MODULE

DATA STRUCTURES

RENAISSANCE 2017

Name –	______________________
Mob No. –	______________________

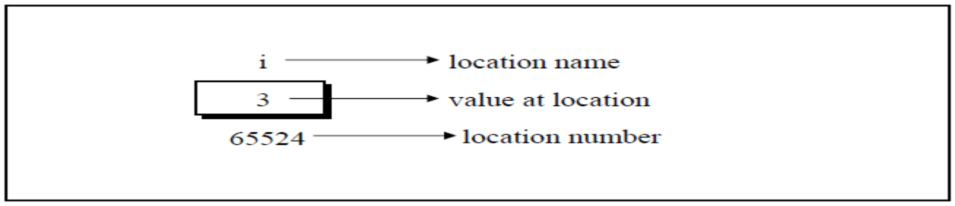
CHAPTER-1
POINTERS

We will learn to implement the pointers using C that is of immense help later, when we construct data structures using pointers in C.

Pointer Notation

Consider the declaration,

int i = 3 ;   // i is a variable. Its value is set as 3.



We see that the computer has selected memory location 65524 as the place to store the value 3.The location number 65524 is not a number to be relied upon, because some other time the computer may choose a different location for storing the value 3.The important point is, i's address in memory is a number.

Reference Operator

You can obtain the address of a variable by using the reference operator &, also called the address operator. The expression &n evaluates to the address of the variable n. We can print this address number through the following program:

```
1. main( )
2. {    int i = 3 ;
3. printf ( "\nAddress of i = %u", &i ) ;     // & - " address of " operator
4. printf ( "\nValue  of i = %d", i ) ;  }
```

The output of the above program would be:

```
Address of i = 65524
Value of i = 3
```

The expression &i returns the address of the variable i, which in this case happens to be 65524. Since 65524 represent an address, there is no question of a sign being associated with it. Hence it is printed out using %u, which is a format specifier for printing an unsigned integer.

Pointer Operator

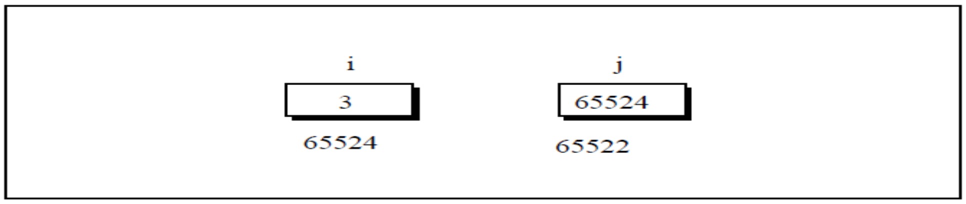
The other pointer operator available in C is '*', called 'value at address' operator. It gives the value stored at a particular address. The 'value at address' operator is also called 'indirection' operator. Note that printing the value of *(&i) is same as printing the value of i.

Pointer Declaration

Consider the declaration

```
j = &i;                         // j points to i. ( j stores address of i)
```

j is a variable that contains the address of other variable (i in this case).



Points to Note:

1. We can't use j in a program without declaring it. it is declared as,
   a. int *j ;   // declaring j as a pointer.
2. This declaration tells the compiler that j will be used to store the address of an integer value. In other words j points to an integer.
3. Let us go by the meaning of *. It stands for 'value at address'. Thus, int *j would mean, the value at the address contained in j is an int.

Look at the following declarations,

```
int *alpha ;
char *ch ;
float *s ;
```

The declaration float *s does not mean that s is going to contain a floating-point value. It means that s is going to contain the address of a floating-point value. Similarly, char *ch means that ch is going to contain the address of a char value.

Here is a program that demonstrates the relationships we have been discussing.

```
main( )
{int i = 3 ;
int *j ;
j = &i ;
printf ( "\nAddress of i = %u", &i ) ;        // & -' address of ' operator
printf ( "\nAddress of i = %u", j ) ;
printf ( "\nAddress of j = %u", &j ) ;
printf ( "\nValue of j = %u", j ) ;           // value of j = address of i ( j points to i )
printf ( "\nValue of i = %d", i ) ;
printf ( "\nValue of i = %d", *( &i ) ) ;
printf ( "\nValue of i = %d", *j ) ;
getch(); }          //  * - ' value at address' operator
```

The output of the above program would be:

```
Address of i = 65524
Address of i = 65524
Address of j = 65522
Value of j = 65524
Value of i = 3
Value of i = 3
Value of i = 3
```

Pointer arithmetic

Not all forms of arithmetic are permissible on a pointer: only those things that make sense. Considering that a pointer is an address somewhere in the computer, it would make sense to add a constant to an address, thereby moving it ahead in memory that number of places. Similarly, subtraction is permissible, moving it back some number of locations. Adding two pointers together would not make sense because absolute memory addresses are not additive. Pointer multiplication is also not allowed, as that would be a 'funny' number.

Consider the following:

int myarray[] = {1,23,17,4,-5,100};

Here we have an array containing 6 integers. We refer to each of these integers by means of a subscript to myarray, i.e. using myarray[0] through myarray[5]. But, we could alternatively access them via a pointer as follows:

1. int *ptr;
2. ptr = &myarray[0]; /* point our pointer at the first integer in our array */

And then we could print out our array either using the array notation or by dereferencing our pointer. Now, pointer equivalent to array notation as follows:

1. ptr = &myarray[0]; (or) *ptr = my_array[0];
2. ptr+1 = &my_array[1]; (or) *(ptr+1) = my_array[1];
3. ptr+2 = &my_array[2]; (or) *(ptr+2) = my_array[2]; and so on…

++ptr and ptr++ are both equivalent to ptr + 1 (though the point in the program when ptr is incremented may be different), incrementing a pointer using the unary ++ operator, which is pre- and post-increments respectively. --ptr and ptr-- are both equivalent to ptr-1 and work in the similar way as pre and post increment operators as illustrated below:

```
#include<stdio.h>
main()
  {
      int my_array[5]={2,-9,12,1,3};
      int *ptr ;
      ptr = &my_array[0];
      printf("\n %d", *(++ptr));
      printf("\n %d", *(++ptr));
      return 0;
  }
```

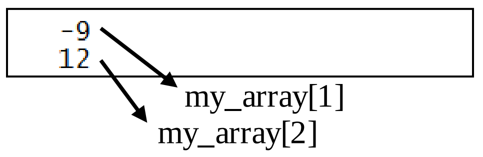
PRE-INCREMENT

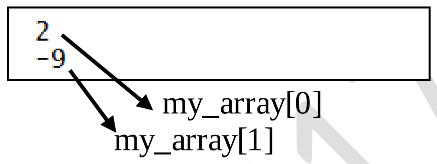
```
#include<stdio.h>
main()
  {
      int my_array[5]={2,-9,12,1,3};
      int *ptr ;
      ptr = &my_array[0];
      printf("\n %d", *(ptr++));
      printf("\n %d", *(ptr++));
      return 0;
  }
```

POST-INCREMENT

PRE –INCREMENT POST-INCREMENT

OUTPUT:





Tutorial

(A)If i is an integer and p and q are pointers to integers , which of the following assignments cause a compilation error ?

a) p = &i;	d) i=*&*p;	g) p =&*&i	j) q=*&p
b) p = *&i;	e) i= &*p;	h) q =*&*p	k) q=&*p
c) p = &*I;	f) i= *&p;	i) q =**&p	

(B)What would be the output of the following programs?

```
#include<stdio.h>
void fun(int*,int*);
main()
{int i=5,j=2 ;
fun(&i,&j);
printf ( "\n%d %d", i, j );
system("PAUSE");
return 0;}
void fun(int *i,int *j)
{*i = *i * *i ;       *j = *j * *j ;}
```

```
#include<stdio.h>
void fun(int*,int);
main()
{int i=5,j=2;
fun(&i,j);
printf ( "\n%d %d", i, j );
system("PAUSE");
return 0;}
void fun(int *i,int j)
{*i = *i * *i ;
j = j * j ;}
```

```
#include<stdio.h>
#include<conio.h>
main( )
{float a = 13.5 ;
float *b, *c ;
b = &a ; /* suppose address of a is 1006 */
c = b ;
printf ( "\n%u %u %u", &a, b, c ) ;
printf ( "\n%f %f %f %f %f", a, *(&a), *&a, *b, *c ) ;
system("PAUSE");
return 0;}
```

(C)Add the missing statement for the following program to print 35.

```
#include<stdio.h>
main( )
{int j, *ptr ;
*ptr = 35 ;
printf ( "\n%d", j ) ;
return 0;}
```

(D)True or false? Explain:

1. If (x == y) then (&x == &y)
2. If (x == y) then (*x == *y)

(E)What is wrong with the following code: int* p = &44;

(F)Determine the value of each of the indicated variables after the following code executes. Assume that each integer occupies 4 bytes and that m is stored in memory starting at byte 0x3fffd00. (pointer increment need to be explained with blocks.)

```
int m = 44;
int* p = &m;
int& r = m;
int n = (*p)++;
int* q = p - 1;
r = *(--p) + 1;
++*q;
```

m, n, &m, *p, r, *q

(G)If p and q are pointers to int and n is an int, which of the following are legal:

p + q
p – q
p + n

p – n
n + p
n – q

Practical

1. Write a function that uses pointers to copy an array of double.
2. Write a function that uses pointers to search for the address of a given integer in a given array. If the given integer is found, the function returns its address; otherwise it returns NULL.

Time for an interview…..

1. **Do you think we can assign a pointer to a function..? If yes , why..?**
2. **Can u provide us with a simple syntax to declare a pointer to the function.**
   My_func()

Notes

CHAPTER-2 STRUCTURES

When we handle real world data, we don't usually deal with little atoms of information by themselves—things like integers, characters and such. Instead, we deal with entities that are collections of things, each thing having its own attributes, just as the entity we call a 'book' is a collection of things such as title, author, number of pages, etc. As you can see, this data is dissimilar, for example, author is a string, whereas number of pages is an integer. For dealing with such collections, C provides a data type called 'structure'. A structure contains a number of data types grouped together. These data types may or may not be of the same type. The following example illustrates the use of this data type.

```
#include <stdio.h>
main()
{struct book
{ char name;
float price;
int pages;};
struct book b1, b2, b3;
printf( "\nEnter names, prices & no. of pages of 3 books\n");
scanf("%c %f %d %c %f %d %c %f %d ", &b1.name, &b1.price, &b1.pages,
        &b2.name,    &b2.price, &b2.pages, &b3.name, &b3.price, &b3.pages);
printf( "\nAnd this is what you entered");
printf( "\n%c %f %d", b1.name, b1.price, b1.pages);
printf( "\n%c %f %d", b2.name, b2.price, b2.pages);
printf( "\n%c %f %d", b3.name, b3.price, b3.pages); system("PAUSE");
return 0;}
```

And here is the output...

```
Enter names, prices and no. of pages of 3 books
A 100.00 354
C 256.50 682
F 233.70 512

And this is what you entered
A 100.000000 354
C 256.500000 682
F 233.700000 512
```

Declaring a Structure

In our example program, the following statement declares the structure type:

```
struct book
{ char name;
float price;
int pages;}
```

This statement defines a new data type called struct book. Each variable of this data type will consist of a character variable called name, a float variable called price and an integer variable called pages. Once the new structure data type has been defined, one or more variables can be declared to be of that type. For example, the variables b1, b2, b3 can be declared to be of the type struct book,

```
struct book b1, b2, b3;
```

This statement sets aside space in memory. It makes available space to hold all the elements in the structure—in this case, seven bytes—one for name, four for price and two for pages. These bytes are always in adjacent memory locations. If we so desire, we can combine the declaration of the structure type and the structure variables in one statement. For example:

```
struct book
{ char name;
float price;
int pages;}
b1,b2,b3;
```

Structure variables can also he initialized where they are declared.

```
struct book
{
char name[10];
float price;
int pages;
struct book b1= { "Basic", 130.00, 550 };
struct book b2= { "Physics", 150.80, 800 };
struct book b3={ 0 };
```

Note the following points while declaring a structure type:

(a) The closing brace in the structure type declaration must be followed by a semicolon.
(b) It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the 'form' of the structure.
(c) Usually structure type declaration appears at the top of the source code file, before any variables or functions are defined.
(d) If a structure variable is initiated to a value {0}, then all its elements are set to value 0, as in b3 above.

Accessing Structure Elements

Having declared the structure type and the structure variables, let us see how the elements of the structure can be accessed. Structures use a dot (.) operator. So to refer to pages of the structure defined in our sample program, we have to use, b1.pages; Similarly, to refer to price, we would use, b1.price ; Note that before the dot, there must always be a structure variable and after the dot, there must always be a structure element.

How Structure Elements are Stored

Whatever be the elements of a structure, they are always stored in contiguous memory locations. The following program would illustrate this:

```
# include <stdio.h>
main()
{struct book
{ char name;
float price;
int pages;};
struct book b1 = { 'B', 130.00, 550 };
printf ( "\nAddress of name = %u", &b1.name);
printf ( "\nAddress of price = %u", &b1.price);
printf ( "\nAddress of pages = %u", &b1.pages); system("PAUSE");
return 0;}
```

Here is the output of the program...

```
Address of name = 65518
Address of price = 65519
Address of pages = 65523
```

Array of Structures

In our sample program, to store data of 100 books, we would be required to use 100 different structure variables from b1 to b100, which is definitely not very convenient. A better approach would be to use an array of structures. Following program shows how to use an array of structures.

```
# include <stdio.h>
main()
{struct book
{ char name;
float price;
int pages;};
struct book b[2];
int i;
for(i=0;i<=1;i++)
{ printf( "\nEnter name, price and pages");
scanf( "%c %f %d",&b[i].name,&b[i].price,&b[i].pages);  }
for(i=0;i<=1;i++)
printf( "\n%c %f %d",b[i].name,b[i].price,b[i].pages);system("pause");
return 0;}
```

Now a few comments about the program:

(a) Notice how the array of structures is declared... struct book b[2]; This provides space in memory for 2 structures of the type struct book.

(b) In an array of structures, all elements of the array are stored in adjacent memory locations. Since each element of this array is a structure, and since all structure elements are always stored in adjacent locations, you can very well visualize the arrangement of array of structures in memory. In our example, b[0]'s name, price and pages in memory would be immediately followed by b[1]'s name, price and pages, and so on.

Additional Features of Structures

(A) The values of a structure variable can be assigned to another structure variable of' the same type using the assignment operator. This is shown in the following example.

```
# include <stdio.h>
# include <string.h>
main()
{struct employee
{ char name[10];
int age;
float salary;};
struct employee e1 = { "Sanjay",  30, 5500.50 };
struct employee e2, e3;
/* piece-meal copying*/
strcpy ( e2.name, e1.name); /*e2.name = el. name is wrong */
e2.age = e1.age;
e2.salary = e1.salary;
/* copying all elements at one go*/
e3=e2;
printf ( "\n%s %d %f", e1.name, e1.age, e1.salary);
printf ( "\n%s %d %f", e2.name, e2.age, e2.salary);
printf ( "\n%s %d %f", e3.name, e3.age, e3.salary);
system("PAUSE");
return 0;}
```

The output of the program would be...

```
Sanjay 30 5500.500000
Sanjay 30 5500.500000
Sanjay 30 5500.500000
```

(B) One structure can be nested within another structure. Using this facility, complex data types can be created. The following program shows nested structures at work.

```
# include <stdio.h>
main()
{struct address
{ char phone[15];
char city[25];
int pin;};
struct emp
{ char name[25];
struct address a;};
struct emp e = { "Nihal", "531046", "Nagpur", 10 };
printf ( "\nname = %s phone = %s", e.name, e.a.phone);
printf ( "\ncity = %s pin = %d", e.a.city, e.a.pin); system("PAUSE");
return 0;}
```

And here is the output...

```
name = Nihal phone = 531046
city = Nagpur pin = 10
```

(C) Like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure variables at one go. Let us examine both the approaches one by one using suitable programs.

```
/* Passing individual structure elements */
# include <stdio.h>
void display (char*, char*, int);
main()
{struct book
{ char name[25];
char author[25];
int callno;};
struct book b1 = { "Data", "YPK", 101};
display (b1.name,b1.author,b1.callno);
return 0;}
void display (char*s,char*t,int n)
{printf ( "\n%s %s %d", s, t, n);};
```

```
And here is the output...
Data YPK 101
```

Observe that in the declaration of the structure, name and author have been declared as arrays. Therefore, when we call the function display using, display (b1.name, b1.author, b1.callno); we are passing the base addresses of the arrays name and author, but the value stored in callno. Thus, this is a mixed call—a call by reference as well as a call by value. It can be immediately realized that to pass individual elements would become more tedious as the number of structure elements goes on increasing. A better way would be to pass the entire structure variable at a time. This method is shown in the following program.

```
# include <stdio.h>
struct book
{char name[25];
char author[25];
int callno;};
void display (struct book);
main()
{struct book b1 = { "Data",  "YPK",  101 };
display(b1); system("pause");
return 0;}
void display ( struct book b)
{printf ( "\n%s %s %d", b.name, b.author, b.callno);}
```

```
And here is the output...
Data YPK 101
```

In this case, it becomes necessary to declare the structure type struct book outside main(), so that it becomes known to all functions in the program.

(D) The way we can have a pointer pointing to an int, or a pointer pointing to a char, similarly we can have a pointer pointing to a struct. Such pointers are known as ‘structure pointers’. Let us look at a program that demonstrates the usage of a structure pointer.

```
# include <stdio.h>
main()
{struct book
{ char name[25];
char author[25];
int callno;};
struct book b1 = { " Data ", "YPK",  101 };
struct book *ptr;
ptr=&b1;
printf( "\n%s %s %d", b1.name, b1.author, b1.callno);
printf ( "\n%s %s %d", ptr->name, ptr->author, ptr->callno);  system("pause");
return 0;}
```

Passing address of a structure variable

```
# include <stdio.h>
struct book
{ char name[25];
char author[25];
int callno; };
void display (struct book *);
main()
{ struct book b1= { " Data ", "YPK",  101 };
display ( &b1); system("PAuse");
return 0;}
void display ( struct book *b)
{ printf("\n %s %s %d", b->name, b->author, b->callno);}
```

And here is the output...

```
Data YPK 101
```

Again note that, to access the structure elements using pointer to a structure, we have to use the '->' operator. Also, the structure struct book should be declared outside main() such that this data type is available to display() while declaring pointer to the structure.

Tutorial

[**A**]What would be the output of the following programs:

```
main( )
{struct gospel
{ int num ;
char mess1[50] ;
char mess2[50] ;
} m ;
m.num = 1 ;
strcpy ( m.mess1, "If all that you have is hammer" ) ;
strcpy ( m.mess2, "Everything looks like a nail" ) ;          /* assume that the structure is
located at address 1004 */
printf ( "\n%u %u %u", &m.num, m.mess1, m.mess2 ) ;
system("pause"); }
```

```
struct gospel
{int num ;
char mess1[50] ;
char mess2[50] ;}
m1 = { 2, "If you are driven by success", "make sure that it is a quality drive"} ;
main( )
{ struct gospel m2, m3 ;
m2 = m1 ;
m3 = m2 ;
printf ( "\n%d %s %s", m1.num, m2.mess1, m3.mess2 ) ;
system("Pause");}
```

[B] Point out the errors, if any, in the following programs:

```
main( )
{ struct employee
{ char name[25] ;
int age ;
float bs ; } ;
struct employee e ;
strcpy ( e.name, "Hacker" ) ;
age = 25 ;
printf ( "\n%s %d", e.name, age ) ;}
```

```
main( )
{struct
{char name[25] ;
char language[10] ;} ;
struct employee e = { "Hacker", "C" } ;
printf ( "\n%s %d", e.name, e.language ) ;}
```

[C] Answer the following:

1. Ten floats are to be stored in memory. What would you prefer, an array or a structure?

2. Given the statement, maruti.engine.bolts = 25 ; which of the following is True?
   a. structure bolts is nested within structure engine
   b. structure engine is nested within structure maruti
   c. structure maruti is nested within structure engine
   d. structure maruti is nested within structure bolts

3. State True or False:
   a. All structure elements are stored in contiguous memory locations.
   b. An array should be used to store dissimilar elements, and a structure to store similar elements.

c. In an array of structures, not only are all structures stored in contiguous memory locations, but the elements of individual structures are also stored in contiguous locations.

4. struct time
   {int hours ;
   int minutes ;
   int seconds ;} t ;
   struct time *tt ;
   tt = &t ;
   Looking at the above declarations, which of the following refers to seconds correctly:
   1. tt.seconds
   2. (*tt).seconds
   3. time.t
   4. tt -> seconds

5. Create a structure to specify data on students given below: Roll number, Name, Department, Course, Year of joining. Assume that there are not more than 450 students in the college.
   a. Write a function to print names of all students who joined in a particular year.
   b. Write a function to print the data of a student whose roll number is given.

6. There is a structure called employee that holds information like employee code, name, date of joining. Write a program to create an array of the structure and enter some data into it. Then ask the user to enter current date. Display the names of those employees whose tenure is 3 or more than 3 years according to the given current date.

7. Write a menu driven program that depicts the working of a library. The menu options should be:
   a. Add book information
   b. Display book information
   c. List all books of given author
   d. List the title of specified book
   e. List the count of books in the library
   f. List the books in the order of accession number
   g. Exit

   Create a structure called library to hold accession number, title of the book, author name, price of the book, and flag indicating whether book is issued or not.

Time for an interview.....

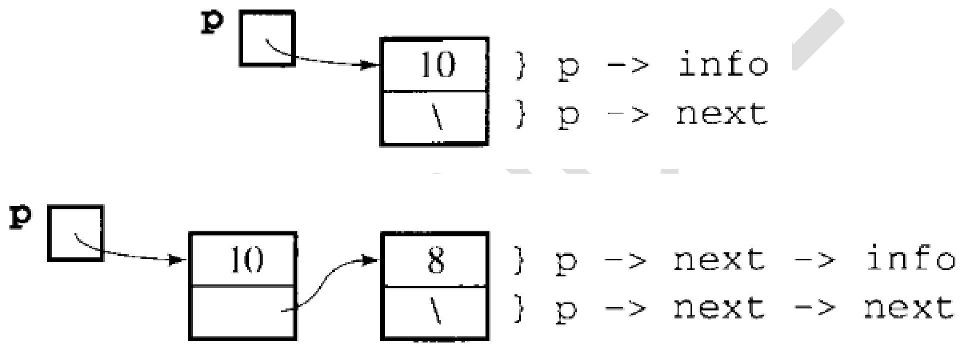
1 . Can we directly compare two structures…

2. Can you explain us what are the default access specifiers in structures and classes .

3. Can you explain us what are the major differences between structures of C and structures of C++ .

CHAPTER-3
LINKED LISTS

Linked list is a collection of items (nodes) arranged in a linear order, each item (node) stores data and links to other item. It is a basic data structure which is mostly implemented using either arrays or pointers. Pointer implementation of linked list is considered efficient. The linear order of the items in array implementation is achieved using array index. Each item in the list is called node and contains two fields, an information field and a next address field. The information field holds the actual element on the list. The next address field contains the address of the next node in the list. Such an address which is used to access a particular node is known as a pointer. The entire linked list is accessed from an external pointer list that points to (contains the address of) the first node in the list. (By an "external" pointer we mean one that is not included within a node)



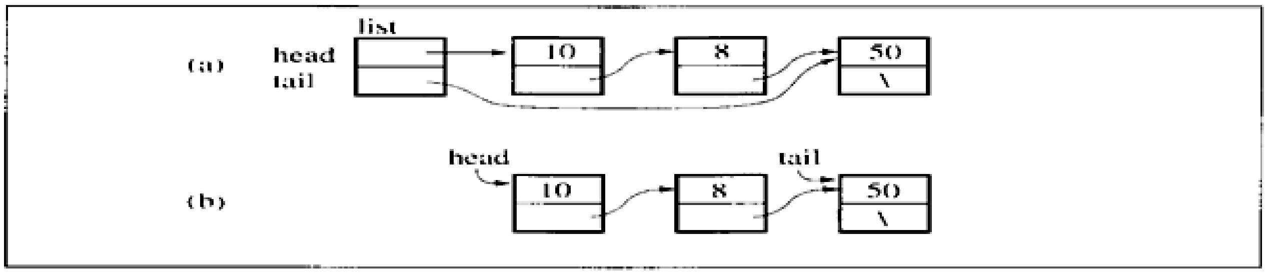
p - external pointer
info - information field
next - next address field (internal pointer)
\ - null pointer

The next address field of the last node in the list contains a special value, known as null, which is not a valid address. The null pointer is used to signal the end of the list. The list with no node in it is called the empty list or a null list. The value of the external pointer list to such a list is the null pointer. A list can be initialized to the empty list by an operation list =null. We now introduce some notation for use in algorithms (not in c programs).

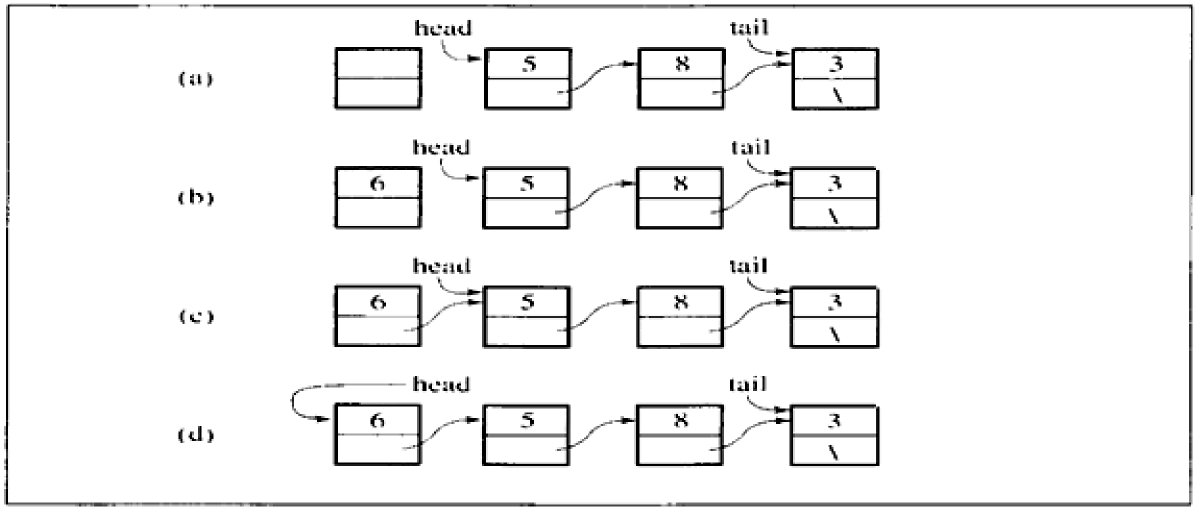
If p is a pointer to a node, node(p) refers to the node pointed by p, info(p) refers to the information portion of that node, and next(p) refers to the next address portion and so, it is a pointer. Thus, if next(p) is not null, info(next(p)) refers to the information portion of the node that follows node(p) in the list.

Singly linked list :

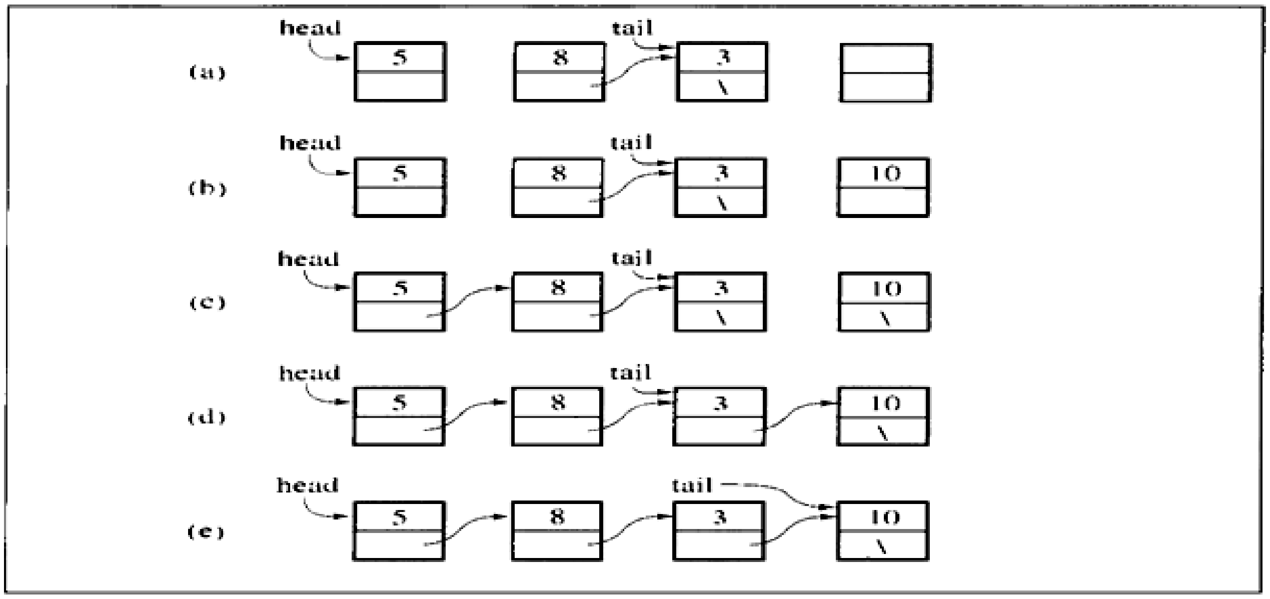
A singly linked list of integers.



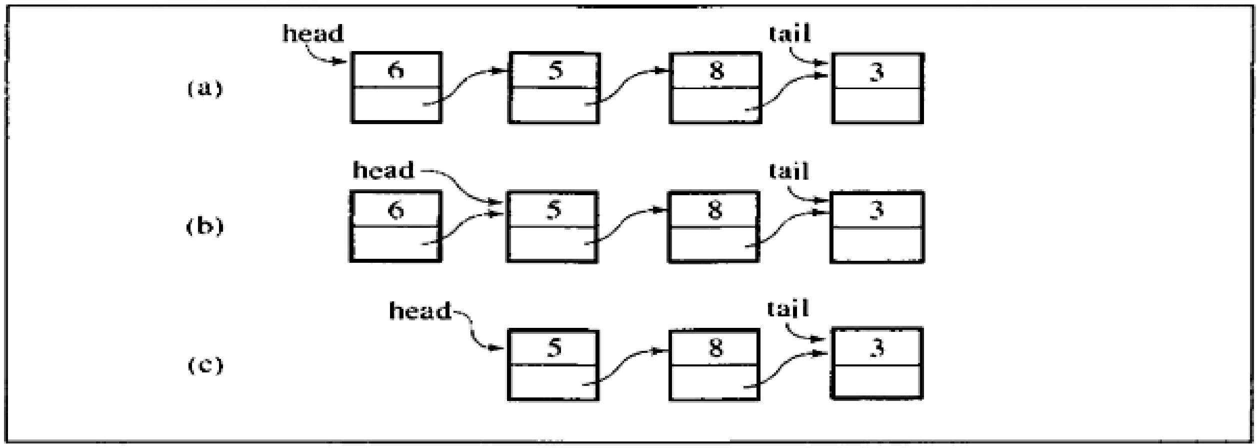
Inserting a new node at the beginning of a singly linked list.



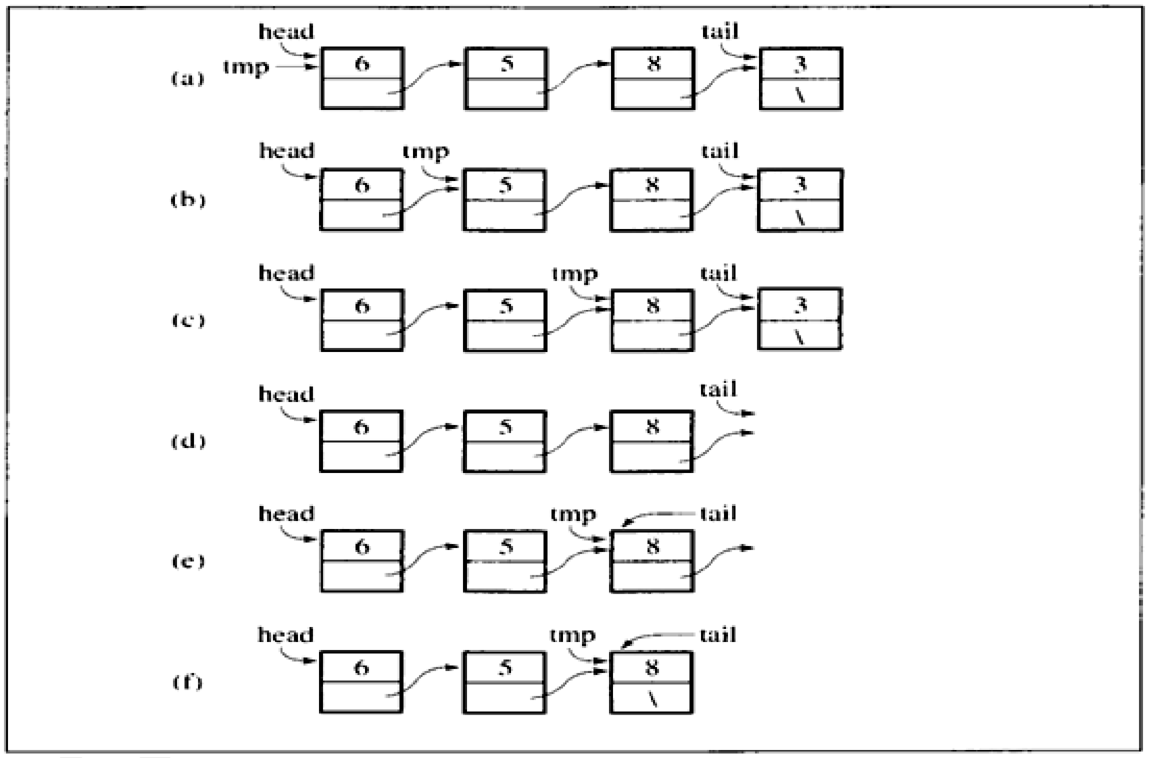
Inserting a new node at the end of a singly linked list.



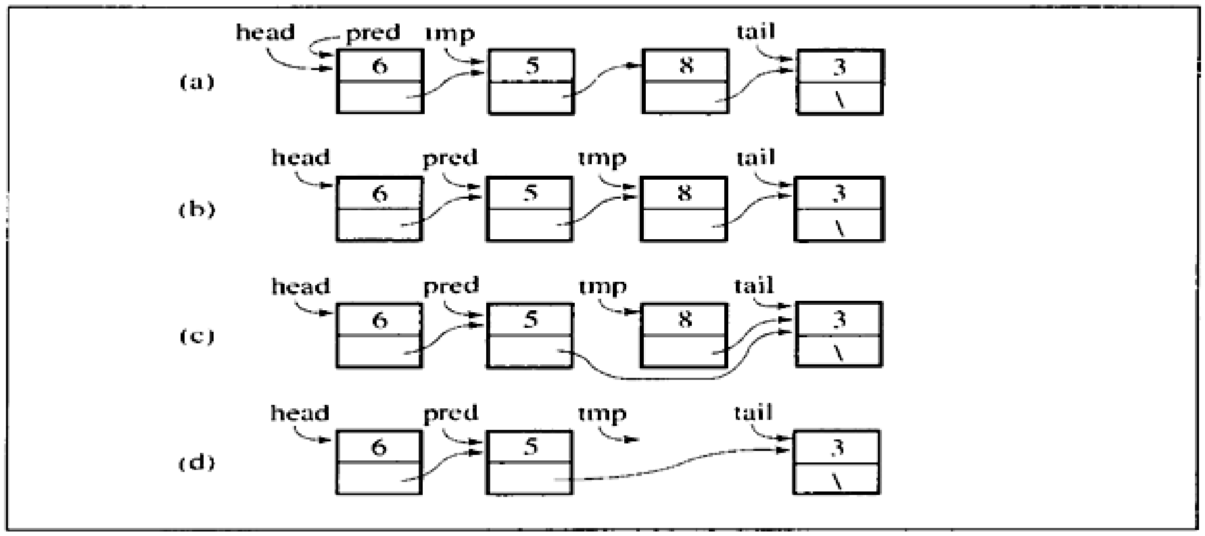
Deleting a node at the beginning of a singly linked list.



Deleting a node from the end of a singly linked list.

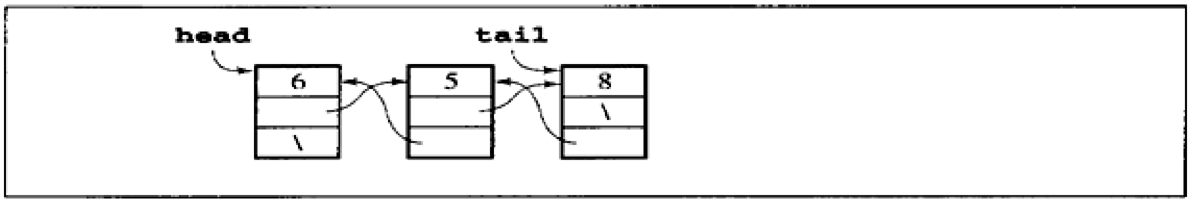


Deleting a node from a singly linked list.

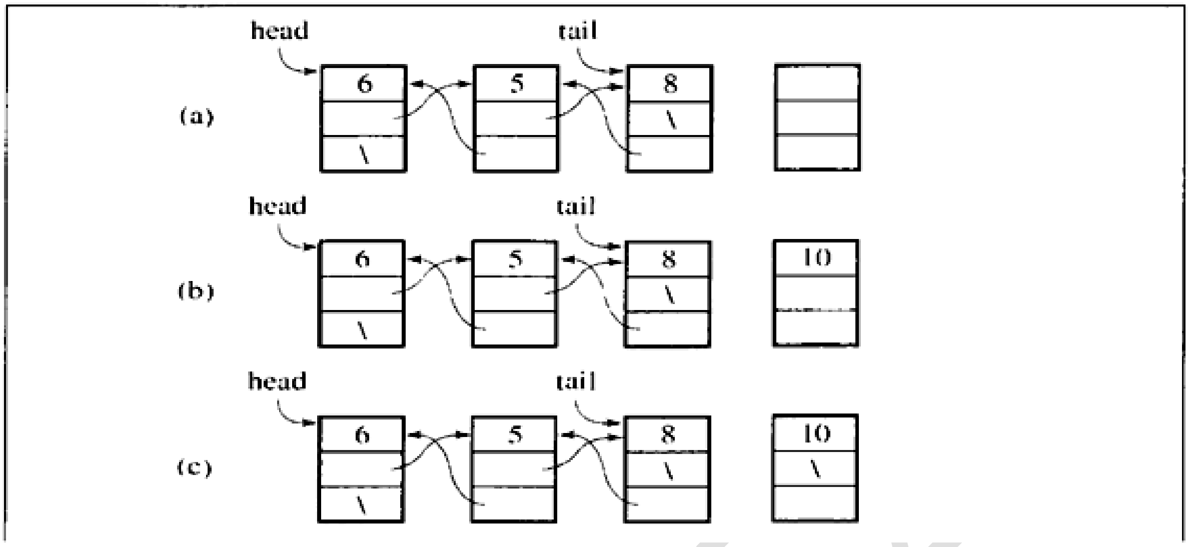


Doubly Linked Lists

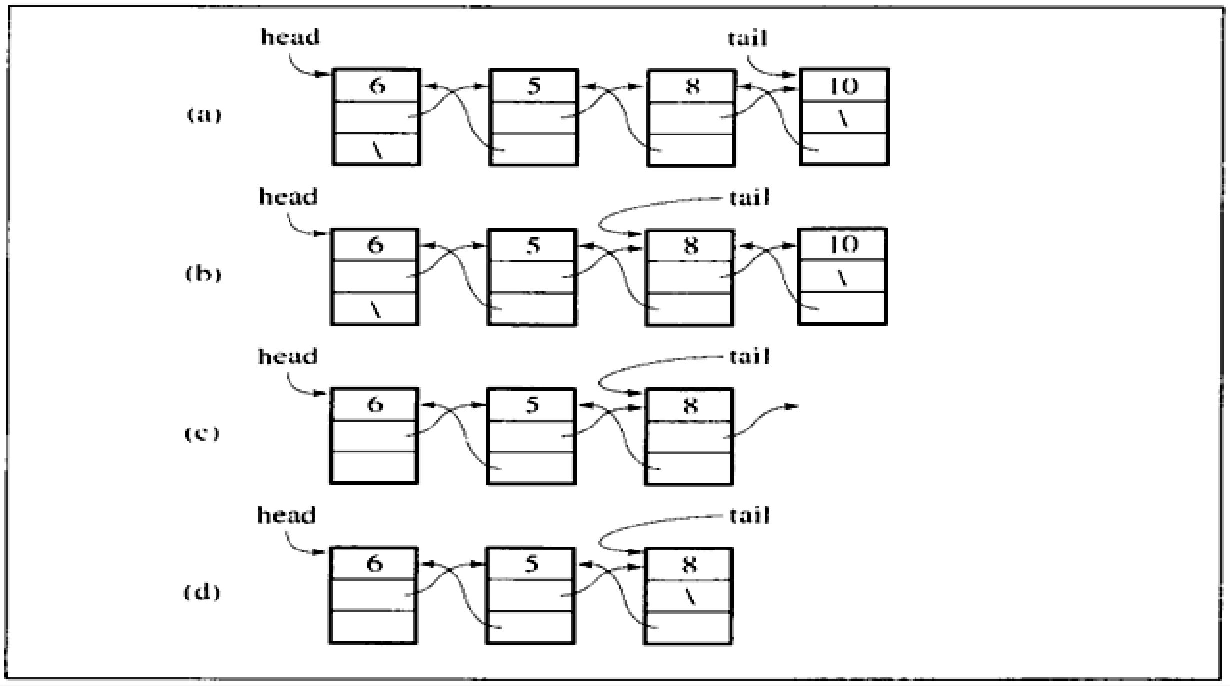
A doubly linked list.



Adding a new node at the end of a doubly linked list.

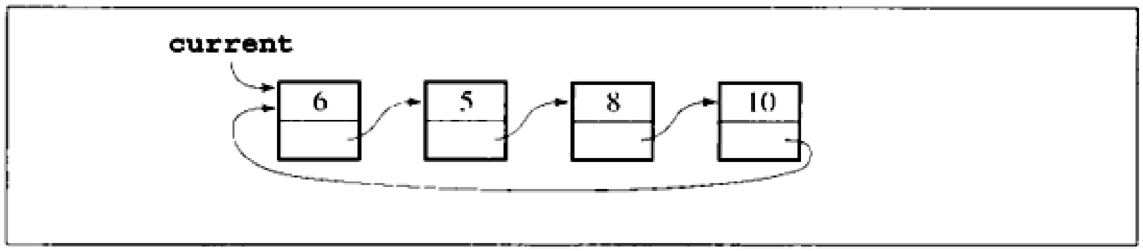


Deleting a node from the end of a doubly linked list.

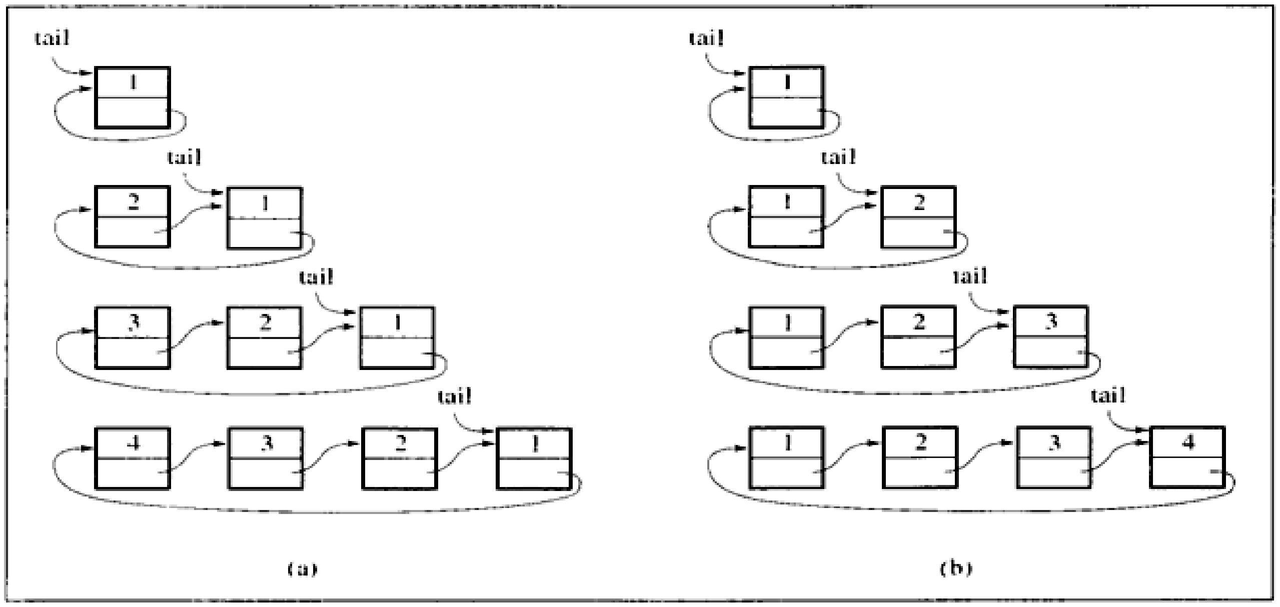


Circular Linked Lists

A circular singly linked list.



Inserting nodes at the front of circular singly linked list (a) and at its end (b).



<u>Implementation of Singly Linked list</u>

<u>Structure of node</u>

```
struct Node
 {int value;
  struct Node *next;};
 struct Node *head;
```

<u>To add node at the end</u>

```
void append_at_end (int number)
 { struct  Node *newNode, *nodePtr;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode -> value  = number;
  newNode -> next = NULL;
  if(head==NULL)
  head = newNode;
  else
  {nodePtr = head;
   while (nodePtr -> next)
   nodePtr = nodePtr -> next;
   nodePtr -> next = newNode;}}
```

<u>To add node at the beginning</u>

```
void append_at_begin(int number)
{  struct  Node *nodeptr,*newNode;
 newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode -> value  = number;
  newNode -> next = NULL;
  if(head==NULL)
  head = newNode;
  else
{ newNode->next=head;
  head=newNode;}}
```

<u>To add node after a particular number say N</u>

```
  void append_after_N(int N,int number)
  { struct  Node *temp,*nodeptr,*newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode -> value  = number;
  newNode -> next = NULL;
  while(nodeptr->value!=N &&nodeptr!=NULL)
  nodeptr=nodeptr->next;
   temp=nodeptr->next;
    nodeptr->next=newNode;
    newNode->next=temp;}
```

<u>Count the number of nodes</u>

```
void count()
{ struct Node *nodeptr;int i=0;
nodeptr=head;
while(nodeptr!=NULL)
{nodeptr=nodeptr->next;
i=i+1;}
printf("\n%d",i);}
```

<u>To delete node with value given</u>

```
void deleteNode(int number)
 { struct  Node *nodePtr, *previousNode;
  if (head == NULL)
  return;
  if (head -> value  == number)
   { nodePtr = head -> next;
     free(head);                                    // delete head;
    head = nodePtr;}
  else
   { nodePtr = head;
```

```
    while( nodePtr != NULL && nodePtr -> value  != number)
     { previousNode = nodePtr;
      nodePtr = nodePtr -> next;}
     previousNode -> next = nodePtr -> next;
          free(head);                                         // delete head;
      }}
```

To display the entire list

```
void displayList ()
 {    struct  Node *nodePtr;
        nodePtr = head;
        while(nodePtr != NULL)
         {printf("%d",nodePtr  -> value);
           nodePtr = nodePtr -> next;} }
```

Tutorial

1. Give functions to implement the following in a Singly Linked List:
    a. Add node before a particular element.
    b. Arrange items in ascending order.
2. Implement all the above functions using a DLL and a CLL.
3. Construct a Singly Linked List for elements whose keys are strings of characters. The left to right sequencing of elements should follow the lexicographic ordering. Perform insert, remove and search operations.

Time for an interview.....

1. Reverse the pointers of a Singly Linked List .
2. Combine two ordered lists into a Single List.
3. Form a list containing the set union and set intersection of the elements of two lists.
4. Give us Some ideas to find a <u>cycle in a Linked list</u>…?

(Asked in interviews by nearly all the top software companies viz., Google, Yahoo, Infosys etc..)

<u>Notes</u>

CHAPTER-4
STACKS AND QUEUES

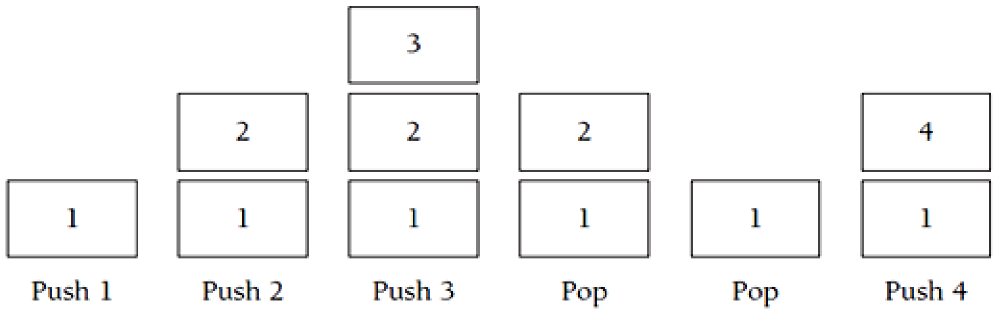
Data organize naturally as lists. We have already learned data organization as a list. Although linked lists helped us group the data in a convenient form for processing, neither structure provides a real abstraction for actually designing and implementing problem solutions. Two list-oriented data structures that provide easy-to-understand abstractions are stacks and queues.

Stacks

The stack is one of the most frequently used data structures, as we just mentioned. Data in a stack are added and removed from only one end of the list. We define a stack as a list of items that are accessible only from the end of the list, which is called the top of the stack. A stack is known as a Last-in, First-out (LIFO) data structure. Stacks are used extensively in programming language implementations, from expression evaluation to handling function calls. Stack is also used to check if a programming statement or a formula has balanced parentheses.

Stack Operations

The two primary operations of a stack are adding items to the stack and taking items off the stack. The Push operation adds an item to a stack. We take an item off the stack with a Pop operation. The other primary operation to perform on a stack is viewing the top item. The Pop operation returns the top item, but the operation also removes it from the stack. We want to just view the top item without actually removing it.



Stacks can be implemented using arrays as well as linked lists.

Implementation using Linked lists

Structure of node

```
struct stackNode
        {  int value;
          struct  stackNode *next; };
 struct stackNode *top;
```

To see whether stack is empty

```
        int  isEmpty()
{ int status;
    if(top==NULL)
```

```
   status=1;
   else
   status=0;
   return status;}
```

Push operation

```
void push(int num)
{  struct stackNode *newnode;
   newnode=(struct stackNode*)malloc(sizeof(struct stackNode));
   newnode->value=num;
   if(isEmpty())
   {  top=newnode;
      newnode->next=NULL; }
   else
   newnode->next=top;
   top=newnode; }
```

pop operation

```
void pop(int &num)
{  struct stackNode *temp;
   if(isEmpty() ==1)
   {  printf("stack is empty");
      }
   else
   { int num;
     num=top->value;
      temp=top->next;
      free(top);
      top=temp;printf("%d",num);
   }
}
```

To display entire stack

```
void displaystack()
{ stuct stackNode *i=top;
   if(i==NULL)
   {  printf("stack is empty");
      }
   while(i!=NULL)
   {  printf("\n%d",i->value);
      i=i->next; }}
```

To view top of the stack

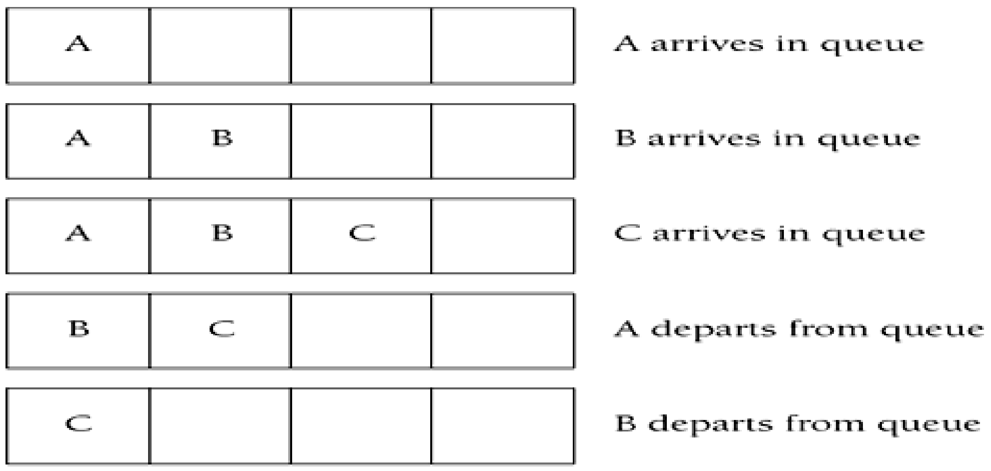
```
void viewtop()
{  if(top==NULL)
   {   printf("stack is empty");
      exit(1);}
   else
    num=top->value;}
```

Queues

A queue is a data structure where data enters at the rear of a list and is removed from the front of the list. Queues are used to store items in the order in which they occur. Queues are an example of a first-in, first-out (FIFO) data structure. Queues are used to order processes submitted to an operating system or a print spooler, and simulation applications use queues to model customers waiting in a line. Queues are also used to simulate events in the real world, the operation of elevators in buildings.

Queue Operations

The two primary operations involving queues are adding a new item to the queue and removing an item from the queue. The operation for adding a new item is called Enqueue, and the operation for removing an item from a queue is called Dequeue. The Enqueue operation adds an item at the end of the queue and the Dequeue operation removes an item from the front (or beginning) of the queue. The other primary operation to perform on a queue is viewing the beginning item. This method simply returns the item without actually removing it from the queue.



Just as Stacks, queues can also be implemented using arrays as well as linked lists.

Implementation using Linked Lists

Structure of node

```
struct queue
  {     int num;
        struct queue *next;};
        struct queue *head;
        struct queue *tail;
```

Enqueue

```
void enqueue(int i)
       {  struct queue *item,*nodeptr;
          item=(struct queue*)malloc(sizeof(struct queue));
```

```
        if(item==NULL)
        {  printf("\nallocation error");
           exit(1);}
        item->num=i;
        item->next=NULL;
        if(head==NULL)
           head=item;
        else
        {  nodeptr=head;
           while(nodeptr->next)
           { nodeptr=nodeptr->next;}
           nodeptr->next=item;}};
```

Dequeue

```
      int dequeue()
      {  struct queue *temp;
        if(head==NULL)
        {  printf("\nallocation error");
           exit(1);}
        else
        {  int i=head->num;
           temp=head;
           head=head->next;
           free(temp);
           return i; }};
```

Display the entire list

```
      void display()
      {   struct queue *nodeptr;
        nodeptr=head;
        if(head==NULL)
       printf("\nqueue is empty");
        else
       { while(nodeptr!=NULL)
        {   printf("\n%d",nodeptr->num);
           nodeptr=nodeptr->next;} };
```

To view the head of the queue

```
int viewhead()
      {   if(head==NULL)
     printf("\nqueue is empty");
       else
     return head->num;};
```

Tutorial

1. Illustrate the result of each operation in the sequence PUSH(S,4), PUSH(S,1), PUSH(S,3), POP(S), PUSH(S, 8), and POP(S) on an initially empty stack S.

2. Illustrate the result of each operation in the sequence ENQUEUE(Q, 4), ENQUEUE(Q, 1), ENQUEUE(Q, 3), DEQUEUE(Q), ENQUEUE(Q, 8), and DEQUEUE(Q) on an initially empty queue Q.
3. Write a program to convert an expression in prefix to postfix and infix.
4. A Deque (double-ended queue) allows insertion and deletion at both ends. Write four procedures to insert elements into and delete elements from both ends of a Deque.
5. Show how to implement a queue using two stacks. Also show how to implement a stack using two queues.
6. Implement a stack and a queue using arrays.
7. Write a program using stacks to check whether an expression has balanced parenthesis or not?

Time for an interview.....

1. Design a Stack. We want to push ,pop, and also " retrieve the minimum element " in constant time...Do u know how to achieve it. --- gOOglE

2. Give the approach you proceed with, in-order to achieve

3 STACKS USING A SINGLE ARRAY.

Notes

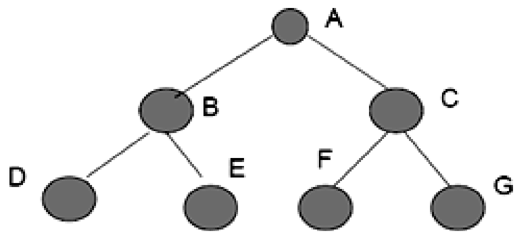
CHAPTER-5
BINARY TREE

Linked list have great advantage of flexibility over the contiguous representation of data structures, but they have one weak feature. They are sequential lists; that is, they are arranged so that it is necessary to move through them only one position at a time. In this chapter we overcome these disadvantages by studying tree as data structure, using the methods of pointer and linked lists for their implementation. Data structure organise as tree will prove valuable for a range of applications, especially for problems of information retrieval.

Definition

A tree is either empty or it consists of a node called the root together with two binary trees called the left subtree and the right subtree of the root. Each element of a binary tree is called a node of a tree.

Array Representation of a binary tree T



p sons are at 2p and 2p+1.

Representation of a binary tree T

For each node we use the following fields:

1. key
2. p (parent)
3. left pointer to left child
4. right pointer to right child

If p[x] = nil then x is the root.

If node x has no left child then left[x] = nil. root[T] root of the entire tree T

Code for implementation

Structure of node

1. struct nodetype {

2. int key;
3. struct nodetype *left;
4. struct nodetype *right;
5. struct nodetype *father; };
6. struct nodetype *Nodeptr;

Nodeptr maketree(int x)

1. { Nodeptr p;
2. p = (Nodeptr) malloc (sizeof (struct nodetype));
3. p→key = x;
4. p→left = null;
5. p→right = null;
6. return p; }

Setleft(Nodeptr p, int x)

1. { if p == null printf("void insertion \n");
2. else if (p→left != null) printf("void insertion \n");
3. else p→left = maketree(x); }

Setright(Nodeptr p, int x)

1. { if p == null printf("void insertion \n");
2. else if (p→right != null) printf("void insertion \n");
3. else p→right = maketree(x); }

Traversal of a Binary Tree

The order in which the nodes of a linear list are visited in a traversal is clearly from first to last. However, there is no such "natural" linear; order for the nodes of a tree. Thus different ordering are used for traversal in different cases. We shall define trees of these traversal methods. In each of these methods, nothing need be done to traverse an empty binary tree. The methods are all defined recursively, so that traversing a binary tree involves visiting the root and traversing its left and right subtree. The only difference among the methods is the order in which these three operations are performed.

Preorder

To traverse a non empty binary tree in preorder(also known as depth-first order), we perform the following three operations:-

1. Visit the root
2. Traverse the left subtree in preorder
3. Traverse the right subtree in preorder

pretrav(Nodeptr p)

```
1. { if ( p != null)
2. { printf(p→info);    // visit the root
3. pretrav(p→left);  // traverse the LST
4. pretrav(p→right); // traverse the RST
5. }
6. }
```

Inorder

To traverse a non empty binary tree inorder (or symmetric order)

1. Traverse the left subtree inorder
2. Visit the root
3. Traverse the right subtree in inorder

intrav(Nodeptr p)

```
1. { if ( p != null)
2. { intrav(p→left);  // traverse the LST
3. printf(p→info);    // visit the root
4. intrav(p→right); // traverse the RST
5. }
6. }
```

Postorder

To traverse a non empty binary tree in postorder

1. Traverse the left subtree in postorder
2. Traverse the right subtree in postorder
3. Visit the root

posttrav(Nodeptr p)

```
1. { if ( p != null)
2. { posttrav(p→left);  // traverse the LST
3. posttrav(p→right); // traverse the RST
4. printf(p→info);    // visit the root
5. }
6. }
```

Tutorial

1. Construct a binary tree with the given keys and write its array representation: 2,5,7,9,5,4,1,8,0,3,6.
2. Construct a binary tree for the expression: 2+(3*6*5)-(5/8)+7.

3. Write the infix, prefix and postfix expressions from the above tree. Evaluate those expressions individually.
4. Write algorithms to determine
    a. Number of nodes in a binary tree
    b. The sum of contents of all nodes in a binary tree
    c. Depth of a binary tree
    d. If a binary tree is strictly binary
5. Prove that a strictly binary tree with n leaves contains 2n-1 nodes.

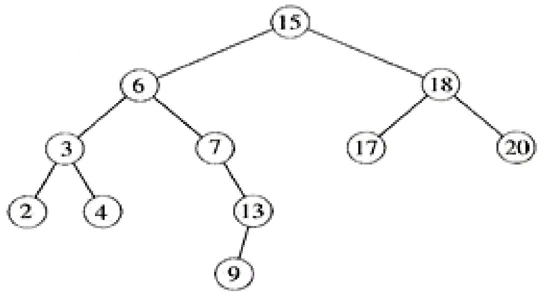
Time for an interview.....

1. **Explain a recursive approach to find depth of node in a binary tree..**
2. **Suggest us a traversal which any simple computer uses to solve an mathematical expression . (eg a+b*c) ------- Very General question**

Notes

CHAPTER-6
BINARY SEARCH TREE

A binary search tree is organized, as the name suggests, binary tree. We can represent such a tree by a linked list data structure in which each node is an object. In addition to a key and satellite data, each node contains attributes left, right, and p that points to the nodes corresponding to its left child, right child, and its parent respectively. If a child or the parent is missing, the appropriate attribute contains the value NIL. The root node is the only node in the tree whose parent is NIL. The binary search tree property allows us to print out all the keys in a binary search tree. The keys in the binary search tree are always stored in such a way as to satisfy the binary-search-tree-property: Let x be a node in a binary search tree. If y is anode in the left sub tree of x, then key[y] $\leq$ key[x]. If y is a node in the right subtree of x, then key[y] $\geq$ key[x].



The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an inorder tree walk.

INORDER-TREE-WALK

```
1. if x ≠ NIL
2. then INORDER-TREE-WALK(left[x])
3. print key[x]
4. INORDER-TREE-WALK(right[x])
```

Querying a binary search tree

Searching

Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

TREE-SEARCH (x,k)

```
1. if x= NIL or k = key[x]
2. then return x
3. if k < key[x]
4. then return TREE-SEARCH(left[x], k)
5. else return TREE-SEARCH(right[x], k)
```

The procedure begins its search at the root and traces a simple path downward in the tree as shown in the figure for each node x it encounters it compares the key k with key[x] .If the two keys are equal the search terminates .If k is smaller than key[x], the search continues in the left subtree of x, since the binary-search-tree property implies that k could not be stored in the right subtree. Symmetrically, if k is larger than key[x], the search continues in the right subtree. The nodes encountered during the recursion from a simple path downward from the root of the tree.

ITERATIVE-TREE-SEARCH(x, k)

```
1. while x ≠ NIL and k ≠ key[x]
2. do if k < key[x]
3. then x ← left[x]
4. else x ← right[x]
5. return x
```

Minimum and Maximum

We can always find an element in a binary search tree whose key is a minimum by following left child pointers from the root until we encounter a NIL. The pseudo code for TREE-MAXIMUM is symmetric.

TREE-MINIMUM(x)

```
1. while left[x] ≠ NIL
2. do x ← left[x]
3. return x
```

TREE-MAXIMUM(x)

```
1. while right[x] ≠ NIL
2. do x ← right[x]
3. return x
```

Successor and predecessor

Given a node in a binary search tree, sometimes we need to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the successor of a node x is the node with the smallest key greater than key[x]. The structure of a binary search tree allows us to determine the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree.

TREE-SUCCESSOR

```
1. if right[x] ≠ NIL
2. then return TREE-MINIMUM (right[x])
3. y ← p[x]
4. while y ≠ NIL and x = right[y]
```

```
5. do x ← y
6. y ← p[y]
7. return y
```

In the same way, predecessor can also be obtained.

Insertion and deletion

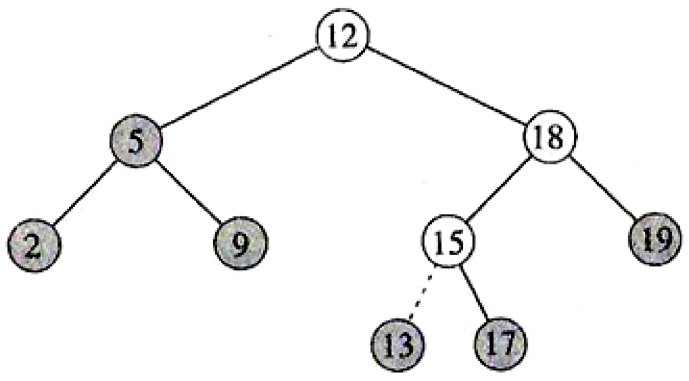
The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold.

Insertion

To insert a new value v into a binary search tree T, we use the procedure TREE-INSERT. The procedure is passed a node z for which key[z] = v, left[z] = NIL, and right[z] = NIL. It modifies T and some of the fields of z in such a way that z is inserted into an appropriate position in the tree.

TREE-INSERT(T, z)

```
1. y ← NIL
2. x ← root[T]
3. while x ≠ NIL
4. do y ← x
5. if key[z] < key[x]
6. then x ← left[x]
7. else x ← right[x]
8. p[z] ← y
9. if y = NIL
10. then root[T] ← z                ⊹ Tree T was empty
11. else if key[z] < key[y]
12. then left[y] ← z
13. else right[y] ← z
```



Deletion

The procedure considers the three cases.

- If z has no children, we modify its parent p[z] to replace z with NIL as its child.
- If the node has only a single child, we "splice out" z by making a new link between its child and its parent.
- Finally, if the node has two children, we splice out z's successor y, which has no left child and replace z's key and satellite data with y's key and satellite data.

TREE-DELETE(T, z)

```
1. if left[z] = NIL or right[z] = NIL
2. then y ← z
3. else y ← TREE-SUCCESSOR(z)
4. if left[y] ≠ NIL
5. then x ← left[y]
6. else x ← right[y]
7. if x ≠ NIL
8. then p[x] ← p[y]
9. if p[y] = NIL
10. then root[T] ← x
11. else if y = left[p[y]]
12. then left[p[y]] ← x
13. else right[p[y]] ← x
14. if y ≠ z
15. then key[z] ← key[y]
16. copy y's satellite data into z
```

Tutorial

1. For the set of keys {1, 4, 5, 10, 16, 17, 21}, draw binary search trees of height 2, 3, 4, 5, and 6.
2. What is the difference between the binary-search-tree property and the min-heap property?
3. Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which of the following sequences could not be the sequence of nodes examined?
   a. 2, 252, 401, 398, 330, 344, 397, 363.
   b. 924, 220, 911, 244, 898, 258, 362, 363.
   c. 925, 202, 911, 240, 912, 245, 363.
   d. 2, 399, 387, 219, 266, 382, 381, 278, 363.
   e. 935, 278, 347, 621, 299, 392, 358, 363.
4. Write the TREE-PREDECESSOR procedure.

5. Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.
6. Give an alternative method of performing in order tree walk.
7. Construct a binary search tree with keys: 11,7,16,5,1,9,18,13,15,6.
8. In the above tree insert keys: 8,12,20. From the above tree delete keys: 7,10,18,11.

Time for an interview.....

1. **Design a node for a tree such that**

   " It can have any number of children."

<u>Notes</u>

CHAPTER-7
HEAPS

The (binary) heap data structure is an array object that can be viewed as a binary tree. Each node of the tree corresponds to an element of the array that stores the value in the node. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes: length [A], which is the number of elements in the array, and heap-size [A], the number of elements in the heap stored within array A. That is, although A [1...length [A]] may contain valid numbers, no element past A [heap-size [A]], where heap-size [A] ≤length [A], is an element of the heap. The root of the tree is A [1], and given the index i of a node, the indices of its parent PARENT (i), left child LEFT (i), and right child RIGHT (i) can be computed simply:

PARENT (i)

```
   return ⌊i/2⌋
```

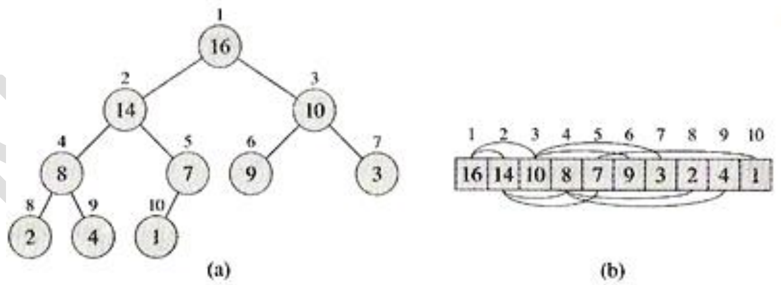
LEFT (i)

```
   return 2i
```

RIGHT (i)

```
   return 2i + 1
```

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap. In a max-heap, the max-heap property is that for every node i other than the root, A [PARENT(i)] ≥ A[i]



A min-heap is organized in the opposite way; the min-heap property is that for every node i other than the root, A [PARENT(i)] ≤ A[i]. For the heapsort algorithm, we use max-heaps.

Height of heap

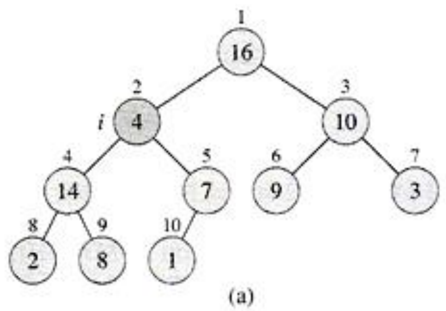
Viewing a heap as a tree, we define the height of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. An n-element heap has height of ⌊lg n⌋.

Maintaining the heap property

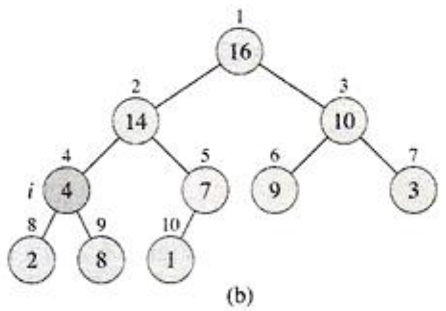
MAX-HEAPIFY is an important subroutine for manipulating max-heaps. Its inputs are an array A and an index i into the array. When MAX-HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that A [i] may be smaller than its children, thus violating the max-heap property.

MAX-HEAPIFY (A, i)

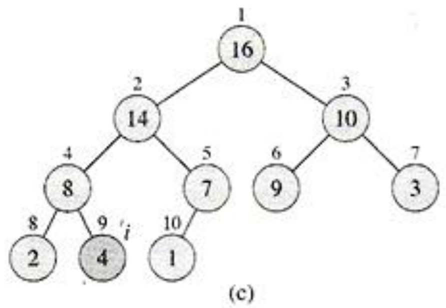
```
1. l ← LEFT(i)
2. r ← RIGHT(i)
3. if l ≤ heap-size[A] and A[l] > A[i]
4. then largest ← l
5. else largest ← i
6. if r ≤ heap-size[A] and A[r] > A[largest]
7. then largest ← r
8. if largest ≠ i
9. then exchange A[i] ↔ A[largest]
10. MAX-HEAPIFY(A, largest)
```



(a)



(b)



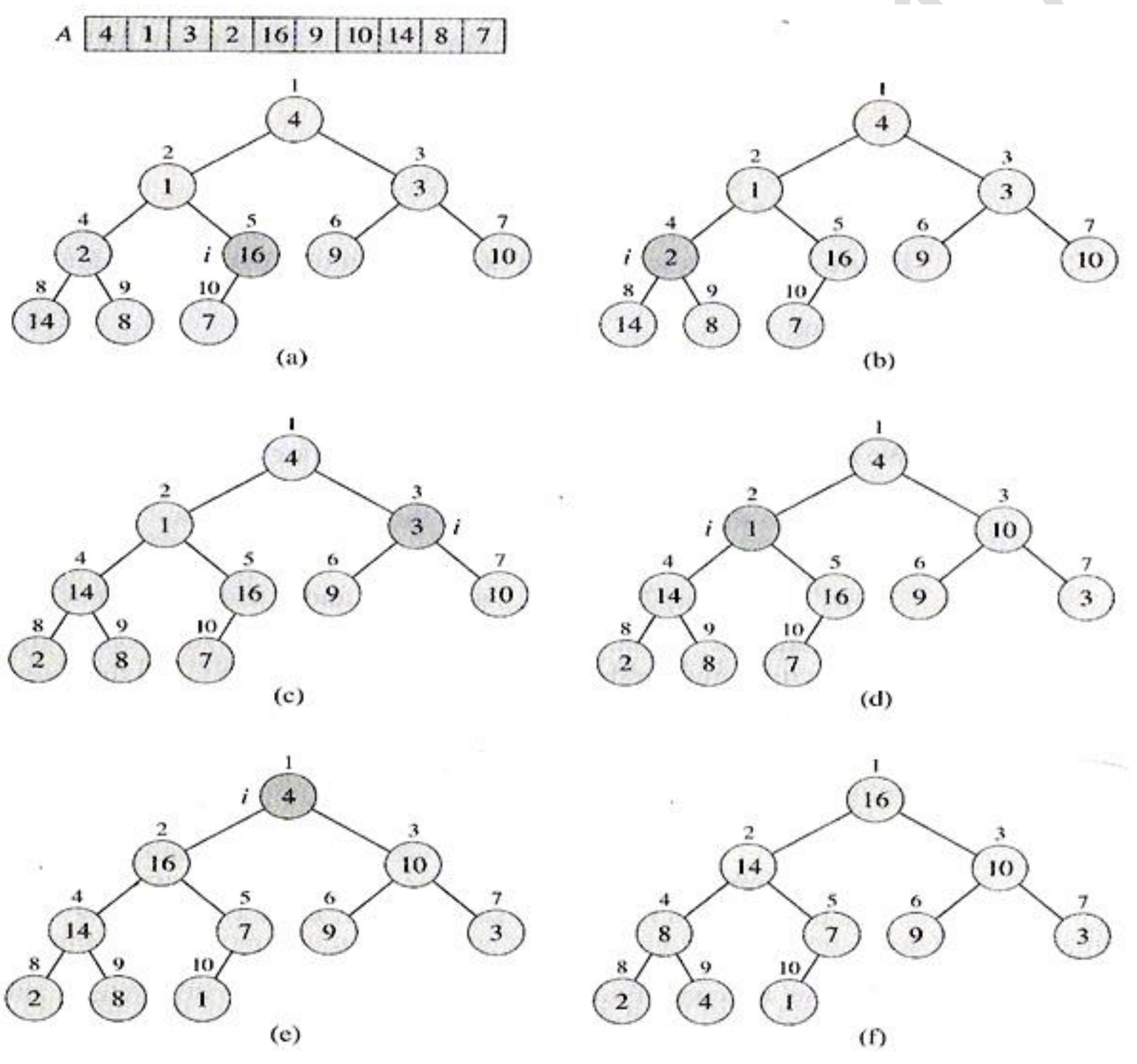
(c)

Building a heap

We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array A [1...n], where n = length[A], into a max-heap. The elements in the subarray A [($\lfloor n/2 \rfloor$+1)...n] are all leaves of the tree, and so each is a 1-element heap to begin with. The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

BUILD-MAX-HEAP(A)

```
1. heap-size[A] ← length[A]
2. for i ← ⌊length[A]/2⌋ downto 1
3. do MAX-HEAPIFY(A, i)
```



The heapsort algorithm

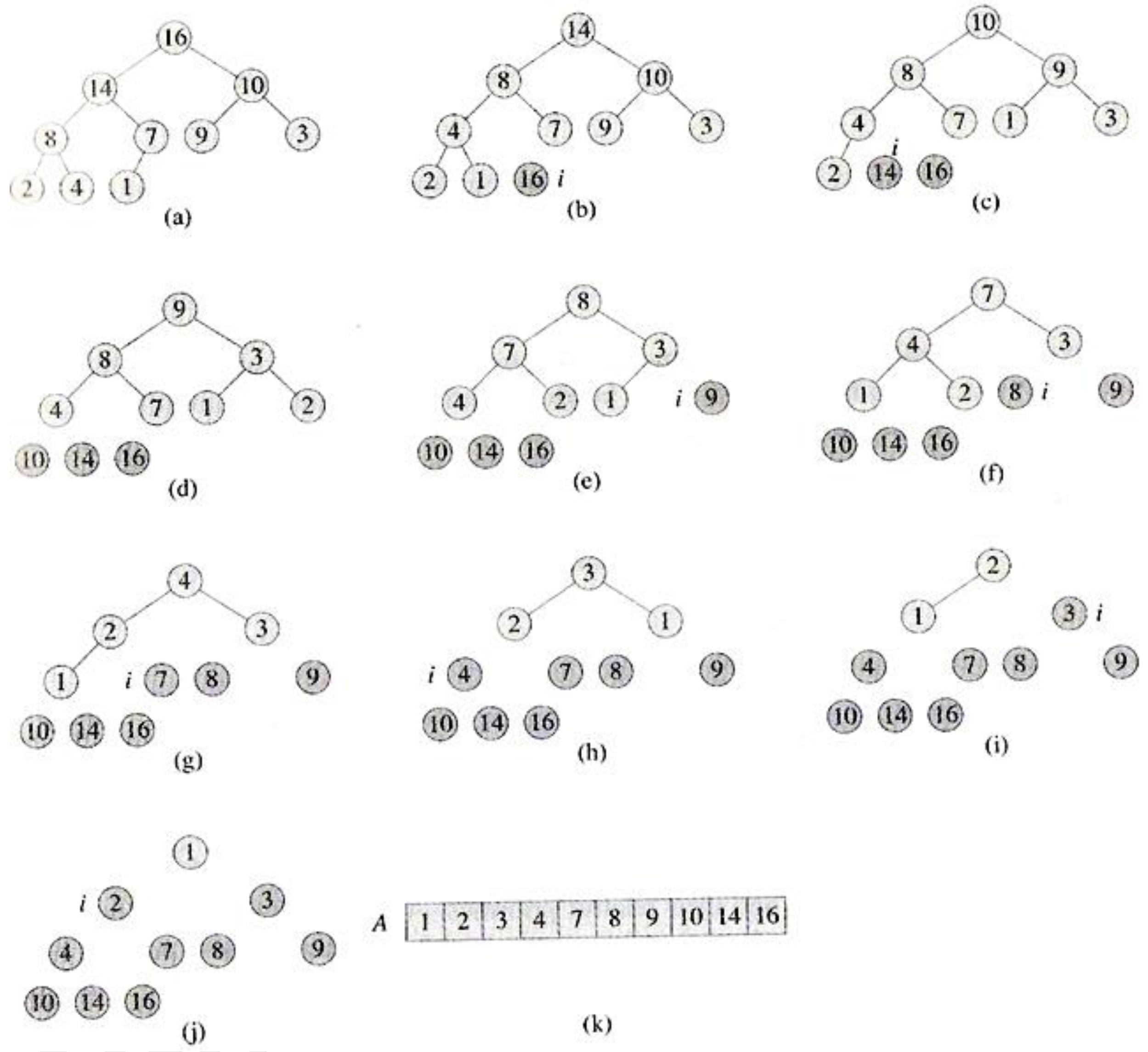
The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array A [1...n], where n = length [A]. Since the maximum element of the array is stored at the root A [1], it can be put into its correct final position by exchanging it with

A [n]. If we now "discard" node n from the heap (by decrementing heap-size [A]), we observe that A [1... (n - 1)] can easily be made into a max-heap. The children of the root remain max-heaps, but the new root element may violate the max-heap property. All that is needed to restore the max- heap property, however, is one call to MAX-HEAPIFY (A, 1), which leaves a max-heap in A [1... (n - 1)]. The heapsort algorithm then repeats this process for the max-heap of size n - 1 down to a heap of size 2.

HEAPSORT (A)

```
1. BUILD-MAX-HEAP(A)
2. for i ← length[A] downto 2
3. do exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] - 1
```

5. MAX-HEAPIFY(A, 1)



Priority queues

The heap data structure itself has enormous utility. In this section, we present one of the most popular applications of a heap: its use as an efficient priority queue. As with heaps, there are two kinds of priority queues: max- priority queues and min-priority queues. We will focus here on how to implement max- priority queues, which are in turn based on max-heaps. A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A max-priority queue supports the following operations.

Inserts the element

HEAP-INSERT (A, key)

```
1. heap-size [A]←heap-size[A]+1
2. i ←heap-size[A]
3. while i>1 and A[parent(i)]<key
4. do A[i]←A[parent(i)]
5. i ←parent(i)
6. A[i]←key
```

Returns the element with the largest key

HEAP-MAXIMUM (A)

```
1. return A[1]
```

Removes and returns the element with the largest key

In the max-heap, root key has the maximum value.

HEAP-EXTRACT-MAX (A)

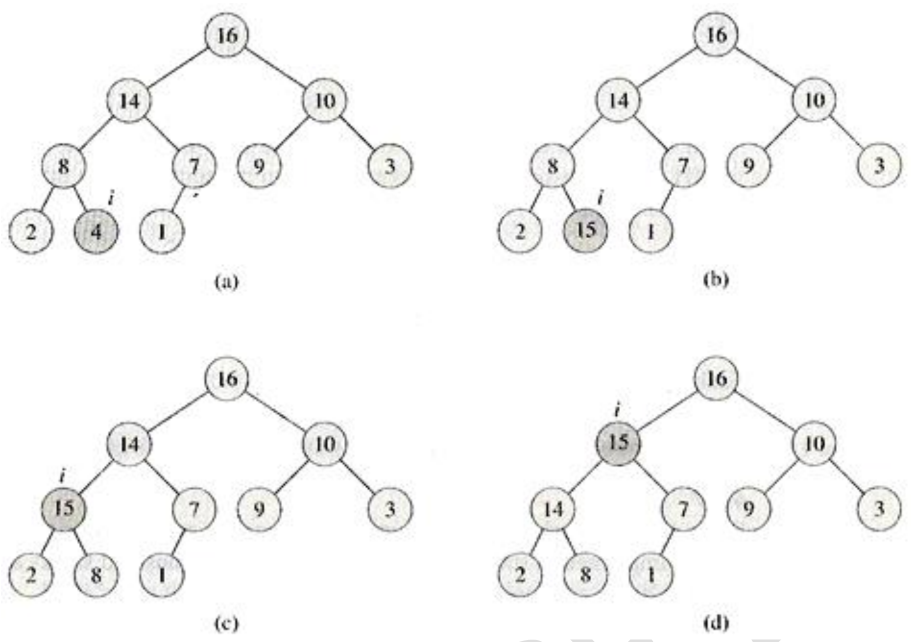
```
1. if heap-size[A]<1
2. then error "heap underflow"
3. max ← A[1]
4. A [1]←A[heap-size[A]]
5. heap-size [A]←heap-size[A]-1
6. MAX_HEAPIFY (A,1)
7. return max
```

Increases key

Increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

HEAP-INCREASE-KEY (A, i, key)

```
1. if key < A[i]
2. then error "new key is smaller than current key"
3. A[i] ← key
4. while i > 1 and A[PARENT(i)] < A[i]
5. do exchange A[i] ↔ A[PARENT(i)]
6. i ← PARENT(i)
```



16 14 10 8 7 9 3 2 i 4 1
(a)
16 14 10 8 7 9 3 2 i 15 1
(b)
16 14 10 i 15 7 9 3 2 8 1
(c)
16 i 15 10 14 7 9 3 2 8 1
(d)

Tutorial

1. What are the minimum and maximum numbers of elements in a heap of height h?
2. Show that an n-element heap has height ⌊lg n⌋.
3. Is the array with values [23,17,14,6,13,10,1,5,7,12] a max-heap?
4. Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by ⌊n/2⌋ + 1, ⌊n/2⌋ + 2 . . . n.
5. Illustrate the operation of MAX-HEAPIFY(A,3) on the array A=[27,17,3,16,13,10,1,5,7,12,4,8,9,0].
6. Write pseudo code for MIN-HEAPIFY(A,i).
7. Illustrate the operation of BUILD-MAX-HEAP on the array [5,3,17,10,84,19,6,22,9].
8. Illustrate the operation of HEAPSORT on the array A = [5, 13,2, 25, 7, 17, 20, 8, 4].
9. Illustrate the operation of HEAP-EXTRACT-MAX on the heap A = [15, 13, 9, 5, 12, 8, 7, 4,0, 6, 2,1].
10. Illustrate the operation of MAX-HEAP-INSERT(A, 10) on the heap A = [15, 13, 9, 5, 12, 8,7, 4, 0, 6, 2, 1].

11. Write pseudo code for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, and MIN-HEAP-INSERT that implement a min-priority queue with a min-heap.
12. Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue.

CHAPTER-8

GRAPHS

This chapter presents methods for representing a graph and for searching a graph. Searching a graph means systematically following the edges of the graph so as to visit the vertices of the graph. A graph-searching algorithm can discover much about the structure of a graph.

Representations of graphs

There are two standard ways to represent a graph G = (V, E): as a collection of adjacency lists or as an adjacency matrix. Either way is applicable to both directed and undirected graphs. The adjacency-list representation is usually preferred, because it provides a compact way to represent sparse graphs-those for which |E| is much less than $|V|^2$. An adjacency-matrix representation may be preferred, however, when the graph is dense-|E| is close to $|V|^2$-or when we need to be able to tell quickly if there is an edge connecting two given vertices.

Adjacency lists

The adjacency-list representation of a graph G = (V, E) consists of an array Adj of |V| lists, one for each vertex in V. For each u V, the adjacency list Adj [u] contains all the vertices v such that there is an edge (u, v) E. That is, Adj [u] consists of all the vertices adjacent to u in G. (Alternatively, it may contain pointers to these vertices.) The vertices in each adjacency list are typically stored in an arbitrary order.If G is a directed graph, the sum of the lengths of all the adjacency lists is |E|, since an edge of the form (u, v) is represented by having v appear in Adj [u]. If G is an undirected graph, the sum of the lengths of all the adjacency lists is 2 |E|, since if (u, v) is an undirected edge, then u appears in v's adjacency list and vice versa. Adjacency lists can readily be adapted to represent weighted graphs, that is, graphs for which each edge has an associated weight, typically given by a weight function w: E →R. For example, let G = (V, E) be a weighted graph with weight function w. The weight w (u, v) of the edge (u, v) E is simply stored with vertex v in u's adjacency list. The adjacency-list representation is quite robust in that it can be modified to support many other graph variants. A potential disadvantage of the adjacency-list representation is that there is no quicker way to determine if a given edge (u, v) is present in the graph than to search for v in the adjacency list Adj [u]. This disadvantage can be remedied by an adjacency-matrix representation of the graph, at the cost of using asymptotically more memory.

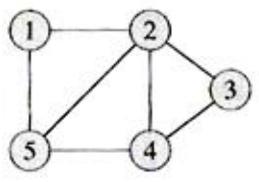
Adjacency-matrix

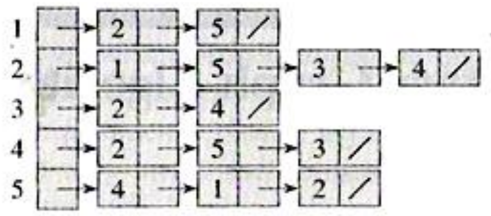
For the adjacency-matrix representation of a graph G = (V, E), we assume that the vertices are numbered 1, 2,..., |V| in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a | V| × |V| matrix A = (a_{ij}) such that

1. a_{ij}= 1, if (i,j)ϵ E

2. a_{ij}= 0, otherwise.

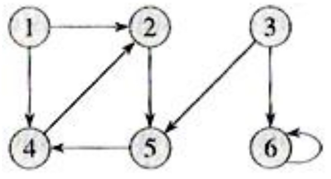
Since in an undirected graph, (u, v) and (v, u) represent the same edge, the adjacency matrix A of an undirected graph is its own transpose: A = A^T. Like the adjacency-list representation of a graph, the adjacency-matrix representation can be used for weighted graphs. For example, if G = (V, E) is a weighted graph with edge-weight function w, the weight w(u, v) of the edge (u, v) E is simply stored as the entry in row u and column v of the adjacency matrix. If an edge does not exist, a NIL value can be stored as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or ∞. Although the adjacency-list representation is asymptotically at least as efficient as the adjacency-matrix representation, the simplicity of an adjacency matrix may make it preferable when graphs are reasonably small.

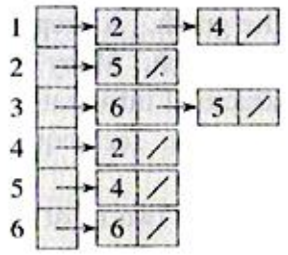

(a)


(b)

$$\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}$$

(c)

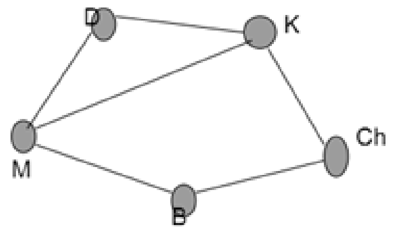

(a)


(b)

$$\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}$$

(c)

Searching a graph



Breadth-first search

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Given a graph G = (V, E) and a distinguished source vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs. Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance k + 1.The breadth-first-search procedure BFS below assumes that the input graph G = (V, E) is represented using adjacency lists. It maintains several additional data structures with each vertex in the graph. The color of each vertex u ϵ V is stored in the variable color[u], and the predecessor of u is stored in the variable π[u]. If u has no predecessor (for example, if u = s or u has not been discovered), then π[u] = NIL. The distance from the source s to vertex u computed by the algorithm is stored in d[u]. The algorithm also uses a first-in, first-out queue Q to manage the set of gray vertices.

BFS (G, s)

```
1.  for each vertex u  V [G] - {s}
2.  do { color[u] ← WHITE
3.  d[u] ← ∞
4.  π[u] ← NIL }
5.  color[s] ← GRAY
6.  d[s] ← 0
7.  π[s] ← NIL
8.  Q ← Ø
9.  ENQUEUE(Q, s)
10. while Q ≠ Ø
11. { do u ← DEQUEUE(Q)
12.   for each v  Adj[u]
13.     {do if color[v] = WHITE
14.        then color[v] ← GRAY
15.         d[v] ← d[u] + 1
16.         π[v] ← u
17.         ENQUEUE(Q, v) }
18. Deque(Q)
19. color[u] ← BLACK}
```

Depth-first search

The strategy followed by depth-first search is, as its name implies, to search "deeper" in the graph whenever possible. In depth-first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered. Besides creating a depth-first forest, depth-first search also timestamps each vertex. Each vertex v has two timestamps: the first timestamp d[v] records when v is first discovered (and grayed), and the second timestamp f[v] records when the search finishes examining v's adjacency list (and blackens v). These timestamps are used in many graph algorithms and are generally helpful in reasoning about the behaviour of depth-first search. The procedure DFS below records when it discovers vertex u in the variable d[u] and when it finishes vertex u in the variable f[u]. These timestamps are integers between 1 and 2 |V|, since there is one discovery event and one finishing event for each of the |V| vertices. For every vertex u, Vertex u is WHITE before time d[u], GRAY between time d[u] and time f[u], and BLACK thereafter. The following pseudo code is the basic depth-first-search algorithm. The input graph G may be undirected or directed. The variable time is a global variable that we use for time stamping.

DFS (G)

```
1. for each vertex u  V [G]
2. do color[u] ← WHITE
3. π[u] ← NIL
4. time ← 0
5. for each vertex u  V [G]
6. do if color[u] = WHITE
7. then DFS-VISIT(u)
```

DFS-VISIT (u)

```
1. color[u] ← GRAY            ▹White vertex u has just been discovered.
2. d[u] ←time ← time +1
3. d[u] time
4. for each v  Adj[u]         ▹Explore edge(u, v).
5. do{ if color[v] = WHITE
6.        then π[v] ← u
7.              DFS-VISIT(v)  }
8. color[u] ← BLACK           ▹ Blacken u; it is finished.
9. f [u] ← time ← time +1
```

Tutorial

1. Give the adjacency-list representation for a complete binary tree on 9 vertices. Give an equivalent adjacency matrix representation. Assume that the vertices are numbered from 1 to 7 as in a binary heap as follows:
    a. Directed downwards
    b. Directed upwards
    c. Undirected
2. The incidence matrix of a directed graph G=(V,E) with no self-loops is a |V|×|E| matrix B=(b_{ij}) such that
    a. b_{ij}= -1 if edge j leaves vertex i;
    b. b_{ij}= 1 if edge j enters vertex i;
    c. b_{ij}= 0 otherwise. Describe what the entries of the matrix product BB^T represent, where B^T is the transpose of B.

NOTES

CHAPTER- 9

INTRODUCTION TO ALGORITHMS

Introduction

The first step towards an understanding of why the study and knowledge of algorithms are so important is to define exactly what we mean by an algorithm. An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output." In other words, algorithms are like road maps for accomplishing a given, well-defined task. So, a chunk of code that calculates the terms of the Fibonacci sequence is an implementation of a particular algorithm. Even a simple function for adding two numbers is an algorithm in a sense is a simple algorithm.

Some algorithms, like those that compute the Fibonacci sequences, are intuitive and may be innately embedded into our logical thinking and problem solving skills. However, for most of us, complex algorithms are best studied so we can use them as building blocks for more efficient logical problem solving in the future. In fact, you may be surprised to learn just how many complex algorithms people use every day when they check their e-mail or listen to music on their computers. This chapter will introduce some basic ideas related to the analysis of algorithms, and then put these into practice with a few examples illustrating why it is important to know about algorithms.

Runtime analysis

One of the most important aspects of an algorithm is how fast it is. It is often easy to come up with an algorithm to solve a problem, but if the algorithm is too slow, it’s back to the drawing board. Since the exact speed of an algorithm depends on where the algorithm is run, as well as the exact details of its implementation, computer scientists typically talk about the runtime relative to the size of the input. For example, if the input consists of N integers, an algorithm might have a runtime proportional to N^2, represented as O(N^2). This means that if you were to run an implementation of the algorithm on your computer with an input of size N, it would take C*N^2 seconds, where C is some constant that doesn’t change with the size of the input.

However, the execution time of many complex algorithms can vary due to factors other than the size of the input. For example, a sorting algorithm may run much faster when given a set of integers that are already sorted than it would when given the same set of integers in a random order. As a result, you often hear people talk about the worst-case runtime, or the average-case runtime. The worst-case runtime is how long it would take for the algorithm to run if it were given the most insidious of all possible inputs. The average-case runtime is the average of how long it would take the algorithm to run if it were given all possible inputs. Of the two, the worst-case is often easier to reason about, and therefore is more frequently used as a benchmark for a given algorithm. The process of determining the worst-case and average-case runtimes for a given algorithm can be tricky, since it is usually impossible to run an algorithm on all possible inputs. There are many good online resources that can help you in estimating these values.

Approximate completion time for algorithms, N = 100

O(Log(N))	10-7 seconds
O(N)	10-6 seconds
O(N*Log(N))	10-5 seconds
O(N^2)	10-4 seconds
O(N^6)	3 minutes
O(2^N)	1014 years.
O(N!)	10142 years.

Sorting

Sorting provides a good example of an algorithm that is very frequently used by computer scientists. The simplest way to sort a group of items is to start by removing the smallest item from the group, and put it first. Then remove the next smallest, and put it next and so on. Unfortunately, this algorithm is O(N^2), meaning that the amount of time it takes is proportional to the number of items squared. If you had to sort a billion things, this algorithm would take around 10^18 operations. To put this in perspective, a desktop PC can do a little bit over 10^9 operations per second, and would take years to finish sorting a billion things this way.

Luckily, there are a number of better algorithms (quicksort, heapsort and mergesort) that have been devised over the years, many of which have a runtime of O(N * Log(N)). This brings the number of operations required to sort a billion items down to a reasonable number that even a cheap desktop could perform. Instead of a billion squared operations (10^18) these algorithms require only about 10 billion operations (10^10), a factor of 100 million faster.

Bubble Sort

One of the first sorting algorithms that is taught to students is bubble sort. While it is not fast enough in practice for all but the smallest data sets, it does serve the purpose of showing how a sorting algorithm works. Typically, it looks something like this:

```
for (int i = 0; i < data.Length; i++)
for (int j = 0; j < data.Length - 1; j++)
    if (data[j] > data[j + 1])
    {
       tmp = data[j];
       data[j] = data[j + 1];
       data[j + 1] = tmp;
    }
```

Insertion Sort

Insertion sort is an algorithm that seeks to sort a list one element at a time. With each iteration, it takes the next element waiting to be sorted, and adds it, in proper location, to those elements that have already been sorted

Merge Sort

A merge sort works recursively. First it divides a data set in half, and sorts each half separately. Next, the first elements from each of the two lists are compared. The lesser element is then removed from its list and added to the final result list.

```
int[] mergeSort (int[] data) {
  if (data.Length == 1)
     return data;
  int middle = data.Length / 2;
  int[] left = mergeSort(subArray(data, 0, middle - 1));
  int[] right = mergeSort(subArray(data, middle, data.Length - 1));
  int[] result = new int[data.Length];
  int dPtr = 0;
  int lPtr = 0;
  int rPtr = 0;
  while (dPtr < data.Length) {
    if (lPtr == left.Length) {
      result[dPtr] = right[rPtr];
      rPtr++;
    } else if (rPtr == right.Length) {
      result[dPtr] = left[lPtr];
      lPtr++;
    } else if (left[lPtr] < right[rPtr]) {
      result[dPtr] = left[lPtr];
      lPtr++;
    } else {
      result[dPtr] = right[rPtr];
      rPtr++;
    }
    dPtr++;
  }
  return result;
}
```

{18, 6, 9, 1, 4, 15, 12, 5, 6, 7, 11}

{18, 6, 9, 1, 4} {15, 12, 5, 6, 7, 11}

{18, 6} {9, 1, 4} {15, 12, 5} {6, 7, 11}

{18} {6} {9} {1, 4} {15} {12, 5} {6} {7, 11}

{18} {6} {9} {1} {4} {15} {12} {5} {6} {7} {11}

{18} {6} {9} {1, 4} {15} {5, 12} {6} {7, 11}

{6, 18} {1, 4, 9} {5, 12, 15} {6, 7, 11}

{1, 4, 6, 9, 18} {5, 6, 7, 11, 12, 15}

{1, 4, 5, 6, 6, 7, 9, 11, 12, 15, 18}

Apart from being fairly efficient, a merge sort has the advantage that it can be used to solve other problems, such as determining how "unsorted" a given list is

Some other sorting algorithms

Some other sorting algorithms include:

1. Heap Sort
2. Quick Sort
3. Counting Sort
4. Radix Sort

Sorting Libraries

Nowadays, most programming platforms include runtime libraries that provide a number of useful and reusable functions for us. The .NET framework, Java API, and C++ STL all provide some built-in sorting capabilities. Even better, the basic premise behind how they work is similar from one language to the next.

For standard data types such as scalars, floats, and strings, everything needed to sort an array is already present in the standard libraries. But what if we have custom data types that require more complicated comparison logic? Fortunately, object-oriented programming provides the ability for the standard libraries to solve this as well.

In both Java and C# (and VB for that matter), there is an interface called Comparable (IComparable in .NET). By implementing the IComparable interface on a user-defined class, you add a method int CompareTo (object other), which returns a negative value if less than, 0 if equal to, or a positive value if greater than the parameter. The library sort functions will then work on arrays of your new data type.

Additionally, there is another interface called Comparator (IComparer in .NET), which defines a single methodint Compare (object obj1, object obj2), which returns a value indicating the results of comparing the two parameters.
The greatest joy of using the sorting functions provided by the libraries is that it saves a lot of coding time, and requires a lot less thought and effort. However, even with the heavy lifting already completed, it is still nice to know how things work under the hood.

Introduction to Dynamic Programming

A **DP** is an algorithmic technique which is usually based on a recurrent formula and one (or some) starting states. A sub-solution of the problem is constructed from previously found ones. DP solutions have a polynomial complexity which assures a much faster running time than other techniques like backtracking, brute-force etc.
Now let's see the base of DP with the help of an example:

Given a list of N coins, their values (**V_1**, **V_2**, … , **V_N**), and the total sum **S**. Find the minimum number of coins the sum of which is **S** (we can use as many coins of one type as we want), or report that it's not possible to select coins in such a way that they sum up to **S**.
Now let's start constructing a DP solution:

First of all we need to find a state for which an optimal solution is found and with the help of which we can find the optimal solution for the next state.

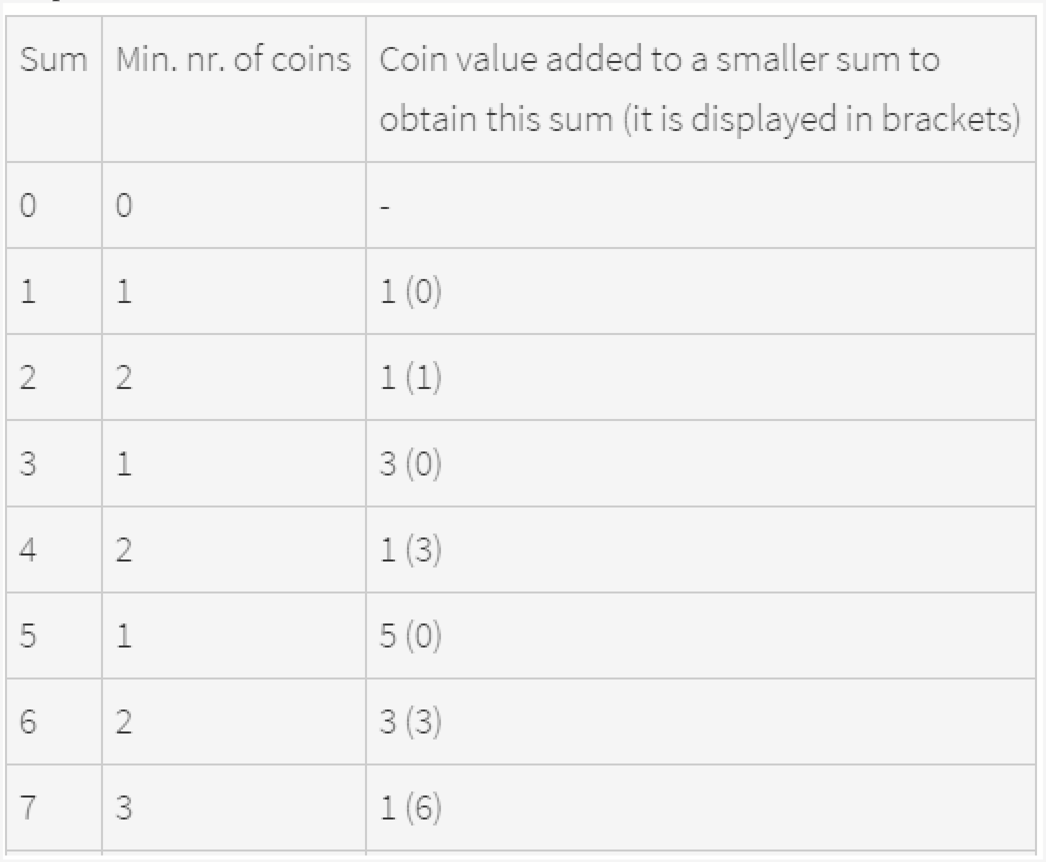
What does a "state" stand for?
It's a way to describe a situation, a sub-solution for the problem. For example a state would be the solution for sum **i**, where **$i \leq S$**. A smaller state than state **i** would be the solution for any sum **j**, where **$j<i$**. For finding a **state i**, we need to first find all smaller states **j ($j<i$)** . Having found the minimum number of coins which sum up to **i**, we can easily find the next state – the solution for **$i+1$**.
How can we find it?
It is simple – for each coin **j, $V_j \leq i$**, look at the minimum number of coins found for the **$i-V_j$**sum (we have already found it previously). Let this number be **m**. If **$m+1$** is less than the minimum number of coins already found for current sum **i**, then we write the new result for it.
For a better understanding let's take this example:
Given coins with values 1, 3, and 5.
And the sum **S** is set to be 11.

First of all we mark that for state 0 (sum 0) we have found a solution with a minimum number of 0 coins. We then go to sum 1. First, we mark that we haven't yet found a solution for this one (a value of Infinity would be fine). Then we see that only coin 1 is less than or equal to the current sum. Analyzing it, we see that for sum 1-$\mathbf{V_1}$= 0 we have a solution with 0 coins. Because we add one coin to this solution, we'll have a solution with 1 coin for sum 1. It's the only solution yet found for this sum. We write (save) it. Then we proceed to the next state – **sum 2**. We again see that the only coin which is less or equal to this sum is the first coin, having a value of 1. The optimal solution found for sum (2-1) = 1 is coin 1. This coin 1 plus the first coin will sum up to 2, and thus make a sum of 2 with the help of only 2 coins. This is the best and only solution for sum 2. Now we proceed to sum 3. We now have 2 coins which are to be analyzed – first and second one, having values of 1 and 3. Let's see the first one. There exists a solution for sum 2 (3 – 1) and therefore we can construct from it a solution for sum 3 by adding the first coin to it. Because the best solution for sum 2 that we found has 2 coins, the new solution for sum 3 will have 3 coins. Now let's take the second coin with value equal to 3. The sum for which this coin needs to be added to make 3, is 0. We know that sum 0 is made up of 0 coins. Thus we can make a sum of 3 with only one coin – 3. We see that it's better than the previous found solution for sum 3, which was composed of 3 coins. We update it and mark it as having only 1 coin. The same we do for sum 4, and get a solution of 2 coins – 1+3. And so on.

```
Set Min[i] equal to Infinity for all of i

Min[0]=0;

For i = 1 to S

For j = 0 to N - 1
If ($V_j$<=i AND Min[i-$V_j$]+1<Min[i])
```

Then Min[i]=Min[i-V_j]+1 Additionally, by tracking data about how we got to a certain sum from a previous one, we can find what coins were used in building it. For example: to sum 11 we got by adding the coin with value 1 to a sum of 10. To sum 10 we got from 5. To 5 – from 0. This way we find the coins used: 1, 5 and 5.

Output Min[S]



Sum	Min. nr. of coins	Coin value added to a smaller sum to obtain this sum (it is displayed in brackets)
0	0	-
1	1	1 (0)
2	2	1 (1)
3	1	3 (0)
4	2	1 (3)
5	1	5 (0)
6	2	3 (3)
7	3	1 (6)

Tutorial

1. Solve turbo sort question on codechef by using various sorting algorithms.
Link- http://www.codechef.com/problems/TSORT
2. You have given an array of integers. (i)Find the length of longest increasing subsequence in given array. (ii) Print it.
3. You have given two strings. (i) Find length of LCS in those strings.(ii) Print it.
4. You have given array. Find the maximum possible sum of a (i) noncontiguous subarray. (ii) Contiguous subarray.
5. Study following algorithms :
   (i) 0-1 knapsack problem
   (ii) Subset sum problem
   (iii) Edit distance
   (iv) Rod cutting

CHAPTER-10
STL CONTAINERS

The STL contains many different container classes that can be used in different situations. Generally speaking, the container classes fall into three basic categories: Sequence containers, Associative containers, and Container adapters. We'll just do a quick overview of the containers here.

Sequence Containers

Sequence contains are container classes that maintain the ordering of elements in the container. A defining characteristic of sequence containers is that you can choose where to insert your element by position. The most common example of a sequence container is the array: if you insert four elements into an array, the elements will be in the exact order you inserted them.

The STL contains 3 sequence containers: vector, deque, and list.

The following program inserts 6 numbers into a vector and uses the overloaded [] operator to access them in order to print them.

```
int main()
{    vector<int> vect;
   for (int nCount=0; nCount < 6; nCount++)
       vect.push_back(10 - nCount); // insert at end of array
   for (int nIndex=0; nIndex < vect.size(); nIndex++)
       cout << vect[nIndex] << " ";
   cout << endl;
}
```

This program produces the result:

```
10 9 8 7 6 5
```

The deque class (pronounced "deck") is a double-ended queue class, implemented as a dynamic array that can grow from both ends.

```
int main()
{
      deque<int> deq;
    for (int nCount=0; nCount < 3; nCount++)
    {
        deq.push_back(nCount); // insert at end of array
        deq.push_front(10 - nCount); // insert at front of array
    }
    for (int nIndex=0; nIndex < deq.size(); nIndex++)
        cout << deq[nIndex]  << " ";
    cout << endl;
}
```

This program produces the result:

```
8 9 10 0 1 2
```

A list is a special type of sequence container called a doubly linked list where each element in the container contains pointers that point at the next and previous elements in the list. Lists only provide access to the start and end of the list -- there is no random access provided. If you want to find a value in the middle, you have to start at one end and “walk the list” until you reach the element you want to find. The advantage of lists is that inserting elements into a list is very fast if you already know where you want to insert them. Generally iterators are used to walk through the list.

Associative Containers

Associative contains are containers that automatically sort their inputs when those inputs are inserted into the container. By default, associative containers compare elements using operator<.

A set is a container that stores unique elements, with duplicate elements disallowed. The elements are sorted according to their values.

A multiset is a set where duplicate elements are allowed.

A map (also called an associative array) is a set where each element is a pair, called a key/value pair. The key is used for sorting and indexing the data, and must be unique. The value is the actual data.

A multimap (also called a dictionary) is a map that allows duplicate keys. Real-life dictionaries are multimaps: the key is the word, and the value is the meaning of the word. All the keys are sorted in ascending order, and you can look up the value by key. Some words can have multiple meanings, which is why the dictionary is a multimap rather than a map.

Container Adapters

Container adapters are special predefined containers that are adapted to specific uses. The interesting part about container adapters is that you can choose which sequence container you want them to use.

A stack is a container where elements operate in a LIFO (Last In, First Out) context, where elements are inserted (pushed) and removed (popped) from the end of the container. Stacks default to using deque as their default sequence container (which seems odd, since vector seems like a more natural fit), but can use vector or list as well.

A queue is a container where elements operate in a FIFO (First In, First Out) context, where elements are inserted (pushed) to the back of the container and removed (popped) from the front. Queues default to using deque, but can also use list.

A priority queue is a type of queue where the elements are kept sorted (via operator<). When elements are pushed, the element is sorted in the queue. Removing an element from the front returns the highest priority item in the priority queue.

STL iterators overview

An Iterator is an object that can traverse (iterate over) a container class without the user having to know how the container is implemented. With many classes (particularly lists and the associative classes), iterators are the primary way elements of these classes are accessed.

An iterator is best visualized as a pointer to a given element in the container, with a set of overloaded operators to provide a set of well-defined functions:

Operator (*) :Dereferencing the iterator returns the element that the iterator is currently pointing at.

Operator (++) :Moves the iterator to the next element in the container. Most iterators also provide Operator (--):To move to the previous element.

Operator (==) and Operator (!=) : Basic comparison operators to determine if two iterators point to the same element. To compare the values that two iterators are pointing at, deference the iterators first, and then use a comparison operator.

Operator (=):Assign the iterator to a new position (typically the start or end of the container's elements). To assign the value of the element the iterator is point at, deference the iterator first, then use the assign operator.

Each container includes four basic member functions for use with Operator(=):

begin() returns an iterator representing the beginning of the elements in the container.

end() returns an iterator representing the element just past the end of the elements.

cbegin() returns a const (read-only) iterator representing the beginning of the elements in the container.

cend() returns a const (read-only) iterator representing the element just past the end of the elements.

It might seem weird that end() doesn't point to the last element in the list, but this is done primarily to make looping easy: iterating over the elements can continue until the iterator reaches end(), and then you know you're done.

Finally, all containers provide (at least) two types of iterators:

container:: iterator provides a read/write iterator

container:: const_iterator provides a read-only iterator

Lets take a look at some examples of using iterators.

Iterating through a vector

```
int main()

{

   vector<int> vect;

   for (int nCount=0; nCount < 6; nCount++)

      vect.push_back(nCount);

   vector<int>::const_iterator it; // declare an read-only iterator

   it = vect.begin(); // assign it to the start of the vector
```

```
    while (it != vect.end()) // while it hasn't reach the end
        {
        cout << *it << " "; // print the value of the element it points to
        it++; // and iterate to the next element
        }
     cout << endl;
}
```

This prints the following:

```
0 1 2 3 4 5
```

Iterating through a list

Now let's do the same thing with a list:

```
int main()
{
    list<int> li;
    for (int nCount=0; nCount < 6; nCount++)
        li.push_back(nCount);
    list<int>::const_iterator it; // declare an iterator
    it = li.begin(); // assign it to the start of the list
    while (it != li.end()) // while it hasn't reach the end
    {
        cout << *it << " "; // print the value of the element it points to
        it++; // and iterate to the next element
    }
    cout << endl;
}
```

This prints:

```
0 1 2 3 4 5
```

Note the code is almost identical to the vector case, even though vectors and lists have almost completely different internal implementations!

Iterating through a set

In the following example, we're going to create a set from 6 numbers and use an iterator to print the values in the set:

```
int main()
{
    set<int> myset;
    myset.insert(7);
    myset.insert(2);
    myset.insert(-6);
    myset.insert(8);
    myset.insert(1);
    myset.insert(-4);
    set<int>::const_iterator it; // declare an iterator
    it = myset.begin(); // assign it to the start of the set
    while (it != myset.end()) // while it hasn't reach the end
    {
        cout << *it << " "; // print the value of the element it points to
        it++; // and iterate to the next element
    }
    cout << endl;
}
```

This program produces the following result:

```
-6 -4 1 2 7 8
```

Note that although populating the set differs from the way we populate the vector and list, the code used to iterate through the elements of the set was essentially identical.

Iterating through a map

This one is a little trickier. Maps and multimaps take pairs of elements (defined as an stl::pair). We use the make_pair() helper function to easily create pairs. std::pair allows access to the elements of the pair via the first() and second() member functions. In our map, we use first() as the key, and second() as the value.

```
int main()
{    map<int, string> mymap;
    mymap.insert(make_pair(4, "apple"));
    mymap.insert(make_pair(2, "orange"));
    mymap.insert(make_pair(1, "banana"));
    mymap.insert(make_pair(3, "grapes"));
    mymap.insert(make_pair(6, "mango"));
    mymap.insert(make_pair(5, "peach"));
    map<int, string>::const_iterator it; // declare an iterator
    it = mymap.begin(); // assign it to the start of the vector
    while (it != mymap.end()) // while it hasn't reach the end
    {
        cout << it->first << "=" << it->second << " "; // print the value of the element it points to
        it++; // and iterate to the next element
    }    cout << endl;
}
```

This program produces the result:

```
1=banana 2=orange 3=grapes 4=apple 5=peach 6=mango
```

Notice here how easy iterators make it to step through each of the elements of the container. You don't have to care at all how map stores its data!

Conclusion

Iterators provide an easy way to step through the elements of a container class without having to understand how the container class is implemented. When combined with STL's algorithms and the member functions of the container classes, iterators become even more powerful.

One point worth noting: Iterators must be implemented on a per-class basis, because the iterator does need to know how a class is implemented. Thus iterators are always tied to specific container classes.

STL algorithms overview

In addition to container classes and iterators, STL also provides a number of generic algorithms for working with the elements of the container classes. These allow you to do things like search, sort, insert, reorder, remove, and copy elements of the container class.

Note that algorithms are implemented as global functions that operate using iterators. This means that each algorithm only needs to be implemented once, and it will generally automatically work for all containers that provides a set of iterators (including your custom container classes). While this is very powerful and can lead to the ability to write complex code very quickly, it's also got a dark side: some combination of algorithms and container types may not work, may cause infinite loops, or may work but be extremely poor performing. So use these at your risk.

STL provides quite a few algorithms .We will only touch on some of the more common and easy to use ones here.

To use any of the STL algorithms, simply include the algorithm header file.

```
#include <algorithm>
```

min_element and max_element

The min_element and max_element algorithms find the min and max element in a container class:

```
int main()

{   list<int> li;

   for (int nCount=0; nCount < 6; nCount++)

      li.push_back(nCount);

   list<int>::const_iterator it; // declare an iterator

   it = min_element(li.begin(), li.end());

      cout << *it << " ";

   it = max_element(li.begin(), li.end());

      cout << *it << " ";

   cout << endl;

}
```

Prints:

```
0 5
```

find (and list::insert)

In this example, we'll use the find() algorithm to find a value in the list class, and then use the list::insert() function to add a new value into the list at that point.

```
int main()

{

   list<int> li;

   for (int nCount=0; nCount < 6; nCount++)

      li.push_back(nCount);

   list<int>::const_iterator it; // declare an iterator
```

```
    it = find(li.begin(), li.end(), 3); // find the value  3 in the list
    li.insert(it, 8); // use list::insert to insert the value 8 before it
    for (it = li.begin(); it != li.end(); it++) // for loop with iterators
        cout << *it << " ";
    cout << endl;
}
```

This prints the value:

```
0 1 2 8 3 4 5
```

sort and reverse

In this example, we'll sort a vector and then reverse it.

```
int main()
{
    vector<int> vect;
    vect.push_back(7);
    vect.push_back(-3);
    vect.push_back(6);
    vect.push_back(2);
    vect.push_back(-5);
    vect.push_back(0);
    vect.push_back(4);
    sort(vect.begin(), vect.end()); // sort the list
    vector<int>::const_iterator it; // declare an iterator
    for (it = vect.begin(); it != vect.end(); it++) // for loop with iterators
```

```
        cout << *it << " ";
    cout << endl;
    reverse(vect.begin(), vect.end()); // reverse the list
    for (it = vect.begin(); it != vect.end(); it++) // for loop with iterators
        cout << *it << " ";
    cout << endl;
}
```

This produces the result:

```
-5 -3 0 2 4 6 7
7 6 4 2 0 -3 -5
```

Note that sort() doesn't work on list container classes -- the list class provides it's own sort() member function, which is much more efficient than the generic version would be.

Conclusion

Although this is just a taste of the algorithms that STL provides, it should suffice to show how easy these are to use in conjunction with iterators and the basic container classes. There are enough other algorithms.

Tutorial

1. You have given an array of integers. Find the number of different integers in the array.
2. You have given an expression in infix form. Transform given expression in postfix form.
   e.g. input- ((a+t)*((b+(a+c))^(c+d)))
   output - at+bac++cd+^*
3. Solve following problem on codechef:
   (i) http://www.codechef.com/problems/CSS2
   (ii) http://www.codechef.com/problems/CHEFBM
4. You have given a string. Reverse it using stack.
5. Pangrams are sentences constructed by using every letter of the alphabet at least once. You have given a string. Check whether it is a pangram or not.
6. You have given a string. Sort it and print it: (i) without using stl sort function.
   (ii) with using sort function.

Some useful sites:

(i) www.codechef.com
(ii) www.codeforces.com
(iii) www.hackerrank.com
(iv) www.geeksforgeeks.org

Some useful pages:

(i) http://discuss.codechef.com/questions/48877/data-structures-and-algorithms (for DS and algorithms).
(ii) http://e-maxx.ru/algo/