and-Conquer Example--Binary Search

The running time of the algorithm is clearly a function of n , the number of elements to be searched. Although Program [□](#) works correctly for arbitrary values of n , it is much easier to determine the running time if we assume that n is a power of two. In this case, the running time is given by the recurrence

$$T(n) = \begin{cases} O(1) & n \leq 1, \\ T(n/2) + O(1) & n > 1. \end{cases} \quad (14.3)$$

Equation [□](#) is easily solved using repeated substitution:

$$\begin{aligned}
 T(n) &= T(n/2) + 1 \\
 &= T(n/4) + 2 \\
 &= T(n/8) + 3 \\
 &\vdots \\
 &= T(n/2^k) + k
 \end{aligned}$$

Setting $n/2^k = 1$ gives $T(n) = \log n + 1 = O(\log n)$.

Example-Computing Fibonacci Numbers

The Fibonacci numbers are given by following recurrence

$$F_n = \begin{cases} 0 & n = 0, \\ 1 & n = 1, \\ F_{n-1} + F_{n-2} & n \geq 2. \end{cases} \quad (14.4)$$

Section [14.1](#) presents a recursive function to compute the Fibonacci numbers by implementing directly Equation [14.4](#). (See Program [14.1](#)). The running time of that program is shown to be $T(n) = \Omega((3/2)^n)$.

In this section we present a divide-and-conquer style of algorithm for computing Fibonacci numbers. We make use of the following identities

$$F_{2k-1} = (F_k)^2 + (F_{k-1})^2$$

$$F_{2k} = (F_k)^2 + 2F_k F_{k-1}$$

for $k \geq 1$. (See Exercise [14.2](#)). Thus, we can rewrite Equation [14.4](#) as

$$F_n = \begin{cases} 0 & n = 0, \\ 1 & n = 1, \\ (F_{\lceil n/2 \rceil})^2 + (F_{\lceil n/2 \rceil - 1})^2 & n \geq 2 \text{ and } n \text{ is odd,} \\ (F_{\lceil n/2 \rceil})^2 + 2F_{\lceil n/2 \rceil} F_{\lceil n/2 \rceil - 1} & n \geq 2 \text{ and } n \text{ is even.} \end{cases} \quad (14.5)$$

Program [14.2](#) defines the function `Fibonacci` which implements directly Equation [14.5](#).

Given $n > 1$ it computes F_n by calling itself recursively to compute $F_{\lceil n/2 \rceil}$ and $F_{\lceil n/2 \rceil - 1}$ and then combines the two results as required.

```

1 unsigned int Fibonacci (unsigned int n)
2 {
3     if (n == 0 || n == 1)
4         return n;
5     else
6     {
7         unsigned int const a = Fibonacci ((n + 1) / 2);
8         unsigned int const b = Fibonacci ((n + 1) / 2 - 1);
9         if (n % 2 == 0)
10            return a * (a + 2 * b);
11        else
12            return a * a + b * b;
13    }
14 }

```

Program: Divide-and-Conquer Example--Computing Fibonacci Numbers

To determine a bound on the running time of the `Fibonacci` routine in Program \square we

assume that $T(n)$ is a non-decreasing function. I.e., $T(n) \geq T(n-1)$ for all $n \geq 1$.

Therefore $T(\lceil n/2 \rceil) \geq T(\lceil n/2 \rceil - 1)$. Although the program works correctly for all values of n , it is convenient to assume that n is a power of 2. In this case, the running time of the routine is upper-bounded by $T(n)$ where

$$T(n) = \begin{cases} O(1) & n \leq 1, \\ 2T(n/2) + O(1) & n > 1. \end{cases} \quad (14.6)$$

Equation \square is easily solved using repeated substitution:

$$\begin{aligned}
 T(n) &= 2T(n/2) + 1 \\
 &= 4T(n/4) + 1 + 2 \\
 &= 8T(n/8) + 1 + 2 + 4 \\
 &\vdots \\
 &= 2^k T(n/2^k) + \sum_{i=0}^{k-1} 2^i \\
 &\vdots \\
 &= nT(1) + n - 1 \quad (n = 2^k).
 \end{aligned}$$

Thus, $T(n) = 2n - 1 = O(n)$.

Example-Merge Sorting

Sorting algorithms and sorters are covered in detail in Chapter 14. In this section we consider a divide-and-conquer sorting algorithm--*merge sort*. Given an array of n items in arbitrary order, the objective is to rearrange the elements of the array so that they are ordered from the smallest element to the largest one.

The merge sort algorithm sorts a sequence of length $n > 1$ by splitting it into two subsequences--one of length $\lfloor n/2 \rfloor$, the other of length $\lceil n/2 \rceil$. Each subsequence is sorted and then the two sorted sequences are merged into one.

Program 14.1 defines the function `MergeSort` which takes three arguments, `array`, `i` and `n`. The routine sorts the following n elements:

`array[i], array[i + 1], array[i + 2], ..., array[i + n - 1]`.

The `MergeSort` routine calls itself as well as the `Merge` routine. The purpose of the `Merge` routine is to merge two sorted sequences, one of length $\lfloor n/2 \rfloor$, the other of length $\lceil n/2 \rceil$, into a single sorted sequence of length n . This can easily be done in $O(n)$ time. (See Program 14.2).

```
1  template <class T>
2  void MergeSort (Array<T>& array, unsigned int i, unsigned int n)
3  {
4      if (n > 1)
5      {
6          MergeSort (array, i, n / 2);
7          MergeSort (array, i + n / 2, n - n / 2);
8          Merge (array, i, n / 2, n - n / 2);
9      }
10 }
```

Program: Divide-and-Conquer Example--Merge Sorting

The running time of the `MergeSort` routine depends on the number of items to be sorted, n . Although Program 14.1 works correctly for arbitrary values of n , it is much easier to determine the running time if we assume that n is a power of two. In this case, the running time is given by the recurrence

$$T(n) = \begin{cases} O(1) & n \leq 1, \\ 2T(n/2) + O(n) & n > 1. \end{cases} \quad (14.7)$$

Equation \square is easily solved using repeated substitution:

$$\begin{aligned}
 T(n) &= 2T(n/2) + n \\
 &= 4T(n/4) + 2n \\
 &= 8T(n/8) + 3n \\
 &\vdots \\
 &= 2^k T(n/2^k) + kn
 \end{aligned}$$

Setting $n/2^k = 1$ gives $T(n) = n + n \log n = O(n \log n)$.

Backtracking Algorithms

A backtracking algorithm systematically considers all possible outcomes for each decision. In this sense, backtracking algorithms are like the brute-force algorithms. However, backtracking algorithms are distinguished by the way in which the space of possible solutions is explored. Sometimes a backtracking algorithm can detect that an exhaustive search is unnecessary and, therefore, it can perform much better.

Example-Balancing Scales

Consider the set of *scales* shown in Figure \square . Suppose we are given a collection of n weights, $\{w_1, w_2, \dots, w_n\}$, and we are required to place *all* of the weights onto the scales so that they are balanced.

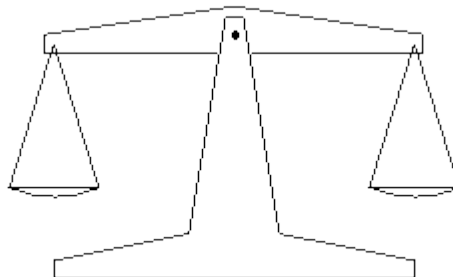


Figure: A Set of Scales

The problem can be expressed mathematically as follows: Let x_i represent the pan in which weight w_i is placed such that

$$x_i = \begin{cases} 0 & w_i \text{ is placed in the left pan,} \\ 1 & w_i \text{ is placed in the right pan.} \end{cases}$$

The scales are balanced when the sum of the weights in the left pan equals the sum of the weights in the right pan,


$$\sum_{i=1}^n w_i x_i = \sum_{i=1}^n w_i (1 - x_i).$$

Given an arbitrary set of n weights, there is no guarantee that a solution to the problem exists. A solution always exists if, instead of balancing the scales, the goal is to minimize the difference between the total weights in the left and right pans. Thus, given $\{w_1, w_2, \dots, w_n\}$, our *objective* is to *minimize* δ where

$$\delta = \left| \sum_{i=1}^n w_i x_i - \sum_{i=1}^n w_i (1 - x_i) \right|$$

subject to the constraint that *all* the weights are placed on the scales.

Given a set of scales and collection of weights, we might solve the problem by trial-and-error: Place all the weights onto the pans one-by-one. If the scales balance, a solution has been found. If not, remove some number of the weights and place them back on the scales in some other combination. In effect, we search for a solution to the problem by first trying one solution and then backing-up to try another.

Figure  shows the *solution space* for the scales balancing problem. In this case the solution space takes the form of a tree: Each node of the tree represents a *partial solution* to the problem. At the root (node A) no weights have been placed yet and the scales are balanced. Let δ be the difference between the the sum of the weights currently placed in the left and right pans. Therefore, $\delta = 0$ at node A.

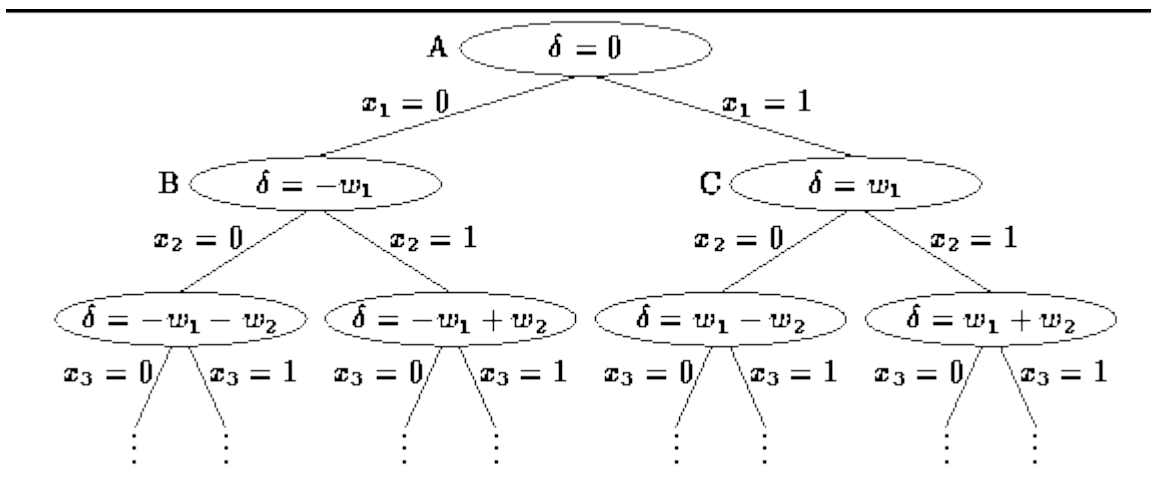


Figure: Solution Space for the Scales Balancing Problem

Node B represents the situation in which weight w_1 has been placed in the left pan. The difference between the pans is $\delta = -w_1$. Conversely, node C represents the situation in which the weight w_1 has been placed in the right pan. In this case $\delta = +w_1$. The complete solution tree has depth n and 2^n leaves. Clearly, the solution is the leaf node having the smallest $|\delta|$ value.

In this case (as in many others) the solution space is a tree. In order to find the best solution a backtracking algorithm visits all the nodes in the solution space. I.e., it does a *tree traversal*. Section [□](#) presents the two most important tree traversals--*depth-first* and *breadth-first*. Both kinds can be used to implement a backtracking algorithm.

Asymptotic Notation

Suppose we are considering two algorithms, A and B , for solving a given problem. Furthermore, let us say that we have done a careful analysis of the running times of each of the algorithms and determined them to be $T_A(n)$ and $T_B(n)$, respectively, where n is a measure of the problem size. Then it should be a fairly simple matter to compare the two functions $T_A(n)$ and $T_B(n)$ to determine which algorithm is *the best*!

But is it really that simple? What exactly does it mean for one function, say $T_A(n)$, to be *better than* another function, $T_B(n)$? One possibility arises if we know the problem size *a priori*. E.g., suppose the problem size is n_0 and $T_A(n_0) < T_B(n_0)$. Then clearly algorithm A is better than algorithm B for problem size n_0 .

In the general case, we have no *a priori* knowledge of the problem size. However, if it can be shown, say, that $\forall n \geq 0 : T_A(n) \leq T_B(n)$, then algorithm A is better than algorithm B regardless of the problem size.



Unfortunately, we usually don't know the problem size beforehand, nor is it true that one of the functions is less than or equal the other over the entire range of problem sizes. In this case, we consider the *asymptotic* behavior of the two functions for very large problem sizes.

An Asymptotic Upper Bound-Big Oh

In 1892, P. Bachmann invented a notation for characterizing the asymptotic behavior of functions. His invention has come to be known as *big oh notation*:

Definition (Big Oh) Consider a function $f(n)$ which is non-negative for all integers $n \geq 0$. We say that " $f(n)$ is big oh $g(n)$," which we write $f(n) = O(g(n))$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$, $f(n) \leq cg(n)$.

A Simple Example

Consider the function $f(n) = 8n + 128$ shown in Figure . Clearly, $f(n)$ is non-negative for all integers $n \geq 0$. We wish to show that $f(n) = O(n^2)$. According to Definition , in order to show this we need to find an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$, $f(n) \leq cn^2$.

It does not matter what the particular constants are--as long as they exist! E.g., suppose we choose $c=1$. Then

$$\begin{aligned} f(n) \leq cn^2 &\Rightarrow 8n + 128 \leq n^2 \\ &\Rightarrow 0 \leq n^2 - 8n - 128 \\ &\Rightarrow 0 \leq (n - 16)(n + 8). \end{aligned}$$

Since $(n+8) > 0$ for all values of $n \geq 0$, we conclude that $(n_0 - 16) \geq 0$. I.e., $n_0 = 16$.

So, we have that for $c=1$ and $n_0 = 16$, $f(n) \leq cn^2$ for all integers $n \geq n_0$. Hence, $f(n) = O(n^2)$. Figure 1 clearly shows that the function $f(n) = n^2$ is greater than the function $f(n) = 8n + 128$ to the right of $n=16$.

Of course, there are many other values of c and n_0 that will do. For example, $c=2$ and $n_0 = 2 + 4\sqrt{17} \approx 10.2$ will do, as will $c=4$ and $n_0 = 1 + \sqrt{33} \approx 6.7$. (See Figure 1).

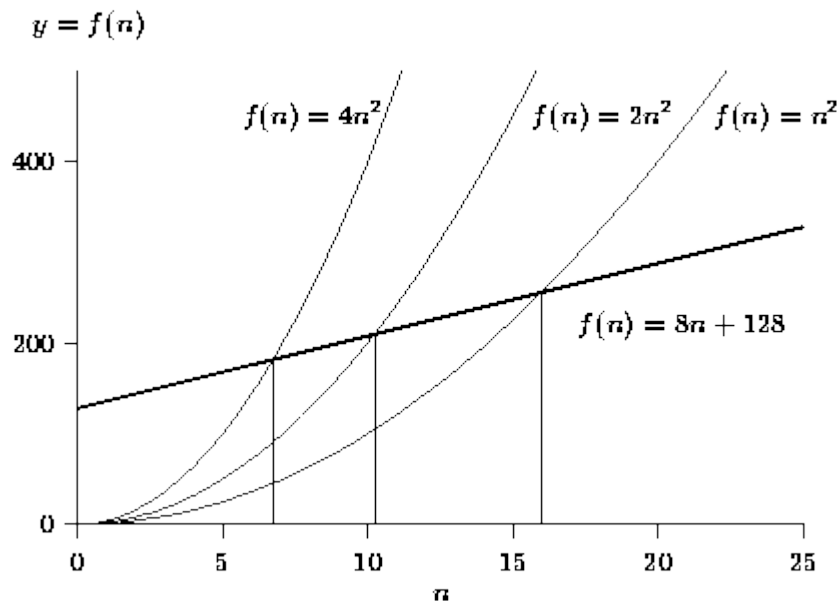


Figure: Showing that $f(n) = 8n + 128 = O(n^2)$.

An Asymptotic Lower Bound-Omega

The big oh notation introduced in the preceding section is an asymptotic *upper bound*. In this section, we introduce a similar notation for characterizing the asymptotic behavior of a function, but in this case it is a *lower bound*.

Definition (Omega) Consider a function $f(n)$ which is non-negative for all integers $n \geq 0$. We say that " $f(n)$ is omega $g(n)$," which we write $f(n) = \Omega(g(n))$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$, $f(n) \geq cg(n)$.

The definition of omega is almost identical to that of big oh. The only difference is in the comparison--for big oh it is $f(n) \leq cg(n)$; for omega, it is $f(n) \geq cg(n)$. All of the same conventions and caveats apply to omega as they do to big oh.

A Simple Example

Consider the function $f(x) = 5x^2 - 64x + 256$ which is shown in Figure [□](#). Clearly, $f(n)$ is non-negative for all integers $n \geq 0$. We wish to show that $f(n) = \Omega(n^2)$. According to Definition [□](#), in order to show this we need to find an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$, $f(n) \geq cn^2$.

As with big oh, it does not matter what the particular constants are--as long as they exist! E.g., suppose we choose $c=1$. Then

$$\begin{aligned} f(n) \geq cn^2 &\Rightarrow 5n^2 - 64n + 256 \geq n^2 \\ &\Rightarrow 4n^2 - 64n + 256 \geq 0 \\ &\Rightarrow 4(n - 8)^2 \geq 0. \end{aligned}$$

Since $(n - 8)^2 > 0$ for all values of $n \geq 0$, we conclude that $n_0 = 0$.

So, we have that for $c=1$ and $n_0 = 0$, $f(n) \geq cn^2$ for all integers $n \geq n_0$. Hence, $f(n) = \Omega(n^2)$. Figure [□](#) clearly shows that the function $f(n) = n^2$ is less than the function $f(n) = 5n - 64n + 256$ for all values of $n \geq 0$. Of course, there are many other values of c and n_0 that will do. For example, $c=2$ and $n_0 = 16$.

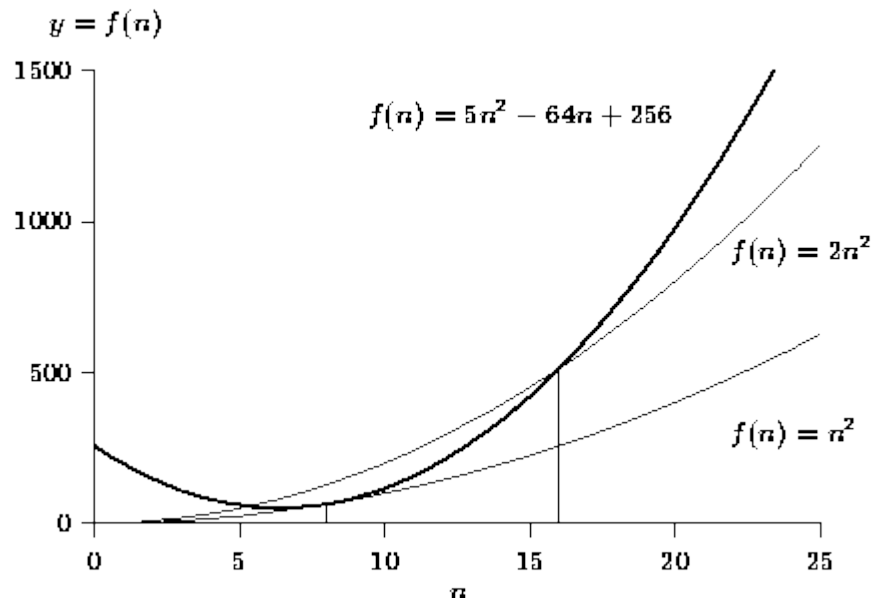


Figure: Showing that $f(n) = 4n^2 - 64n + 288 = \Omega(n^2)$.

More Notation-Theta and Little Oh

This section presents two less commonly used forms of asymptotic notation. They are:

- A notation, $\Theta(\cdot)$, to describe a function which is both $O(g(n))$ and $\Omega(g(n))$, for the same $g(n)$. (Definition \square).
- A notation, $o(\cdot)$, to describe a function which is $O(g(n))$ but not $\Theta(g(n))$, for the same $g(n)$. (Definition \square).

Definition (Theta) Consider a function $f(n)$ which is non-negative for all integers $n \geq 0$. We say that " $f(n)$ is theta $g(n)$," which we write $f(n) = \Theta(g(n))$, if and only if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Recall that we showed in Section \square that a polynomial in n , say $f(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_2 n^2 + a_1 n + a_0$, is $O(n^m)$. We also showed in Section \square that such a polynomial is $\Omega(n^m)$. Therefore, according to Definition \square , we will write $f(n) = \Theta(n^m)$.

Definition (Little Oh) Consider a function $f(n)$ which is non-negative for all integers $n \geq 0$. We say that " $f(n)$ is little oh $g(n)$," which we write $f(n) = o(g(n))$, if and only if $f(n)$ is $O(g(n))$ but $f(n)$ is not $\Theta(g(n))$.

Little oh notation represents a kind of *loose asymptotic bound* in the sense that if we are given that $f(n) = o(g(n))$, then we know that $g(n)$ is an asymptotic upper bound since $f(n) = O(g(n))$, but $g(n)$ is *not* an asymptotic lower bound since $f(n) \neq \Omega(g(n))$ and $f(n) \neq \Theta(g(n))$. \diamond

For example, consider the function $f(n) = n + 1$. Clearly, $f(n) = O(n^2)$. Clearly too, $f(n) \neq \Omega(n^2)$, since no matter what c we choose, for large enough n , $cn^2 \geq n + 1$. $f(n) = n + 1 = o(n^2)$.

NP - Complete Problems

A decision problem D is said to be NP complete if 1. It belongs to class NP. 2. Every problem in NP is polynomially reducible to D .

A problem P_1 can be reduced to P_2 as follows

Provide a mapping so that any instance of P_1 can be transformed to an instance of P_2 . Solve P_2 and then map the answer back to the original. As an example, numbers are entered into a pocket calculator in decimal. The decimal numbers are converted to binary, and all calculations are performed in binary. Then the final answer is converted back to decimal for display. For P_1 to be polynomially reducible to P_2 , all the work associated with the transformations must be performed in polynomial time.

The reason that NP - complete problems are the hardest NP problems is that a problem that is NP - complete can essentially be used as a subroutine for any problem in NP, with only a polynomial amount of overhead. Suppose we have an NP - complete problem P_1 . Suppose P_2 is known to be in NP. Suppose further that P_1 polynomially reduces to P_2 , so that we can solve P_1 by using P_2 with only a polynomial time penalty. Since P_1 is NP - complete, every problem in NP polynomially reduces to P_1 . By applying the closure property to polynomials, we see that every problem in NP is polynomially reducible to P_2 ; we reduce the problem to P_1 and then reduce P_1 to P_2 . Thus, P_2 is NP - Complete. Travelling salesman problem is NP - complete. It is easy to see that a solution can be checked in polynomial time, so it is certainly in NP. Hamilton cycle problem can be polynomially reduced to travelling salesman problem.

Hamilton Cycle Problem transformed to travelling salesman problem.

Some of the examples of NP - complete problems are Hamilton circuit, Travelling salesman, knapsack, graph coloring, Bin packing and partition problem.

EE2204 Data Structure Answer

Unit 1 part a

1. What is meant by an abstract data type?

An ADT is a set of operation. Abstract data types are mathematical abstractions. Eg. Objects such as list, set and graph along their operations can be viewed as ADT's.

2. What are the operations of ADT?

Union, Intersection, size, complement and find are the various operations of ADT.

3. What is meant by list ADT?

List ADT is a sequential storage structure. General list of the form $a_1, a_2, a_3, \dots, a_n$ and the size of the list is 'n'. Any element in the list at the position i is defined to be a_i , a_{i+1} the successor of a_i and a_{i-1} is the predecessor of a_i .

4. What are the various operations done under list ADT?

Print list

Insert

Make empty

Remove

Next

Previous

Find kth

5. What are the different ways to implement list?

Simple array implementation of list

Linked list implementation of list

6. What are the advantages in the array implementation of list?

a) Print list operation can be carried out at the linear time

b) Find Kth operation takes a constant time

7. What is a linked list?

Linked list is a kind of series of data structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a record containing its successor.

8. What is a pointer?

Pointer is a variable, which stores the address of the next element in the list.

Pointer is basically a number.

9. What is a doubly linked list?

In a simple linked list, there will be one pointer named as 'NEXT POINTER' to point the next element, whereas in a doubly linked list, there will be two pointers one to point the next element and the other to point the previous element location.

10. Define double circularly linked list?

In a doubly linked list, if the last node or pointer of the list, point to the first element of the list, then it is a circularly linked list.

11. What is the need for the header?

Header of the linked list is the first element in the list and it stores the number of elements in the list. It points to the first data element of the list.

12. List three examples that uses linked list?

Polynomial ADT

Radix sort

Multi lists

13. Give some examples for linear data structures?

Stack

Queue

14. What is a stack?

Stack is a data structure in which both insertion and deletion occur at one end only. Stack is maintained with a single pointer to the top of the list of elements. The other name of stack is Last-in -First-out list.

15. Write postfix from of the expression $-A+B-C+D$?

$A-B+C-D+$

16. How do you test for an empty queue?

To test for an empty queue, we have to check whether $READ=HEAD$ where REAR is a pointer pointing to the last node in a queue and HEAD is a pointer that pointer to the dummy header. In the case of array implementation of queue, the condition to be checked for an empty queue is $READ<FRONT$.

17. What are the postfix and prefix forms of the expression?

$A+B*(C-D)/(P-R)$

Postfix form: $ABCD-*PR-/+$

Prefix form: $+A/*B-CD-PR$

18. Explain the usage of stack in recursive algorithm implementation?

In recursive algorithms, stack data structures is used to store the return address when a recursive call is

Encountered and also to store the values of all the parameters essential to the current state of the procedure.

19. Write down the operations that can be done with queue data structure?

Queue is a first – in -first out list. The operations that can be done with queue are addition and deletion.

20. What is a circular queue?

The queue, which wraps around upon reaching the end of the array is called as circular queue.

Unit 1 part B

1. Explain the linked list implementation of list ADT in Detail?

_ Definition for linked list

_ Figure for linked list

_ Next pointer

- _ Header or dummy node

- _ Various operations

Explanation

Example figure

Coding

2. Explain the cursor implementation of linked list?

- _ Definition for linked list

- _ Figure for linked list

- _ Next pointer

- _ Header or dummy node

- _ Various operations

Explanation

Example figure

Coding

3. Explain the various applications of linked list?

- _ Polynomical ADT

Operations

Coding

Figure

- _ Radix Sort

Explanation

Example

- _ Multilist

Explanation

Example figure

4. Explain the linked list implementation of stack ADT in detail?

- _ Definition for stack

- _ Stack model

- _ Figure

- _ Pointer-Top

- _ Operations

Coding

Example figure

5. Explain the array implementation of queue ADT in detail?

- _ Definition for stack

- _ Stack model

- _ Figure

- _ Pointer-FRONT, REAR

- _ Operations

Coding

Example figure

Unit 2& 3 part A

1. Define non-linear data structure

Data structure which is capable of expressing more complex relationship than that of physical adjacency is called non-linear data structure.

2. Define tree?

A tree is a data structure, which represents hierarchical relationship between individual data items.

3. Define leaf?

In a directed tree any node which has out degree 0 is called a terminal node or a leaf.

4. What is meant by directed tree?

Directed tree is an acyclic diagraph which has one node called its root with indegree 0 while all other nodes have indegree 1.

5. What is a ordered tree?

In a directed tree if the ordering of the nodes at each level is prescribed then such a tree is called ordered tree.

6. What are the applications of binary tree?

Binary tree is used in data processing.

- a. File index schemes
- b. Hierarchical database management system

7. What is meant by traversing?

Traversing a tree means processing it in such a way, that each node is visited only once.

8. What are the different types of traversing?

The different types of traversing are

- a. Pre-order traversal-yields prefix form of expression.
- b. In-order traversal-yields infix form of expression.
- c. Post-order traversal-yields postfix form of expression.

9. What are the two methods of binary tree implementation?

Two methods to implement a binary tree are,

- a. Linear representation.
- b. Linked representation

10. Define pre-order traversal?

Pre-order traversal entails the following steps;

- a. Process the root node
- b. Process the left subtree
- c. Process the right subtree

11. Define post-order traversal?

Post order traversal entails the following steps;

- a. Process the left subtree
- b. Process the right subtree
- c. Process the root node

12. Define in -order traversal?

In-order traversal entails the following steps;

- a. Process the left subtree
- b. Process the root node
- c. Process the right subtree

13. What is a balance factor in AVL trees?

Balance factor of a node is defined to be the difference between the height of the node's left subtree and the height of the node's right subtree.

14. What is meant by pivot node?

The node to be inserted travel down the appropriate branch track along the way of the deepest level node on the branch that has a balance factor of +1 or -1 is called pivot node.

15. What is the length of the path in a tree?

The length of the path is the number of edges on the path. In a tree there is exactly one path from the root to each node.

16. Define expression trees?

The leaves of an expression tree are operands such as constants or variable names and the other nodes contain operators.

17. What is the need for hashing?

Hashing is used to perform insertions, deletions and find in constant average time.

18. Define hash function?

Hash function takes an identifier and computes the address of that identifier in the hash table using some function.

19. List out the different types of hashing functions?

The different types of hashing functions are,

- a. The division method
- b. The mind square method
- c. The folding method
- d. Multiplicative hashing
- e. Digit analysis

20. What are the problems in hashing?

- a. Collision
- b. Overflow

21. What are the problems in hashing?

When two keys compute in to the same location or address in the hash table through any of the hashing function then it is termed collision.

Unit 2 and 3 part B

1. Explain the different tree traversals with an application?

_ In order

Explanation with an example

Figure

_ Preorder

Explanation with an example

Figure

_ Postorder

Explanation with an example

Figure

2. Define binary search tree? Explain the various operations with an example?

- _ Definition
- _ Figure for binary search tree
- _ Operations
- Codings
- Explanation
- Example
- 3. Define AVL trees? Explain the LL, RR, RL, LR case with an example?
 - _ Definition
 - _ LL, RR, RL, LR case
 - Figure
 - Example
 - Explanation
- 4. Define priority queue? Explain the basic heap operation with an example?
 - _ Definition
 - _ Basic operation
 - Insert
 - Delmin
 - Delmax
 - _ Coding
 - _ Explanation
 - _ Example
- 5. Explain any two techniques to overcome hash collision?
 - _ Separate chaining
 - Example
 - Explanation
 - Coding
 - _ Open addressing
 - Linear probing
 - Quadratic probing

Unit 4 part A

1. Define Graph?

A graph G consists of a nonempty set V which is a set of nodes of the graph, a set E which is the set of edges of the graph, and a mapping from the set for edge E to a set of pairs of elements of V . It can also be represented as $G=(V, E)$.

2. Define adjacent nodes?

Any two nodes which are connected by an edge in a graph are called adjacent nodes. For example, if an edge x is associated with a pair of nodes $u, v \in V$, then we say that the edge x connects the nodes u and v . $x \in (u, v)$ where $u, v \in V$.

3. What is a directed graph?

A graph in which every edge is directed is called a directed graph.

4. What is an undirected graph?

A graph in which every edge is undirected is called an undirected graph.

5. What is a loop?

An edge of a graph which connects to itself is called a loop or sling.

6. What is a simple graph?

A simple graph is a graph, which has not more than one edge between a pair of nodes than such a graph is called a simple graph.

7. What is a weighted graph?

A graph in which weights are assigned to every edge is called a weighted graph.

8. Define out degree of a graph?

In a directed graph, for any node v , the number of edges which have v as their initial node is called the out degree of the node v .

9. Define indegree of a graph?

In a directed graph, for any node v , the number of edges which have v as their terminal node is called the indegree of the node v .

10. Define path in a graph?

The path in a graph is the route taken to reach terminal node from a starting node.

11. What is a simple path?

A path in a diagram in which the edges are distinct is called a simple path.

It is also called as edge simple.

12. What is a cycle or a circuit?

A path which originates and ends in the same node is called a cycle or circuit.

13. What is an acyclic graph?

A simple diagram which does not have any cycles is called an acyclic graph.

14. What is meant by strongly connected in a graph?

An undirected graph is connected, if there is a path from every vertex to every other vertex. A directed graph with this property is called strongly connected.

15. When is a graph said to be weakly connected?

When a directed graph is not strongly connected but the underlying graph is connected, then the graph is said to be weakly connected.

16. Name the different ways of representing a graph?

a. Adjacency matrix

b. Adjacency list

17. What is an undirected acyclic graph?

When every edge in an acyclic graph is undirected, it is called an undirected acyclic graph. It is also called as undirected forest.

18. What are the two traversal strategies used in traversing a graph?

a. Breadth first search

b. Depth first search

19. What is a minimum spanning tree?

A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at the lowest total cost.

20. What is NP?

NP is the class of decision problems for which a given proposed solution for a given input can be checked quickly to see if it is really a solution.

Unit 4 part B

1. Explain the various representation of graph with example in detail?

_ Adjacency matrix

Figure

Explanation

Table

_ Adjacency list

Figure

Explanation

Table

2. Define topological sort? Explain with an example?

_ Definition

_ Explanation

_ Example

_ Table

_ Coding

3. Explain Dijkstra's algorithm with an example?

_ Explanation

_ Example

_ Graph

_ Table

_ Coding

4. Explain Prim's algorithm with an example?

_ Explanation

_ Example

_ Graph

_ Table

_ Coding

5. Explain Krushal's algorithm with an example?

_ Explanation

_ Example

_ Graph

_ Table

_ Coding

Unit V part A

1. Write down the definition of data structures?

A data structure is a mathematical or logical way of organizing data in the memory that consider not only the items stored but also the relationship to each other and also it is characterized by accessing functions.

2. What is meant by problem solving?

Problem solving is a creative process, which needs systemization and mechanization.

3. Give few examples for data structures?

Stacks, Queue, Linked list, Trees, graphs

4. What is problem definition phase?

The first step in solving a problem is to understand problem clearly. Hence, the first phase is the problem definition phase. That is, to extract the task from the problem statement. If the problem is not understood, then the solution will not be correct and it may result in wastage of time and effort.

5. Define Algorithm?

Algorithm is a solution to a problem independent of programming language. It consist of set of finite steps which, when carried out for a given set of inputs, produce the corresponding output and terminate in a finite time.

6. Define Program?

Set of instructions to find the solution to a problem. It is expressed in a programming language in an explicit and unambiguous manner.

7. Mention how similarities among the problems are used in problem solving?

This method is used to find out if a problem of this sort has been already solved and to adopt a similar method in solving the problem. The contribution of experience in the previous problem with help and enhance the method of problem for the current problem.

8. What is working backward from the solution?

When we have a solution to the problem then we have to work backward to find the starting condition. Even a guess can take us to the starting of the problem. This is very important to sytematize the investigation to avoid duplication of our effort.

9. Mention some of the problem solving strategies?

The most widely strategies are listed below

Divide and conquer

Binary doubling strategy

Dynamic programming

10. What is divide and conquer method?

The basic idea is to divide the problem into several sub problems beyond which cannot be further subdivided. Then solve the sub problems efficiently and join then together to get the solution for the main problem.

11. What are the features of an efficient algorithm?

Free of ambiguity

Efficient in execution time

Concise and compact

Completeness

Definiteness

Finiteness

12. List down any four applications of data structures?

Compiler design

Operating System

Database Management system

Network analysis

13. What is binary doubling strategy?

The reverse of binary doubling strategy, i.e. combining small problems in to one is known as binary doubling strategy. This strategy is used to avoid the generation of intermediate results.

14. Where is dynamic programming used?

Dynamic programming is used when the problem is to be solved in a sequence of intermediate steps. It is particularly relevant for many optimization problems, i.e. frequently encountered in Operations research.

15. Define top-down design?

Top-down design is a strategy that can be applied to find a solution to a problem from a vague outline to precisely define the algorithm and program implementation by stepwise refinement.

16. Mention the types of bugs that may arise in a program?

The different types of bugs that can arise in a program are

Syntactic error

Semantic error

Logical error

17. What is program testing?

Program testing is process to ensure that a program solves the smallest possible problem, when all the variables have the same value, the biggest possible problem, unusual cases etc.

18. What is program verification?

Program verification refers to the application of mathematical proof techniques, to verify that the results obtained by the execution of the program with arbitrary inputs are in accord with formally defined output Specifications.

19. How will you verify branches with segments?

To handle the branches that appear in the program segments, it is necessary to set-up and proves verification conditions individually.

20. What is proof of termination?

To prove that a program accomplishes its stated objective in a finite number of steps is called program termination. The proof of termination is obtained directly from the properties of the interactive constructs.

Unit V part B

1. Explain top-down design in detail?

Definition

Breaking a problem in to subproblems

Choice of a suitable data structure

Constructions of loops

Establishing initial conditions for loops

Finding the iterative construct

Terminations of loops

2. What are the steps taken to improve the efficiency of an algorithm?

Definition

Redundant computations

Referencing array elements

Inefficiency due to late termination

Early detection of desired output conditions

Trading storage for efficiency gains

3. Design an algorithm for sine function computation. Explain it with an example?

Algorithm development

Algorithm description

Pascal implementation

Application

4. Design an algorithm for reversing the digit of an integer. Explain it with an example?

Algorithm development

Algorithm description

Pascal implementation

Application

5. Design an algorithm for base conversion. Explain it with an example?

Algorithm development

Algorithm description

Pascal implementation

Application

EE 2204 Data Structure and Algorithms analysis

Unit 1 2 marks

1. Define ADT.
2. Give the structure of Queue model.
3. What are the basic operations of Queue ADT?
4. What is Enqueue and Dequeue?
5. Give the applications of Queue.
6. What is the use of stack pointer?
7. What is an array?
8. Define ADT (Abstract Data Type).
9. Swap two adjacent elements by adjusting only the pointers (and not the data) using singly linked list.
10. Define a queue model.
11. What are the advantages of doubly linked list over singly linked list?
12. Define a graph
13. What is a Queue?
14. What is a circularly linked list?
15. What is linear list?
16. How will you delete a node from a linked list?
17. What is linear pattern search?
18. What is recursive data structure?
19. What is doubly linked list?

Unit 1 part B

1. Explain the implementation of stack using Linked List.
2. Explain Prefix, Infix and postfix expressions with example.
3. Explain the operations and the implementation of list ADT.
4. Give a procedure to convert an infix expression $a+b*c+(d*e+f)*g$ to postfix notation
5. Design and implement an algorithm to search a linear ordered linked list for a given alphabetic key or name.
6. (a) What is a stack? Write down the procedure for implementing various stack operations(8)
(b) Explain the various application of stack? (8)
7. (a) Given two sorted lists L1 and L2 write a procedure to compute L1_L2 using only the basic operations (8)
(b) Write a routine to insert an element in a linked list (8)
8. What is a queue? Write an algorithm to implement queue with example.

Unit 2 and 3 part a

1. Explain Tree concept?
2. What is meant by traversal?
3. What is meant by depth first order?
4. What is In order traversal?
5. What is Pre order traversal?
6. What is Post order traversal?
7. Define Binary tree.
8. What is meant by BST?
9. Define AVL trees.
10. Give example for single rotation and double rotation.
11. Define Hashing.
12. Define Double Hashing.
13. What is meant by Binary Heap?
14. Mention some applications of Priority Queues.
15. Define complete binary tree.
16. How a binary tree is represented using an array? Give an example
17. A full node is a node with two children. Prove that the number of full nodes plus one is equal to the number of leaves in a non empty binary tree.
18. Define (i) inorder (ii) preorder (iii) postorder traversal of a binary tree.
19. Suppose that we replace the deletion function, which finds, return, and removes the minimum element in the priority queue, with find min, can both insert and find min be implemented in constant time?
20. What is an expression tree?

1. Explain the representation of priority queue
2. Compare the various hashing Techniques.
3. List out the steps involved in deleting a node from a binary search tree.
4. What is binary heap?
5. Define Binary search tree.
6. List out the various techniques of hashing
7. Define hash function.
8. Show that maximum number of nodes in a binary tree of height H is $2^{H+1} - 1$.
9. Define hashing.
10. Define AVL tree.

Unit 2 and 3 part B

1. Explain the operation and implementation of Binary Heap.
2. Explain the implementation of different Hashing techniques.
3. Give the prefix, infix and postfix expressions corresponding to the tree given in figure.
4. (a) How do you insert an element in a binary search tree? (8)
(b) Show that for the perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $2^{h+1} - 1 - (h+1)$. (8)
5. Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function $h(X) = X \pmod{10}$, show the resulting:

- (a) Separate chaining table (4)
- (b) Open addressing hash table using linear probing (4)
- (c) Open addressing hash table using quadratic probing (4)
- (d) Open addressing hash table with second hash function $h_2(X) = 7 - (X \bmod 7)$. (4)
- 6. Explain in detail (i) Single rotation (ii) double rotation of an AVL tree.
- 7. Explain the efficient implementation of the priority queue ADT
- 8. Explain how to find a maximum element and minimum element in BST? Explain detail about Deletion in Binary Search Tree?
- 1. (a) Construct an expression tree for the expression $A + (B - C) * D + (E * F)$ (8)
- (b) Write a function to delete the minimum element from a binary heap (8)
- 2. Write a program in C to create an empty binary search tree & search for an element X in it. (16)
- 3. Explain in detail about Open Addressing (16)
- 4. Explain in detail insertion into AVL Trees (16)
- 5. Write a recursive algorithm for binary tree traversal with an example. (16)
- 6. Write an algorithm for initializing the hash table and insertion in a separate chaining (16)

Unit 4 part A

Define Graph.

- 2. What is meant by directed graph?
- 3. Give a diagrammatic representation of an adjacency list representation of a graph.
- 4. What is meant by topological sort?
- 5. What is meant by acyclic graph?
- 6. What is meant by Shortest Path Algorithm?
- 7. What is meant by Single-Source Shortest path problem?
- 8. What is meant by Dijkstra's Algorithm?
- 9. What is minimum spanning tree?
- 10. Mention the types of algorithm.
- 11. Define NP- complete problems
- 12. What is space requirement of an adjacency list representation of a graph
- 13. What is topological sort?
- 14. What is breadth-first search?
- 15. Define minimum spanning tree
- 16. Define undirected graph
- 17. What is depth-first spanning tree
- 18. What is Bio connectivity?
- 19. What is Euler Circuit?
- 20. What is a directed graph?
- 21. What is meant by 'Hamiltonian Cycle'?
- 22. Define (i) indegree (ii) outdegree
- 1. Explain the topological sort.

2. Define NP hard & NP complete problem.
3. Prove that the number of odd degree vertices in a connected graph should be even.
4. What is an adjacency list? When it is used?
5. What is an activity node of a graph?
6. Define Breadth First Search.
7. Define Depth First Search.
8. Define Minimum Spanning Tree.
9. Define Shortest Path of a graph.
10. Define Biconnectivity.
1. Define Graph.
2. What is meant by directed graph?
3. Give a diagrammatic representation of an adjacency list representation of a graph.
4. What is meant by topological sort?
5. What is meant by acyclic graph?
6. What is meant by Shortest Path Algorithm?
7. What is meant by Single-Source Shortest path problem?
8. What is meant by Dijkstra's Algorithm?
9. What is minimum spanning tree?
10. Mention the types of algorithm.
11. Define NP- complete problems
12. What is space requirement of an adjacency list representation of a graph
13. What is topological sort?
14. What is breadth-first search?
15. Define minimum spanning tree
16. Define undirected graph
17. What is depth-first spanning tree
18. What is Bi connectivity?
19. What is Euler Circuit?
20. What is a directed graph?
21. What is meant by 'Hamiltonian Cycle'?
22. Define (i)indegree (ii)outdegree

Unit 4 part B

1. Explain Prim's & Kruskal's Algorithm with an example.
 2. Describe Dijkstra's algorithm with an example.
 3. Explain how to find shortest path using Dijkstra's algorithm with an example.
 4. Explain the application of DFS.
 5. Find a minimum spanning tree for the graph using both Prim's and Kruskal's algorithms.
 6. Explain in detail the simple topological sort pseudo code
 7. Write notes on NP-complete problems
-
1. Formulate an algorithm to find the shortest path using Dijkstra's algorithm and explain with example. (16)
 2. Explain the minimum spanning tree algorithms with an example. (16)

3. (a) Write short notes on Biconnectivity. (8)
- (b) Write an algorithm for Topological Sort of a graph. (8)
4. Write and explain weighted and unweighted shortest path algorithm (16)
5. Explain the various applications of Depth First Search. (16)

1. Explain Prim's & Kruskal's Algorithm with an example.
2. Describe Dijkstra's algorithm with an example.
3. Explain how to find shortest path using Dijkstra's algorithm with an example.
4. Explain the application of DFS.
5. Find a minimum spanning tree for the graph

using both Prim's and Kruskal's algorithms.

6. Explain in detail the simple topological sort pseudo code
7. Write notes on NP-complete problems

Unit V part A

1. Define Program
2. Define Algorithm
3. Define Problem Definition Phase
4. What are the problem solving strategies?
5. Define Top Down Design.
6. What is the basic idea behind Divide & Conquer Strategy?
7. Define Program Verification.
8. Define Input & Output Assertion.
9. Define Symbolic Execution
10. Define Verification Condition
11. Define the qualities of good algorithm.
12. Define Computational Complexity.
13. What is O – notation?
14. What is Recursion? Explain with an example.
15. List out the performance measures of an algorithm.
1. Define Algorithm & Notion of algorithm.
2. What is analysis framework?
3. What are the algorithm design techniques?
4. How is an algorithm's time efficiency measured?
5. Mention any four classes of algorithm efficiency.
6. Define Order of Growth.
7. State the following Terms.
 - (i) Time Complexity
 - (ii) Space Complexity
8. What are the various asymptotic Notations?
9. What are the important problem types?
10. Define algorithmic Strategy (or) Algorithmic Technique.
11. What are the various algorithm strategies (or) algorithm Techniques?
12. What are the ways to specify an algorithm?

13. Define Best case Time Complexity .
14. Define Worst case Time Complexity.
15. Define Average case time complexity.
16. What are the Basic Efficiency Classes.
17. Define Asymptotic Notation.
18. How to calculate the GCD value?
1. What is meant by algorithm? What are its measures?
2. Give any four algorithmic techniques.
3. Write an algorithm to find the factorial of a given number?
4. Define the worst case & average case complexities of an algorithm
5. What is divide & conquer strategy?
6. What is dynamic programming?
7. Write at least five qualities & capabilities of a good algorithm
8. Write an algorithm to exchange the values of two variables
- 9 Write an algorithm to find N factorial (written as n!) where $n \geq 0$.

Unit V part B

1. (a) Explain in detail the types on analysis that can be performed on an algorithm (8)
- (b) Write an algorithm to perform matrix multiplication algorithm and analyze the same (8)
2. Design an algorithm to evaluate the function $\sin(x)$ as defined by the infinite series expansion $\sin(x) = x/1! - x^3/3! + x^5/5! - x^7/7! + \dots$
3. Write an algorithm to generate and print the first n terms of the Fibonacci series where $n \geq 1$ the first few terms are 0, 1, 1, 2, 3, 5, 8, 13.
4. Design an algorithm that accepts a positive integer and reverse the order of its digits.
5. Explain the Base conversion algorithm to convert a decimal integer to its corresponding octal representation.
6. Explain in detail about Greedy algorithm with an example(16).
7. Explain in detail about Divide and conquer algorithm with an example also mark the difference between Greedy and divide and conquer algorithm.(16).
8. Describe the backtracking problem using knapsack problem .(16).