Unit 1 Introduction to Data Structures and Algorithms (4 Hrs)

1.1 Data Types, Data Structure and Abstract Data Type

1.2 Dynamic memory allocation in C

1.3 Introduction to Algorithms

1.4 Asymptotic notations and common functions

**What is Data structure ?**

Data structure is the way of storing data in a computer so that it can be used efficiently. It can also be defined as a mathematical or logical model which relates to a particular organization of different data elements.

**Types of Data Structure**

* Primitive Data Structure
* Non-primitive Data Structure

**Primitive Data Structure**

The primitive data structures are also known as basic data structures. These structures are directly operated upon by the machine instructions. Examples of primitive data structures are:

* Integer
* Float
* Character
* Pointer

**Non-primitive Data Structure**

The non-primitive data structures are highly developed complex data structures. Basically, these are developed from basic data structure. Examples of non-primitive data structures are:

* Arrays
* Lists
* Files

There are **two** **types** of **non-primitive data structure**

1. **Linear Data Structure**: When the elements are stored on contiguous memory locations then data structure is called linear data structure. For example, array, stack, queue etc.
2. **Non Linear Data Structure**: In nonlinear data structure, elements are stored in non-contiguous memory locations. Eg tree, graphs, etc.

**What is Static Data Structure ?**

A static data structure is one whose capacity is fixed at creation. An array is an example of static data structure.

**What is dynamic data Structure ?**

A dynamic data structure is one whose capacity is variable, so it can expand or contract at any time. Linked List, binary tree are example of dynamic data structure.

**Operations on Data Structure**

* Traversing
* Searching
* Sorting
* Insertion
* Deletion

**Whati is Algorithm ?**

An algorithm is finite set of instructions to perform the computational task of finite number of steps. To develop a program of an algorithm, we select an appropriate data structure for that algorithm. Therefore algorithm and its associated data structures form a program.

**Algorithm + Data Structure = Program**

Data structures are building blocks of a program.

**What is a good algorithm ?**

A good algorithm must contain the following properties :-

* **Input**: The quantity that is given to algorithm initially is called input.
* **Output:** The quantity produced by algorithm is called output. The output will have some relationship with input.
* **Finiteness**: The algorithm must terminate after finite number of steps.
* **Definiteness**: Each step of the algorithm must be precisely defined that is each step should be unambiguous.

**What are the different structures used in algorithms ?**

* Natural Language
* Pseudo code
* Flowchart
* Programming Language

**Complexity Analysis of Algorithm**

Several algorithms could be created to solve a single problem. These algorithms may vary in the way they get, process and output data. They could have significant differences in terms of performance and space utilization.

**What is time Complexity** ?

The time complexity of an algorithm measures the amount of time taken by an algorithm to run as a function of input.

**What is Space Complexity ?**

The space complexity of an algorithm measures the amount of space taken by an algorithm to run as function of input.

**Types of Analysis**

* **Worst Case Running Time:**  The worst case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer. For expressing worst case running time of an algorithm Big O notation is used.
* **Best Case Running Time:** The best case running time of an algorithm is lower bound on the running time for any input. The best rarely occurs in practice. For expressing best case running time of an algorithm Big Ω notation is used.
* **Average Case Running Time:** The average case running time of an algorithm is an estimate of the running time for an “average input”. For expressing, average case running time of an algorithm Big Ɵ notation is used.
* **Amortized Analysis:** In amortized analysis, the time required to perform a sequence of (related) operations is averaged over all the operations performed. It is the average performance of each operation in the worst case. It guarantees the average performance of each operation in the worst case.

**What is Big O(oh) Notation ?**

The big O notation gives the asymptotic upper bound of the running time of an algorithm.

**What is Big Omega Notation?**

The big Ω notation gives the asymptotic lower bound of the running time of an algorithm.

**What is Big Theta Notation?**

The big theta notation gives the average case of running time of an algorithm.

**Examples of Complexities**

There are different Big- O expressions such as O(1), O(log n), O(n), O(nlogn), O(n2), O(n3) and O(2n)

**What is Dynamic Memory Allocation in C ?**

The concept of dynamic memory allocation in C language enables the C programmer to allocate memory at run time. Dynamic memory allocation in C language is possible by 4 functions of stdlib.h header file.

1. **malloc()**
2. **calloc()**
3. **realloc()**
4. **free()**

**Difference between Dynamic Memory Allocation and Static Allocation**

|  |  |
| --- | --- |
| **Static Memory Allocation** | **Dynamic Memory Allocation** |
| Memory is allocated at compile time | Memory is allocated at run time |
| Memory can n't be increased while executing program | Memory can be increased while executing program |
| Used in array | Used in linked list |

**Malloc function in C**

The malloc() function allocates single block of requested memory. It does not initialize memory at execution time, so it has garbage value initially. It returns NULL if memory is not sufficient. The syntax of malloc() function is:

**ptr = (cast\_type)malloc(byte-size)**

**Calloc() function in C**

The calloc() function allocates multiple block of requested memory. It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient. The syntax of calloc() function is given below:

**ptr=(cast-type\*)calloc(number, byte-size)**

**Realloc() function in C**

If memory is not sufficient for **malloc()** or **calloc()**, we can reallocate the memory by **realloc()** function. In short, it changes the memory size. The syntax of realloc() function.

**ptr=realloc(ptr, new-size)**

**Free() function in C**

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

The syntax of free() function.

**free(ptr)**

**The following program reads n numbers from user and find their sum using malloc & calloc function**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int n,i,\*ptr,sum=0;

printf("Enter number of elements: ");

scanf("%d",&n);

ptr=(int\*)malloc(n\*sizeof(int)); **// using malloc**

if(ptr==NULL)

{

printf("Sorry! unable to allocate memory");

exit(0);

}

printf("Enter elements of array: ");

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

{

scanf("%d",ptr+i);

sum+=\*(ptr+i);

}

printf("Sum=%d",sum);

free(ptr);

return 0;

}

The output of above program:

Enter number of elements: 10

Enter elements of array: 1 2 3 4 5 6 7 8 9 10

Sum=55

#include<stdio.h>

#include<stdlib.h>

int main()

{

int n,i,\*ptr,sum=0;

printf("Enter number of elements: ");

scanf("%d",&n);

ptr=(int\*)calloc(n,sizeof(int)); **// using calloc**

if(ptr==NULL)

{

printf("Sorry! unable to allocate memory");

exit(0);

}

printf("Enter elements of array: ");

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

{

scanf("%d",ptr+i);

sum+=\*(ptr+i);

}

printf("Sum=%d",sum);

free(ptr);

return 0;

}

Output:

Enter number of elements: 5

Enter elements of array: 1 2 3 4 5

Sum=15

**Unit 2 Stack (4 Hrs)**

**2.1 Basic Concept of Stack, Stack as an ADT, Stack Operations, Stack Applications**

**What is Stack ?**

Stack is a linear data structure in which an element is inserted or deleted at only one end called top of the stack. The stack follows the principle of last in first out (**LIFO**) principle.

**Operation On A Stack:-**

**Push:-** The insertion of an item into a stack is called **push** operation.

**Pop:-** The deletion of an item from a stack is called pop operation.

**Application of Stack**

* To evaluate infix and postfix expression
* To perform the undo sequence in text editor
* To check the correctness of parenthesis
* To pass the parameters in functions in C
* To keep the function call history in recursion
* To keep the page visited history in browsers

**The Stack as ADT**

* **Create\_empty\_stack(S)** : Creates an stack S without elements
* **Push(S,x)**: Inserts **x** at the top of stack S
* **Top(S):** If stack is not empty, return top element of stack S
* **POP(S):** Deletes top element of stack S
* **Is\_Full():** Checks if the stack **S** is full or not. Returns 1 if stack is full and 0 otherwise
* **IsEmpty ()**: Checks if stack is empty or not. Return 1 if stack is empty and 0 otherwise

**Implementation of Stack**

Stack can be implemented in two ways

* Array implementation of stack (static)
* Linked list implementation of stack (dynamic)

**Array Implementation of Stack**

Array implementation of stack use one array with fixed size and pointer called top that indicates topmost position of stack. Each time an item is pushed into stack, top is incremented and top is decremented for pop operation. By convention, in C implementation the empty stack is indicated by setting the value of top to -1

**Algorithm for PUSH Operation**

void push (struct Stack \*ps, char val)

{

if (ps->top== STACKSIZE -1)

{

printf ("\nStack Overflow");

exit(1);

}

ps->items[++ps->top]=val;

}

1. Starts
2. Check for stack overflow:

If top== STACKSIZE -1 then print “ Stack Overflow”

else

top = top + 1

1. Read the element to be inserted say ‘val’
2. Set stk[top]=val;
3. Stop

Int pop (struct Stack \*ps)

{

if(ps->top==-1)

{

printf ("\nStack Underflow");

exit(1);

}

return (ps->items[ps->top--]);

}

**Algorithm for POP Operation**

1. Starts
2. Check for stack underflow:

If top==-1 then print “Stack Undertflow”

else

return stk[top--];

1. Stop

int topStack (struct Stack St)

{

return St.top;

}

int IsEmpty (struct Stack St)

{

if (St.top==-1)

return 1;

else

return 0;

}

int IsFull (struct Stack St)

{

if (S.top==STACKSIZE)

return 1;

else

return 0;

}

**Infix, Prefix and Postfix Notation**

We can represent the expression using following expressions.

* Infix Expression
* Prefix Expression
* Postfix Expression

**Infix Expression**

It is an **ordinary mathematical notation of expression** where an **operator** is written **between** the **operands**. For example A+B. Here '+' is operator and 'A' and 'B' are called operands. It needs extra information to make the order of evaluation of the operators. Each operator has precedence. Operator with higher precedence is evaluated first. If two operators have same precedence then operators are evaluated according to left to right or right to left associativity rule.

**Prefix Notation**

In prefix notation the operator precedes the two operands. The operator is written before the operands. It is also called police notation. For example. +AB

**Postfix Notation**

In postfix notation the operator is written after the operands. It is also called reverse police notation. For example.

AB+

The order of evaluation of operators is always left to right and brackets cannot be used to change those order. Operators act on values immediately to the left of them.

**Why do we use postfix expression**

* Postfix expression specify the actual order of operations without any parenthesis
* No need to check of precedence
* Cumbersome parenthesis are avoided
* Compiler uses postfix expression

**Order of precedence for evaluation of Arithmetic Expression**

1. Parenthesis
2. Exponent
3. Division and multiplication
4. Addition and Substraction

**Algorithm to convert infix to postfix notation**

1. Start
2. Read infix expression
3. Create one empty stack for operator called ‘**opstack**’ and an empty array for postfix expression called ‘**postfix**’
4. Scan one character at a time from infix expression
5. Repeat till there is data in infix expression
   1. If scanned character is ‘(‘ then push it into **opstack**
   2. If the scanned character is operand then push into **opstack**
   3. If the scanned character is operator then

If(opstack is not empty)

**While** (precedence (scanned character) <=precedence (**opstack [top]**)

and push into **opstack**

Otherwise

Push scanned character into **opstack**

* 1. If scanned character is ‘)’ then

Pop and add into postfix array until **‘(‘** is not found and ignore both symbols

1. Pop and push into **poststack** and until stack **opstack** is not empty
2. Stop

**Algorithm to evaluate the postfix expression**

1. Read the postfix expression
2. Scan character by character from left to write of given postfix expression
   1. If scanned symbol is operand then

Read its corresponding value and push it into vstack

* 1. If scanned symbol is operator then
* Pop and place into op2
* Pop and place into op1
* Compute result according to operator and push this result into vstack

1. Pop and display the result
2. Stop

**Infix to Prefix Conversion**

1. Start
2. Input infix expression and store it in an array infix[].
3. Reverse this expression
4. Find postfix expression and store result in postfix[] array
5. Reverse the postfix[] array
6. Print this reversed array postfix[]
7. Stop

**Trace of Conversion Algorithm**

**Example 1 Convert ((A-(B+C)\*D)$(E+F)**

|  |  |  |
| --- | --- | --- |
| Scan Character | Poststack | opstack |
| ( | …… | ( |
| ( | …… | (( |
| A | A | (( |
| - | A | ((- |
| ( | A | ((-( |
| B | AB | ((-( |
| + | AB | ((-(+ |
| C | ABC | ((-(+ |
| ) | ABC+ | ((- |
| ) | ABC+- | ( |
| \* | ABC+- | (\* |
| D | ABC+-D | (\* |
| ) | ABC+-D\* | …. |
| $ | ABC+-D\* | $ |
| ( | ABC+-D\* | $( |
| E | ABC+-D\*E | $( |
| + | ABC+-D\*E | $(+ |
| F | ABC+-D\*EF | $(+ |
| ) | ABC+-D\*E+ | $ |
| …………… | ABC+-D\*E+$ | …… |

**Example 2: Convert the infix expression ((A+B)\*D)^E into postfix expression. Show all the possible steps in stack**

|  |  |  |  |
| --- | --- | --- | --- |
| Steps | Input | Stack | Postfix |
| 1 | ( | ( | …. |
| 2 | ( | (( | …. |
| 3 | A | (( | A |
| 4 | + | ((+ | A |
| 5 | B | ((+ | AB |
| 6 | ) | ( | AB+ |
| 7 | \* | (\* | AB+ |
| 8 | D | (\* | AB+D |
| 9 | ) | - | AB+D\* |
| 10 | ^ | ^ | AB+D\* |
| 11 | E | ^ | AB+D\*E |
| 12 | … | Poped stack | AB+D\*E^ |

**Evaluating the postfix expression**

Each operator in a postfix expression refers to the previous two operands in the expression. To evaluate the postfix expression, we use the following procedure

* Each time we read an operand we push it onto a stack
* When we reach an operator, its two operands will be the top two elements on the stack
* We can then pop these two elements perform the indicated operation on them and then we push result back to stack.

**Example:**

10 20 + 30\*

30 30 \*

900

**Trace of Evaluation**

**ABC+\*CBA-+\***

**123+\*321-+\***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Scanned Character | Value | Op2 | Op1 | Result | vstack |
| A | 1 | … | … | … | 1 |
| B | 2 | …. | … | … | 1 2 |
| C | 3 | …. | …. | … | 1 2 3 |
| + | …. | 3 | 2 | 5 | 1 5 |
| \* | …. | 5 | 1 | 5 | 5 |
| C | 3 | …. | ….. | … | 5 3 |
| B | 2 | …. | ….. | …. | 5 3 2 |
| A | 1 | …. | ….. | … | 5 3 2 1 |
| - | …. | 2 | 2 | 1 | 5 3 1 |
| + | …. | 3 | 3 | 4 | 5 4 |
| \* | …. | 5 | 5 |  | 20 |

**Application of Stack**

**Delimiter Matching**

The delimiter are the braces ‘{‘and ‘}’ brackets ‘[‘ and ‘]’ and parentheses ‘(‘ and ‘)’. Each opening or left delimiter should be matched by a closing delimiter. Also, opening delimiter that occurs later in the string should be closed before those occurring earlier.

**Delimiter matching Algorithm**

1. start
2. Read character string from keyboard as **“input”**
3. Read one character at a time say ‘ch’ from **“input”** string
4. While there is character in **“input**
5. If ch == ‘(‘ OR ch == ‘[‘ OR ch==’{‘

Push (ch)

1. Else if ch alphabet then continue
2. Else if ch ==’)’ OR ch==’}’ OR ch==’]’ then

p = pop ()

If **ch** and **p** do not match the print “**Error**”

1. Read next character
2. If stack is empty

Print” **Correct”**

Else

Print**” Incorrect**”

1. Stop

**Example**: Convert the given infix expression (A+B\*D) into prefix expression

Solution: Reverse the given string: )D\*B+A)

Apply infix to postfix conversion algorithm

|  |  |  |
| --- | --- | --- |
| Input | Prefix | Stack |
| ) | - | ) |
| D | D | ) |
| \* | D | )\* |
| B | DB | )\* |
| + | DB\* | )+ |
| A | DB\*A | )+ |
| C | DB\*A | ……. |
| .. | DB\*A+ | ……. |
| .. | Reverse it | ……. |
| .. | +A\*BD | ……. |

Infixfix String : (A+B\*D)

Prefix String : +A\*BD

**Unit 3: Queue (4 Hrs)**

3.1 Basic Concept of Queue, Queue as an ADT, Primitive Operations in Queue

**What is Queue?**

**Queue** is a linear data structure in which insertion and deletion operation performed at two different ends. The insertion operation is performed at rear end and deletion operation is performed at front end. The insertion operation is called **enQueue** operation and deletion operation is called **deQueue** operation. Rear is incremented while inserting an element and front is incremented while deleting an element.

Queue follows FIFO principle.

**Queue as an ADT**

A queue Q of type T is a finite set of elements together with the following operations.

* **isFull (Q):** Checks if the queue is full or not. Returns 1 if queue is full and 0 otherwis
* **isEmpty(Q):** Checks if the queue is not empty or not. Return 1 if queue is empty and 0 otherewise.
* **enQueue(Q, el):** Inserts an item el at the rear end of queue
* **deQueue(Q):** Deletes an item from front part of queue.

**Implementation of Queue**

* Array implementation of Queue (Static Memory Allocation)
* Linked List Implementation of Queue (Dynamic Memory Allocation)

**Array Implementation of Queue:** In array implementation of queue, we take an array of fixed size. Two pointers front and rear are used to point front part and rear part of queue.

* Linear Array Implementation (Linear Queue)
* Circular Array Implementation (Circular Queue)

**3.2 Linear Queue, Circular Queue, Priority Queue, Queue Applications**

Linear Array Implementation (Linear Queue)

Algorithm | Program

Insert

if queue is full

print” Queue is Full”

else

set rear = rear+1

queue[rear]=element

If queue is empty

print” Queue is empty”

else

set front = front+1

element = queue[front]

print “Element is deleted”

IsFull

if q.rear==MAXSIZE-1

return 1

else

return 0

To Initialize

Set front=0, rear=-1;

IsEmpty

.

      if q.rear<q.front

return 1

else

return 0

**Problems in Linear Queue**

The linear arrangement of the queue always considers the elements in forward direction. The pointers front and rear always incremented as and when we delete or insert respectively. When the situation arises where rear is size-1 and front is more than 0 and according to our logic, queue is full even if there are some positions available. To overcome this situation the concept of circular queue was introduced.

**Circular Queue**

A circular queue is a queue in which the insertion of a new element is done at very first location of  the queue if the queue is full. When the rear part reaches at the last index of the linear queue but front part is not at first index of queue then in this case if we try to insert some elements then according to the logic when rear is N-1 then it encounters an overflow situation But are some elements are left blank at the beginning part of the array. To utilize those left over spaces more efficiently, a circular fashion is implemented in queue representation. The circular queue reassigns the rear pointer with 0 if it reaches N-1 and beginning elements are free and the process continues. Such queue are called circular queue.

Insertion

1. Check for full Queue as

if(front==(rear+1)%MAXSIZE)

print " Queue is full"

else

set rear = (rear+1)%MAXSIZE

1. cque[rear]=item;
2. Stop

void enQueue(int el)

{

if(isFull())

printf("Queue is full");

else

{

cq.rear = (cq.rear+1)%MAXSIZE;

cq.cque[cq.rear]=el;

}

}

Initialization

rear = front = MAXSIZE-1

Deletion

1. Checking for empty queue as

if (rear==front)

print" Queue is empty"

else

front = (front+1)%MAXSIZE

1. item = qque[front]
2. return item
3. stop

int deQueue()

{

if(isEmpty())

printf("Queue is empty");

else

{

cq.front = (cq.front+1)%MAXSIZE;

return cq.cque[cq.front];

}

}

**Priority Queue :** Priority Queue is an extension of queue with following properties

1. Every item has a priority associated with it
2. An element with high priority is dequeued before an element with low priority
3. if tw
4. elements have the same priority, they are served according to their order in the queue.

**Types Priority Queue**

**Descending Order Priority Queue (Max Priority Queue):-**  In this priority queue, the elements are inserted as in ordinary queue but while dequeuing, the maximum element is deleted. We perform following operations on max priority queue

* **isEmpty():-** To check if the queue is empty or not
* **insert():-** Inserts a new value into the queue
* **findMax():-** Find maximum value in the queue
* **remove():-** Delete maximum value from the queue

**Ascending Order Priority Queue(Min Priority Queue):-** Min Priority Queue is similar to max priority queue but while removing, we remove a minimum element. The following operations are performed on min priority queue

* **isEmpty() :-**To check whether queue is Empty
* **insert ():-** Inserts a new value into the queue
* **remove():-** Delete minimum value from the queue
* **findMin():-** Find minimum value in the queue

**Priority Queue as ADT**

An ascending priority queue of elements of type T is a finite sequence of elements of type T together with the following operations

* **MakeEmpty(P):-** Create an empty priority queue p
* **isEmpty(p):-** Check to see if the priority queue is empty or not
* **isFull(p):** Check to see if priority queue is full or not
* **insert(p,x):-** Add element x on the priority queue p
* **deleteMin(p):-**If the priority queue p is not empty, remove the minimum element of the queue and return it.
* **findMin(p):-** Retrieve the minimum element of the priority queue p

**Unit 4: Recursion (3 Hrs)**

4.1 Principle of Recursion, Comparison between Recursion and Iteration, Tail recursion

**What is Recursion ?**

Recursion is a technique that allows us to break down a problem into one or more sub-problems that are similar in form to the original problem. An algorithm is called recursive if it solves the problem by reducing it to an instance of the same problem with smaller input.

A function is called recursive function if it calls itself.

**Comparison between Recursion and Iteration**

|  |  |  |
| --- | --- | --- |
| S.N | Recursion | Iteration |
| 1 | It is a technique of defining anything in term of itself | It executes until some condition is satisfied |
| 2 | Not all problems have recursive solution | Any recursive problem can be solved using iteration |
| 3 | No need of looping | Need of looping |
| 4 | Its uses system stack | It does not use system stack |
| 5 | It is slower technique | It is faster technique |
| 6 |  |  |
| 7 |  |  |
| 8 | int sum(int n)  {  if (n==1) return n;  else  return (n+sum(n-1)  } | int sum(int n)  {  Int I, s=0;  for(i=1;i<=n;i++)  s = s+I;  return s;  } |

4.2: Factorial, Fibonacci Sequence, GCD, Towers of Hanoi (TOH)

Factorial

int factorial(int n)

{

if(n<=1)

return 1;

else

return n\*factorial(n-1);

}

Fibonacci Sequence

int fib(int n)

{

if(n<=1)

return n;

else

return fib(n-1)+fib(n-2);

}

Power x^y

int power(int x,int y)

{

if(y==0)

return 1;

else

return x\*power(x,y-1);

}

Product

int product(int x,int y)

{

if(y==0)

return 0;

else

return x+product(x,y-1);

}

Sum of digits of any number

int digitsum(int n)

{

if(n<10)

return n;

else

return (n%10+digitsum(n/10));

}

Sum Of no from 0 to 100

int sum(int n)

{

if(n==100)

return n;

else

return n+sum(n+1);

}

GCD

int gcd(int a,int b)

{

if(b==0)

return a;

else

return gcd(b,a%b);

}

Reverse of any number

int reverse(int n,int len)

{

if(n==1)

return n;

else

return ((n%10)\*pow(10,len-1))+reverse(n/10,len-1);

}

**Tower of Hanoi Problem**

Initial State

* There are three poles named as origin, intermediate and destination
* n number of different sized disks having hole at the center is stacked around the origin pole in decreasing order
* The disks are numbered as 1,2,3,4...n

**Objective:**

Transfer all disks from origin pole to destination pole using intermediate pole for temporary storage

Condition:

* Move only one disk at a time
* Each disk must always be placed around one of the pole
* Never place larger disk on the top of the smaller disk

Algorithm:

To move a tower of n disks from source to destination ( where n is positive integers)

1. If n==1

1.1. Move a single disk from source to destination

1. if n>1

2.1 Move n-1 disk from source to intermediate disk

2.2 Move a single disk from source to destination

2.3 Move n-1 disk from intermediate disk to destination

1. Stop

#include<stdio.h>

#include<conio.h>

#include<math.h>

void TOH(int n,char A, char B, char C);

int main()

{

int n=3;

TOH(n,'A','B','C');

getch();

return 0;

}

void TOH(int n,char A, char B, char C)

{

if(n==1)

printf("Move disk %d from %c to %c\n",n,A,C);

else

{

TOH(n-1,A,C,B);

printf("Move disk %d from %c to %c\n",n,A,C);

TOH(n-1,B,A,C);

}

}

4.3 Application and Efficiency of Recursion

**Unit 5 – Lists**

**Basic Concept of List**

**Linked list** is a linear collection of data elements, whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence.

Each item in the list is called a **node** and contains two fields, an **information field** and a **next address field**. The information filed holds the actual element in the list. The next address field contains the address of next node in the list. Such an address which is used to access a particular node is called **pointer**. The next address field of the last node in the list contains a special value, known as **null**, which is not valid address. This **null pointer** is used to signal the end of the list. The list with no nodes on it is called the **empty list** or the **null list**.

**List as an ADT**

Linked List is an Abstract Data Type (ADT) that holds a collection of Nodes, the nodes can be accessed in a sequential way. Linked List doesn’t provide a random access to a Node.

**List Operations**

Insertion

* Add at beg :- add a node at beginning of the list
* Add at end :- add a node at end of the list
* Add at pos :- add a node at a given position of the list

Deletion

* Del from beg :- delete a node from beginning of the list
* Del from end :- delete a node from end of the list
* Del a node :- delete a given node of the list
* Del at pos :- delete a node at given position of the list

**Array Implementation of Lists**

Let Initially,

List->5->7->6->9->N

List->4->5->7->6->9->N //Add 4 at beg

List->4->5->7->6->9->10->N //Add 10 at end

List->4->5->11->7->6->9->10->N //Add 11 at pos 3

List->5->11->7->6->9->10->N //Del from beg

List->5->11->7->6->9->N //Del from end

List->5->11->6->9->N //Del node 7

List->5->11->9->N //Del node at pos 3

**Types of Linked Lists**

**Singly Linked List**

In a singly linked list, each node in the list stores the **contents** and a **pointer** or reference to the next node in the list. It does not store any pointer or reference to the previous node.

**List->5->7->9->11->N**

**Doubly Linked List**

A doubly linked list is a kind of linked list with a link to the **previous node** as well as a **data point** and the link to the **next node** in the list as with singly linked list. A sentinel or null node indicates the end of the list.

**DList->|N|5| <->|7|<->|9|<->|11|N|**

**Circular Linked List**

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

**CList->5->7->9->11->**

**Linked List Implementation of stack and queue**

Stack

The operation of adding an element to the front of a linked list is similar to that of pushing an element onto a stack. Similarly, the operation of deleting the first element of a linked list is similar to that of popping an element from a stack.

Queue

The insertion operation in a queue is similar to add at end of the list. Similarly, the deletion operation in a queue is similar to that of the delete from beginning of the list.

**Unit 6: Sorting (8 Hrs)**

**Introduction**

**Sorting** is the process of ordering elements in an array in some specific order. For example ascending order or descending order on the basis of values. Sorting is categorized as internal sorting or external sorting.

1. **Internal Sorting**: By internal sorting means we are arranging the numbers within the array only which is in computer primary memory.
2. **External Sorting**: External sorting is the sorting of numbers from the external file by reading it from secondary memory.

**Importance of Sorting**

Searching a sorted list is more faster than unsorted list. Some other examples are

* Words in a dictionary are sorted
* Files in a directory are often listed in sorted order
* The index of a book is sorted
* A listing of course offerings at a university is sorted, first by department then by course number

**Bubble Sorting**

The simplest sorting algorithm is the bubble sorting. The basic idea behind the bubble sorting is to pass through the array sequentially several times comparing each adjacent elements in each pass and swapping adjacent elements if they are out of order.

Characteristics of Bubble Sorting

* Large elements are sorted first
* The best time complexity of bubble sorting is O(n) and worst time complexity is O(n2)
* The space complexity of bubble sorting is O(1) because only single additional memory space is required.

**Selection Sort**

The idea of algorithm is quite simple. Array is imaginary divided into two parts- sorted and unsorted one. At the beginning, sorted part is empty, while unsorted one contains whole array. At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one. When unsorted part becomes empty, algorithm stops.

total number of comparisons: N-1+N-2+……+3+2+1 = N(N-1)/2

The complexity of the selection sort is O(N2). Moreover, the complexity of bubble sort and selection sort is same but selection sort executes faster than bubble sort. This is because of less number of swapping operations.

**Insertion Sort**

In the insertion sort, elements are inserted into their proper location such that array will be sorted.

Total number of comparisons will be:

1+2+3+4+………………….(N-1) = N\*(N-1)/2.

Therefore complexity of insertion sort in worst case: O(N2)

**Shell Sort**

The first algorithm to improve on the insertion sort substantially was Shell sort which was discovered in 1959 by Donald Shell. Though it is not -the fastest algorithm known, Shell sort is a sub-quadratic algorithm whose code is only slightly longer than the insertion sort making it the simplest of the faster algorithms. Shells idea was to avoid the large amount of data movement first by comparing elements that were far apart and then by comparing elements that were lass far apart and so on, gradually shrinking towards the basic insertion sort.

**Divide and Conquer Algorithms**

An important problem solving technique that makes use of recursion is divide and conquer. A divide and conquer algorithm is an efficient recursive algorithm that consists of two parts.

* Divide : in which smaller problems are solved recursively (except of course , base case)
* Conquer: in which the solution to the original in then formed from the solutions to the sub-problems.

**Quick Sort**

As its name implies, quick sort is a fast divide and conquer algorithm. Its average running time is O(nlogn). Its speed is mainly due to a very tight and highly optimized inner loop. It has quadratic worse case performance which can be made statistically unlikely to occur with a little effort.

Quick sort is very efficient sorting algorithm invented by CAR Hoare. It has two phases:

* The partition phase
* The sort Phase

**Quick sort is a divide and conquer algorithm**

* Divide: Partition the array A[l...r] into two sub-arrays A[l.....p] and A[p+1.....r] such that each element of first sub array is less than second sub-array.
* Conquer: Recursively sort each sub-array
* Combine: Trivial. The arrays are sorted in place. No additional task is required to combine them.

The recurrence relation for quick sort can be written as:

T(n) = 1 if n==1

2T(n/2)+n if n>1

Solving we get :- T(n) = O(nlogn)

**Merge sort**

The problem with quick sort is that its complexity in the worst case is O(n2) because it is difficult to control the partitioning process. Another strategy is to make portioning as simple as possible and concentrate on merging the two sorted arrays. This strategy is characteristic of merge sort.

The key process in merge sort is merging sorted halves of an array into one sorted array. However, these halves have to be sorted first, which is accomplished by merging the already sorted halves of these halves. The process of dividing arrays into two halves stops when the array has fewer than two elements. The algorithm is recursive in nature and can be summarized in the following pseudocode:

mergesort(data, first, last)

if first<last

mid = (first+last)/2;

mergesort(data, first, mid);

mergesort(data, mid+1, last);

merge(data, first, last);

**Heap Sort**

To have the array in ascending order, heap sort puts the largest element at the end of the array, then the second largest in front of it, and so on. Heap sort starts from the end of the array by finding the largest elements, whereas selection sort starts from the beginning using the smallest element. The final order in both cases is indeed the same.

A heap is a binary tree with the following two properties.

* The value of each node is not less than the values stored in each of its children.
* The tree is perfectly balanced and the leaves in the last level are all in the leftmost positions.

A tree has the heap property if it satisfies condition 1. Both conditions are useful for sorting, although this is not immediately apparent for the second condition. The goal is to use only the array being sorted without using additional storage for the array elements; by condition 2, all elements are located in consecutive positions in the array starting from position 0, with no unusual position inside the array. In other words, condition 2 reflects the packing of an array with no gaps.

Elements in a heap are not perfectly ordered. It is known only that the largest element is in the root node and that, for each other node, all its descendants are not greater than the element in this node. Heap sort thus starts from the heap, puts the largest element at the end of the array, and restores the heap that now has one less element. From the new heap, the largest element is removed and put in its final position and then the heap property is restored for the remaining elements. Thus, in each round, one element of the array ends up in its final position, and the heap becomes smaller by this one element. The process ends with exhausting all elements from the heap.

The Buildheap function takes O(n). Function Heapsort used Buildheap and heapify function. A call to buildheap function takes O(N) time and there are N-1 calls to heapify that takes O(log2N) time. Therefore running time of Heapsort is O(NlogN).

**Unit 7 Searching and Hashing 6hrs**

**Searching**

Searching refers to finding a given data item in a set, list or array. In general searching is of two types: Linear search and binary search.

**Linear search or sequential search**

In sequential search we search for a key in a sequential manner, accessing each element only once from beginning of the data structure. These include arrays, lists and sequential files. This search will take O(n) worst case. We would generally do a sequential search when:

* There are very few items or search is rare compared to other operations.
* The items are ordered by frequency and there is a strong likelihood that only the top few items will be needed.
* Items cannot be ordered (keys can be equal or not equal to other keys)
* It is too expensive to convert the data structure to a more flexible one.

If elements is found then search is said to be successful else unsuccessful. The C code for the same is give below. Here array is assumed to be unsorted.

Algorithm

1. Start
2. Read the search element from the user
3. Compare the search element with the first element in the list
4. If both are matching, then display " Given element found" and terminate the operation
5. If both are not matching then compare search element with the next element in the list
6. Repeat steps 4 and five until we are at last element
7. If the search element is not matched with the last element in the list then print " search element not found" and terminate the operation
8. Stop

**Complexity of Linear Search**

Complexity of any searching method depends on the number of comparisons performed to search for a specific element in the array of N elements. In case element is found within the very first comparison, it will be the best case for searching (least probable) and time complexity will be O(1). In the worst case element may be at the end of the list so that number of comparisons will be N. Time complexity in this case will be O(N). On an average case the number of comparisons will be approximately (N+1)/2. But still complexity will be O(N).

**Binary Search**

In binary search over the array of N elements, we first find out the middle position of the array. We then compare the middle element of the array with the data to be searched. If data is equal to the middle element, it is found. If data is element is less than middle element, it resides into the lower half of the array else it resides into the upper half of the array. The process is repeated for the other half of the array (lower and upper) and finally the element will be found as the middle element or search will be unsuccessful.

**Algorithm**

1. Start
2. Read the search element from the user
3. Find the middle element in the sorted list
4. Compare the search element with the middle element in the sorted list
5. If both are matching then display "Given element found" and terminate the function
6. If both are not matching then check whether the search element is smaller or larger than middle element
7. If the search element is smaller than middle element then repeat steps 2, 3, 4 and 5 for the left half of the list
8. If the search element is larger than middle element the repeat steps 2,3,4,and 5 for the right half of the list
9. Repeat the same process until search element is found or array size becomes 1.
10. stop

**Complexity of Binary Search**

Let us consider the size of an array be n. If key is in the middle of the array, the loop executes only one time. Otherwise, the algorithm looks at one of the halves of size n/2, then at one of the halves of this half, of size n/22 and so on, until the array is of size 1. Hence, we have the sequence n/2, n/22,……….n/2m and we want to know the value of m. But the last term of this sequence n/2m equals 1, from which we have m = logn. So the fact that k is not in the array can be determined after logn iterations of the loop.

**Hashing**

It is an efficient searching technique in which key is placed in direct accessible address for rapid search. Hashing provides the direct access of records from the file no matter where the records are in the file, Due to which it reduces the unnecessary comparisons. This technique uses a hashing function say h which maps the key with the corresponding key address or location.

**Hash Table**

A data structure that implements an associative array abstract data type, a structure that can map keys to values is hash table. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

**Hash Function**

A function that transforms a key into a table index is called a hash function. A hash function is any function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called hash values, hash codes, hash sums or simply hashes.

**Types of Hash Functions**

* Division
* Folding
* Mid Square Function
* Extraction
* Collision Resolution
* Open Addressing
* Chaining
* Bucketing Addressing

**Hash Collision or hash clash**

A situation in which two records cannot occupy the same position is called hash collision.

**Dealing with hash collision**

**Rehashing**

It involves using a secondary hash function on the hash key of the item. The rehash function is applied successively until an empty position is found where the item can be inserted. If the hash position of the item is found to be occupied during a search, the rehash function is again used to locate the item.

**Chaining**

It builds a linked list of all the items whose keys hash to the same values. During search this sort linked list is traversed sequentially for the desired key. The technique involves adding extra link field to each table position.

Inserting an element in hash table

Deleting an element from hash table

**Unit 4: Trees and Graphs (8 Hrs)**

**Trees**

Linked list usually provide greater flexibility than array, but they are linear structures and it is difficult to use them to organize a hierarchical representation of objects. To overcome these limitations, we create a new data type called a tree that consists of nodes and arcs. A tree can be defined recursively as the following:

1. An empty structure is an empty tree.
2. If t1,……tk are disjointed trees, then the structure whose root has its children the roots of t1,,……… tk is also a tree.
3. Only structures generated by rules 1 and 2 are trees.

**Key Terminologies**

**Root:**  A tree contains a unique first node which is shown at the top of the tree structure. This node is called root of the tree.

**Leaf Nodes**: The nodes which do not have children are called leaf nodes.

**Interior Nodes:** The nodes which have children nodes are called interior nodes.

**Siblings:** If two or more nodes have same parent, then these nodes are called siblings to each other.

**Ancestor:** A node is called ancestor of another node if either it is the parent of that node or it is the parent of some other ancestor of that node.

**Descendents:** A node is called Descendents of another node if it is the child of that node or the child of some other descendents of that node.

**Depth of tree:** The length of the longest path from root to any other node is known as the depth of the tree. Path is the number of edges from root to any node.

**Binary Tree**

A binary tree is type of tree with finite number of elements and is divided into three main parts. The first part is called root of the tree and other two parts are itself binary tree which exists towards left and right of the tree. Each of the elements in the binary tree is considered a node and a node will have three piece of information: data, two references

**Advantages of Binary Tree**

Searching in Binary Tree becomes faster

Maximum and minimum element can be directly picked up.

It is used for graph traversal

It is used to convert infix expression to postfix and prefix expression

**Types of Binary Tree**

**Strictly Binary Tree**

It is a binary tree with non-empty right and left sub trees. In other words, it is binary tree with every node N has either 0 or 2 tree. The strictly binary tree is also known as extended 2 Tree or simply 2 –tree. Sometimes nodes with 2 children's are known as internal nodes and nodes with 0 children are known as external nodes.

**Complete Binary Tree**

It is a special type of strictly binary tree where all the leaves of the tree reside at the same level. Using the depth we always say a complete binary tree of depth d where all leaves with be at level d

**Properties of Complete Binary Tree**

1. A binary tree of height h with no missing node
2. All leaves are at height h and all other nodes have no children
3. All the nodes that are at a level less than h have two children

**Almost Complete Binary Tree**

A binary tree of depth d is an almost complete binary tree if

* Each leaf in the tree is either at level d or at level d-1
* For any node "nd" in the tree with right child descendant at level d, all the left descendants of nd that are leaves are also at level d. It means nodes should be present in left to right at any level: there should be any missing nodes in the traversal

**Binary Search Tree**

The application of binary tree is searching and sorting. By enforcing certain rules on the values of the elements stored in a binary tree, it could be used to search and sort.

Binary search tree is a tree in which value of each node in the tree is greater than the value of node in its left (if exists) and it is less than the value in its right child (if it exists).

As name suggests, binary search tree (BST) is used for searching purpose. For every node N in the BST, the following property will be true.

* The data at left node will be smaller than data at node N
* The data at right node will be larger than data at node N

**Figure: Binary Search Tree**

**Implementing Binary Trees**

**Array Implementation of Binary Tree**

The sequential representation of binary tree uses array for storing the data for each node. This is very simple. If any parent node is stored at index I then its left child will be stored at 2\*I+1 and right child will be stored at 2\*I+2. The root of the tree is stored at first index of the array (index 0).

The array representation of above BST is as shown below:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 12 | 9 | 18 | 1 |  | 14 | 21 |  |  |  |

It is clear that the most of the locations in array are empty. This causes wastage of memory space. That is, the array representation of binary tree is quite inefficient. In general for a binary tree with height H, the size of the array will be approximately be 2H.

**Linked List Representation of Binary Tree**

The linked list representation is the most popular, efficient and most frequently used representation of binary tree. In the linked list representation every node is represented by data, a reference to left child and a reference to right child. The node of binary tree can be created as follows:

struct BNode

{

int info;

struct BNode \*left;

struct BNode \*right;

}

A binary search tree(BST) is a binary tree that is either empty or in which every node contains a key and satisfies following conditions.

* All keys in the left sub-tree of the root are smaller than the key in the root node.
* All keys in the right sub-tree of the root are greater than the key in the root node.
* The left and right sub-tree of the root are again binary search trees

**Construct a BST from the following sequence of numbers: 14, 15, 4, 9,7,18,3,5,16,,4,20,17,9**

**Insertion**

To insert a new node with key el, a tree node with a dead end has to be reached, and new node has to be attached to it The algorithm for inserting an item into binary search tree is as follows:

1. Start
2. Set r = root, save = root
3. Create new node

BNode temp = (struct BNode \*) malloc (sizeof(BNode))

1. Temp->info = data, temp->right = null, temp->right = null
2. If root == null

root = temp;

1. Otherwise
2. While (r!=null)

7.1 Save = r

7.2 if(data<r->data)

r = r->left;

else

r = r->right;

1. If(data<save->data)

Save->left = temp;

1. Else
2. Save->right = temp;
3. stop

**Deletion**

Algorithm for deletion of a node in BST.

1. Start
2. If a node to be deleted is a left leaf node then simply delete this node and set null pointer to its parent’s left pointer.
3. If a node to be deleted is right leaf node then simply delete this node and set null pointer to it’s parent’s right child.
4. If a node to be deleted has one child then connect it’s child pointer with it’s parent pointer and delete it from the tree.
5. If a node to be deleted has two children then replace the node being deleted either by
6. Right most of its left sub sub-tree or
7. Left most node of it’s right sub-tree.
8. End