DATA STRUCTURES

CS201

LECTURE NOTES

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

NATIONAL INSTITUTE OF TECHNOLOGY MANIPUR

Chapter 1

Basic Concepts

The term data structure is used to describe the way data is stored, and the term algorithm is used to describe the way data is processed. Data structures and algorithms are interrelated. Choosing a data structure affects the kind of algorithm you might use, and choosing an algorithm affects the data structures we use.

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

1.1. Introduction to Data Structures:

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

Algorithm + Data structure = Program

A data structure is said to be *linear* if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be *non linear* if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Data structures are divided into two types:

- Primitive data structures.
- Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category. Figure 1.1 shows the classification of data structures.

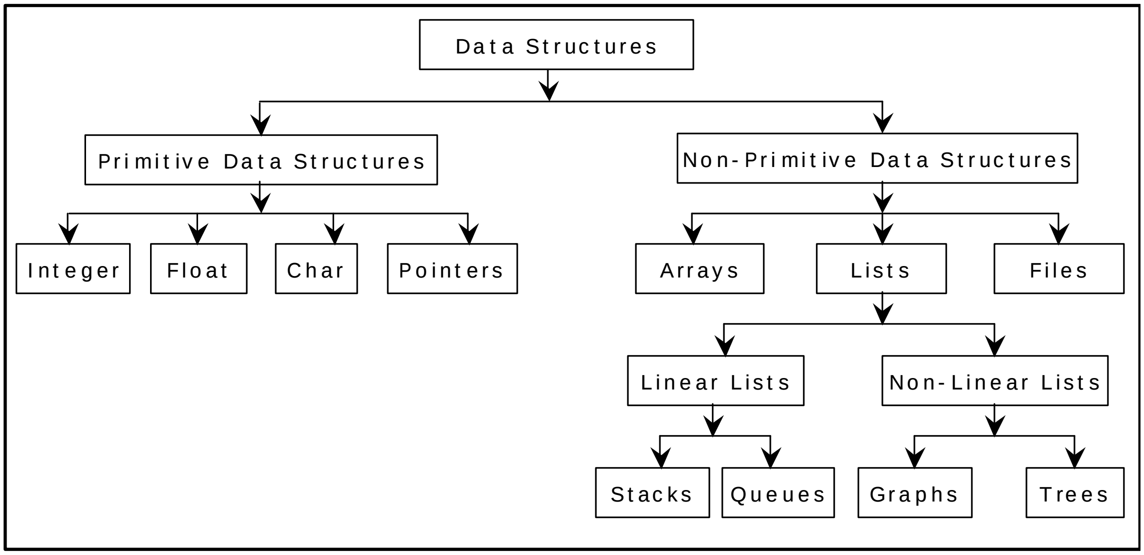


Figure 1.1. Classification of Data Structures

1.2. Data structures: Organization of data

The collection of data you work with in a program have some kind of structure or organization. No matte how complex your data structures are they can be broken down into two fundamental types:

- Contiguous
- Non-Contiguous.

In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An array is an example of a contiguous structure. Since each element in the array is located next to one or two other elements. In contrast, items in a non-contiguous structure and scattered in memory, but we linked to each other in some way. A linked list is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node. Figure 1.2 below illustrates the difference between contiguous and non-contiguous structures.

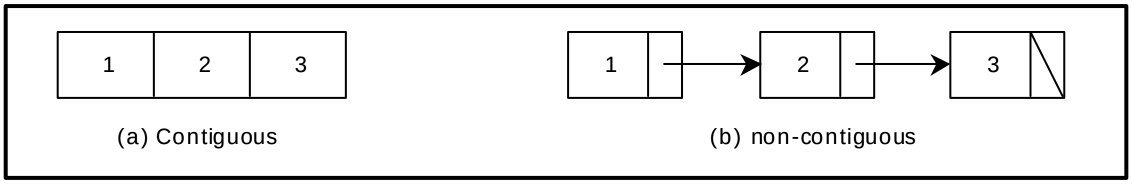


Figure 1.2 Contiguous and Non-contiguous structures compared

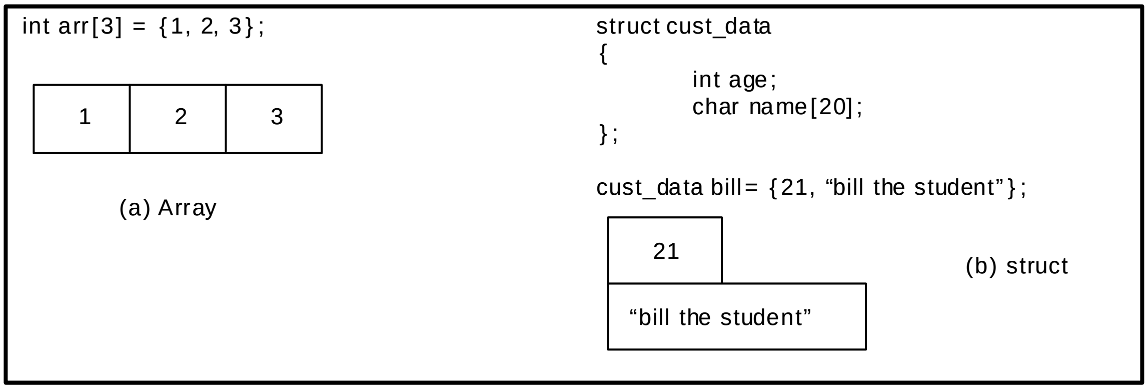
Contiguous structures:

Contiguous structures can be broken drawn further into two kinds: those that contain data items of all the same size, and those where the size may differ. Figure 1.2 shows example of each kind. The first kind is called the array. Figure 1.3(a) shows an example of an array of numbers. In an array, each element is of the same type, and thus has the same size.

The second kind of contiguous structure is called structure, figure 1.3(b) shows a simple structure consisting of a person's name and age. In a struct, elements may be of different data types and thus may have different sizes.

For example, a person's age can be represented with a simple integer that occupies two bytes of memory. But his or her name, represented as a string of characters, may require many bytes and may even be of varying length.

Couples with the atomic types (that is, the single data-item built-in types such as integer, float and pointers), arrays and structs provide all the "mortar" you need to built more exotic form of data structure, including the non-contiguous forms.



```
int arr[3] = {1, 2, 3};
```

```
struct cust_data
{
        int age;
        char name[20];
};

cust_data bill= {21, "bill the student"};
```

Figure 1.3 Examples of contiguous structures.

Non-contiguous structures:

Non-contiguous structures are implemented as a collection of data-items, called nodes, where each node can point to one or more other nodes in the collection. The simplest kind of non-contiguous structure is linked list.

A linked list represents a linear, one-dimension type of non-contiguous structure, where there is only the notation of backwards and forwards. A tree such as shown in figure 1.4(b) is an example of a two-dimensional non-contiguous structure. Here, there is the notion of up and down and left and right.

In a tree each node has only one link that leads into the node and links can only go down the tree. The most general type of non-contiguous structure, called a graph has no such restrictions. Figure 1.4(c) is an example of a graph.

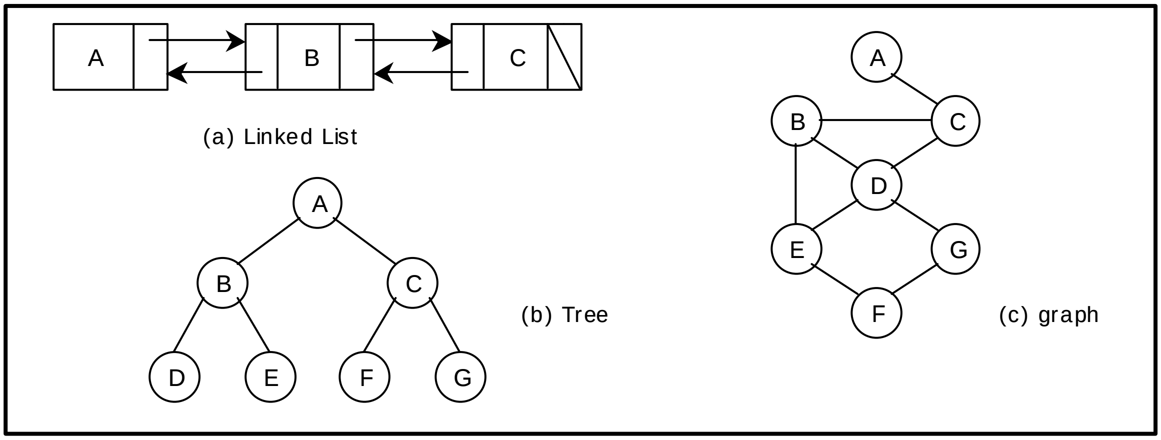
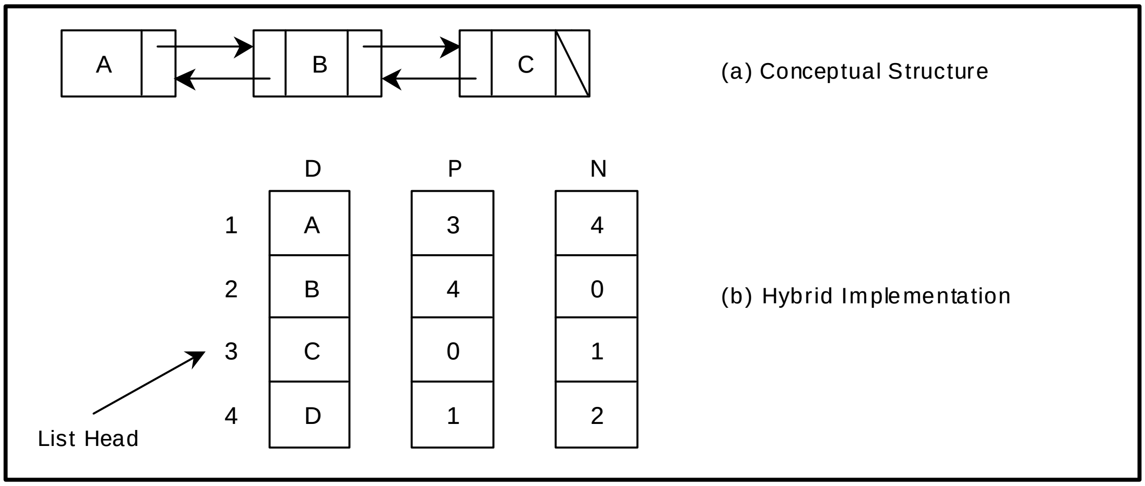


Figure 1.4. Examples of non-contiguous structures

Hybrid structures:

If two basic types of structures are mixed then it is a hybrid form. Then one part contiguous and another part non-contiguous. For example, figure 1.5 shows how to implement a double–linked list using three parallel arrays, possibly stored a past from each other in memory.



	D	P	N
1	A	3	4
2	B	4	0
3	C	0	1
4	D	1	2

Figure 1.5. A double linked list via a hybrid data structure

The array D contains the data for the list, whereas the array P and N hold the previous and next "pointers''. The pointers are actually nothing more than indexes into the D array. For instance, D[i] holds the data for node i and p[i] holds the index to the node previous to i, where may or may not reside at position i–1. Like wise, N[i] holds the index to the next node in the list.

1.3. Abstract Data Type (ADT):

The design of a data structure involves more than just its organization. You also need to plan for the way the data will be accessed and processed – that is, how the data will be interpreted actually, non-contiguous structures – including lists, tree and graphs – can be implemented either contiguously or non- contiguously like wise, the structures that are normally treated as contiguously - arrays and structures – can also be implemented non-contiguously.

The notion of a data structure in the abstract needs to be treated differently from what ever is used to implement the structure. The abstract notion of a data structure is defined in terms of the operations we plan to perform on the data.

Considering both the organization of data and the expected operations on the data, leads to the notion of an abstract data type. An *abstract data type* in a theoretical construct that consists of data as well as the operations to be performed on the data while hiding implementation.

For example, a stack is a typical abstract data type. Items stored in a stack can only be added and removed in certain order – the last item added is the first item removed. We call these operations, pushing and popping. In this definition, we haven't specified have items are stored on the stack, or how the items are pushed and popped. We have only specified the valid operations that can be performed.

For example, if we want to read a file, we wrote the code to read the physical file device. That is, we may have to write the same code over and over again. So we created what is known today as an ADT. We wrote the code to read a file and placed it in a library for a programmer to use.

As another example, the code to read from a keyboard is an ADT. It has a data structure, character and set of operations that can be used to read that data structure.

To be made useful, an abstract data type (such as stack) has to be implemented and this is where data structure comes into ply. For instance, we might choose the simple data structure of an array to represent the stack, and then define the appropriate indexing operations to perform pushing and popping.

1.4. Selecting a data structure to match the operation:

The most important process in designing a problem involves choosing which data structure to use. The choice depends greatly on the type of operations you wish to perform.

Suppose we have an application that uses a sequence of objects, where one of the main operations is delete an object from the middle of the sequence. The code for this is as follows:

```
void delete (int *seg, int &n, int posn)
// delete the item at position from an array of n elements.
{
        if (n)
        {
                 int i=posn;
                 n--;
                while (i < n)
                {
                        seq[i] = seg[i+1];
                        i++;
                }
        }
        return;
}
```

This function shifts towards the front all elements that follow the element at position *posn*. This shifting involves data movement that, for integer elements, which is too costly. However, suppose the array stores larger objects, and lots of them. In this case, the overhead for moving data becomes high. The problem is that, in a contiguous structure, such as an array the logical ordering (the ordering that we wish to interpret our elements to have) is the same as the physical ordering (the ordering that the elements actually have in memory).

If we choose non-contiguous representation, however we can separate the logical ordering from the physical ordering and thus change one without affecting the other. For example, if we store our collection of elements using a double–linked list (with previous and next pointers), we can do the deletion without moving the elements, instead, we just modify the pointers in each node. The code using double linked list is as follows:

```
void delete (node * beg, int posn)
//delete the item at posn from a list of elements.
{
        int i = posn;
        node *q = beg;
        while (i && q)
        {
                i--;
                q = q → next;
        }

        if (q)
        {               /* not at end of list, so detach P by making previous and
                           next nodes point to each other */
                node *p = q -> prev;
                node *n = q -> next;
                if (p)
                        p -> next = n;
                if (n)
                        n -> prev = P;
        }
        return;
}
```

The process of detecting a node from a list is independent of the type of data stored in the node, and can be accomplished with some pointer manipulation as illustrated in figure below:

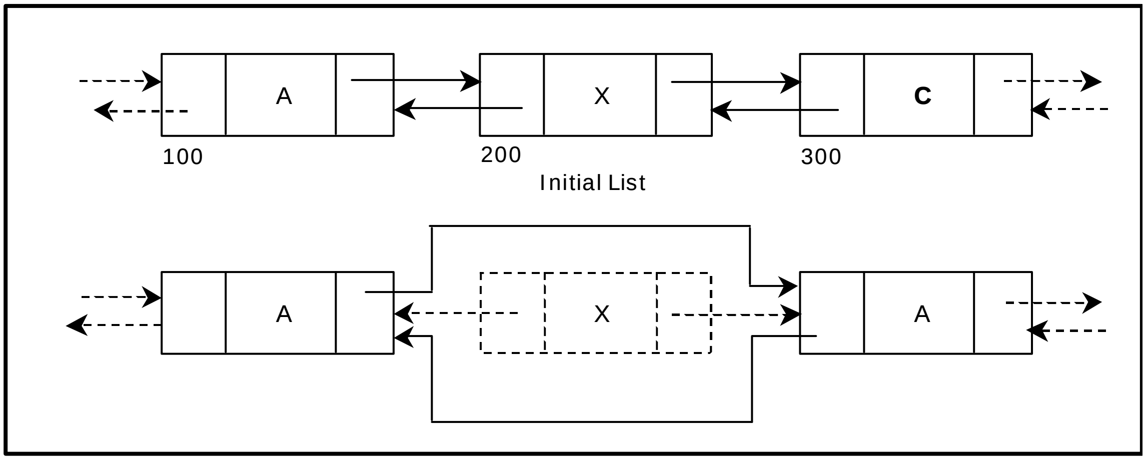


Figure 1.6 Detaching a node from a list

Since very little data is moved during this process, the deletion using linked lists will often be faster than when arrays are used.

It may seem that linked lists are superior to arrays. But is that always true? There are trade offs. Our linked lists yield faster deletions, but they take up more space because they require two extra pointers per element.

1.5. Algorithm

An **a**lgorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent an algorithm using pseudo language that is a combination of the constructs of a programming language together with informal English statements.

1.6. Practical Algorithm design issues:

Choosing an efficient algorithm or data structure is just one part of the design process. Next, will look at some design issues that are broader in scope. There are three basic design goals that we should strive for in a program:

1. Try to save time (Time complexity).
2. Try to save space (Space complexity).
3. Try to have face.

A program that runs faster is a better program, so saving time is an obvious goal. Like wise, a program that saves space over a competing program is considered desirable. We want to “save face” by preventing the program from locking up or generating reams of garbled data.

1.7. Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the ***TIME COMPLEXITY*** of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

1.8. Classification of Algorithms

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

1 Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

Log n When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, log n is a doubled whenever n doubles, log n increases by a constant, but log n does not double until n increases to n^2.

n When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

n. log n This running time arises for algorithms but solve a problem by breaking it up into smaller sub-problems, solving them independently, and then combining the solutions. When n doubles, the running time more than doubles.

n^2 When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold.

n^3 Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold.

2^n Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as "brute–force" solutions to problems. Whenever n doubles, the running time squares.

1.9. Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.

1.10. Rate of Growth

Big–Oh (O), Big–Omega (Ω), Big–Theta (Θ) and Little–Oh

1. T(n) = O(f(n)), (pronounced order of or big oh), says that the growth rate of T(n) is less than or equal ($\leq$) that of f(n)

2. T(n) = Ω(g(n)) (pronounced omega), says that the growth rate of T(n) is greater than or equal to ($\geq$) that of g(n)

3. T(n) = Θ(h(n)) (pronounced theta), says that the growth rate of T(n) equals (=) the growth rate of h(n) [if T(n) = O(h(n)) and T(n) = Ω (h(n))]

4. T(n) = o(p(n)) (pronounced little oh), says that the growth rate of T(n) is less than the growth rate of p(n) [if T(n) = O(p(n)) and T(n) $\neq$ Θ (p(n))].

Some Examples:

$2n^2 + 5n - 6 = O(2^n)$
$2n^2 + 5n - 6 = O(n^3)$
$2n^2 + 5n - 6 = O(n^2)$
$2n^2 + 5n - 6 \neq O(n)$

$2n^2 + 5n - 6 \neq \Omega(2^n)$
$2n^2 + 5n - 6 \neq \Omega(n^3)$
$2n^2 + 5n - 6 = \Omega(n^2)$
$2n^2 + 5n - 6 = \Omega(n)$

$2n^2 + 5n - 6 \neq \Theta(2^n)$
$2n^2 + 5n - 6 \neq \Theta(n^3)$
$2n^2 + 5n - 6 = \Theta(n^2)$
$2n^2 + 5n - 6 \neq \Theta(n)$

$2n^2 + 5n - 6 = o(2^n)$
$2n^2 + 5n - 6 = o(n^3)$
$2n^2 + 5n - 6 \neq o(n^2)$
$2n^2 + 5n - 6 \neq o(n)$

1.11. Analyzing Algorithms

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing times are:

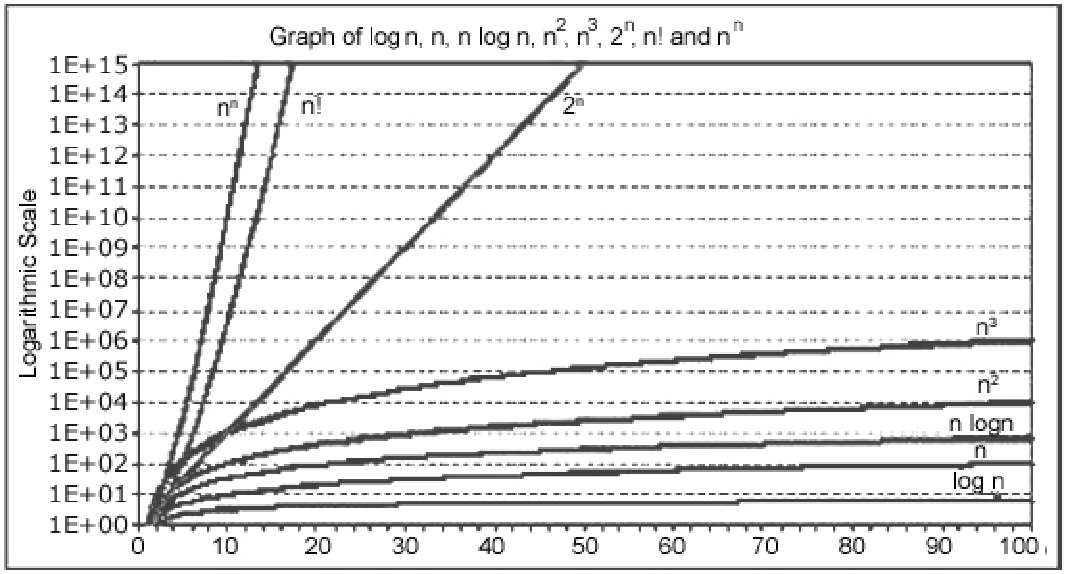
$O(1)$, $O(\log_2 n)$, $O(n)$, $O(n. \log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, $n!$ and n^n

Numerical Comparison of Different Algorithms

The execution time for six of the typical functions is given below:

S.No	log n	n	n. log n	n^2	n^3	2^n
1	0	1	1	1	1	2
2	1	2	2	4	8	4
3	2	4	8	16	64	16
4	3	8	24	64	512	256
5	4	16	64	256	4096	65536

Graph of log n, n, n log n, n^2, n^3, 2^n, n! and n^n



O(log n) does not depend on the base of the logarithm. To simplify the analysis, the convention will not have any particular units of time. Thus we throw away leading constants. We will also throw away low–order terms while computing a Big–Oh running time. Since Big-Oh is an upper bound, the answer provided is a guarantee that the program will terminate within a certain time period. The program may stop earlier than this, but never later.

One way to compare the function f(n) with these standard function is to use the functional 'O' notation, suppose f(n) and g(n) are functions defined on the positive integers with the property that f(n) is bounded by some multiple g(n) for almost all 'n'. Then,

$$f(n) = O(g(n))$$

Which is read as "f(n) is of order g(n)". For example, the order of complexity for:

- Linear search is O(n)
- Binary search is O(log n)
- Bubble sort is $O(n^2)$
- Quick sort is O(n log n)

For example, if the first program takes $100n^2$ milliseconds. While the second taken $5n^3$ milliseconds. Then might not $5n^3$ program better than $100n^2$ program?

As the programs can be evaluated by comparing their running time functions, with constants by proportionality neglected. So, $5n^3$ program be better than the $100n^2$ program.

$$5\ n^3/100\ n^2 = n/20$$

for inputs n < 20, the program with running time $5n^3$ will be faster those the one with running time $100\ n^2$.

Therefore, if the program is to be run mainly on inputs of small size, we would indeed prefer the program whose running time was $O(n^3)$

However, as 'n' gets large, the ratio of the running times, which is n/20, gets arbitrarily larger. Thus, as the size of the input increases, the $O(n^3)$ program will take significantly more time than the $O(n^2)$ program. So it is always better to prefer a program whose running time with the lower growth rate. The low growth rate function's such as O(n) or O(n log n) are always better.

Exercises

1. Define algorithm.

2. State the various steps in developing algorithms?

3. State the properties of algorithms.

4. Define efficiency of an algorithm?

5. State the various methods to estimate the efficiency of an algorithm.

6. Define time complexity of an algorithm?

7. Define worst case of an algorithm.

8. Define average case of an algorithm.

9. Define best case of an algorithm.

10. Mention the various spaces utilized by a program.

11. Define space complexity of an algorithm.

12. State the different memory spaces occupied by an algorithm.

Multiple Choice Questions

1. _____ is a step-by-step recipe for solving an instance of problem. [A]
A. Algorithm B. Complexity
C. Pseudocode D. Analysis

2. ______ is used to describe the algorithm, in less formal language. [C]
A. Cannot be defined B. Natural Language
C. Pseudocode D. None

3. ______ of an algorithm is the amount of time (or the number of steps) needed by a program to complete its task. [D]
A. Space Complexity B. Dynamic Programming
C. Divide and Conquer D. Time Complexity

4. ______ of a program is the amount of memory used at once by the algorithm until it completes its execution. [C]

A. Divide and Conquer B. Time Complexity
C. Space Complexity D. Dynamic Programming

5. ______ is used to define the worst-case running time of an algorithm. [A]
A. Big-Oh notation B. Cannot be defined
C. Complexity D. Analysis

Chapter 2

Recursion

Recursion is deceptively simple in statement but exceptionally complicated in implementation. Recursive procedures work fine in many problems. Many programmers prefer recursion through simpler alternatives are available. It is because recursion is elegant to use through it is costly in terms of time and space. But using it is one thing and getting involved with it is another.

In this unit we will look at "recursion" as a programmer who not only loves it but also wants to understand it! With a bit of involvement it is going to be an interesting reading for you.

2.1. Introduction to Recursion:

A function is recursive if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself. For a computer language to be recursive, a function must be able to call itself.

For example, let us consider the function factr() shown below, which computers the factorial of an integer.

```
#include <stdio.h>
int factorial (int);
main()
{
        int num, fact;
        printf (“Enter a positive integer value: ");
        scanf (“%d”, &num);
        fact = factorial (num);
        printf ("\n Factorial of %d =%5d\n", num, fact);
}

int factorial (int n)
{
        int result;
        if (n == 0)
                return (1);
        else
                result = n * factorial (n-1);

        return (result);
}
```

A non-recursive or iterative version for finding the factorial is as follows:

```
factorial (int n)
{
        int i, result = 1;
        if (n == 0)
```

```
                return (result);
        else
        {
                for (i=1; i<=n; i++)
                        result = result * i;
        }
        return (result);
}
```

The operation of the non-recursive version is clear as it uses a loop starting at 1 and ending at the target value and progressively multiplies each number by the moving product.

When a function calls itself, new local variables and parameters are allocated storage on the stack and the function code is executed with these new variables from the start. A recursive call does not make a new copy of the function. Only the arguments and variables are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function.

When writing recursive functions, you must have a exit condition somewhere to force the function to return without the recursive call being executed. If you do not have an exit condition, the recursive function will recurse forever until you run out of stack space and indicate error about lack of memory, or stack overflow.

2.2. Differences between recursion and iteration:

- Both involve repetition.
- Both involve a termination test.
- Both can occur infinitely.

Iteration	Recursion
Iteration explicitly user a repetition structure.	Recursion achieves repetition through repeated function calls.
Iteration terminates when the loop continuation.	Recursion terminates when a base case is recognized.
Iteration keeps modifying the counter until the loop continuation condition fails.	Recursion keeps producing simple versions of the original problem until the base case is reached.
Iteration normally occurs within a loop so the extra memory assigned is omitted.	Recursion causes another copy of the function and hence a considerable memory space’s occupied.
It reduces the processor’s operating time.	It increases the processor’s operating time.

2.3. Factorial of a given number:

The operation of recursive factorial function is as follows:

Start out with some natural number N (in our example, 5). The recursive definition is:

n = 0, 0 ! = 1	Base Case
n > 0, n ! = n * (n - 1) !	Recursive Case

Recursion Factorials:

5! =5 * 4! = 5 *___ = ____ factr(5) = 5 * factr(4) = __

4! = 4 *3! = 4 *___ = ___ factr(4) = 4 * factr(3) = __

3! = 3 * 2! = 3 * ___ = ___ factr(3) = 3 * factr(2) = __

2! = 2 * 1! = 2 * ___ = ___ factr(2) = 2 * factr(1) = __

1! = 1 * 0! = 1 * __ = __ factr(1) = 1 * factr(0) = __

0! = 1 factr(0) = __

5! = 5*4! = 5*4*3! = 5*4*3*2! = 5*4*3*2*1! = 5*4*3*2*1*0! = 5*4*3*2*1*1 =120

We define 0! to equal 1, and we define factorial N (where N > 0), to be N * factorial (N-1). All recursive functions must have an exit condition, that is a state when it does not recurse upon itself. Our exit condition in this example is when N = 0.

Tracing of the flow of the factorial () function:

When the factorial function is first called with, say, N = 5, here is what happens:

FUNCTION:
Does N = 0? No
Function Return Value = 5 * factorial (4)

At this time, the function factorial is called again, with N = 4.

FUNCTION:
Does N = 0? No
Function Return Value = 4 * factorial (3)

At this time, the function factorial is called again, with N = 3.

FUNCTION:
Does N = 0? No
Function Return Value = 3 * factorial (2)

At this time, the function factorial is called again, with N = 2.

FUNCTION:
Does N = 0? No
Function Return Value = 2 * factorial (1)

At this time, the function factorial is called again, with N = 1.

FUNCTION:
Does N = 0? No
Function Return Value = 1 * factorial (0)

At this time, the function factorial is called again, with N = 0.

FUNCTION:
Does N = 0? Yes
Function Return Value = 1

Now, we have to trace our way back up! See, the factorial function was called six times. At any function level call, all function level calls above still exist! So, when we have N = 2, the function instances where N = 3, 4, and 5 are still waiting for their return values.

So, the function call where N = 1 gets retraced first, once the final call returns 0. So, the function call where N = 1 returns 1*1, or 1. The next higher function call, where N = 2, returns 2 * 1 (1, because that's what the function call where N = 1 returned). You just keep working up the chain.

When N = 2, 2 * 1, or 2 was returned.
When N = 3, 3 * 2, or 6 was returned.
When N = 4, 4 * 6, or 24 was returned.
When N = 5, 5 * 24, or 120 was returned.

And since N = 5 was the first function call (hence the last one to be recalled), the value 120 is returned.

2.4. The Towers of Hanoi:

In the game of Towers of Hanoi, there are three towers labeled 1, 2, and 3. The game starts with n disks on tower A. For simplicity, let n is 3. The disks are numbered from 1 to 3, and without loss of generality we may assume that the diameter of each disk is the same as its number. That is, disk 1 has diameter 1 (in some unit of measure), disk 2 has diameter 2, and disk 3 has diameter 3. All three disks start on tower A in the order 1, 2, 3. The objective of the game is to move all the disks in tower 1 to entire tower 3 using tower 2. That is, at no time can a larger disk be placed on a smaller disk.

Figure 3.11.1, illustrates the initial setup of towers of Hanoi. The figure 3.11.2, illustrates the final setup of towers of Hanoi.

The rules to be followed in moving the disks from tower 1 tower 3 using tower 2 are as follows:

- Only one disk can be moved at a time.
- Only the top disc on any tower can be moved to any other tower.
- A larger disk cannot be placed on a smaller disk.

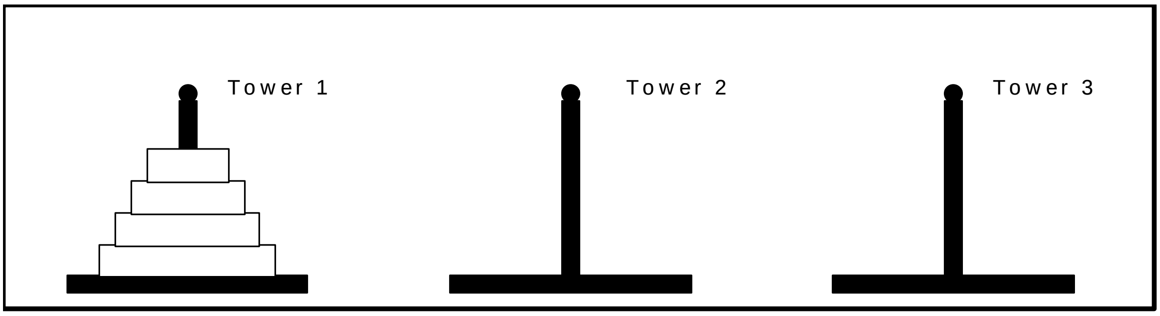


Fig. 3.11.1. Initial setup of Towers of Hanoi

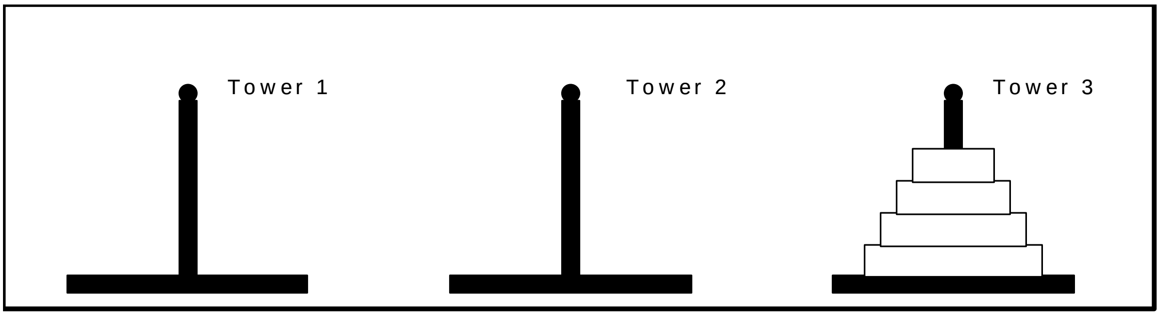


Fig 3.11.2. Final setup of Towers of Hanoi

The towers of Hanoi problem can be easily implemented using recursion. To move the largest disk to the bottom of tower 3, we move the remaining n – 1 disks to tower 2 and then move the largest disk to tower 3. Now we have the remaining n – 1 disks to be moved to tower 3. This can be achieved by using the remaining two towers. We can also use tower 3 to place any disk on it, since the disk placed on tower 3 is the largest disk and continue the same operation to place the entire disks in tower 3 in order.

The program that uses recursion to produce a list of moves that shows how to accomplish the task of transferring the n disks from tower 1 to tower 3 is as follows:

```
#include <stdio.h>
#include <conio.h>

void towers_of_hanoi (int n, char *a, char *b, char *c);

int cnt=0;

int main (void)
{
        int n;
        printf("Enter number of discs: ");
        scanf("%d",&n);
        towers_of_hanoi (n, "Tower 1", "Tower 2", "Tower 3");
        getch();
}

void towers_of_hanoi (int n, char *a, char *b, char *c)
{
        if (n == 1)
        {
                ++cnt;
                printf ("\n%5d: Move disk 1 from %s to %s", cnt, a, c);
                return;
        }
        else
        {
                towers_of_hanoi (n-1, a, c, b);
                ++cnt;
                printf ("\n%5d: Move disk %d from %s to %s", cnt, n, a, c);
                towers_of_hanoi (n-1, b, a, c);
                return;
        }
}
```

Output of the program:

RUN 1:

```
Enter the number of discs: 3

1:      Move disk  1  from tower  1  to tower  3.
2:      Move disk  2  from tower  1  to tower  2.
3:      Move disk  1  from tower  3  to tower  2.
4:      Move disk  3  from tower  1  to tower  3.
5:      Move disk  1  from tower  2  to tower  1.
6:      Move disk  2  from tower  2  to tower  3.
7:      Move disk  1  from tower  1  to tower  3.
```

RUN 2:

```
Enter the number of discs: 4

1:      Move disk  1  from tower  1  to tower  2.
2:      Move disk  2  from tower  1  to tower  3.
3:      Move disk  1  from tower  2  to tower  3.
4:      Move disk  3  from tower  1  to tower  2.
5:      Move disk  1  from tower  3  to tower  1.
6:      Move disk  2  from tower  3  to tower  2.
7:      Move disk  1  from tower  1  to tower  2.
8:      Move disk  4  from tower  1  to tower  3.
9:      Move disk  1  from tower  2  to tower  3.
10:     Move disk  2  from tower  2  to tower  1.
11:     Move disk  1  from tower  3  to tower  1.
12:     Move disk  3  from tower  2  to tower  3.
13:     Move disk  1  from tower  1  to tower  2.
14:     Move disk  2  from tower  1  to tower  3.
15:     Move disk  1  from tower  2  to tower  3.
```

2.5. Fibonacci Sequence Problem:

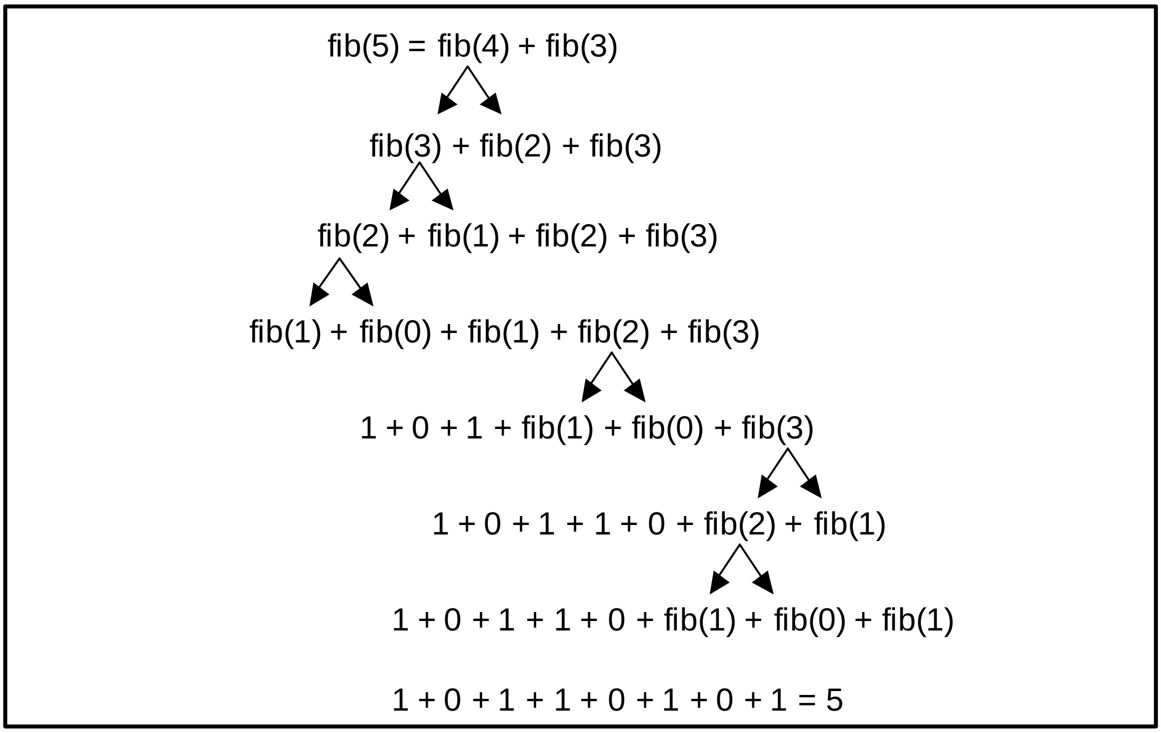
A Fibonacci sequence starts with the integers 0 and 1. Successive elements in this sequence are obtained by summing the preceding two elements in the sequence. For example, third number in the sequence is 0 + 1 = 1, fourth number is 1 + 1= 2, fifth number is 1 + 2 = 3 and so on. The sequence of Fibonacci integers is given below:

0 1 1 2 3 5 8 13 21 . . . .

A recursive definition for the Fibonacci sequence of integers may be defined as follows:

Fib (n) = n if n = 0 or n = 1
Fib (n) = fib (n-1) + fib (n-2) for n >=2

We will now use the definition to compute fib(5):


fib(5) = fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
fib(1) + fib(0) + fib(1) + fib(2) + fib(3)
1 + 0 + 1 + fib(1) + fib(0) + fib(3)
1 + 0 + 1 + 1 + 0 + fib(2) + fib(1)
1 + 0 + 1 + 1 + 0 + fib(1) + fib(0) + fib(1)
1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 = 5

We see that fib(2) is computed 3 times, and fib(3), 2 times in the above calculations. We save the values of fib(2) or fib(3) and reuse them whenever needed.

A recursive function to compute the Fibonacci number in the n^{th} position is given below:

```
main()
{
        clrscr ();
        printf (“=nfib(5) is %d”, fib(5));
}

fib(n)
int n;
{
        int x;
        if (n==0 | | n==1)
        return n;
        x=fib(n-1) + fib(n-2);
        return (x);
}
```

Output:

```
fib(5) is 5
```

Chapter 3

LINKED LISTS

In this chapter, the list data structure is presented. This structure can be used as the basis for the implementation of other data structures (stacks, queues etc.). The basic linked list can be used without modification in many programs. However, some applications require enhancements to the linked list design. These enhancements fall into three broad categories and yield variations on linked lists that can be used in any combination: circular linked lists, double linked lists and lists with header nodes.

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collections of elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast. The disadvantages of arrays are:

- The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.

- Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

- Deleting an element from an array is not possible.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Here is a quick review of the terminology and rules of pointers. The linked list code will depend on the following functions:

malloc() is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype of malloc() and other heap functions are in stdlib.h. malloc() returns NULL if it cannot fulfill the request. It is defined by:

*void *malloc (number_of_bytes)*

Since a void * is returned the C standard states that this pointer can be converted to any type. For example,

```
char *cp;
cp = (char *) malloc (100);
```

Attempts to get 100 bytes and assigns the starting address to cp. We can also use the sizeof() function to specify the number of bytes. For example,

```
int *ip;
ip = (int *) malloc (100*sizeof(int));
```

free() is the opposite of malloc(), which de-allocates memory. The argument to free() is a pointer to a block of memory in the heap — a pointer which was obtained by a malloc() function. The syntax is:

free (ptr);

The advantage of free() is simply memory management when we no longer need a block.

3.1. Linked List Concepts:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

3.2. Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

A circular double linked list is one, which has both the successor pointer and predecessor pointer in the circular manner.

Comparison between array and linked list:

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Trade offs between linked lists and arrays:

FEATURE	ARRAYS	LINKED LISTS
Sequential access	efficient	efficient
Random access	efficient	inefficient
Resigning	inefficient	efficient
Element rearranging	inefficient	efficient
Overhead per elements	none	1 or 2 links

Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example:

   $P(x) = a_0 X^n + a_1 X^{n-1} + \ldots\ldots + a_{n-1} X + a_n$

2. Represent very large numbers and operations of the large number such as addition, multiplication and division.

3. Linked lists are to implement stack, queue, trees and graphs.

4. Implement the symbol table in compiler construction

3.3. Single Linked List:

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the "start" node.

A single linked list is shown in figure 3.2.1.

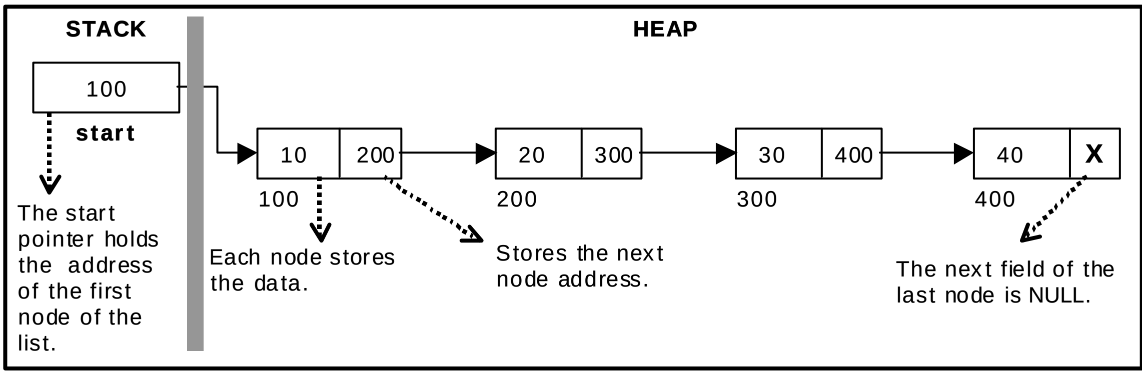

STACK
HEAP
100
start
10
200
20
300
30
400
40
X
100
200
300
400
The start pointer holds the address of the first node of the list.
Each node stores the data.
Stores the next node address.
The next field of the last node is NULL.

Figure 3.2.1. Single Linked List

The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the **start** and following the next pointers.

The **start** pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

Implementation of Single Linked List:

Before writing the code to build the above list, we need to create a **start** node**,** used to create and access other nodes in the linked list. The following structure definition will do (see figure 3.2.2):

- Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.
- Initialise the start pointer to be NULL.

```
struct slinklist
{
        int data;
        struct slinklist* next;
};

typedef struct slinklist node;

node *start = NULL;
```

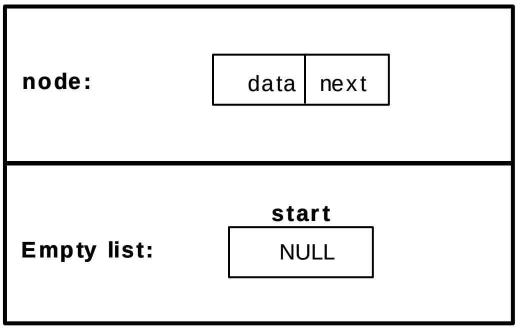


Figure 3.2.2. Structure definition, single link node and empty list

The basic operations in a single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node. Figure 3.2.3 illustrates the creation of a node for single linked list.

```
node* getnode()
{
    node* newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> next = NULL;
    return newnode;
}
```

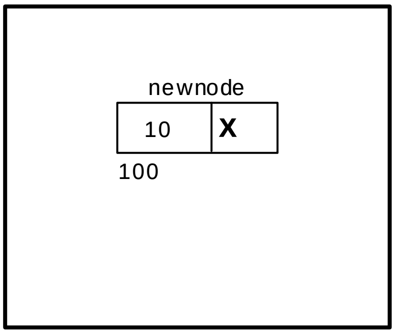


Figure 3.2.3. new node with a value of 10

Creating a Singly Linked List with ‘n’ number of nodes:

The following steps are to be followed to create ‘n’ number of nodes:

- Get the new node using getnode().
  newnode = getnode();

- If the list is empty, assign new node as start.
  start = newnode;

- If the list is not empty, follow the steps given below:

  - The next field of the new node is made to point the first node (i.e. start node) in the list by assigning the address of the first node.

  - The start pointer is made to point the new node by assigning the address of the new node.

- Repeat the above steps ‘n’ times.

Figure 3.2.4 shows 4 items in a single linked list stored at different locations in memory.

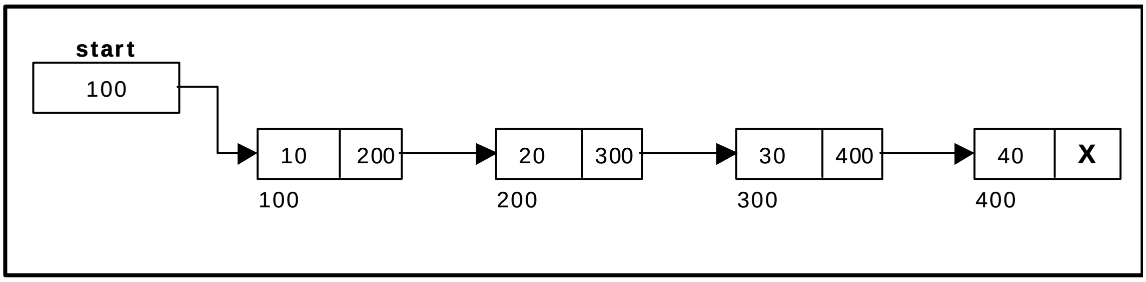


Figure 3.2.4. Single Linked List with 4 nodes

The function createlist(), is used to create ‘n’ number of nodes:

```
void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        for(i = 0; i < n ; i++)
        {
                newnode = getnode();
                if(start == NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> next != NULL)
                                temp = temp -> next;
                        temp -> next = newnode;
                }
        }
}
```

Insertion of a Node:

One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

- Inserting a node at the beginning.
- Inserting a node at the end.
- Inserting a node at intermediate position.

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using getnode().
  ```
  newnode = getnode();
  ```
- If the list is empty then *start = newnode.*
- If the list is not empty, follow the steps given below:
  ```
  newnode -> next = start;
  start = newnode;
  ```

Figure 3.2.5 shows inserting a node into the single linked list at the beginning.

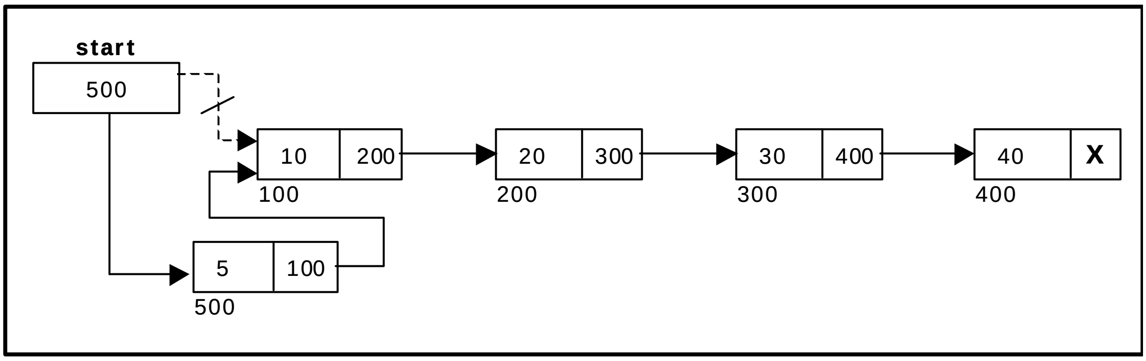


Figure 3.2.5. Inserting a node at the beginning

The function insert_at_beg(), is used for inserting a node at the beginning

```
void insert_at_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                newnode -> next = start;
                start = newnode;
        }
}
```

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode()
  ```
  newnode = getnode();
  ```

- If the list is empty then *start = newnode*.

- If the list is not empty follow the steps given below:
  ```
  temp = start;
  while(temp -> next != NULL)
          temp = temp -> next;
  temp -> next = newnode;
  ```

Figure 3.2.6 shows inserting a node into the single linked list at the end.

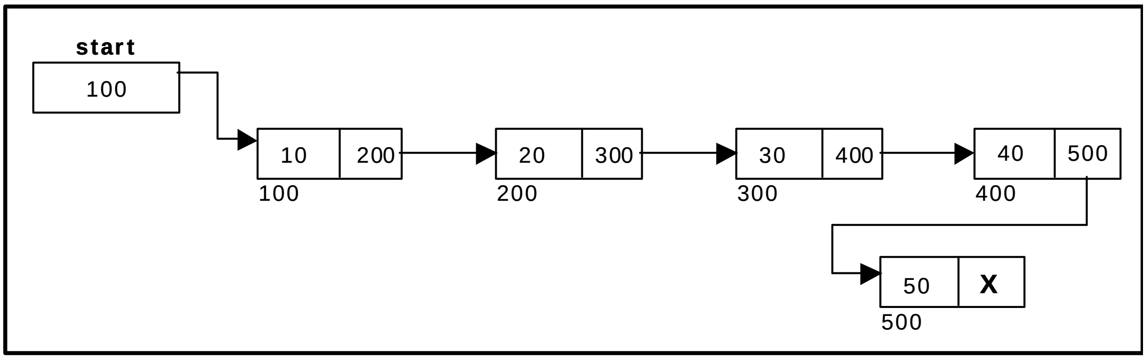


Figure 3.2.6. Inserting a node at the end.

The function insert_at_end(), is used for inserting a node at the end.

```
void insert_at_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                temp = start;
                while(temp -> next != NULL)
                        temp  = temp -> next;
                temp -> next = newnode;
        }
}
```

Inserting a node at intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using getnode().
  ```
  newnode = getnode();
  ```

- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

- After reaching the specified position, follow the steps given below:
  prev -> next = newnode;
  newnode -> next = temp;

- Let the intermediate position be 3.

Figure 3.2.7 shows inserting a node into the single linked list at a specified intermediate position other than beginning and end.

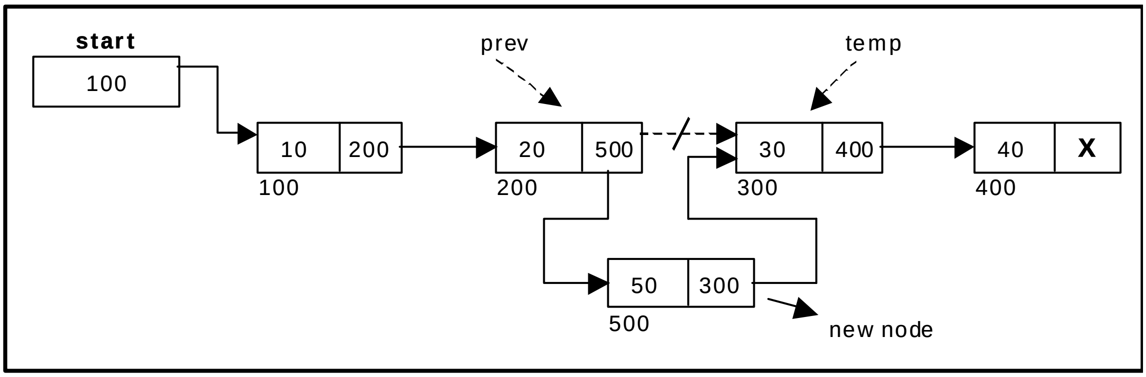


Figure 3.2.7. Inserting a node at an intermediate position.

The function insert_at_mid(), is used for inserting a node in the intermediate position.

```
void insert_at_mid()
{
        node *newnode, *temp, *prev;
        int  pos, nodectr, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > 1 && pos < nodectr)
        {
                temp = prev = start;
                while(ctr < pos)
                {
                        prev = temp;
                        temp = temp -> next;
                        ctr++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
        }
        else
        {
                printf("position %d is not a middle position", pos);
        }
}
```

Deletion of a node:

Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

- Deleting a node at the beginning.
- Deleting a node at the end.
- Deleting a node at intermediate position.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> next;
free(temp);
```

Figure 3.2.8 shows deleting a node at the beginning of a single linked list.

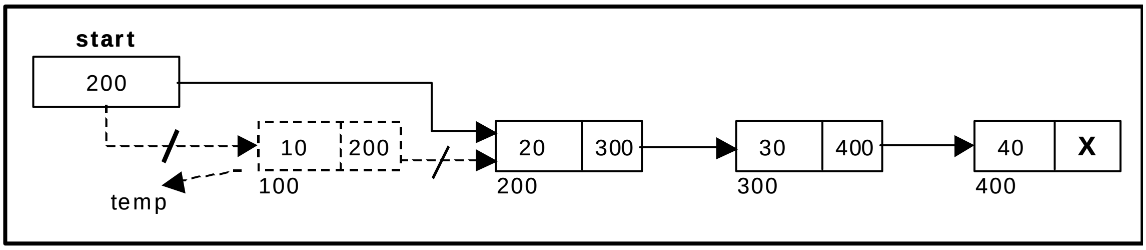


Figure 3.2.8. Deleting a node at the beginning.

The function delete_at_beg(), is used for deleting the first node in the list.

```
void delete_at_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes are exist..");
                return ;
        }
        else
        {
                temp = start;
                start = temp -> next;
                free(temp);
                printf("\n Node deleted ");
        }
}
```

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = prev = start;
while(temp -> next != NULL)
{
        prev = temp;
        temp = temp -> next;
}
prev -> next = NULL;
free(temp);
```

Figure 3.2.9 shows deleting a node at the end of a single linked list.

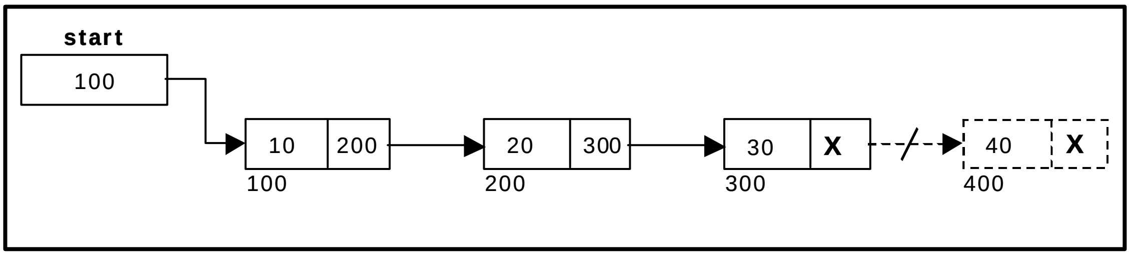


Figure 3.2.9. Deleting a node at the end.

The function delete_at_last(), is used for deleting the last node in the list.

```
void delete_at_last()
{
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != NULL)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = NULL;
                free(temp);
                printf("\n Node deleted ");
        }
}
```

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

- If list is empty then display 'Empty List' message

- If the list is not empty, follow the steps given below.

```
    if(pos > 1 && pos < nodectr)
    {
            temp = prev = start;
            ctr = 1;
            while(ctr < pos)
            {
                    prev = temp;
                    temp = temp -> next;
                    ctr++;
            }
            prev -> next = temp -> next;
            free(temp);
            printf("\n node deleted..");
    }
```

Figure 3.2.10 shows deleting a node at a specified intermediate position other than beginning and end from a single linked list.

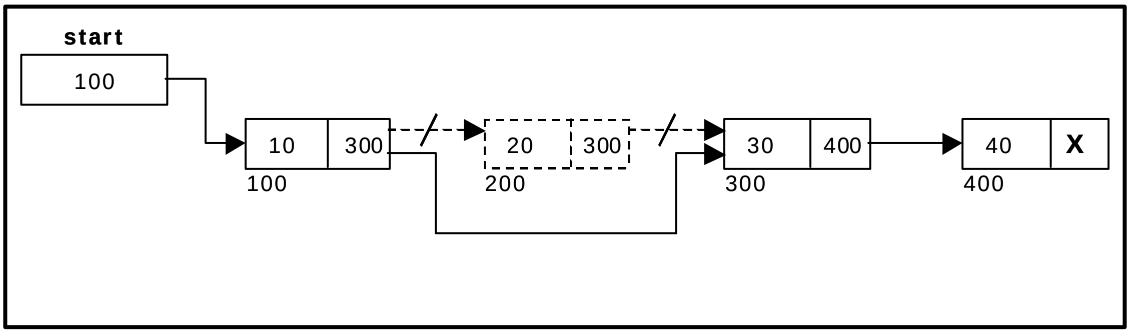


Figure 3.2.10. Deleting a node at an intermediate position.

The function delete_at_mid(), is used for deleting the intermediate node in the list.

```
void delete_at_mid()
{
        int ctr = 1, pos, nodectr;
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
                return ;
        }
        else
        {
                printf("\n Enter position of node to delete: ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos > nodectr)
                {
                        printf("\nThis node doesnot exist");
                }
                if(pos > 1 && pos < nodectr)
                {
                        temp = prev = start;
                        while(ctr < pos)
                        {
                                prev = temp;
                                temp = temp -> next;
                                ctr ++;
                        }
                        prev -> next = temp -> next;
                        free(temp);
                        printf("\n Node deleted..");
                }
                else
                {
                        printf("\n Invalid position..");
                        getch();
                }

        }
}
```

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves the following steps:

- Assign the address of start pointer to a temp pointer.
- Display the information from the data field of each node.

The function *traverse*() is used for traversing and displaying the information stored in the list from left to right.

```
void traverse()
{
        node *temp;
        temp = start;
        printf("\n The contents of List (Left to Right): \n");
        if(start == NULL )
                printf("\n Empty List");
        else
        {
                while (temp != NULL)
                {
                        printf("%d ->", temp -> data);
                        temp = temp -> next;
                }
        }
        printf("X");
}
```

Alternatively there is another way to traverse and display the information. That is in reverse order. The function rev_traverse(), is used for traversing and displaying the information stored in the list from right to left.

```
void rev_traverse(node *st)
{
        if(st == NULL)
        {
                return;
        }
        else
        {
                rev_traverse(st -> next);
                printf("%d ->", st -> data);
        }
}
```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list using *recursion*.

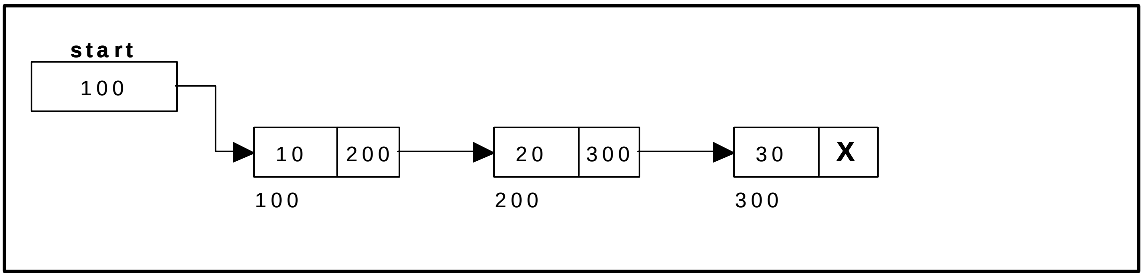
```
int countnode(node *st)
{
        if(st == NULL)
                return 0;
        else
                return(1 + countnode(st -> next));
}
```

3.3.1. Source Code for the Implementation of Single Linked List:

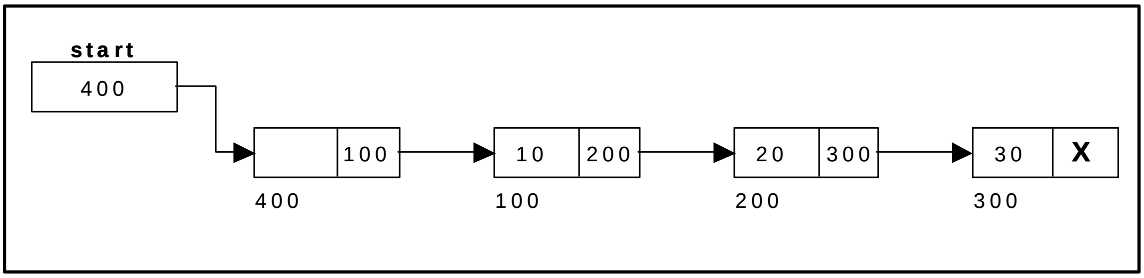
```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

struct slinklist
{
        int data;
        struct slinklist *next;
};

typedef struct slinklist node;

node *start = NULL;
int menu()
{
        int ch;
        clrscr();
        printf("\n 1.Create a list ");
        printf("\n-------------------------");
        printf("\n 2.Insert a node at beginning ");
        printf("\n 3.Insert a node at end");
        printf("\n 4.Insert a node at middle");
        printf("\n-------------------------");
        printf("\n 5.Delete a node from beginning");
        printf("\n 6.Delete a node from Last");
        printf("\n 7.Delete a node from Middle");
        printf("\n-------------------------");
        printf("\n 8.Traverse the list (Left to Right)");
        printf("\n 9.Traverse the list (Right to Left)");
        printf("\n-------------------------");
        printf("\n 10. Count nodes ");
        printf("\n 11. Exit ");
        printf("\n\n Enter your choice: ");
        scanf("%d",&ch);
        return ch;
}

node* getnode()
{
        node * newnode;
        newnode = (node *) malloc(sizeof(node));
        printf("\n Enter data: ");
        scanf("%d", &newnode -> data);
        newnode -> next = NULL;
        return newnode;
}

int countnode(node *ptr)
{
        int count=0;
        while(ptr != NULL)
        {
                count++;
                ptr = ptr -> next;
        }
        return (count);
}

void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        for(i = 0; i < n; i++)
        {
                newnode = getnode();
                if(start == NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> next != NULL)
                                temp = temp -> next;
                        temp -> next = newnode;
                }
        }
}

void traverse()
{
        node *temp;
        temp = start;
        printf("\n The contents of List (Left to Right): \n");
        if(start == NULL)
        {
                printf("\n Empty List");
                return;
        }
        else
        {
                while(temp != NULL)
                {
                        printf("%d-->", temp -> data);
                        temp = temp -> next;
                }
        }
        printf(" X ");
}

void rev_traverse(node *start)
{
        if(start == NULL)
        {
                return;
        }
        else
        {
                rev_traverse(start -> next);
                printf("%d -->", start -> data);
        }
}

void insert_at_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                newnode -> next = start;
                start = newnode;
        }
}

void insert_at_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                temp = start;
                while(temp -> next != NULL)
                        temp  = temp -> next;
                temp -> next = newnode;
        }
}

void insert_at_mid()
{
        node *newnode, *temp, *prev;
        int  pos, nodectr, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > 1 && pos < nodectr)
        {
                temp = prev = start;
                while(ctr < pos)
                {
                        prev = temp;
                        temp = temp -> next;
                        ctr++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
        }
        else
                printf("position %d is not a middle position", pos);
}

void delete_at_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes are exist..");
                return ;
        }
        else
        {
                temp = start;
                start = temp -> next;
                free(temp);
                printf("\n Node deleted ");
        }
}

void delete_at_last()
{
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != NULL)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = NULL;
                free(temp);
                printf("\n Node deleted ");
        }
}

void delete_at_mid()
{
        int ctr = 1, pos, nodectr;
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
                return ;
        }
        else
        {
                printf("\n Enter position of node to delete: ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos > nodectr)
                {
                        printf("\nThis node doesnot exist");

                }
                if(pos > 1 && pos < nodectr)
                {
                        temp = prev = start;
                        while(ctr < pos)
                        {
                                prev = temp;
                                temp = temp -> next;
                                ctr ++;
                        }
                        prev -> next = temp -> next;
                        free(temp);
                        printf("\n Node deleted..");
                }
                else
                {
                        printf("\n Invalid position..");
                        getch();
                }
        }
}

void main(void)
{
        int  ch, n;
        clrscr();
        while(1)
        {
                ch = menu();
                switch(ch)
                {
                case 1:
                        if(start == NULL)
                        {
                                printf("\n Number of nodes you want to create: ");
                                scanf("%d", &n);
                                createlist(n);
                                printf("\n List created..");
                        }
                        else
                                printf("\n List is already created..");
                                break;
                case 2:
                        insert_at_beg();
                        break;
                case 3:
                        insert_at_end();
                        break;
                case 4:
                        insert_at_mid();
                        break;
                case 5:
                        delete_at_beg();
                        break;
                case 6:
                        delete_at_last();
                        break;
                case 7:
                        delete_at_mid();
                        break;
                case 8:
                        traverse();
                        break;
                case 9:
                        printf("\n The contents of List (Right to Left): \n");
                        rev_traverse(start);
                        printf(" X ");
                        break;
                case 10:
                        printf("\n No of nodes : %d ", countnode(start));
                        break;
                case 11 :
                        exit(0);
                }
                getch();
        }
}
```

3.4. Using a header node:

A header node is a special dummy node found at the front of the list. The use of header node is an alternative to remove the first node in a list. For example, the picture below shows how the list with data 10, 20 and 30 would be represented using a linked list without and with a header node:



Single Linked List without a header node



Single Linked List with header node

Note that if your linked lists do include a header node, there is no need for the special case code given above for the *remove* operation; node ***n*** can never be the first node in the list, so there is no need to check for that case. Similarly, having a header node can simplify the code that adds a node before a given node ***n***.

Note that if you do decide to use a header node, you must remember to initialize an empty list to contain one (dummy) node, you must remember not to include the header node in the count of "real" nodes in the list.

It is also useful when information other than that found in each node of the list is needed. For example, imagine an application in which the number of items in a list is often calculated. In a standard linked list, the list function to count the number of nodes has to traverse the entire list every time. However, if the current length is maintained in a header node, that information can be obtained very quickly.

3.5. Array based linked lists:

Another alternative is to allocate the nodes in blocks. In fact, if you know the maximum size of a list a head of time, you can pre-allocate the nodes in a single array. The result is a hybrid structure – an array based linked list. Figure 3.5.1 shows an example of null terminated single linked list where all the nodes are allocated contiguously in an array.

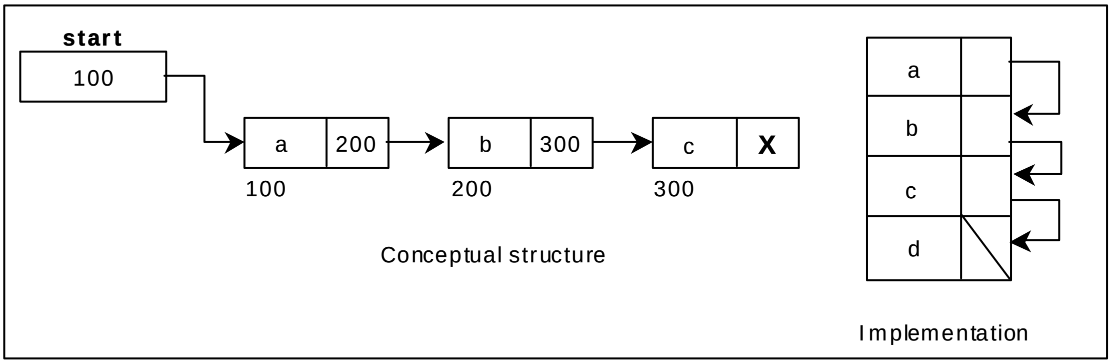


Figure 3.5.1. An array based linked list

3.6. Double Linked List:

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

- Left link.
- Data.
- Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data.

Many applications require searching forward and backward thru nodes of a list. For example searching for a name in a telephone directory would need forward and backward scanning thru a region of the whole list.

The basic operations in a double linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

A double linked list is shown in figure 3.3.1.

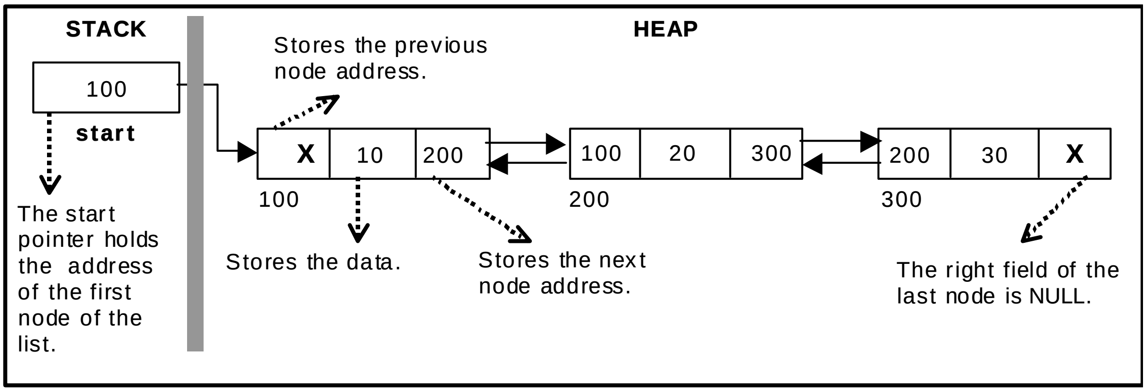

STACK
HEAP
100
start
The start pointer holds the address of the first node of the list.
Stores the previous node address.
X
10
200
100
20
300
200
30
X
100
200
300
Stores the data.
Stores the next node address.
The right field of the last node is NULL.

Figure 3.3.1. Double Linked List

The beginning of the double linked list is stored in a "**start**" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

The following code gives the structure definition:

```
struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;

};

typedef struct dlinklist node;
node *start = NULL;
```

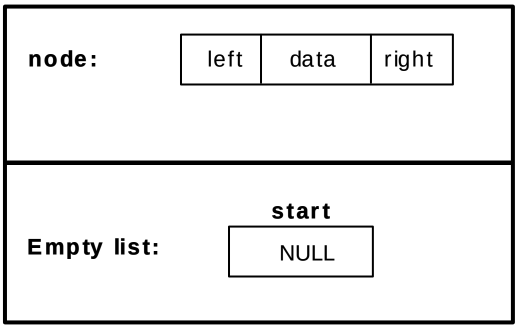


Figure 3.4.1. Structure definition, double link node and empty list

Creating a node for Double Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user and set left field to NULL and right field also set to NULL (see figure 3.2.2).

```
node* getnode()
{
        node* newnode;
        newnode = (node *) malloc(sizeof(node));
        printf("\n Enter data: ");
        scanf("%d", &newnode -> data);
        newnode -> left = NULL;
        newnode -> right = NULL;
        return newnode;
}
```

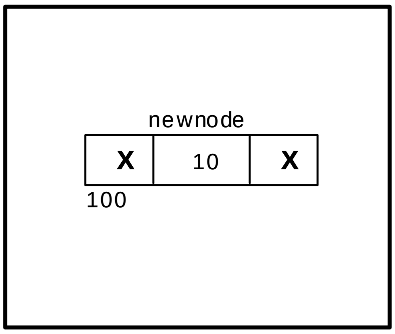


Figure 3.4.2. new node with a value of 10

Creating a Double Linked List with ‘n’ number of nodes:

The following steps are to be followed to create ‘n’ number of nodes:

- Get the new node using getnode().

  newnode =getnode();

- If the list is empty then *start = newnode*.
- If the list is not empty, follow the steps given below:
  - The left field of the new node is made to point the previous node.
  - The previous nodes right field must be assigned with address of the new node.
- Repeat the above steps ‘n’ times.

The function createlist(), is used to create ‘n’ number of nodes:

```
void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        for(i = 0; i < n; i++)
        {
                newnode = getnode();
                if(start == NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> right)
                                temp = temp -> right;
                        temp -> right = newnode;
                        newnode -> left = temp;
                }
        }
}
```

Figure 3.4.3 shows 3 items in a double linked list stored at different locations.

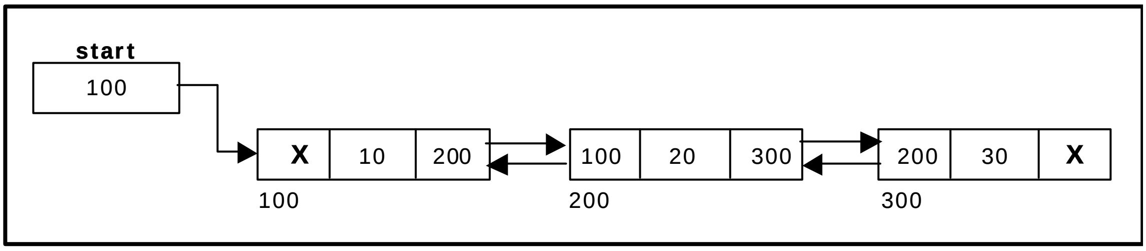


Figure 3.4.3. Double Linked List with 3 nodes

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using getnode().

```
newnode=getnode();
```

- If the list is empty then *start = newnode*.

- If the list is not empty, follow the steps given below:

```
newnode -> right = start;
start -> left = newnode;
start = newnode;
```

The function dbl_insert_beg(), is used for inserting a node at the beginning. Figure 3.4.4 shows inserting a node into the double linked list at the beginning.

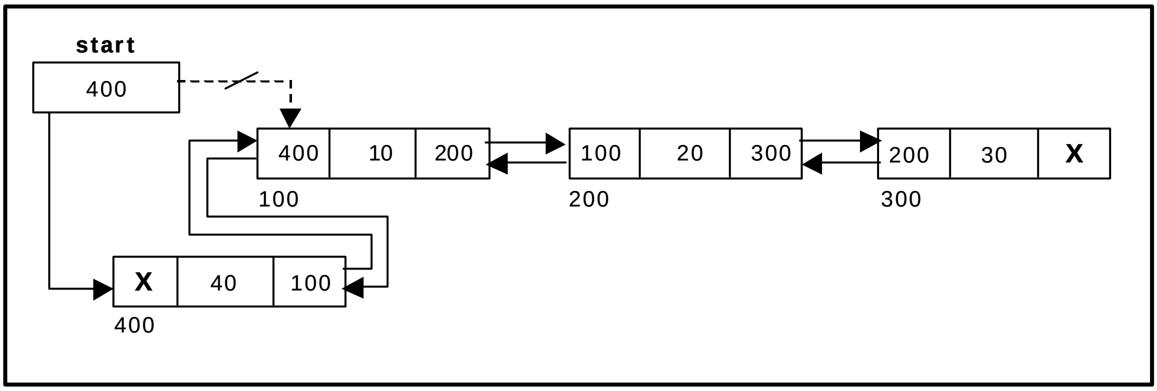


Figure 3.4.4. Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode()

```
newnode=getnode();
```

- If the list is empty then *start = newnode*.

- If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
        temp = temp -> right;
temp -> right = newnode;
newnode -> left = temp;
```

The function dbl_insert_end(), is used for inserting a node at the end. Figure 3.4.5 shows inserting a node into the double linked list at the end.

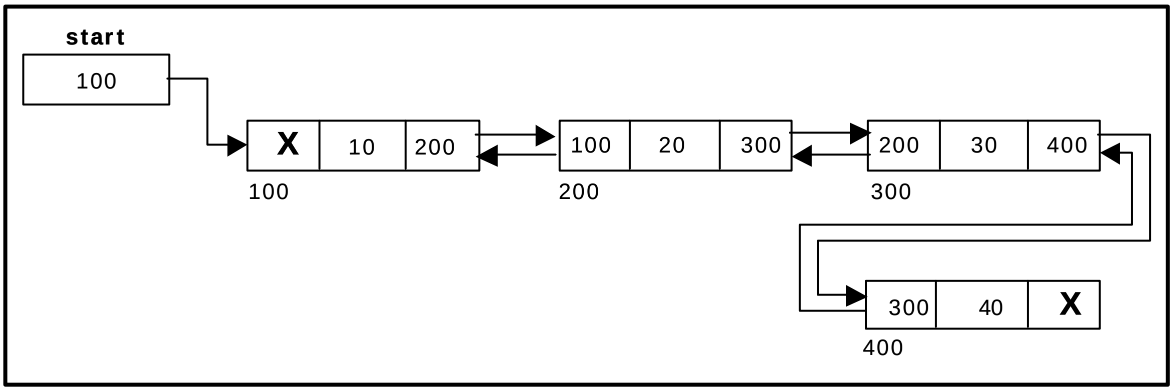


Figure 3.4.5. Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using getnode().

```
newnode=getnode();
```

- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below:

```
newnode -> left = temp;
newnode -> right = temp -> right;
temp -> right -> left = newnode;
temp -> right = newnode;
```

The function dbl_insert_mid(), is used for inserting a node in the intermediate position. Figure 3.4.6 shows inserting a node into the double linked list at a specified intermediate position other than beginning and end.

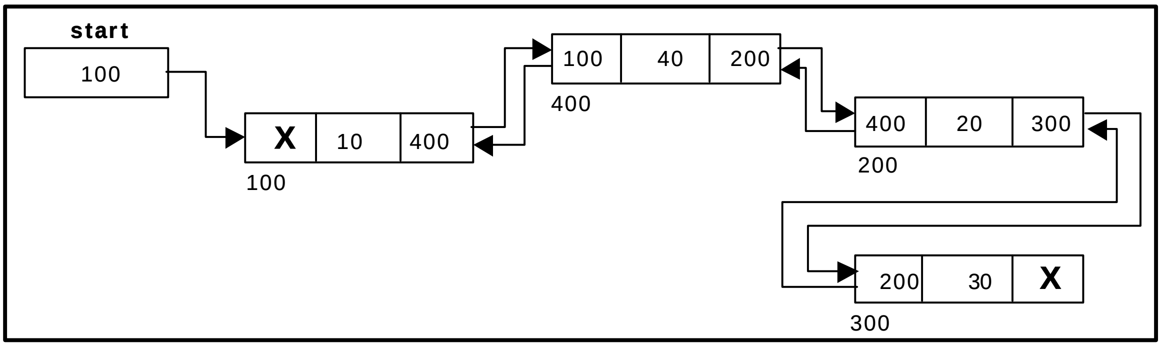


Figure 3.4.6. Inserting a node at an intermediate position

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.

- If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> right;
start -> left = NULL;
free(temp);
```

The function dbl_delete_beg(), is used for deleting the first node in the list. Figure 3.4.6 shows deleting a node at the beginning of a double linked list.

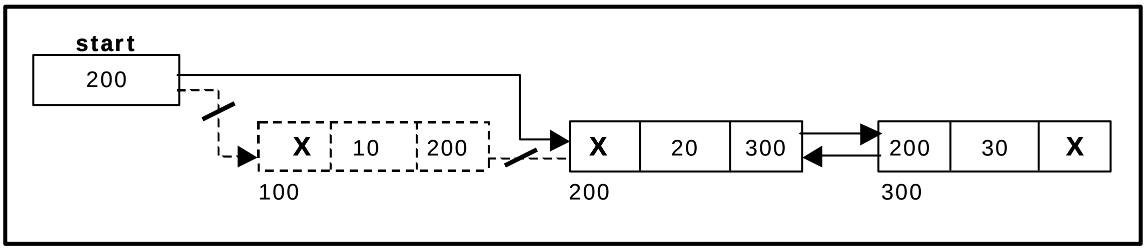


Figure 3.4.6. Deleting a node at beginning

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message

- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
{
        temp = temp -> right;
}
temp -> left -> right = NULL;
free(temp);
```

The function dbl_delete_last(), is used for deleting the last node in the list. Figure 3.4.7 shows deleting a node at the end of a double linked list.

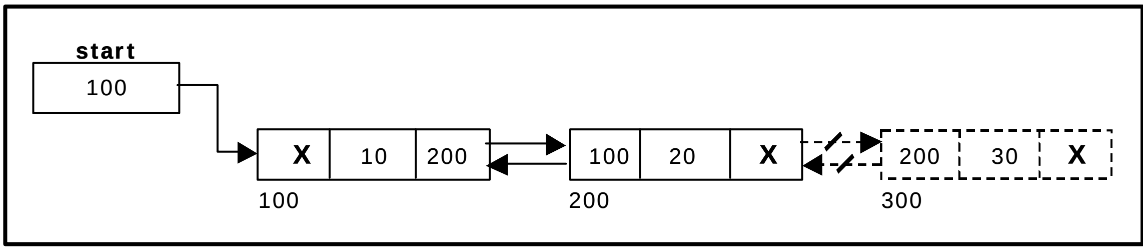


Figure 3.4.7. Deleting a node at the end

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two nodes).

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
  - Get the position of the node to delete.
  - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
  - Then perform the following steps:

```
if(pos > 1 && pos < nodectr)
{
        temp = start;
        i = 1;
        while(i < pos)
        {
                temp = temp -> right;
                i++;
        }
        temp -> right -> left = temp -> left;
        temp -> left -> right = temp -> right;
        free(temp);
        printf("\n node deleted..");
}
```

The function delete_at_mid(), is used for deleting the intermediate node in the list. Figure 3.4.8 shows deleting a node at a specified intermediate position other than beginning and end from a double linked list.

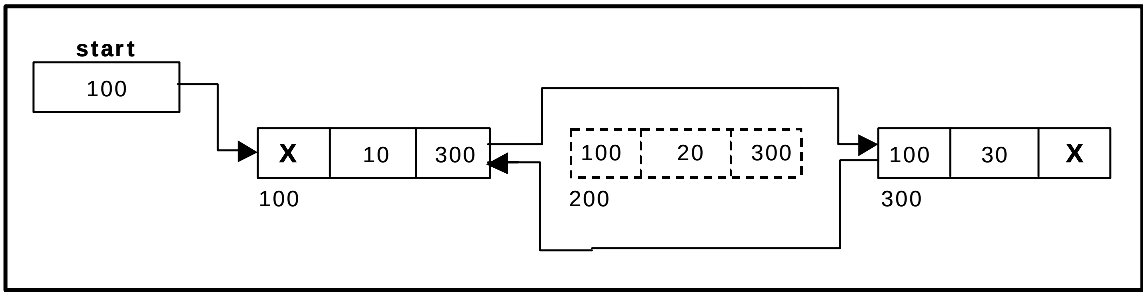


Figure 3.4.8 Deleting a node at an intermediate position

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_left_right*() is used for traversing and displaying the information stored in the list from left to right.

The following steps are followed, to traverse a list from left to right:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp != NULL)
{
        print  temp -> data;
        temp = temp -> right;
}
```

Traversal and displaying a list (Right to Left):

To display the information from right to left, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_right_left*() is used for traversing and displaying the information stored in the list from right to left. The following steps are followed, to traverse a list from right to left:

- If list is empty then display 'Empty List' message.

- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
        temp = temp -> right;
while(temp != NULL)
{
        print temp -> data;
        temp = temp -> left;
}
```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list (using recursion).

```
int countnode(node *start)
{
        if(start == NULL)
                return 0;
        else
                return(1 + countnode(start ->right ));
}
```

3.5. A Complete Source Code for the Implementation of Double Linked List:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;
};

typedef struct dlinklist node;
node *start = NULL;
```

```
                                printf("\n List created..");
                                break;
                        case 2 :
                                dll_insert_beg();
                                break;
                        case 3 :
                                dll_insert_end();
                                break;
                        case 4 :
                                dll_insert_mid();
                                break;
                        case 5 :
                                dll_delete_beg();
                                break;
                        case 6 :
                                dll_delete_last();
                                break;
                        case 7 :
                                dll_delete_mid();
                                break;
                        case 8 :
                                traverse_left_to_right();
                                break;
                        case 9 :
                                traverse_right_to_left();
                                break;

                        case 10 :
                                printf("\n Number of nodes: %d", countnode(start));
                                break;
                        case 11:
                                exit(0);
                }
                getch();
        }
}
```

3.7. Circular Single Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

A circular single linked list is shown in figure 3.6.1.

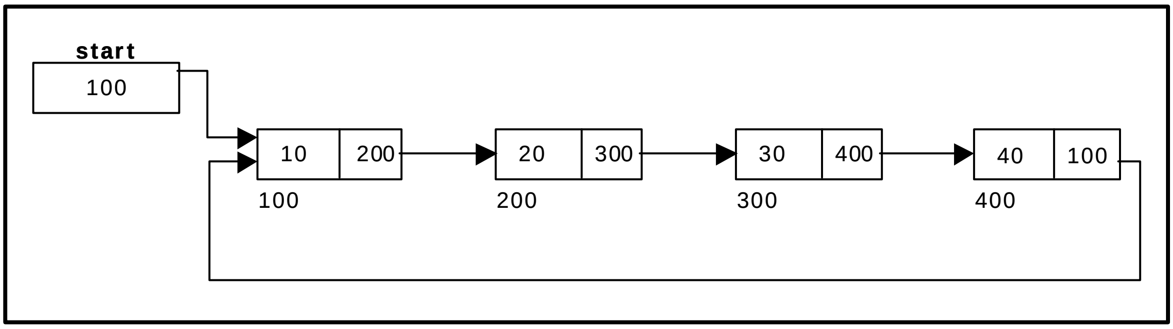


Figure 3.6.1. Circular Single Linked List

The basic operations in a circular single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a circular single Linked List with ‘n’ number of nodes:

The following steps are to be followed to create ‘n’ number of nodes:

- Get the new node using getnode().

```
newnode = getnode();
```

- If the list is empty, assign new node as start.

```
start = newnode;
```

- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> next != NULL)
        temp = temp -> next;
temp -> next = newnode;
```

- Repeat the above steps ‘n’ times.

- newnode -> next = start;

The function createlist(), is used to create ‘n’ number of nodes:

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the circular list:

- Get the new node using getnode().

```
newnode = getnode();
```

- If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next = start;
```

- If the list is not empty, follow the steps given below:

```
last = start;
while(last -> next != start)
        last = last -> next;
newnode -> next = start;
start = newnode;
last -> next = start;
```

The function cll_insert_beg(), is used for inserting a node at the beginning. Figure 3.6.2 shows inserting a node into the circular single linked list at the beginning.

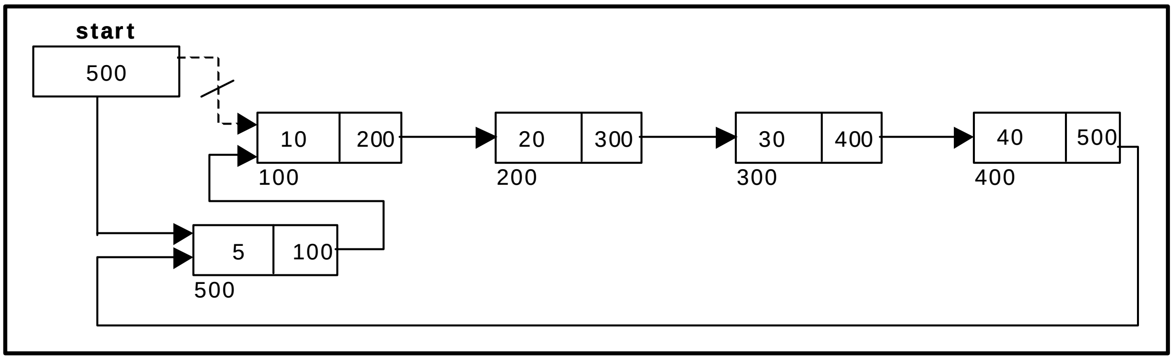


Figure 3.6.2. Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode().

```
newnode = getnode();
```

- If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next = start;
```

- If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> next != start)
        temp = temp -> next;
temp -> next = newnode;
newnode -> next = start;
```

The function cll_insert_end(), is used for inserting a node at the end.

Figure 3.6.3 shows inserting a node into the circular single linked list at the end.

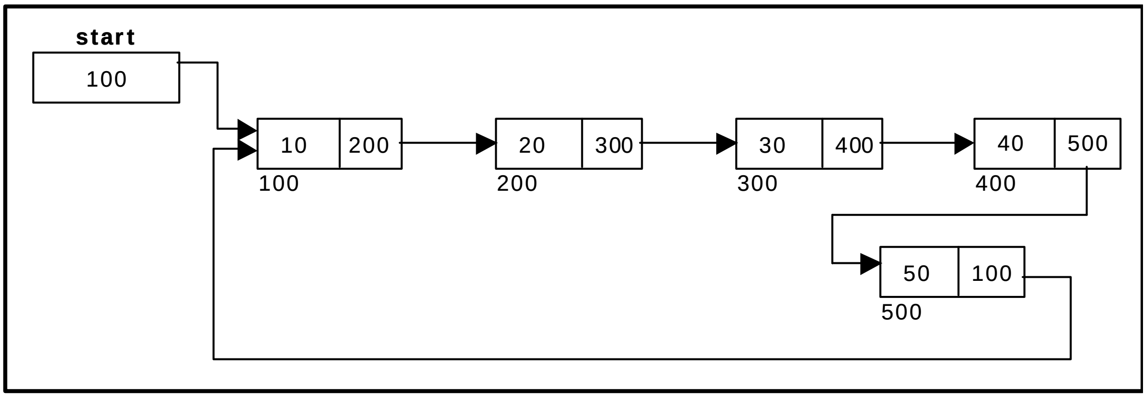


Figure 3.6.3 Inserting a node at the end.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below:

```
last = temp = start;
while(last -> next != start)
        last = last -> next;
start = start -> next;
last -> next = start;
```

- After deleting the node, if the list is empty then *start = NULL.*

The function cll_delete_beg(), is used for deleting the first node in the list. Figure 3.6.4 shows deleting a node at the beginning of a circular single linked list.

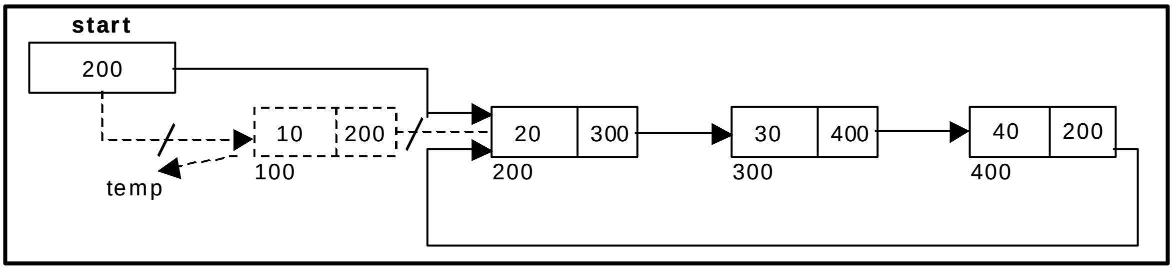


Figure 3.6.4. Deleting a node at beginning.

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below:

```
temp = start;
prev = start;
while(temp -> next != start)
{
        prev = temp;
        temp = temp -> next;
}
prev -> next = start;
```

- After deleting the node, if the list is empty then *start = NULL.*

The function cll_delete_last(), is used for deleting the last node in the list.

Figure 3.6.5 shows deleting a node at the end of a circular single linked list.

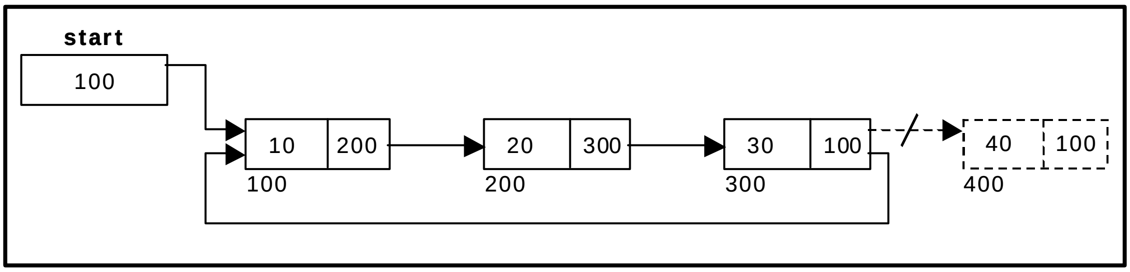


Figure 3.6.5. Deleting a node at the end.

Traversing a circular single linked list from left to right:

The following steps are followed, to traverse a list from left to right:

- If list is empty then display ‘Empty List’ message.
- If the list is not empty, follow the steps given below:

```
temp = start;
do
{
        printf("%d ", temp -> data);
        temp = temp -> next;
} while(temp != start);
```

3.7.1. Source Code for Circular Single Linked List:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

struct cslinklist
{
        int data;
        struct cslinklist *next;
};

typedef struct cslinklist node;

node *start = NULL;

int nodectr;

node* getnode()
{
        node * newnode;
        newnode = (node *) malloc(sizeof(node));
        printf("\n Enter data: ");
        scanf("%d", &newnode -> data);
        newnode -> next = NULL;
        return newnode;
}
```

```
                                        else
                                                printf("\n List is already Exist..");
                                        break;
                                case 2 :
                                        cll_insert_beg();
                                        break;
                                case 3 :
                                        cll_insert_end();
                                        break;
                                case 4 :
                                        cll_insert_mid();
                                        break;
                                case 5 :
                                        cll_delete_beg();
                                        break;
                                case 6 :
                                        cll_delete_last();
                                        break;
                                case 7 :
                                        cll_delete_mid();
                                        break;
                                case 8 :
                                        display();
                                        break;
                                case 9 :
                                        exit(0);
                        }
                        getch();
                }
        }
```

3.8. Circular Double Linked List:

A circular double linked list has both successor pointer and predecessor pointer in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list. In circular double linked list the *right* link of the right most node points back to the *start* node and *left* link of the first node points to the last node. A circular double linked list is shown in figure 3.8.1.

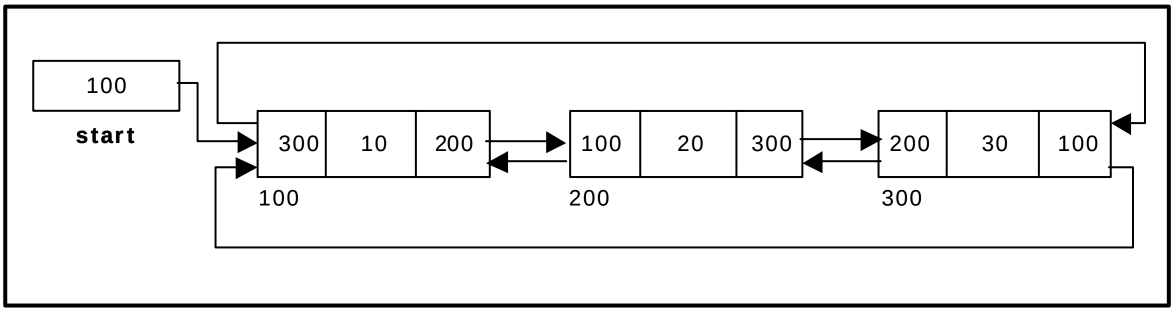


Figure 3.8.1. Circular Double Linked List

The basic operations in a circular double linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a Circular Double Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

- Get the new node using getnode().
  ```
  newnode = getnode();
  ```

- If the list is empty, then do the following
  ```
  start = newnode;
  newnode -> left = start;
  newnode ->right = start;
  ```

- If the list is not empty, follow the steps given below:
  ```
  newnode -> left =  start -> left;
  newnode -> right = start;
  start -> left->right = newnode;
  start -> left = newnode;
  ```

- Repeat the above steps 'n' times.

The function cdll_createlist(), is used to create 'n' number of nodes:

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using getnode().
  ```
  newnode=getnode();
  ```

- If the list is empty, then
  ```
  start = newnode;
  newnode -> left = start;
  newnode -> right = start;
  ```

- If the list is not empty, follow the steps given below:
  ```
  newnode -> left = start -> left;
  newnode -> right = start;
  start -> left -> right = newnode;
  start -> left = newnode;
  start = newnode;
  ```

The function cdll_insert_beg(), is used for inserting a node at the beginning. Figure 3.8.2 shows inserting a node into the circular double linked list at the beginning.

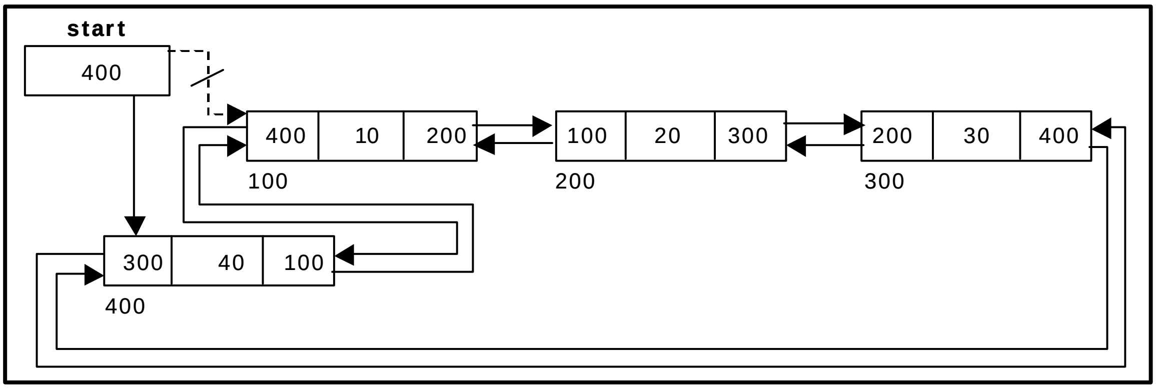


Figure 3.8.2. Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode()
  ```
  newnode=getnode();
  ```

- If the list is empty, then
  ```
  start = newnode;
  newnode -> left = start;
  newnode -> right = start;
  ```

- If the list is not empty follow the steps given below:
  ```
  newnode -> left = start -> left;
  newnode -> right = start;
  start -> left -> right = newnode;
  start -> left = newnode;
  ```

The function cdll_insert_end(), is used for inserting a node at the end. Figure 3.8.3 shows inserting a node into the circular linked list at the end.

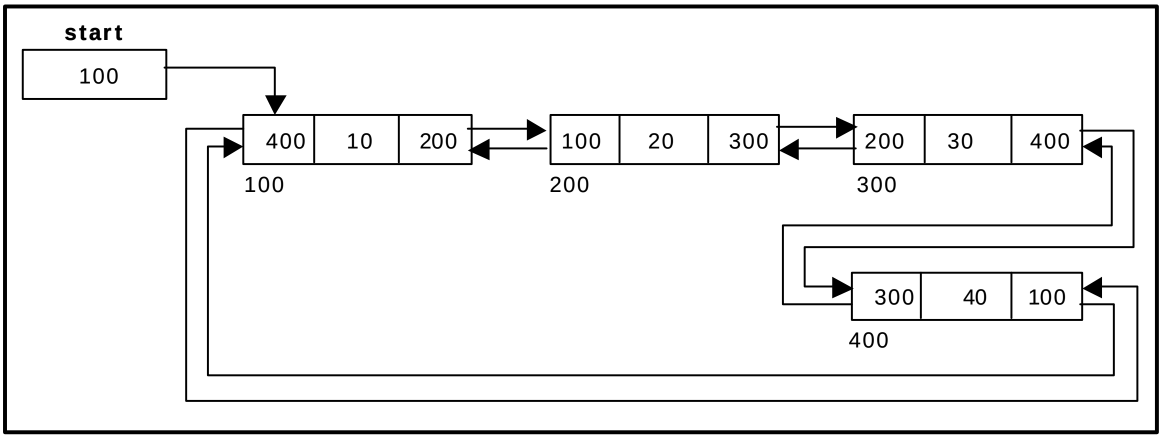


Figure 3.8.3. Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using getnode().
  ```
  newnode=getnode();
  ```

- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

- Store the starting address (which is in start pointer) in temp. Then traverse the temp pointer upto the specified position.

- After reaching the specified position, follow the steps given below:
  ```
  newnode -> left = temp;
  newnode -> right = temp -> right;
  temp -> right -> left = newnode;
  temp -> right = newnode;
  nodectr++;
  ```

The function cdll_insert_mid(), is used for inserting a node in the intermediate position. Figure 3.8.4 shows inserting a node into the circular double linked list at a specified intermediate position other than beginning and end.

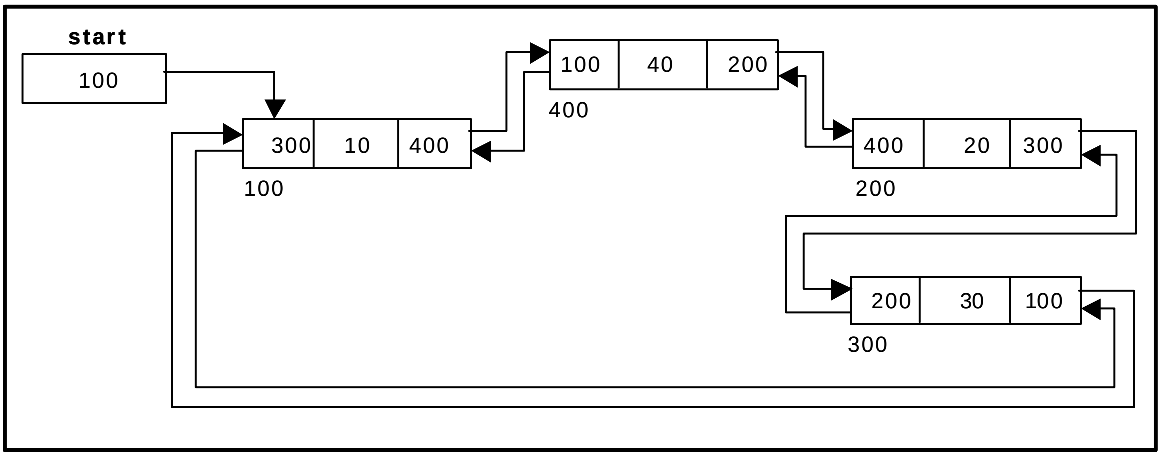


Figure 3.8.4. Inserting a node at an intermediate position

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display ‘Empty List’ message.
- If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> right;
temp -> left -> right = start;
start -> left = temp -> left;
```

The function cdll_delete_beg(), is used for deleting the first node in the list. Figure 3.8.5 shows deleting a node at the beginning of a circular double linked list.

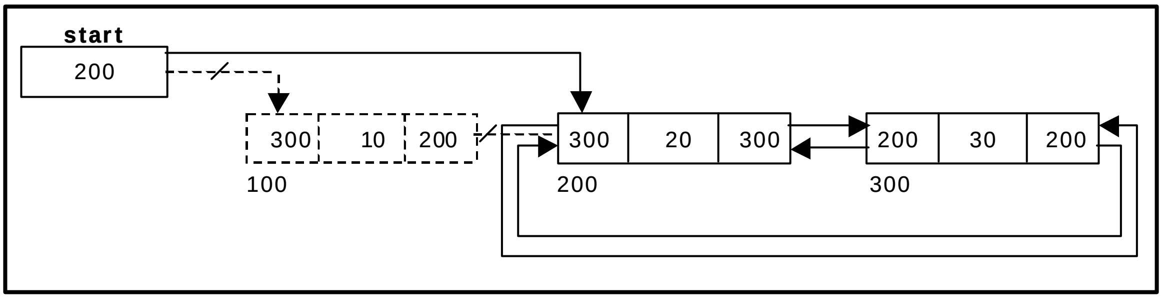


Figure 3.8.5. Deleting a node at beginning

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If list is empty then display ‘Empty List’ message
- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != start)
{
        temp = temp -> right;
}
temp -> left -> right = temp -> right;
temp -> right -> left = temp -> left;
```

The function cdll_delete_last(), is used for deleting the last node in the list. Figure 3.8.6 shows deleting a node at the end of a circular double linked list.

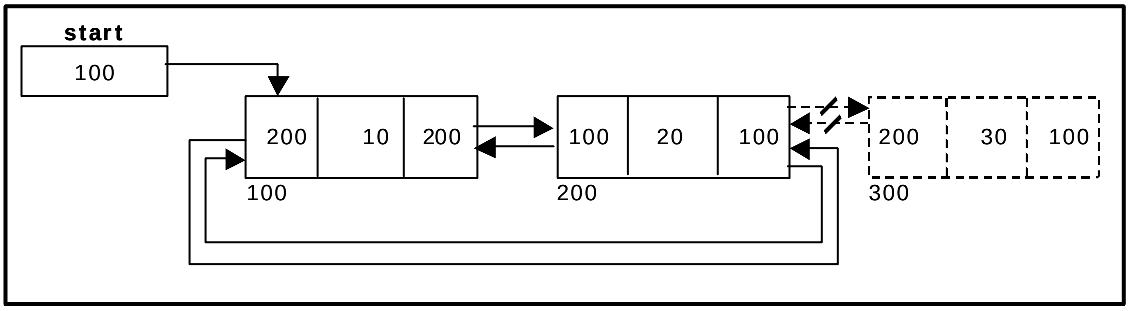


Figure 3.8.6. Deleting a node at the end

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
  - Get the position of the node to delete.
  - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
  - Then perform the following steps:

```
if(pos > 1 && pos < nodectr)
{

        temp = start;
        i = 1;
        while(i < pos)
        {
                temp = temp -> right ;
                i++;
        }
        temp -> right -> left = temp -> left;
        temp -> left -> right = temp -> right;
        free(temp);
        printf("\n node deleted..");
        nodectr--;
}
```

The function cdll_delete_mid(), is used for deleting the intermediate node in the list.

Figure 3.8.7 shows deleting a node at a specified intermediate position other than beginning and end from a circular double linked list.

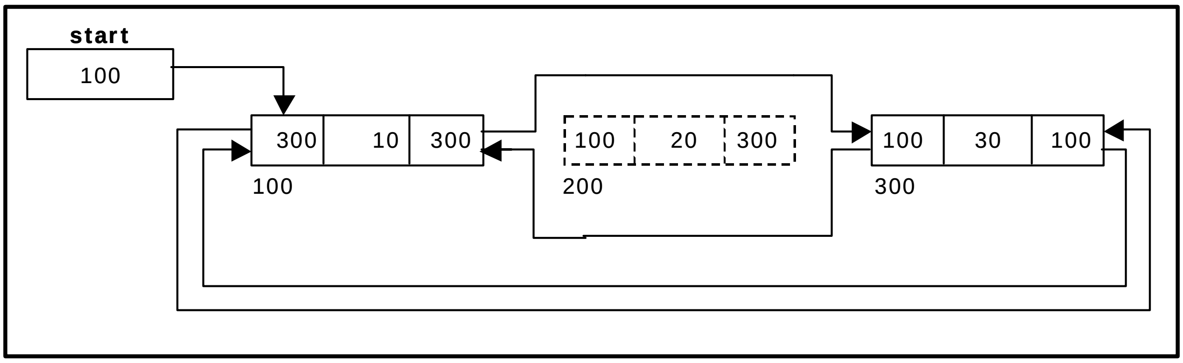


Figure 3.8.7. Deleting a node at an intermediate position

Traversing a circular double linked list from left to right:

The following steps are followed, to traverse a list from left to right:

- If list is empty then display 'Empty List' message.

- If the list is not empty, follow the steps given below:

```
temp = start;
Print  temp -> data;
temp = temp -> right;
while(temp != start)
{
        print  temp -> data;
        temp = temp -> right;
}
```

The function cdll_display_left _right(), is used for traversing from left to right.

Traversing a circular double linked list from right to left:

The following steps are followed, to traverse a list from right to left:

- If list is empty then display 'Empty List' message.

- If the list is not empty, follow the steps given below:

```
temp = start;
do
{
        temp = temp -> left;
        print temp -> data;
} while(temp != start);
```

The function cdll_display_right_left(), is used for traversing from right to left.

3.8.1. Source Code for Circular Double Linked List:

```
# include <stdio.h>
# include <stdlib.h>
# include <conio.h>
```

Chapter 4

Stack and Queue

There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of such data structures that are useful are:

- *Stack.*
- *Queue.*

Linear lists and arrays allow one to insert and delete elements at any place in the list i.e., at the beginning, at the end or in the middle.

4.1. STACK:

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

- *Push*: is the term used to insert an element into a stack.
- *Pop*: is the term used to delete an element from a stack.

"Push" is the term used to insert an element into a stack. "Pop" is the term used to delete an element from the stack.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

4.1.1. Representation of Stack:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a *stack underflow* condition.

When an element is added to a stack, the operation is performed by push(). Figure 4.1 shows the creation of a stack and addition of elements using push().

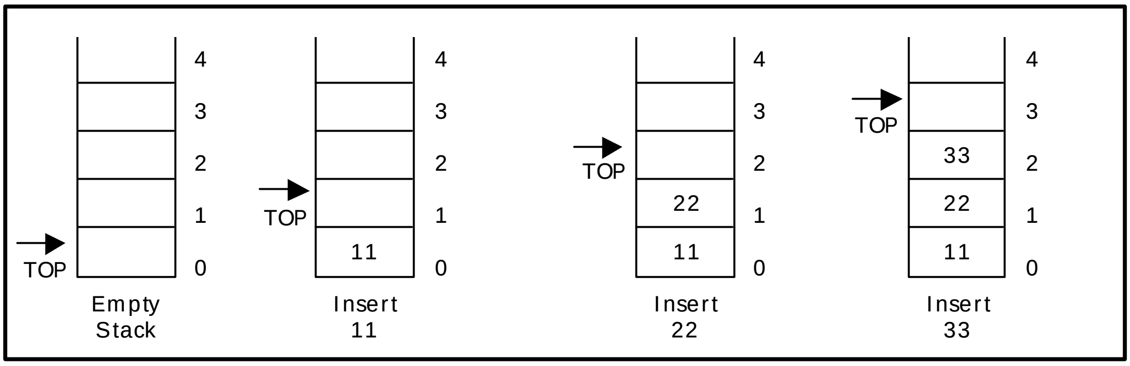


Figure 4.1. Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). Figure 4.2 shows a stack initially with three elements and shows the deletion of elements using pop().

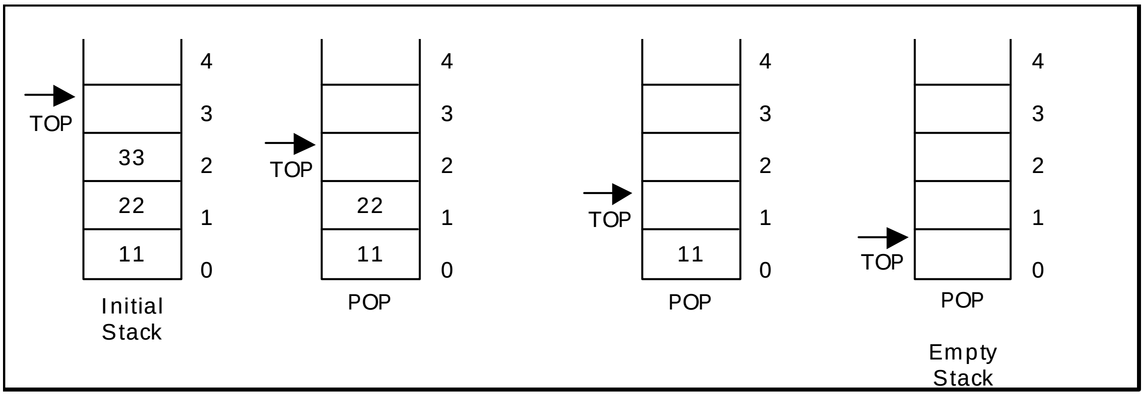


Figure 4.2. Pop operations on stack

4.1.2. Source code for stack operations, using array:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
# define MAX 6
int stack[MAX];
int top = 0;
int menu()
{
        int ch;
        clrscr();
        printf("\n … Stack operations using ARRAY... ");
        printf("\n -----------**********-------------\n");
        printf("\n 1. Push ");
        printf("\n 2. Pop ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}
void display()
{
        int i;
        if(top == 0)
        {
                printf("\n\nStack empty..");
                return;
        }
        else
        {
                printf("\n\nElements in stack:");
                for(i = 0; i < top; i++)
                        printf("\t%d", stack[i]);
        }
}

void pop()
{
        if(top == 0)
        {
                printf("\n\nStack Underflow..");
                return;
        }
        else
                printf("\n\npopped element is: %d ", stack[--top]);
}

void push()
{
        int data;
        if(top == MAX)
        {
                printf("\n\nStack Overflow..");
                return;
        }
        else
        {
                printf("\n\nEnter data: ");
                scanf("%d", &data);
                stack[top] = data;
                top = top + 1;
                printf("\n\nData Pushed into the stack");
        }
}

void main()
{
        int ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case 1:
                                push();
                                break;
                        case 2:
                                pop();
                                break;
                        case 3:
                                display();
                                break;

                        case 4:
                                exit(0);
                }
                getch();
        } while(1);
}
```

4.1.3. Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using *top* pointer. The linked stack looks as shown in figure 4.3.

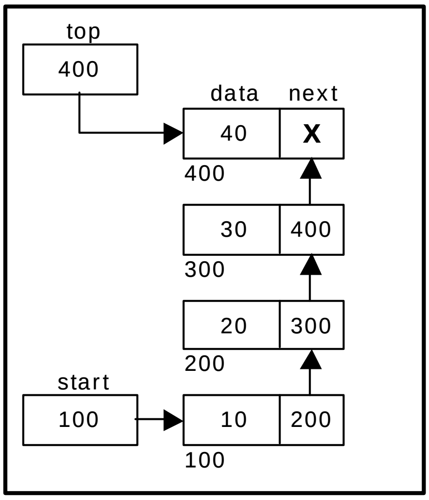


Figure 4.3. Linked stack representation

4.1.4. Source code for stack operations, using linked list:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

struct stack
{
        int data;
        struct stack *next;
};

void push();
void pop();
void display();
typedef struct stack node;
node *start=NULL;
node *top = NULL;

node* getnode()
{
        node *temp;
        temp=(node *) malloc( sizeof(node)) ;
        printf("\n Enter data ");
        scanf("%d", &temp -> data);
        temp -> next = NULL;
        return temp;
}
void push(node *newnode)
{
        node *temp;
        if( newnode == NULL )
        {
                printf("\n Stack Overflow..");
                return;
        }
        if(start == NULL)
        {
                start = newnode;
                top = newnode;
        }
        else
        {
                temp = start;
                while( temp -> next != NULL)
                        temp = temp -> next;
                temp -> next = newnode;
                top = newnode;
        }
        printf("\n\n\t Data pushed into stack");
}
void pop()
{
        node *temp;
        if(top == NULL)
        {
                printf("\n\n\t Stack underflow");
                return;
        }
        temp = start;
        if( start -> next == NULL)
        {
                printf("\n\n\t Popped element is %d ", top -> data);
                start = NULL;
                free(top);
                top = NULL;
        }
        else
        {
                while(temp -> next != top)
                {
                        temp = temp -> next;
                }
                temp -> next = NULL;
                printf("\n\n\t Popped element is %d ", top -> data);
                free(top);
                top = temp;
        }
}
void display()
{
        node *temp;
        if(top == NULL)
        {
                printf("\n\n\t\t Stack is empty ");
        }
        else
        {
                temp = start;
                printf("\n\n\n\t\t Elements in the stack: \n");
                printf("%5d  ", temp -> data);
                while(temp != top)
                {
                        temp = temp -> next;
                        printf("%5d   ", temp -> data);
                }
        }
}
```

```
char menu()
{
        char ch;
        clrscr();
        printf("\n \tStack operations using pointers.. ");
        printf("\n -----------**********-------------\n");
        printf("\n 1. Push ");
        printf("\n 2. Pop ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        ch = getche();
        return ch;
}

void main()
{
        char ch;
        node *newnode;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case '1' :
                                newnode = getnode();
                                push(newnode);
                                break;
                        case '2' :
                                pop();
                                break;
                        case '3' :
                                display();
                                break;
                        case '4':
                                return;
                }
                getch();
        } while( ch != '4' );
}
```

4.2. Algebraic Expressions:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: (A + B) * (C - D)

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).

Example: * + A B – C D

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.

Example: A B + C D - *

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression un-ambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: +, -, *, / and \$ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation (\$ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

4.3. Converting expressions using Stack:

Let us convert the expressions from one type to another. These can be done as follows:

1. Infix to postfix
2. Infix to prefix
3. Postfix to infix
4. Postfix to prefix
5. Prefix to infix
6. Prefix to postfix

4.3.1. Conversion from infix to postfix:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.

2. a) If the scanned symbol is left parenthesis, push it onto the stack.

   b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or greater than or equal*) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example 1:

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert a + b * c + (d * e + f) * g the infix expression into postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
a	a		
+	a	+	
b	a b	+	

*	a b	+ *	
c	a b c	+ *	
+	a b c * +	+	
(	a b c * +	+ (	
d	a b c * + d	+ (	
*	a b c * + d	+ (*	
e	a b c * + d e	+ (*	
+	a b c * + d e *	+ (+	
f	a b c * + d e * f	+ (+	
)	a b c * + d e * f +	+	
*	a b c * + d e * f +	+ *	
g	a b c * + d e * f + g	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 4:

Convert the following infix expression A + (B * C – (D / E ↑ F) * G) * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	

(	A	+ (	
B	A B	+ (	
*	A B	+ (*	
C	A B C	+ (*	
-	A B C *	+ (-	
(	A B C *	+ (- (	
D	A B C * D	+ (- (	
/	A B C * D	+ (- (/	
E	A B C * D E	+ (- (/	
↑	A B C * D E	+ (- (/ ↑	
F	A B C * D E F	+ (- (/ ↑	
)	A B C * D E F ↑ /	+ (-	
*	A B C * D E F ↑ /	+ (- *	
G	A B C * D E F ↑ / G	+ (- *	
)	A B C * D E F ↑ / G * -	+	
*	A B C * D E F ↑ / G * -	+ *	
H	A B C * D E F ↑ / G * - H	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

4.3.2. Program to convert an infix to postfix expression:

```
# include <string.h>

char postfix[50];
char infix[50];
char opstack[50];                /* operator stack */
int i, j, top = 0;

int lesspriority(char op, char op_at_stack)
{
        int k;
        int pv1;                 /* priority value of op */
        int pv2;                 /* priority value of op_at_stack */
        char operators[] = {'+', '-', '*', '/', '%', '^', '(' };
        int priority_value[] = {0,0,1,1,2,3,4};
        if( op_at_stack == '(' )
                return 0;
        for(k = 0; k < 6; k ++)
        {
                if(op == operators[k])
                        pv1 = priority_value[k];
        }
        for(k = 0; k < 6; k ++)
        {
                if(op_at_stack == operators[k])
                        pv2 = priority_value[k];
        }
        if(pv1 < pv2)
                return 1;
        else
                return 0;
}
```

```
/* before pushing the operator
'op' into the stack check priority
of op  with top of opstack if less
then pop the operator from stack
then push into postfix string else
push op onto stack itself  */
```

```
void push(char op)        /* op - operator  */
{
        if(top == 0)
        {
                opstack[top] = op;
                top++;
        }
        else
        {
                if(op != '(' )
                {
                        while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
                        {
                                        postfix[j] = opstack[--top];
                                        j++;
                        }
                }
                opstack[top] = op;        /* pushing onto stack  */
                top++;
        }
}

pop()
{
        while(opstack[--top] != '(' )             /*  pop until '(' comes   */
        {
                postfix[j] = opstack[top];
                j++;
        }
}

void main()
{
        char ch;
        clrscr();
        printf("\n Enter Infix Expression  : ");
        gets(infix);
        while( (ch=infix[i++]) != ‘\0’)
        {
                switch(ch)
                {
                        case ' ' : break;
                        case '(' :
                        case '+' :
                        case '-' :
                        case '*' :
                        case '/' :
                        case '^' :
                        case '%' :
                                push(ch);               /* check priority and push  */
                                break;
                        case ')' :
                                pop();
                                break;
                        default :
                                postfix[j] = ch;
                                j++;
                }
        }
        while(top >= 0)
        {
                postfix[j] =  opstack[--top];
                j++;
        }
        postfix[j] = '\0';
        printf("\n Infix  Expression    :  %s ", infix);
        printf("\n Postfix Expression  :  %s ", postfix);
        getch();
}
```

4.3.3. Conversion from infix to prefix:

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that traverse the expression from right to left and the operator is placed before the operands rather than after them. The prefix form of a complex expression is not the mirror image of the postfix form.

Example 1:

Convert the infix expression A + B - C into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
C	C		
-	C	-	
B	B C	-	
+	B C	- +	
A	A B C	- +	
End of string	- + A B C	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert the infix expression (A + B) * (C - D) into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
D	D	)	
-	D	) -	
C	C D	) -	
(	- C D		
*	- C D	*	
)	- C D	*)	
B	B - C D	*)	
+	B - C D	*) +	
A	A B - C D	*) +	
(	+ A B – C D	*	
End of string	* + A B – C D	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the infix expression A ↑ B * C – D + E / F / (G + H) into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
H	H	)	
+	H	) +	
G	G H	) +	
(	+ G H		
/	+ G H	/	
F	F + G H	/	
/	F + G H	/ /	
E	E F + G H	/ /	
+	/ / E F + G H	+	
D	D / / E F + G H	+	
-	D / / E F + G H	+ -	
C	C D / / E F + G H	+ -	
*	C D / / E F + G H	+ - *	
B	B C D / / E F + G H	+ - *	
↑	B C D / / E F + G H	+ - * ↑	
A	A B C D / / E F + G H	+ - * ↑	
End of string	+ - * ↑ A B C D / / E F + G H	The input is now empty. Pop the output symbols from the stack until it is empty.	

4.3.4. Program to convert an infix to prefix expression:

```
# include <conio.h>
# include <string.h>

char prefix[50];
char infix[50];
char opstack[50];                    /* operator stack  */
int j, top = 0;

void insert_beg(char ch)
{
        int k;
        if(j == 0)
                prefix[0] =  ch;
        else
        {
                for(k = j + 1; k > 0; k--)
                        prefix[k] = prefix[k - 1];
                prefix[0] = ch;
        }
        j++;
}
```

```
int lesspriority(char op, char op_at_stack)
{
        int k;
        int pv1;                        /* priority value of  op  */
        int pv2;                        /* priority value of op_at_stack  */
        char operators[] = {'+', '-', '*', '/', '%', '^', ')'};
        int priority_value[] = {0, 0, 1, 1, 2, 3, 4};
        if(op_at_stack == ')' )
                return 0;
        for(k = 0; k < 6; k ++)
        {
                if(op == operators[k])
                        pv1 = priority_value[k];
        }
        for(k = 0; k < 6; k ++)
        {
                if( op_at_stack == operators[k] )
                        pv2 = priority_value[k];
        }
        if(pv1 < pv2)
                return 1;
        else
                return 0;
}

void push(char op)              /* op – operator  */
{
        if(top == 0)
        {
                opstack[top] = op;
                top++;
        }
        else
        {
                if(op != ')')
                {
                        /* before pushing the operator 'op' into the stack check priority of op
                with top of operator stack if less pop the operator from stack then push into
                postfix string else push op onto stack itself  */

                        while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
                        {
                                        insert_beg(opstack[--top]);
                        }
                }
                opstack[top] = op;                      /* pushing onto stack   */
                top++;
        }
}

void pop()
{
        while(opstack[--top] != ')')                    /*  pop until ')' comes;  */
                insert_beg(opstack[top]);
}


void main()
{
        char ch;
        int l, i = 0;
        clrscr();
        printf("\n Enter Infix Expression  : ");
        gets(infix);
        l = strlen(infix);
        while(l > 0)
        {
                ch = infix[--l];
                switch(ch)
                {
                        case ' ' : break;
                        case ')' :
                        case '+' :
                        case '-' :
                        case '*' :
                        case '/' :
                        case '^' :
                        case '%' :
                                push(ch);               /* check priority and push  */
                                break;
                        case '(' :
                                pop();
                                break;
                        default :
                                insert_beg(ch);
                }
        }
        while( top > 0 )
        {
                insert_beg( opstack[--top] );
                j++;
        }
        prefix[j] = '\0';
        printf("\n Infix Expression    :  %s ", infix);
        printf("\n Prefix Expression  :  %s ", prefix);
        getch();
}
```

4.3.5. Conversion from postfix to infix:

Procedure to convert postfix expression to infix expression is as follows:

1. Scan the postfix expression from left to right.

2. If the scanned symbol is an operand, then push it onto the stack.

3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.

4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following postfix expression A B C * D E F ^ / G * - H * + into its equivalent infix expression.

Symbol	Stack					Remarks
A	A					Push A
B	A	B				Push B
C	A	B	C			Push C
*	A	(B*C)				Pop two operands and place the operator in between the operands and push the string.
D	A	(B*C)	D			Push D
E	A	(B*C)	D	E		Push E
F	A	(B*C)	D	E	F	Push F
^	A	(B*C)	D	(E^F)		Pop two operands and place the operator in between the operands and push the string.
/	A	(B*C)	(D/(E^F))			Pop two operands and place the operator in between the operands and push the string.
G	A	(B*C)	(D/(E^F))	G		Push G
*	A	(B*C)	((D/(E^F))*G)			Pop two operands and place the operator in between the operands and push the string.
-	A	((B*C) – ((D/(E^F))*G))				Pop two operands and place the operator in between the operands and push the string.
H	A	((B*C) – ((D/(E^F))*G))	H			Push H
*	A	(((B*C) – ((D/(E^F))*G)) * H)				Pop two operands and place the operator in between the operands and push the string.
+	(A + (((B*C) – ((D/(E^F))*G)) * H))					
End of string	The input is now empty. The string formed is infix.					

4.3.6. Program to convert postfix to infix expression:

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
# define MAX 100

void pop (char*);
void push(char*);

char stack[MAX] [MAX];
int top = -1;
```

```
void main()
{
        char s[MAX], str1[MAX], str2[MAX], str[MAX];
        char s1[2],temp[2];
        int i=0;
        clrscr( ) ;
        printf("\Enter the postfix expression; ");
        gets(s);
        while (s[i]!='\0')
        {
                if(s[i] == ' ' )                        /*skip whitespace, if any*/
                        i++;
                if (s[i] == '^' || s[i] == '*'|| s[i] == '-' || s[i] == '+' || s[i] == '/')
                {
                        pop(str1);
                        pop(str2);
                        temp[0] ='(';
                        temp[1] ='\0';
                        strcpy(str, temp);
                        strcat(str, str2);
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcat(str,temp);
                        strcat(str, str1);
                        temp[0] =')';
                        temp[1] ='\0';
                        strcat(str,temp);
                        push(str);
                }
                else
                {
                        temp[0]=s[i];
                        temp[1]='\0';
                        strcpy(s1, temp);
                        push(s1);
                }
                i++;
        }
        printf("\nThe Infix expression is: %s", stack[0]);

}

void pop(char *a1)
{
        strcpy(a1,stack[top]);
        top--;
}

void push (char*str)
{
        if(top == MAX - 1)
                printf("\nstack is full");
        else
        {
                top++;
                strcpy(stack[top], str);
        }
}
```

4.3.7. Conversion from postfix to prefix:

Procedure to convert postfix expression to prefix expression is as follows:

1. Scan the postfix expression from left to right.

2. If the scanned symbol is an operand, then push it onto the stack.

3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in front of the operands and push it onto the stack.

5. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following postfix expression A B C * D E F ^ / G * - H * + into its equivalent prefix expression.

Symbol	Stack					Remarks
A	A					Push A
B	A	B				Push B
C	A	B	C			Push C
*	A	*BC				Pop two operands and place the operator in front the operands and push the string.
D	A	*BC	D			Push D
E	A	*BC	D	E		Push E
F	A	*BC	D	E	F	Push F
^	A	*BC	D	^EF		Pop two operands and place the operator in front the operands and push the string.
/	A	*BC	/D^EF			Pop two operands and place the operator in front the operands and push the string.
G	A	*BC	/D^EF	G		Push G
*	A	*BC	*/D^EFG			Pop two operands and place the operator in front the operands and push the string.
-	A	- *BC*/D^EFG				Pop two operands and place the operator in front the operands and push the string.
H	A	- *BC*/D^EFG	H			Push H
*	A	*- *BC*/D^EFGH				Pop two operands and place the operator in front the operands and push the string.
+	+A*-*BC*/D^EFGH					
End of string	The input is now empty. The string formed is prefix.					

4.3.8. Program to convert postfix to prefix expression:

```
# include <conio.h>
# include <string.h>

#define MAX 100
void pop (char *a1);
void push(char *str);
char stack[MAX][MAX];
int top =-1;

main()
{
        char s[MAX], str1[MAX], str2[MAX], str[MAX];
        char s1[2], temp[2];
        int i = 0;
        clrscr();
        printf("Enter the postfix expression; ");
        gets (s);
        while(s[i]!='\0')
        {
        /*skip whitespace, if any */
                if (s[i] == ' ')
                        i++;
                if(s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i]== '+' || s[i] == '/')
                {
                        pop (str1);
                        pop (str2);
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcpy (str, temp);
                        strcat(str, str2);
                        strcat(str, str1);
                        push(str);
                }
                else
                {
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcpy (s1, temp);
                        push (s1);
                }
                i++;
        }
        printf("\n The prefix expression is: %s", stack[0]);
}

void pop(char*a1)
{
        if(top == -1)
        {
                printf("\nStack is empty");
                return ;
        }
        else
        {
                strcpy (a1, stack[top]);
                top--;
        }
}
```

```
void push (char *str)
{
        if(top == MAX - 1)
                printf("\nstack is full");
        else
        {
                top++;
                strcpy(stack[top], str);
        }
}
```

4.3.9. Conversion from prefix to infix:

Procedure to convert prefix expression to infix expression is as follows:

1. Scan the prefix expression from right to left (reverse order).
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following prefix expression + A * - * B C * / D ^ E F G H into its equivalent infix expression.

Symbol	Stack	Remarks
H	H	Push H
G	H G	Push G
F	H G F	Push F
E	H G F E	Push E
^	H G (E^F)	Pop two operands and place the operator in between the operands and push the string.
D	H G (E^F) D	Push D
/	H G (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
*	H ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
C	H ((D/(E^F))*G) C	Push C
B	H ((D/(E^F))*G) C B	Push B
*	H ((D/(E^F))*G) (B*C)	Pop two operands and place the operator in front the operands and push the string.
-	H ((B*C)-((D/(E^F))*G))	Pop two operands and place the operator in front the operands and push the

		string.
*	(((B*C)-((D/(E^F))*G))*H)	Pop two operands and place the operator in front the operands and push the string.
A	(((B*C)-((D/(E^F))*G))*H) A	Push A
+	(A+(((B*C)-((D/(E^F))*G))*H))	Pop two operands and place the operator in front the operands and push the string.
End of string	The input is now empty. The string formed is infix.	

4.3.10. Program to convert prefix to infix expression:

```
# include <string.h>
# define MAX 100

void pop (char*);
void push(char*);
char stack[MAX] [MAX];
int top = -1;

void main()
{
        char s[MAX], str1[MAX], str2[MAX], str[MAX];
        char s1[2],temp[2];
        int i=0;
        clrscr( ) ;
        printf("\Enter the prefix expression; ");
        gets(s);
        strrev(s);
        while (s[i]!='\0')
        {
                /*skip whitespace, if any*/
                if(s[i] == ' ' )
                        i++;
                if (s[i] == '^' || s[i] == '*'|| s[i] == '-' || s[i] == '+' || s[i] == '/')
                {
                        pop(str1);
                        pop(str2);
                        temp[0] ='(';
                        temp[1] ='\0';
                        strcpy(str, temp);
                        strcat(str, str1);
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcat(str,temp);
                        strcat(str, str2);
                        temp[0] =')';
                        temp[1] ='\0';
                        strcat(str,temp);
                        push(str);
                }
                else
                {
                        temp[0]=s[i];
                        temp[1]='\0';
                        strcpy(s1, temp);
                        push(s1);
                }
                i++;
        }
        printf("\nThe infix expression is: %s", stack[0]);
}

void pop(char *a1)
{
        strcpy(a1,stack[top]);
        top--;
}

void push (char*str)
{
        if(top == MAX - 1)
                printf("\nstack is full");
        else
        {
                top++;
                strcpy(stack[top], str);
        }
}
```

4.3.11. Conversion from prefix to postfix:

Procedure to convert prefix expression to postfix expression is as follows:

1. Scan the prefix expression from right to left (reverse order).

2. If the scanned symbol is an operand, then push it onto the stack.

3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator after the operands and push it onto the stack.

4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following prefix expression + A * - * B C * / D ^ E F G H into its equivalent postfix expression.

Symbol	Stack	Remarks
H	H	Push H
G	H G	Push G
F	H G F	Push F
E	H G F E	Push E
^	H G EF^	Pop two operands and place the operator after the operands and push the string.
D	H G EF^ D	Push D

/	H G DEF^/	Pop two operands and place the operator after the operands and push the string.
*	H DEF^/G*	Pop two operands and place the operator after the operands and push the string.
C	H DEF^/G* C	Push C
B	H DEF^/G* C B	Push B
*	H DEF^/G* BC*	Pop two operands and place the operator after the operands and push the string.
-	H BC*DEF^/G*-	Pop two operands and place the operator after the operands and push the string.
*	BC*DEF^/G*-H*	Pop two operands and place the operator after the operands and push the string.
A	BC*DEF^/G*-H* A	Push A
+	ABC*DEF^/G*-H*+	Pop two operands and place the operator after the operands and push the string.
End of string	The input is now empty. The string formed is postfix.	

4.3.12. Program to convert prefix to postfix expression:

```
# include <stdio.h>
# include <conio.h>
# include <string.h>

#define MAX 100

void pop (char *a1);
void push(char *str);
char stack[MAX][MAX];
int top =-1;

void main()
{
        char s[MAX], str1[MAX], str2[MAX], str[MAX];
        char s1[2], temp[2];
        int i = 0;
        clrscr();
        printf("Enter the prefix expression; ");
        gets (s);
        strrev(s);
        while(s[i]!='\0')
        {
                if (s[i] == ' ')                        /*skip whitespace, if any */
                        i++;
                if(s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i]== '+' || s[i] == '/')
                {
                        pop (str1);
                        pop (str2);
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcat(str1,str2);
                        strcat (str1, temp);
                        strcpy(str, str1);
                        push(str);
                }
                else
                {
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcpy (s1, temp);
                        push (s1);
                }
                i++;
        }
        printf("\nThe postfix expression is: %s", stack[0]);
}
void pop(char*a1)
{
        if(top == -1)
        {
                printf("\nStack is empty");
                return ;
        }
        else
        {
                strcpy (a1, stack[top]);
                top--;
        }
}
void push (char *str)
{
        if(top == MAX - 1)
                printf("\nstack is full");
        else
        {
                top++;
                strcpy(stack[top], str);
        }
}
```

4.4. Evaluation of postfix expression:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed

8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example 2:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

4.4.1. Program to evaluate a postfix expression:

```
# include <conio.h>
# include <math.h>
# define MAX 20

int isoperator(char ch)
{
        if(ch == '+' || ch == '-' ||  ch == '*' || ch == '/' ||  ch == '^')
                return 1;
        else
                return 0;
}
```

```
void main(void)
{
        char postfix[MAX];
        int val;
        char ch;
        int i = 0, top = 0;
        float val_stack[MAX], val1, val2, res;
        clrscr();
        printf("\n Enter a postfix expression: ");
        scanf("%s", postfix);
        while((ch = postfix[i]) != '\0')
        {
                if(isoperator(ch) == 1)
                {
                        val2 = val_stack[--top];
                        val1 = val_stack[--top];
                        switch(ch)
                        {
                                case '+':
                                        res = val1 + val2;
                                        break;
                                case '-':
                                        res = val1 - val2;
                                        break;
                                case '*':
                                        res = val1 * val2;
                                        break;
                                case '/':
                                        res = val1 / val2;
                                        break;
                                case '^':
                                        res = pow(val1, val2);
                                        break;
                        }
                        val_stack[top] = res;
                }
                else
                        val_stack[top] = ch-48;  /*convert character digit to integer digit */
                top++;
                i++;
        }
        printf("\n Values of %s is : %f ",postfix, val_stack[0] );
        getch();
}
```

4.5. Applications of stacks:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.

2. Stack is used to evaluate a postfix expression.

3. Stack is used to convert an infix expression into postfix/prefix form.

4. In recursion, all intermediate arguments and return values are stored on the processor’s stack.

5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

4.6. Queue:

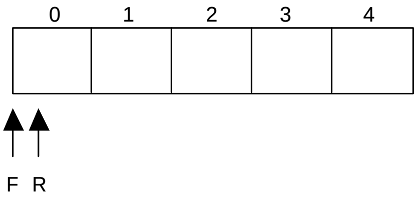
A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a “FIFO” or “First-in-first-out” list.

The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations on queues:

- *enqueue*: which inserts an element at the end of the queue.
- *dequeue*: which deletes an element at the start of the queue.

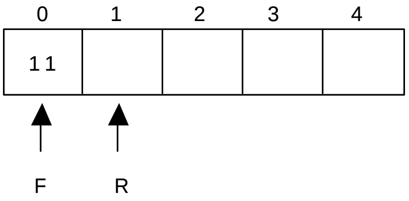
4.6.1. Representation of Queue:

Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



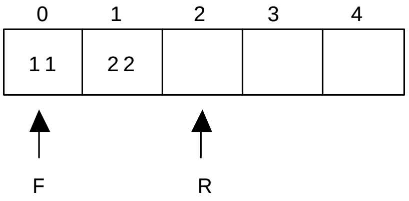
Queue Empty
FRONT = REAR = 0

Now, insert 11 to the queue. Then queue status will be:



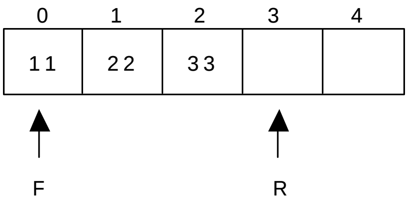
REAR = REAR + 1 = 1
FRONT = 0

Next, insert 22 to the queue. Then the queue status is:



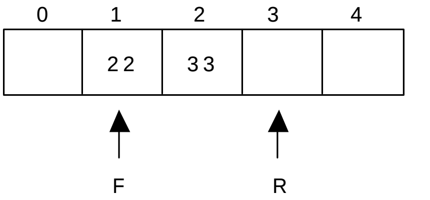
REAR = REAR + 1 = 2
FRONT = 0

Again insert another element 33 to the queue. The status of the queue is:



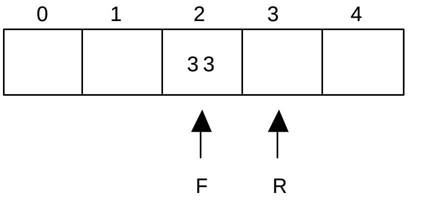
REAR = REAR + 1 = 3
FRONT = 0

Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



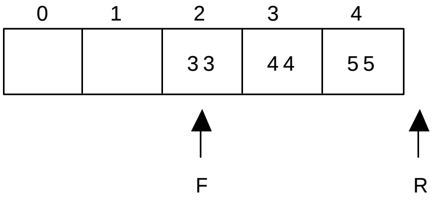
REAR = 3
FRONT = FRONT + 1 = 1

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



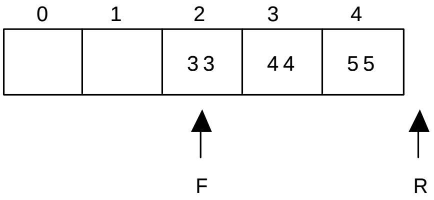
REAR = 3
FRONT = FRONT + 1 = 2

Now, insert new elements 44 and 55 into the queue. The queue status is:



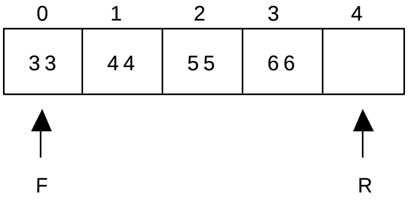
REAR = 5
FRONT = 2

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



REAR = 5
FRONT = 2

Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To over come this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



REAR = 4
FRONT = 0

This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

4.6.2. Source code for Queue operations using array:

In order to create a queue we require a one dimensional array Q(1:n) and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus, front = rear if and only if there are no elements in the queue. The initial condition then is front = rear = 0. The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

1. insertQ(): inserts an element at the end of queue Q.
2. deleteQ(): deletes the first element of Q.
3. displayQ(): displays the elements in the queue.

```
# include <conio.h>
# define MAX 6
int Q[MAX];
int front, rear;

void insertQ()
{
        int data;
        if(rear == MAX)
        {
                printf("\n Linear Queue is full");
                return;
        }
        else
        {
                printf("\n Enter data: ");
                scanf("%d", &data);
                Q[rear] = data;
                rear++;
                printf("\n Data Inserted in the Queue ");
        }
}
void deleteQ()
{
        if(rear == front)
        {
                printf("\n\n Queue is Empty..");
                return;
        }
        else
        {
                printf("\n Deleted element from Queue is %d", Q[front]);
                front++;
        }
}
void displayQ()
{
        int i;
        if(front == rear)
        {
                printf("\n\n\t Queue is Empty");
                return;
        }
        else
        {
                printf("\n Elements in Queue are: ");
                for(i = front; i < rear; i++)
                {
                        printf("%d\t", Q[i]);
                }
        }
}
int menu()
{
        int ch;
        clrscr();
        printf("\n \tQueue operations using ARRAY..");
        printf("\n ----------**********-------------\n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}
void main()
{
        int ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case 1:
                                insertQ();
                                break;
                        case 2:
                                deleteQ();
                                break;
                        case 3:
                                displayQ();
                                break;
                        case 4:
                                return;
                }
                getch();
        } while(1);
}
```

4.6.3. Linked List Implementation of Queue:

We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers *front* and *rear* for our linked queue implementation.

The linked queue looks as shown in figure 4.4:

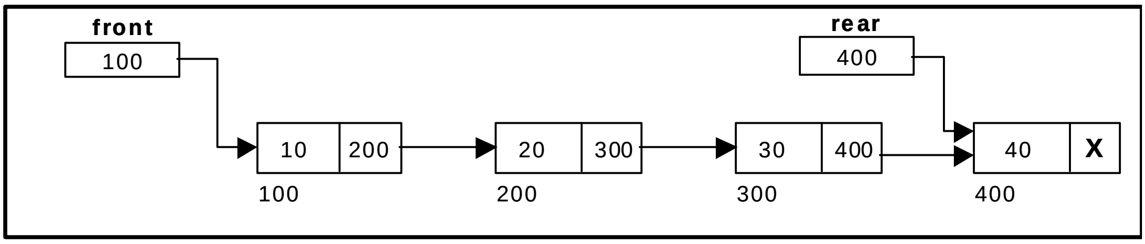


Figure 4.4. Linked Queue representation

4.6.4. Source code for queue operations using linked list:

```
# include <stdlib.h>
# include <conio.h>

struct queue
{
        int data;
        struct queue *next;
};
typedef struct queue node;
node *front = NULL;
node *rear =  NULL;

node* getnode()
{
        node *temp;
        temp = (node *) malloc(sizeof(node)) ;
        printf("\n Enter data ");
        scanf("%d", &temp -> data);
        temp -> next = NULL;
        return temp;
}
void insertQ()
{
        node *newnode;
        newnode = getnode();
        if(newnode == NULL)
        {
                printf("\n Queue Full");
                return;
        }
        if(front == NULL)
        {
                front = newnode;
                rear = newnode;
        }
        else
        {
                rear -> next = newnode;
                rear = newnode;
        }
        printf("\n\n\t Data Inserted into the Queue..");
}
void deleteQ()
{
        node *temp;
        if(front == NULL)
        {
                printf("\n\n\t Empty Queue..");
                return;
        }
        temp = front;
        front = front -> next;
        printf("\n\n\t Deleted element from queue is %d ", temp -> data);
        free(temp);
}
```

```
void displayQ()
{
        node *temp;
        if(front == NULL)
        {
                printf("\n\n\t\t Empty Queue ");
        }
        else
        {
                temp = front;
                printf("\n\n\n\t\t Elements in the Queue are: ");
                while(temp != NULL )
                {
                        printf("%5d  ", temp -> data);
                        temp = temp -> next;
                }
        }
}

char menu()
{
        char ch;
        clrscr();
        printf("\n \t..Queue operations using pointers.. ");
        printf("\n\t -----------**********-------------\n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        ch = getche();
        return ch;
}

void main()
{
        char ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case '1' :
                                insertQ();
                                break;
                        case '2' :
                                deleteQ();
                                break;
                        case '3' :
                                displayQ();
                                break;
                        case '4':
                                return;
                }
                getch();
        } while(ch != '4');
}
```

4.7. Applications of Queue:

1. It is used to schedule the jobs to be processed by the CPU.

2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.

3. Breadth first search uses a queue data structure to find an element from a graph.

4.8. Circular Queue:

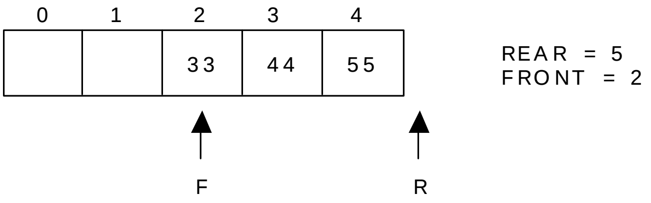
A more efficient queue representation is obtained by regarding the array Q[MAX] as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

There are two problems associated with linear queue. They are:

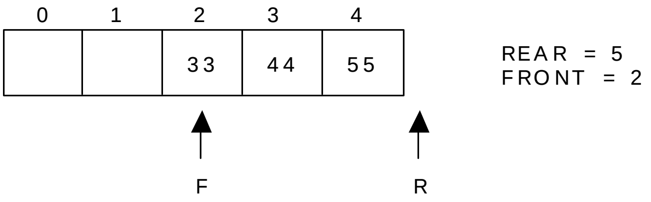
- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.

- Signaling queue full: even if the queue is having vacant position.

For example, let us consider a linear queue status as follows:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:

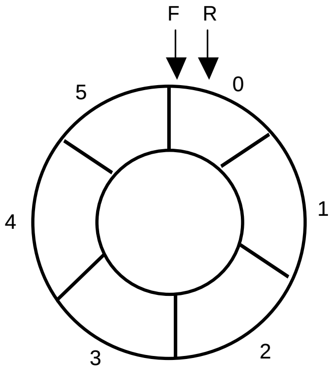


This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a **circular queue.**

In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the slots at the beginning of the circular queue are empty.

4.8.1. Representation of Circular Queue:

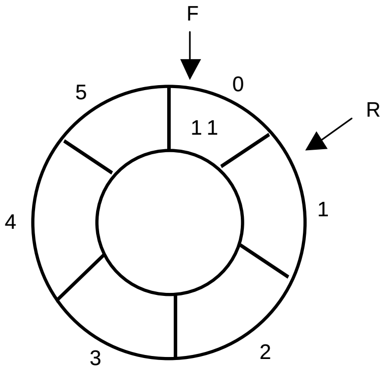
Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



Circular Queue

Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

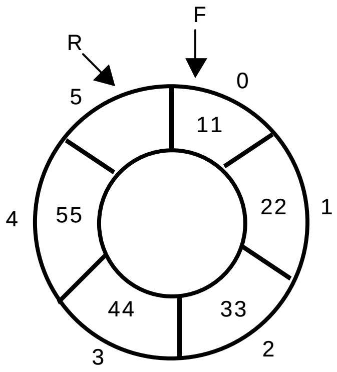
Now, insert 11 to the circular queue. Then circular queue status will be:



Circular Queue

FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1

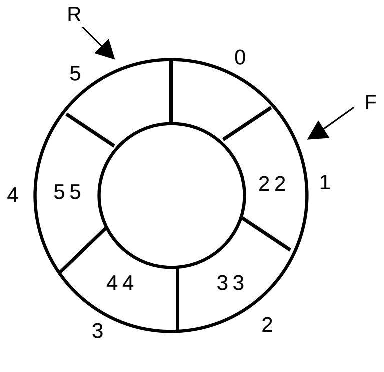
Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



Circular Queue

FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

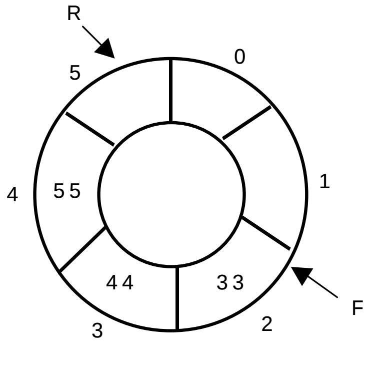
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



Circular Queue

FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4

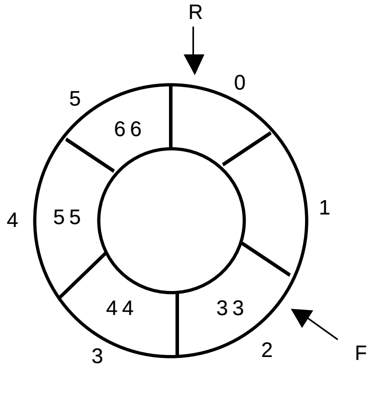
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circular Queue

FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3

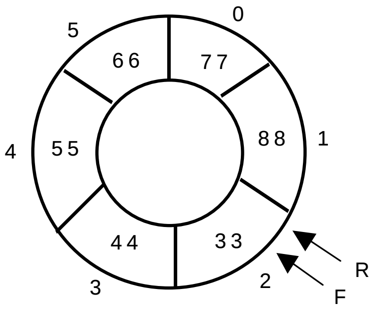
Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circular Queue

FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



Circular Queue

FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

4.8.2. Source code for Circular Queue operations, using array:

```
# include <stdio.h>
# include <conio.h>
# define MAX 6

int CQ[MAX];
int front = 0;
int rear = 0;
int count = 0;

void insertCQ()
{
        int data;
        if(count == MAX)
        {
                printf("\n Circular Queue is Full");
        }
        else
        {
                printf("\n Enter data: ");
                scanf("%d", &data);
                CQ[rear] = data;
                rear = (rear + 1) % MAX;
                count ++;
                printf("\n Data Inserted in the Circular Queue ");
        }
}

void deleteCQ()
{
        if(count == 0)
        {
                printf("\n\nCircular Queue is Empty..");
        }
        else
        {
                printf("\n Deleted element from Circular Queue is %d ", CQ[front]);
                front = (front + 1) % MAX;
                count --;
        }
}
```

```
void displayCQ()
{
        int i, j;
        if(count == 0)
        {
                printf("\n\n\t Circular Queue is Empty ");
        }
        else
        {
                printf("\n Elements in Circular Queue are: ");
                j = count;
                for(i = front; j != 0; j--)
                {
                        printf("%d\t", CQ[i]);
                        i = (i + 1) % MAX;
                }
        }
}

int menu()
{
        int ch;
        clrscr();
        printf("\n \t Circular Queue Operations using ARRAY..");
        printf("\n -----------**********-------------\n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter Your Choice: ");
        scanf("%d", &ch);
        return ch;
}

void main()
{
        int ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case 1:
                                insertCQ();
                                break;
                        case 2:
                                deleteCQ();
                                break;
                        case 3:
                                displayCQ();
                                break;
                        case 4:
                                return;
                        default:
                                printf("\n Invalid Choice ");
                }
                getch();
        } while(1);
}
```

4.9. Deque:

In the preceding section we saw that a queue in which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a *deque*. The word *deque* is an acronym derived from *double-ended queue*. Figure 4.5 shows the representation of a deque.

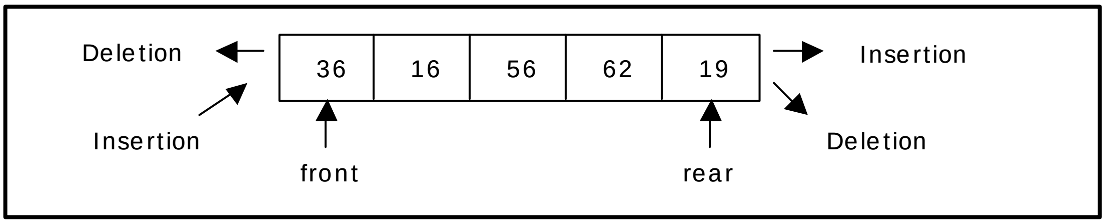


Figure 4.5. Representation of a deque.

A deque provides four operations. Figure 4.6 shows the basic operations on a deque.

- enqueue_front: insert an element at front.
- dequeue_front: delete an element at front.
- enqueue_rear: insert element at rear.
- dequeue_rear: delete element at rear.

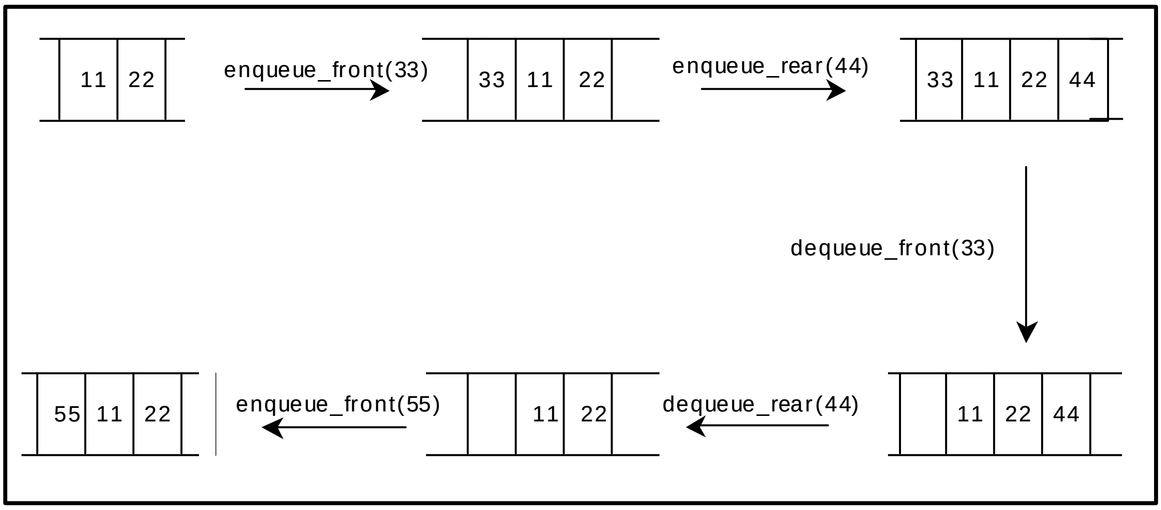


Figure 4.6. Basic operations on deque

There are two variations of deque. They are:

- Input restricted deque (IRD)
- Output restricted deque (ORD)

An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.

An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.

4.10. Priority Queue:

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.

2. two elements with same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort.

Exercises

1. What is a linear data structure? Give two examples of linear data structures.

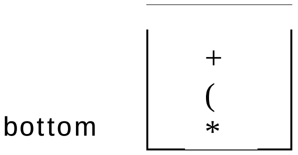
2. Is it possible to have two designs for the same data structure that provide the same functionality but are implemented differently?

3. What is the difference between the logical representation of a data structure and the physical representation?

4. Transform the following infix expressions to reverse polish notation:
   a) A ↑ B * C – D + E / F / (G + H)
   b) ((A + B) * C – (D – E)) ↑ (F + G)
   c) A – B / (C * D ↑ E)
   d) (a + b ↑ c ↑ d) * (e + f / d))
   f) 3 – 6 * 7 + 2 / 4 * 5 – 8
   g) (A – B) / ((D + E) * F)
   h) ((A + B) / D) ↑ ((E – F) * G)

5. Evaluate the following postfix expressions:
   a) P_1: 5, 3, +, 2, *, 6, 9, 7, -, /, -
   b) P_2: 3, 5, +, 6, 4, -, *, 4, 1, -, 2, ↑, +
   c) P_3 : 3, 1, +, 2, ↑, 7, 4, -, 2, *, +, 5, -

6. Consider the usual algorithm to convert an infix expression to a postfix expression. Suppose that you have read 10 input characters during a conversion and that the stack now contains these symbols:



   Now, suppose that you read and process the 11th symbol of the input. Draw the stack for the case where the 11th symbol is:
   A. A number:
   B. A left parenthesis:
   C. A right parenthesis:
   D. A minus sign:
   E. A division sign:

7. Write a program using stack for parenthesis matching. Explain what modifications would be needed to make the parenthesis matching algorithm check expressions with different kinds of parentheses such as (), [] and {}'s.

8. Evaluate the following prefix expressions:
   a) + * 2 + / 14 2 5 1
   b) - * 6 3 – 4 1
   c) + + 2 6 + - 13 2 4

9. Convert the following infix expressions to prefix notation:
   a) ((A + 2) * (B + 4)) -1
   b) Z – ((((X + 1) * 2) – 5) / Y)
   c) ((C * 2) + 1) / (A + B)
   d) ((A + B) * C – (D - E)) ↑ (F + G)
   e) A – B / (C * D ↑ E)

10. Write a “C” function to copy one stack to another assuming
    a) The stack is implemented using array.
    b) The stack is implemented using linked list.

11. Write an algorithm to construct a fully parenthesized infix expression from its postfix equivalent. Write a “C” function for your algorithm.

12. How can one convert a postfix expression to its prefix equivalent and vice-versa?

13. A double-ended queue (deque) is a linear list where additions and deletions can be performed at either end. Represent a deque using an array to store the elements of the list and write the “C” functions for additions and deletions.

14. In a circular queue represented by an array, how can one specify the number of elements in the queue in terms of “front”, “rear” and MAX-QUEUE-SIZE? Write a “C” function to delete the K-th element from the “front” of a circular queue.

15. Can a queue be represented by a circular linked list with only one pointer pointing to the tail of the queue? Write “C” functions for the “add” and “delete” operations on such a queue

16. Write a “C” function to test whether a string of opening and closing parenthesis is well formed or not.

17. Represent N queues in a single one-dimensional array. Write functions for “add” and “delete” operations on the i^{th} queue

18. Represent a stack and queue in a single one-dimensional array. Write functions for “push”, “pop” operations on the stack and “add”, “delete” functions on the queue.

Chapter 5

Binary Trees

A data structure is said to be linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

In this chapter in particular, we will explain special type of trees known as binary trees, which are easy to maintain in the computer.

5.1. TREES:

A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure 5.1.1 shows a tree and a non-tree.

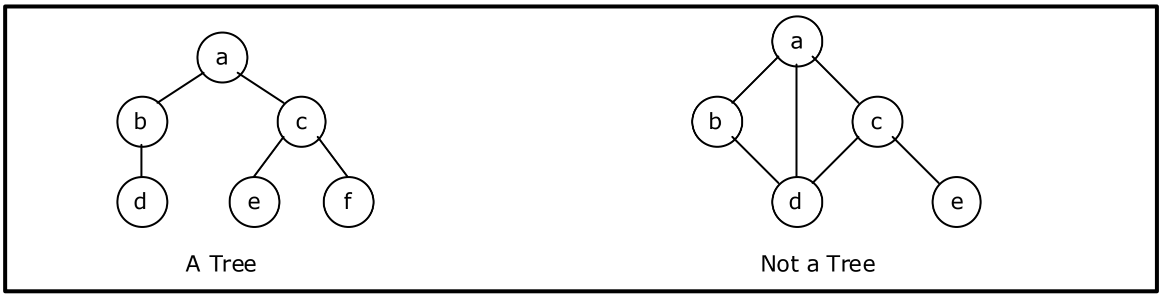


Figure 5.1.1 A Tree and a not a tree

In a tree data structure, there is no distinction between the various children of a node i.e., none is the "first child" or "last child". A tree in which such distinctions are made is called an **ordered tree**, and data structures built on them are called **ordered tree data structures**. Ordered trees are by far the commonest form of tree data structure.

5.2. BINARY TREE:

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.

A tree with no nodes is called as a **null** tree. A binary tree is shown in figure 5.2.1.

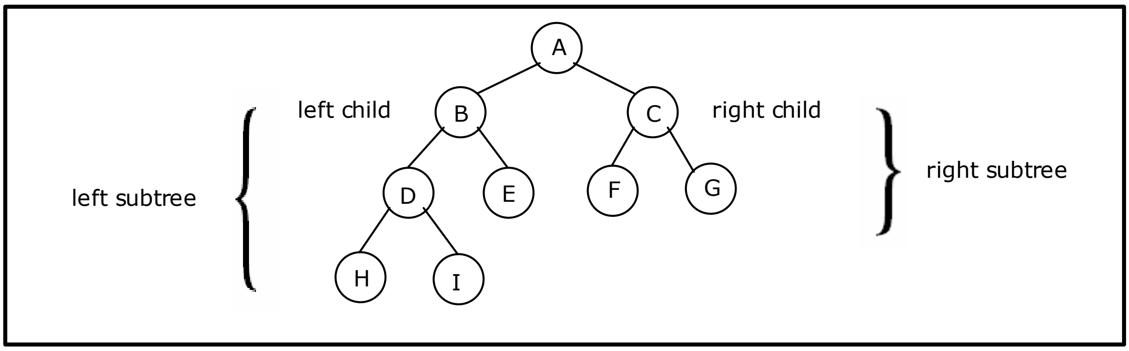


Figure 5.2.1. Binary Tree

Binary trees are easy to implement because they have a small, fixed number of child links. Because of this characteristic, binary trees are the most common types of trees and form the basis of many important data structures.

Tree Terminology:

Leaf node

A node with no children is called a *leaf* (or *external node*). A node which is not a leaf is called an *internal node*.

Path

A sequence of nodes n_1, n_2, . . ., n_k, such that n_i is the parent of n_{i+1} for i = 1, 2,. . ., k - 1. The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself.

For the tree shown in figure 5.2.1, the path between A and I is A, B, D, I.

Siblings

The children of the same parent are called siblings.

For the tree shown in figure 5.2.1, F and G are the siblings of the parent node C and H and I are the siblings of the parent node D.

Ancestor and Descendent

If there is a path from node A to node B, then A is called an ancestor of B and *B* is called a descendent of A.

Subtree

Any node of a tree, with all of its descendants is a subtree.

Level

The level of the node refers to its distance from the root. The root of the tree has level O, and the level of any other node in the tree is one more than the level of its parent. For example, in the binary tree of Figure 5.2.1 node F is at level 2 and node H is at level 3. *The maximum number of nodes at any level is 2^n.*

Height

The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree. The height of the tree of Figure 5.2.1 is 3.

Depth

The depth of a node is the number of nodes along the path from the root to that node. For instance, node 'C' in figure 5.2.1 has a depth of 1.

Assigning level numbers and Numbering of nodes for a binary tree:

The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent. For example, see Figure 5.2.2.

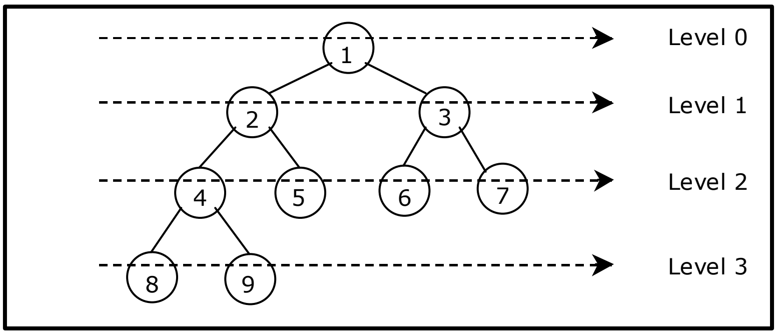


Figure 5.2.2. Level by level numbering of binary tree

Properties of binary trees:

Some of the important properties of a binary tree are as follows:

1. If *h* = height of a binary tree, then
    a. Maximum number of leaves = 2^h
    b. Maximum number of nodes = $2^{h+1} - 1$
2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l node at level l.
4. The total number of edges in a full binary tree with n node is n - 1.

Strictly Binary tree:

If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed as strictly binary tree. Thus the tree of figure 5.2.3(a) is strictly binary. A strictly binary tree with n leaves always contains 2n - 1 nodes.

Full Binary tree:

A full binary tree of height h has all its leaves at level h. Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height *h* has 2^{h+1} - 1 nodes. A full binary tree of height h is a *strictly binary tree* all of whose leaves are at level h. Figure 5.2.3(d) illustrates the full binary tree containing 15 nodes and of height 3.
A full binary tree of height h contains 2^h leaves and, 2^h - 1 non-leaf nodes.

Thus by induction, total number of nodes $(tn) = \sum_{l=0}^{h} 2^{l} = 2^{h+1} - 1$.

For example, a full binary tree of height 3 contains 2^{3+1} – 1 = 15 nodes.

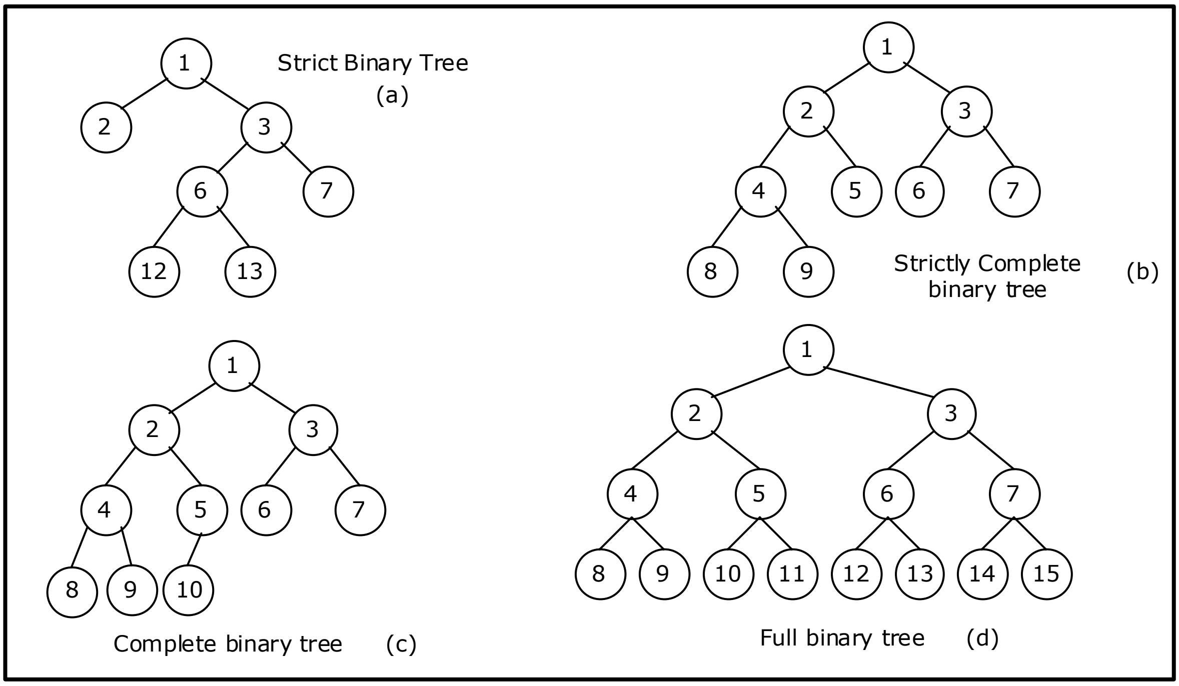


Figure 5.2.3. Examples of binary trees

Complete Binary tree:

A binary tree with *n* nodes is said to be **complete** if it contains all the first *n* nodes of the above numbering scheme. Figure 5.2.4 shows examples of complete and incomplete binary trees.

A complete binary tree of height h looks like a full binary tree down to level h-1, and the level h is filled from left to right.

A complete binary tree with n leaves that is *not strictly* binary has 2n nodes. For example, the tree of Figure 5.2.3(c) is a complete binary tree having 5 leaves and 10 nodes.

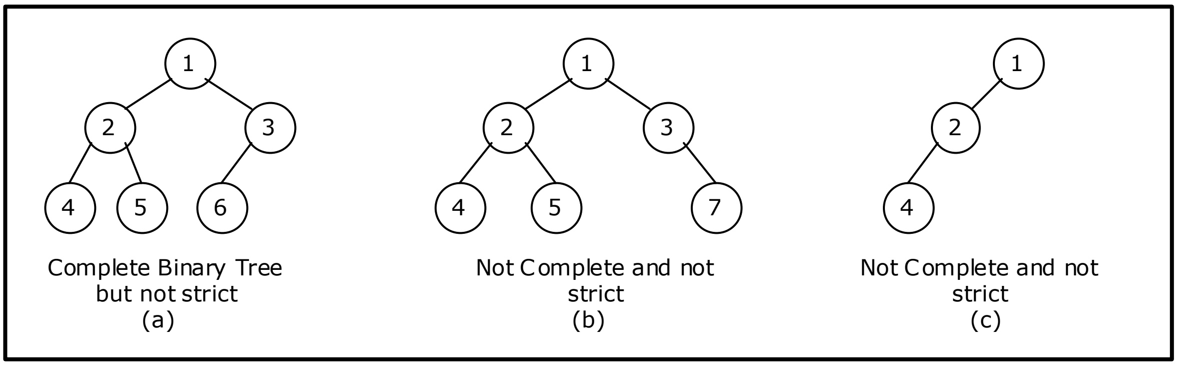

1
2
3
4
5
6
Complete Binary Tree but not strict
(a)
1
2
3
4
5
7
Not Complete and not strict
(b)
1
2
4
Not Complete and not strict
(c)

Figure 5.2.4. Examples of complete and incomplete binary trees

Internal and external nodes:

We define two terms: Internal nodes and external nodes. An internal node is a tree node having at least one–key and possibly some children. It is some times convenient to have another types of nodes, called an external node, and pretend that all null child links point to such a node. An external node doesn't exist, but serves as a conceptual place holder for nodes to be inserted.

We draw internal nodes using circles, with letters as labels. External nodes are denoted by squares. The square node version is sometimes called an extended binary tree. A binary tree with n internal nodes has n+1 external nodes. Figure 5.2.6 shows a sample tree illustrating both internal and external nodes.

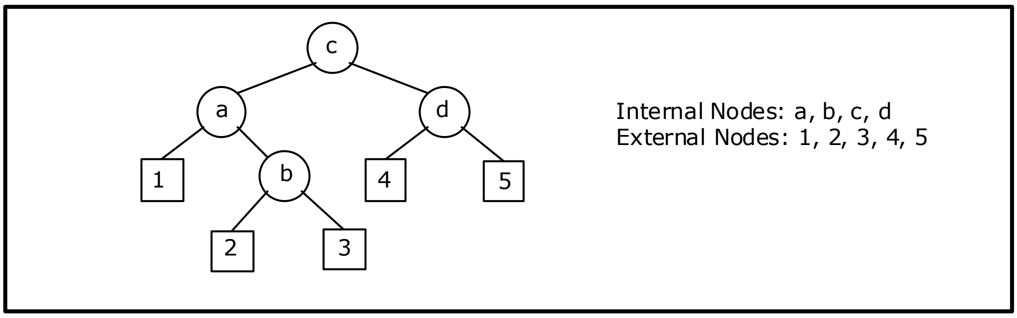


Figure 5.2.6. Internal and external nodes

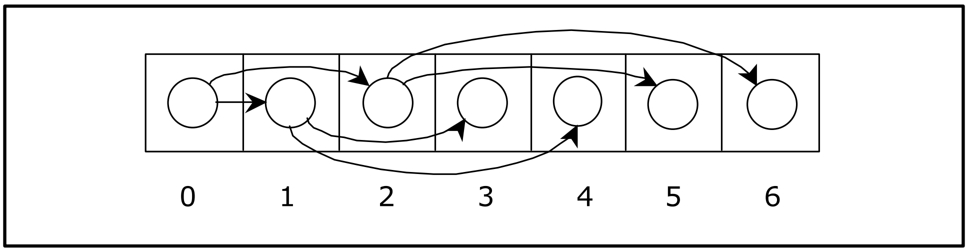
Data Structures for Binary Trees:

1. Arrays; especially suited for complete and full binary trees.
2. Pointer-based.

Array-based Implementation:

Binary trees can also be stored in arrays, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index *i*, its children are found at indices 2*i*+1 and 2*i*+2, while its parent (if any) is found at index *floor((i-1)/2)* (assuming the root of the tree stored in the array at an index zero).

This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal. However, it requires contiguous memory, expensive to grow and wastes space proportional to 2^h - n for a tree of height h with n nodes.



Linked Representation (Pointer based):

Array representation is good for complete binary tree, but it is wasteful for many other binary trees. The representation suffers from insertion and deletion of node from the middle of the tree, as it requires the moment of potentially many nodes to reflect the change in level number of this node. To overcome this difficulty we represent the binary tree in linked representation.

In linked representation each node in a binary has three fields, the left child field denoted as *LeftChild*, data field denoted as *data* and the right child field denoted as *RightChild*. If any sub-tree is empty then the corresponding pointer's LeftChild and RightChild will store a NULL value. If the tree itself is empty the root pointer will store a NULL value.

The advantage of using linked representation of binary tree is that:

- Insertion and deletion involve no data movement and no movement of nodes except the rearrangement of pointers.

The disadvantages of linked representation of binary tree includes:

- Given a node structure, it is difficult to determine its parent node.

- Memory spaces are wasted for storing NULL pointers for the nodes, which have no subtrees.

The structure definition, node representation empty binary tree is shown in figure 5.2.6 and the linked representation of binary tree using this node structure is given in figure 5.2.7.

```
struct binarytree
{
        struct binarytree *LeftChild;
        int data;
        struct binarytree *RightChild;
};

typedef struct binarytree node;

node *root = NULL;
```

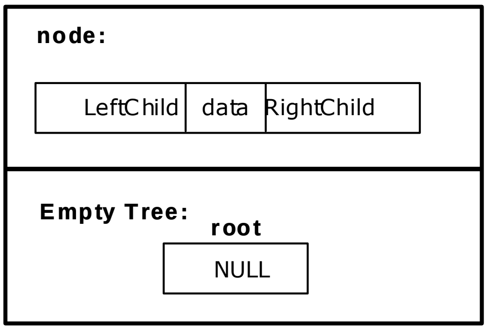


Figure 5.2.6. Structure definition, node representation and empty tree

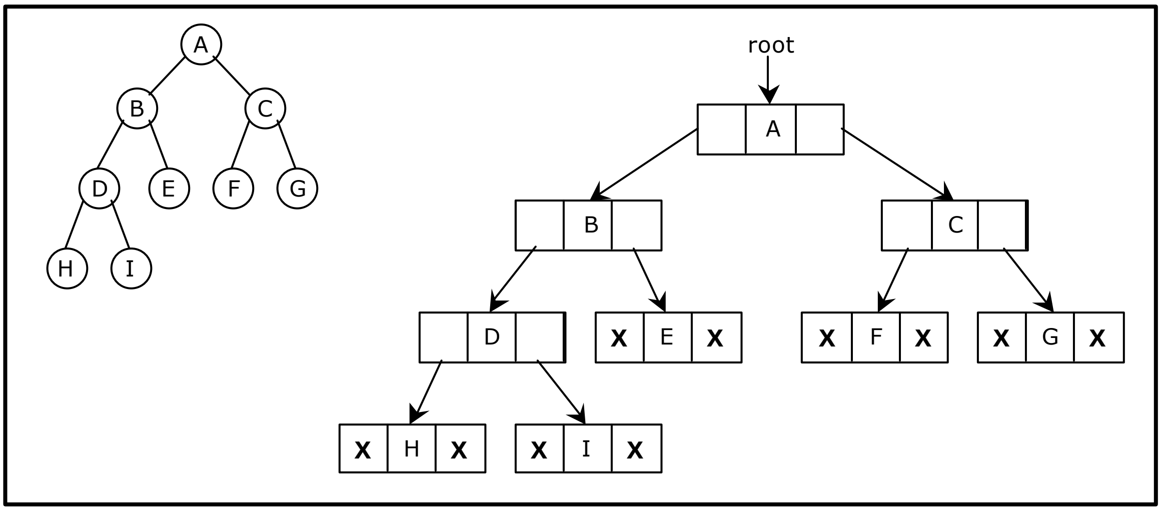


Figure 5.2.7. Linked representation for the binary tree

5.3. Binary Tree Traversal Techniques:

A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

There are four common ways to traverse a binary tree:

1. *Preorder*
2. *Inorder*
3. *Postorder*
4. *Level order*

In the first three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among them comes from the difference in the time at which a root node is visited.

5.3.1. Recursive Traversal Algorithms:

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```
void inorder(node *root)
{
        if(root != NULL)
        {
                inorder(root->lchild);
                print root -> data;
                inorder(root->rchild);
        }
}
```

Preorder Traversal:

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```
void preorder(node *root)
{
        if( root != NULL )
        {
                print root -> data;
                preorder (root -> lchild);
                preorder (root -> rchild);
        }
}
```

Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

The algorithm for postorder traversal is as follows:

```
void postorder(node *root)
{
        if( root != NULL )
        {
                postorder (root -> lchild);
                postorder (root -> rchild);
                print (root -> data);
        }
}
```

Level order Traversal:

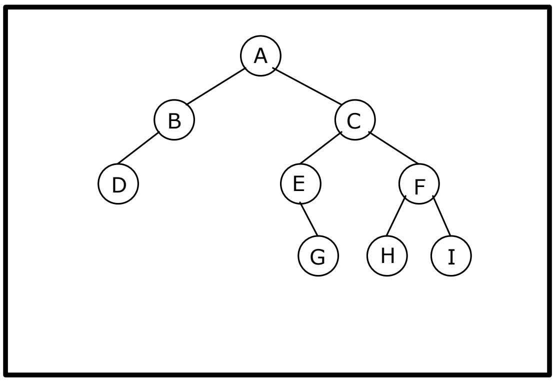
In a level order traversal, the nodes are visited level by level starting from the root, and going from left to right. The level order traversal requires a queue data structure. So, it is not possible to develop a recursive procedure to traverse the binary tree in level order. This is nothing but a breadth first search technique.

The algorithm for level order traversal is as follows:

```
void levelorder()
{
        int j;
        for(j = 0; j < ctr; j++)
        {
                if(tree[j] != NULL)
                        print tree[j] -> data;
        }
}
```

Example 1:

Traverse the following binary tree in pre, post, inorder and level order.



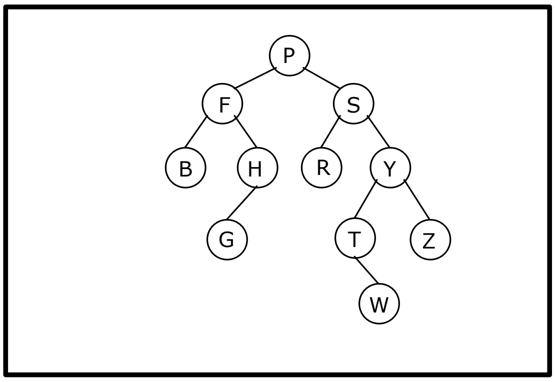
Binary Tree

- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



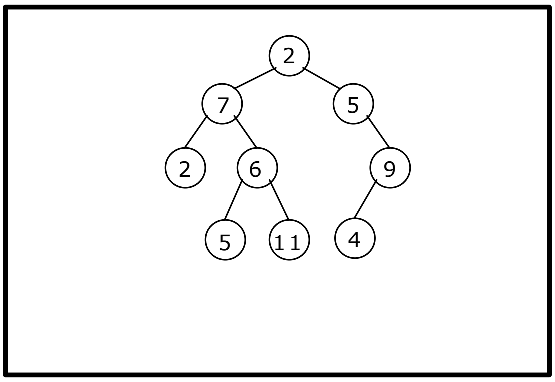
Binary Tree

- Preorder traversal yields:
  P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
  B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
  B, F, G, H, P, R, S, T, W, Y, Z
- Level order traversal yields:
  P, F, S, B, H, R, Y, G, T, Z, W

Pre, Post, Inorder and level order Traversing

Example 3:

Traverse the following binary tree in pre, post, inorder and level order.



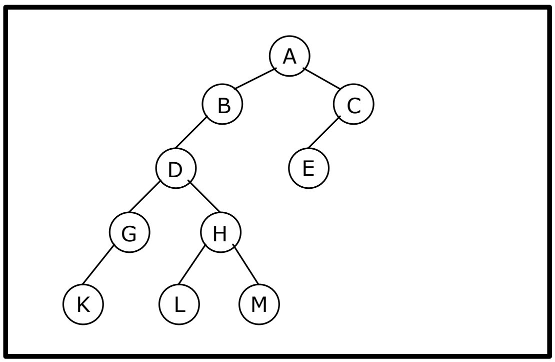
Binary Tree

- Preorder traversal yields:
  2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder travarsal yields:
  2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder travarsal yields:
  2, 7, 5, 6, 11, 2, 5, 4, 9
- Level order traversal yields:
  2, 7, 5, 2, 6, 9, 5, 11, 4

Pre, Post, Inorder and level order Traversing

Example 4:

Traverse the following binary tree in pre, post, inorder and level order.



Binary Tree

- Preorder traversal yields:
  A, B, D, G, K, H, L, M, C, E
- Postorder travarsal yields:
  K, G, L, M, H, D, B, E, C, A
- Inorder travarsal yields:
  K, G, D, L, H, M, B, A, E, C
- Level order traversal yields:
  A, B, C, D, E, G, H, K, L, M

Pre, Post, Inorder and level order Traversing

5.3.2. Building Binary Tree from Traversal Pairs:

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

- Inorder and preorder
- Inorder and postorder
- Inorder and level order

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified using inorder.

Same technique can be applied repeatedly to form sub-trees.

It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be inorder traversal and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

Example 1:

Construct a binary tree from a given preorder and inorder sequence:

Preorder: A B D G C E H I F
Inorder: D G B A H E I C F

Solution:

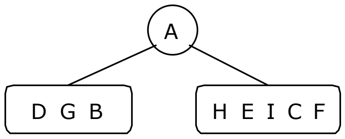
*From Preorder sequence **A** B D G C E H I F, the root is: A*

From Inorder sequence *D G B* ***A*** *H E I C F*, we get the left and right sub trees:

Left sub tree is: D G B

Right sub tree is: H E I C F

The Binary tree upto this point looks like:

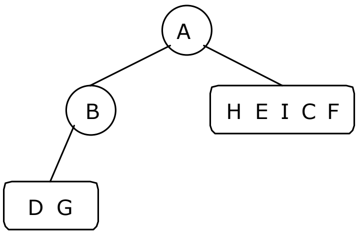


To find the root, left and right sub trees for D G B:

*From the preorder sequence **B** D G, the root of tree is: B*

From the inorder sequence D G **B,** we can find that D and G are to the left of B.

The Binary tree upto this point looks like:

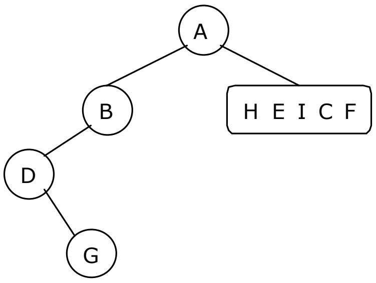


To find the root, left and right sub trees for D G:

From the preorder sequence **D** *G, the root of the tree is: D*

From the inorder sequence ***D*** *G*, we can find that there is no left node to D and G is at the right of D.

The Binary tree upto this point looks like:

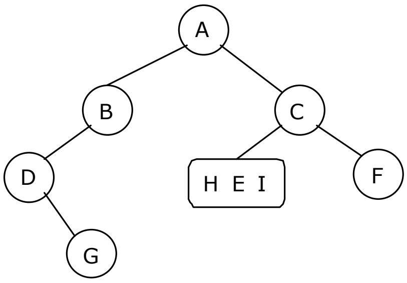


To find the root, left and right sub trees for H E I C F:

From the preorder sequence ***C*** *E H I F, the root of the left sub tree is: C*

From the inorder sequence *H E I* ***C*** *F*, we can find that H E I are at the left of C and F is at the right of C.

The Binary tree upto this point looks like:

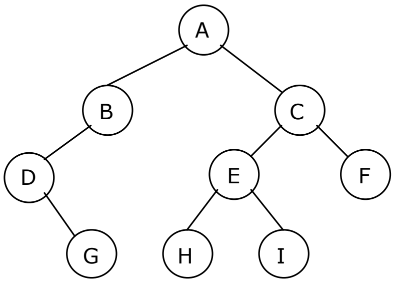


To find the root, left and right sub trees for H E I:

From the preorder sequence ***E*** *H I, the root of the tree is: E*

From the inorder sequence *H* ***E*** *I*, we can find that H is at the left of E and I is at the right of E.

The Binary tree upto this point looks like:



Example 2:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: D G B A H E I C F
Postorder: G D B H I E F C A

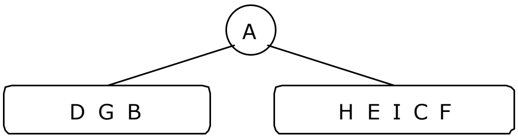
Solution:

From Postorder sequence G D B H I E F C **A**, *the root is: A*

From Inorder sequence D G B **A** H E I C F, we get the left and right sub trees:

> *Left sub tree is: D G B*
> *Right sub tree is: H E I C F*

The Binary tree upto this point looks like:

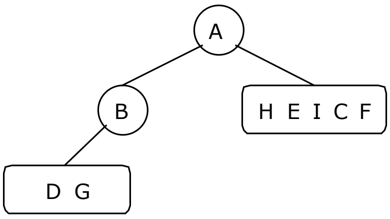


To find the root, left and right sub trees for D G B:

From the postorder sequence G D B, the root of tree is: B

From the inorder sequence D G **B,** we can find that D G are to the left of B and there is no right subtree for B.

The Binary tree upto this point looks like:

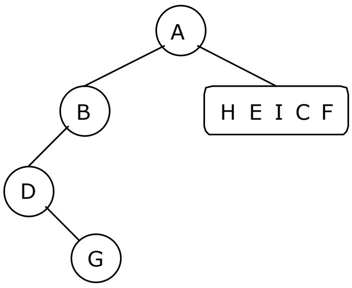


To find the root, left and right sub trees for D G:

From the postorder sequence G ***D****, the root of the tree is: D*

From the inorder sequence **D** G, we can find that is no left subtree for D and G is to the left right of D.

The Binary tree upto this point looks like:

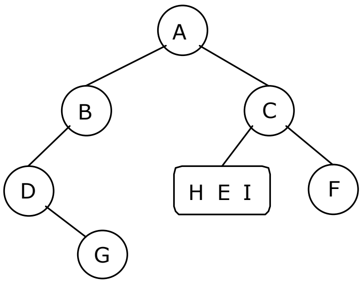


To find the root, left and right sub trees for H E I C F:

From the postorder sequence H I E F ***C****, the root of the left sub tree is: C*

From the inorder sequence H E I ***C*** F, we can find that H E I are to the left of C and F is the right subtree for C.

The Binary tree upto this point looks like:

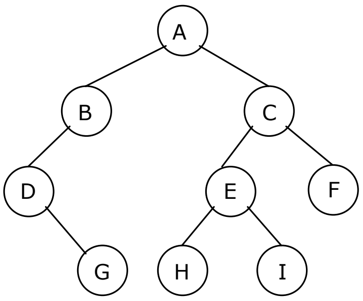


<u>To find the root, left and right sub trees for H E I:</u>

*From the postorder sequence H I **E**, the root of the tree is: E*

From the inorder sequence *<u>H</u>* **E** *<u>I</u>*, we can find that H is left subtree for E and I is to the right of E.

The Binary tree upto this point looks like:



Example 3:

Construct a binary tree from a given preorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Preorder: n6 n2 n1 n4 n3 n5 n9 n7 n8

Solution:

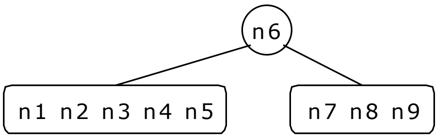
From Preorder sequence **n6** n2 n1 n4 n3 n5 n9 n7 n8, *the root is: n6*

From Inorder sequence *<u>n1 n2 n3 n4 n5</u>* **n6** *<u>n7 n8 n9</u>*, we get the left and right sub trees:

Left sub tree is: n1 n2 n3 n4 n5

Right sub tree is: n7 n8 n9

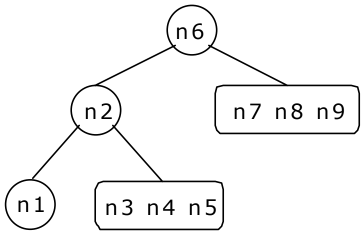
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n1 n2 n3 n4 n5:

*From the preorder sequence **n2** n1 n4 n3 n5, the root of tree is: n2*

From the inorder sequence *n1* **n2** *n3 n4 n5*, we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2. The Binary tree upto this point looks like:

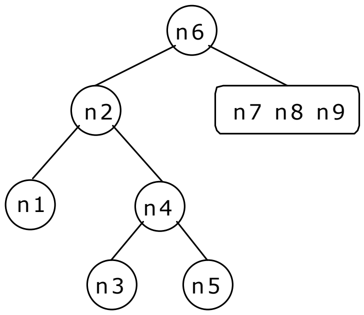


To find the root, left and right sub trees for n3 n4 n5:

*From the preorder sequence **n4** n3 n5, the root of the tree is: n4*

From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is at the right of n4.

The Binary tree upto this point looks like:

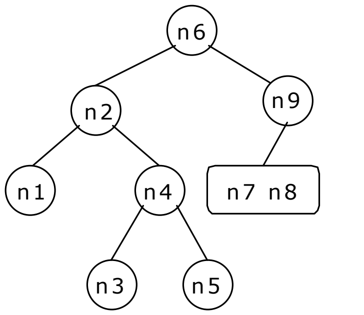


To find the root, left and right sub trees for n7 n8 n9:

*From the preorder sequence **n9** n7 n8, the root of the left sub tree is: n9*

From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are at the left of n9 and no right subtree of n9.

The Binary tree upto this point looks like:

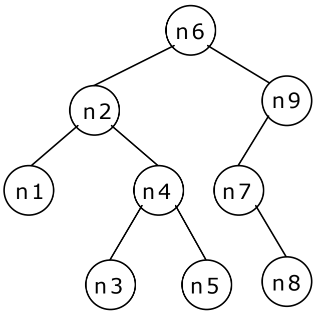


To find the root, left and right sub trees for n7 n8:

*From the preorder sequence **n7** n8, the root of the tree is: n7*

From the inorder sequence **n7** *n8*, we can find that is no left subtree for n7 and n8 is at the right of n7.

The Binary tree upto this point looks like:



Example 4:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

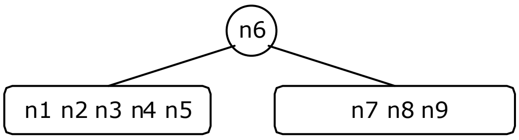
Solution:

From Postorder sequence n1 n3 n5 n4 n2 n8 n7 n9 **n6**, *the root is: n6*

From Inorder sequence *n1 n2 n3 n4 n5* **n6** *n7 n8 n9*, we get the left and right sub trees:

> *Left sub tree is: n1 n2 n3 n4 n5*
> *Right sub tree is: n7 n8 n9*

The Binary tree upto this point looks like:

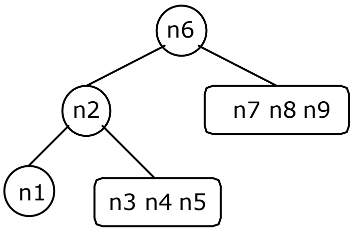


To find the root, left and right sub trees for *n1 n2 n3 n4 n5*:

From the postorder sequence n1 n3 n5 n4 **n2**, *the root of tree is: n2*

From the inorder sequence *n1* **n2** *n3 n4 n5*, we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2.

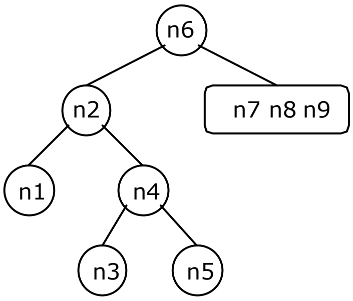
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n3 n4 n5:

From the postorder sequence n3 n5 **n4***, the root of the tree is: n4*

From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is to the right of n4. The Binary tree upto this point looks like:

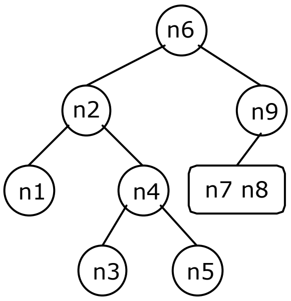


To find the root, left and right sub trees for n7 n8 and n9:

From the postorder sequence n8 n7 **n9***, the root of the left sub tree is: n9*

From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are to the left of n9 and no right subtree for n9.

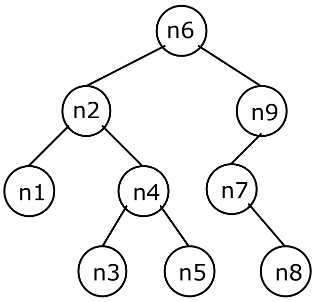
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n7 and n8:

From the postorder sequence n8 **n7***, the root of the tree is: n7*

From the inorder sequence **n7** *n8*, we can find that there is no left subtree for n7 and n8 is to the right of n7. The Binary tree upto this point looks like:



5.3.3. Binary Tree Creation and Traversal Using Arrays:

This program performs the following operations:

1. Creates a complete Binary Tree
2. Inorder traversal
3. Preorder traversal
4. Postorder traversal
5. Level order traversal
6. Prints leaf nodes
7. Finds height of the tree created

```
# include <stdio.h>
# include <stdlib.h>

struct tree
{
        struct tree* lchild;
        char  data[10];
        struct tree* rchild;
};

typedef struct tree node;
int ctr;
node *tree[100];

node* getnode()
{
        node *temp ;
        temp = (node*) malloc(sizeof(node));
        printf("\n Enter Data: ");
        scanf("%s",temp->data);
        temp->lchild = NULL;
        temp->rchild = NULL;
        return temp;
}

void create_fbinarytree()
{
        int j, i=0;
        printf("\n How many nodes you want: ");
        scanf("%d",&ctr);
        tree[0] = getnode();
        j = ctr;
        j--;
        do
        {
                if( j > 0 )                                     /* left child  */
                {
                        tree[ i * 2 + 1 ] = getnode();
                        tree[i]->lchild = tree[i * 2 + 1];
                        j--;
                }
                if( j > 0 )                                     /* right child */
                {
                        tree[i * 2 + 2] = getnode();
                        j--;
                        tree[i]->rchild = tree[i * 2 + 2];
                }
                i++;
        } while( j > 0);
}
```

```
void inorder(node *root)
{
        if( root != NULL )
        {
                inorder(root->lchild);
                printf("%3s",root->data);
                inorder(root->rchild);
        }
}


void preorder(node *root)
{
        if( root != NULL )
        {
                printf("%3s",root->data);
                preorder(root->lchild);
                preorder(root->rchild);
        }
}

void postorder(node *root)
{
        if( root != NULL )
        {
                postorder(root->lchild);
                postorder(root->rchild);
                printf("%3s",root->data);
        }
}



void levelorder()
{
        int j;
        for(j = 0; j < ctr; j++)
        {
                if(tree[j] != NULL)
                        printf("%3s",tree[j]->data);
        }
}



void print_leaf(node *root)
{
        if(root != NULL)
        {
                if(root->lchild == NULL && root->rchild == NULL)
                  printf("%3s ",root->data);
                  print_leaf(root->lchild);
                  print_leaf(root->rchild);
        }
}

int height(node *root)
{
        if(root == NULL)
        {
                return 0;
        }
        if(root->lchild == NULL && root->rchild == NULL)
                return 0;
        else
                return (1 + max(height(root->lchild), height(root->rchild)));
}

void main()
{
        int i;
        create_fbinarytree();
        printf("\n Inorder Traversal: ");
        inorder(tree[0]);
        printf("\n Preorder Traversal: ");
        preorder(tree[0]);
        printf("\n Postorder Traversal: ");
        postorder(tree[0]);
        printf("\n Level Order Traversal: ");
        levelorder();
        printf("\n Leaf Nodes: ");
        print_leaf(tree[0]);
        printf("\n Height of Tree: %d ", height(tree[0]));
}
```

5.3.4. Binary Tree Creation and Traversal Using Pointers:

This program performs the following operations:

1. Creates a complete Binary Tree
2. Inorder traversal
3. Preorder traversal
4. Postorder traversal
5. Level order traversal
6. Prints leaf nodes
7. Finds height of the tree created
8. Deletes last node
9. Finds height of the tree created

```
# include <stdio.h>
# include <stdlib.h>

struct tree
{
        struct tree* lchild;
        char  data[10];
        struct tree* rchild;
};

typedef struct tree node;
node *Q[50];
int node_ctr;

node* getnode()
{
        node *temp ;
        temp = (node*) malloc(sizeof(node));
        printf("\n Enter Data: ");
        fflush(stdin);
        scanf("%s",temp->data);
        temp->lchild = NULL;
        temp->rchild = NULL;
        return temp;
}
```

```
void create_binarytree(node *root)
{
        char option;
        node_ctr = 1;
        if( root != NULL )
        {
                printf("\n Node  %s has Left SubTree(Y/N)",root->data);
                fflush(stdin);
                scanf("%c",&option);
                if( option=='Y' || option == 'y')
                {
                        root->lchild = getnode();
                        node_ctr++;
                        create_binarytree(root->lchild);
                }
                else
                {
                        root->lchild = NULL;
                        create_binarytree(root->lchild);
                }

                printf("\n Node %s has Right SubTree(Y/N) ",root->data);
                fflush(stdin);
                scanf("%c",&option);
                if( option=='Y' || option == 'y')
                {
                        root->rchild = getnode();
                        node_ctr++;
                        create_binarytree(root->rchild);
                }
                else
                {
                        root->rchild = NULL;
                        create_binarytree(root->rchild);
                }
        }
}

void make_Queue(node *root,int parent)
{
        if(root != NULL)
        {
                node_ctr++;
                Q[parent] = root;
                make_Queue(root->lchild,parent*2+1);
                make_Queue(root->rchild,parent*2+2);
        }
}

delete_node(node *root, int parent)
{
        int index = 0;
        if(root == NULL)
                printf("\n Empty TREE ");
        else
        {
                node_ctr = 0;
                make_Queue(root,0);
                index = node_ctr-1;
                Q[index] = NULL;
                parent = (index-1) /2;
                if( 2* parent + 1 == index )
                        Q[parent]->lchild = NULL;
```

```
				else
					Q[parent]->rchild = NULL;
		}
		printf("\n Node Deleted ..");
}

void inorder(node *root)
{
		if(root != NULL)
		{
				inorder(root->lchild);
				printf("%3s",root->data);
				inorder(root->rchild);
		}
}

void preorder(node *root)
{
		if( root != NULL )
		{
				printf("%3s",root->data);
				preorder(root->lchild);
				preorder(root->rchild);
		}
}

void postorder(node *root)
{
		if( root != NULL )
		{
				postorder(root->lchild);
				postorder(root->rchild);
				printf("%3s", root->data);
		}
}

void print_leaf(node *root)
{
		if(root != NULL)
		{
				if(root->lchild == NULL && root->rchild == NULL)
				  printf("%3s ",root->data);
				  print_leaf(root->lchild);
				  print_leaf(root->rchild);
		}
}

int height(node *root)
{
		if(root == NULL)
				return -1;
		else
				return (1 + max(height(root->lchild), height(root->rchild)));
}

void print_tree(node *root, int line)
{
		int i;
		if(root != NULL)
		{
				print_tree(root->rchild,line+1);
				printf("\n");
				for(i=0;i<line;i++)
```

```
                        printf(" ");
                printf("%s", root->data);
                print_tree(root->lchild,line+1);
        }
}

void level_order(node *Q[],int ctr)
{
        int i;
        for( i = 0; i < ctr ; i++)
        {
                if( Q[i] != NULL )
                        printf("%5s",Q[i]->data);
        }
}

int menu()
{
        int ch;
        clrscr();
        printf("\n 1. Create Binary Tree ");
        printf("\n 2. Inorder Traversal ");
        printf("\n 3. Preorder Traversal ");
        printf("\n 4. Postorder Traversal ");
        printf("\n 5. Level Order Traversal");
        printf("\n 6. Leaf Node ");
        printf("\n 7. Print Height of Tree ");
        printf("\n 8. Print Binary Tree ");
        printf("\n 9. Delete a node ");
        printf("\n 10. Quit ");
        printf("\n Enter Your choice: ");
        scanf("%d", &ch);
        return ch;
}

void main()
{
        int i,ch;
        node *root = NULL;
        do
        {
                ch = menu();
                switch( ch)
                {
                        case 1 :
                                if( root == NULL )
                                {
                                        root = getnode();
                                        create_binarytree(root);
                                }
                                else
                                {
                                        printf("\n Tree is already Created ..");
                                }
                                break;
                        case 2 :
                                printf("\n Inorder Traversal: ");
                                inorder(root);
                                break;
                        case 3 :
                                printf("\n Preorder Traversal: ");
                                preorder(root);
                                break;
                        case 4 :
                                printf("\n Postorder Traversal: ");
                                postorder(root);
                                break;
                        case 5:
                                printf("\n Level Order Traversal ..");
                                make_Queue(root,0);
                                level_order(Q,node_ctr);
                                break;
                        case 6 :
                                printf("\n Leaf Nodes: ");
                                print_leaf(root);
                                break;
                        case 7 :
                                printf("\n Height of Tree: %d ", height(root));
                                break;
                        case 8 :
                                printf("\n Print Tree \n");
                                print_tree(root, 0);
                                break;
                        case 9 :
                                        delete_node(root,0);
                                        break;
                        case 10 :
                                exit(0);
                }
                getch();
        }while(1);
}
```

5.3.5. Non Recursive Traversal Algorithms:

At first glance, it appears that we would always want to use the flat traversal functions since they use less stack space. But the flat versions are not necessarily better. For instance, some overhead is associated with the use of an explicit stack, which may negate the savings we gain from storing only node pointers. Use of the implicit function call stack may actually be faster due to special machine instructions that can be used.

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.

2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

```
Algorithm inorder()
{
        stack[1] = 0
        vertex = root
top:    while(vertex ≠ 0)
        {
                push the vertex into the stack
                vertex = leftson(vertex)
        }

        pop the element from the stack and make it as vertex

        while(vertex ≠ 0)
        {
                print the vertex node
                if(rightson(vertex) ≠ 0)
                {
                        vertex = rightson(vertex)
                        goto top
                }
                pop the element from the stack and made it as vertex
        }
}
```

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

```
Algorithm preorder( )
{
        stack[1] = 0
        vertex = root.
        while(vertex ≠ 0)
        {
                print vertex node
                if(rightson(vertex) ≠ 0)
                        push the right son of vertex into the stack.
                if(leftson(vertex) ≠ 0)
                        vertex = leftson(vertex)
                else
                        pop the element from the stack and made it as vertex
        }
}
```

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

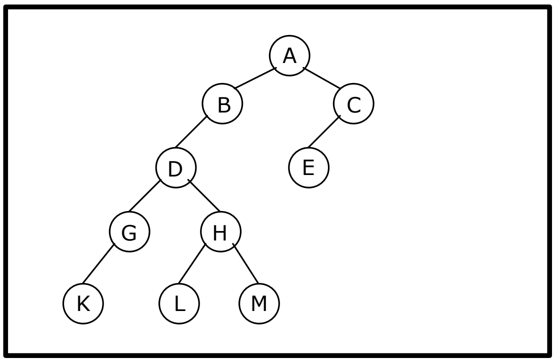
1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

```
Algorithm postorder( )
{
        stack[1] = 0
        vertex = root

 top: while(vertex ≠ 0)
        {
                push vertex onto stack
                if(rightson(vertex) ≠ 0)
                        push – (vertex) onto stack
                vertex = leftson(vertex)
        }
        pop from stack and make it as vertex
        while(vertex > 0)
        {
                print the vertex node
                pop from stack and make it as vertex
        }
        if(vertex < 0)
        {
                vertex = - (vertex)
                goto top
        }
}
```

Example 1:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Binary Tree

- Preorder traversal yields:
  A, B, D, G, K, H, L, M, C, E
- Postorder travarsal yields:
  K, G, L, M, H, D, B, E, C, A
- Inorder travarsal yields:
  K, G, D, L, H, M, B, A, E, C

Pre, Post and Inorder Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.

2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
A	0		PUSH 0
	0 A B D G K		PUSH the left most path of A
K	0 A B D G	K	POP K
G	0 A B D	K G	POP G since K has no right son
D	0 A B	K G D	POP D since G has no right son
H	0 A B	K G D	Make the right son of D as vertex
	0 A B H L	K G D	PUSH the leftmost path of H
L	0 A B H	K G D L	POP L
H	0 A B	K G D L H	POP H since L has no right son
M	0 A B	K G D L H	Make the right son of H as vertex
	0 A B M	K G D L H	PUSH the left most path of M
M	0 A B	K G D L H M	POP M
B	0 A	K G D L H M B	POP B since M has no right son
A	0	K G D L H M B A	Make the right son of A as vertex
C	0 C E	K G D L H M B A	PUSH the left most path of C
E	0 C	K G D L H M B A E	POP E
C	0	K G D L H M B A E C	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
A	0		PUSH 0
	0 A –C B D –H G K		PUSH the left most path of A with a -ve for right sons
	0 A –C B D –H	K G	POP all +ve nodes K and G
H	0 A –C B D	K G	Pop H

	0 A –C B D H –M L	K G	PUSH the left most path of H with a -ve for right sons
L	0 A –C B D H –M	K G L	POP all +ve nodes L
M	0 A –C B D H	K G L	Pop M
	0 A –C B D H M	K G L	PUSH the left most path of M with a -ve for right sons
	0 A –C	K G L M H D B	POP all +ve nodes M, H, D and B
C	0 A	K G L M H D B	Pop C
	0 A C E	K G L M H D B	PUSH the left most path of C with a -ve for right sons
	0	K G L M H D B E C A	POP all +ve nodes E, C and A
	0	K G L M H D B E C A	Stop since stack is empty

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

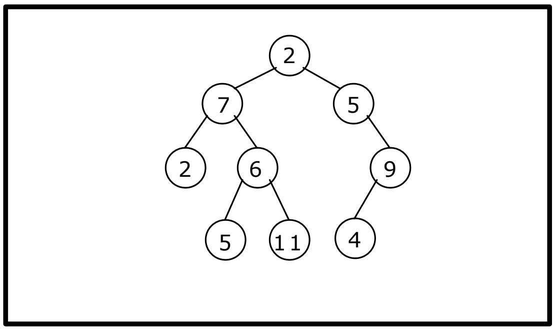
1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
A	0		PUSH 0
	0 C H	A B D G K	PUSH the right son of each vertex onto stack and process each vertex in the left most path
H	0 C	A B D G K	POP H
	0 C M	A B D G K H L	PUSH the right son of each vertex onto stack and process each vertex in the left most path
M	0 C	A B D G K H L	POP M
	0 C	A B D G K H L M	PUSH the right son of each vertex onto stack and process each vertex in the left most path; M has no left path
C	0	A B D G K H L M	Pop C
	0	A B D G K H L M C E	PUSH the right son of each vertex onto stack and process each vertex in the left most path; C has no right son on the left most path
	0	A B D G K H L M C E	Stop since stack is empty

Example 2:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Binary Tree

- Preorder traversal yields:
  2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder travarsal yields:
  2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder travarsal yields:
  2, 7, 5, 6, 11, 2, 5, 4, 9

Pre, Post and In order Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.

2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
2	0		
	0 2 7 2		
2	0 2 7	2	
7	0 2	2 7	
6	0 2 6 5	2 7	
5	0 2 6	2 7 5	
6	0 2	2 7 5 6	
11	0 2 11	2 7 5 6	
11	0 2	2 7 5 6 11	
2	0	2 7 5 6 11 2	
5	0 5	2 7 5 6 11 2	
5	0	2 7 5 6 11 2 5	
9	0 9 4	2 7 5 6 11 2 5	
4	0 9	2 7 5 6 11 2 5 4	
9	0	2 7 5 6 11 2 5 4 9	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
2	0		
	0 2 –5 7 –6 2		
2	0 2 –5 7 –6	2	
6	0 2 –5 7	2	
	0 2 –5 7 6 –11 5	2	
5	0 2 –5 7 6 –11	2 5	
11	0 2 –5 7 6 11	2 5	
	0 2 –5	2 5 11 6 7	
5	0 2 5 –9	2 5 11 6 7	
9	0 2 5 9 4	2 5 11 6 7	
	0	2 5 11 6 7 4 9 5 2	Stop since stack is empty

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
2	0		
	0 5 6	2 7 2	
6	0 5 11	2 7 2 6 5	
11	0 5	2 7 2 6 5 11	
	0 5	2 7 2 6 5 11	
5	0 9	2 7 2 6 5 11 5	
9	0	2 7 2 6 5 11 5 9 4	
	0	2 7 2 6 5 11 5 9 4	Stop since stack is empty

5.4. Expression Trees:

Expression tree is a binary tree, because all of the operations are binary. It is also possible for a node to have only one child, as is the case with the unary minus operator. The leaves of an expression tree are operands, such as constants or variable names, and the other (non leaf) nodes contain operators.

Once an expression tree is constructed we can traverse it in three ways:

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

Figure 5.4.1 shows some more expression trees that represent arithmetic expressions given in infix form.

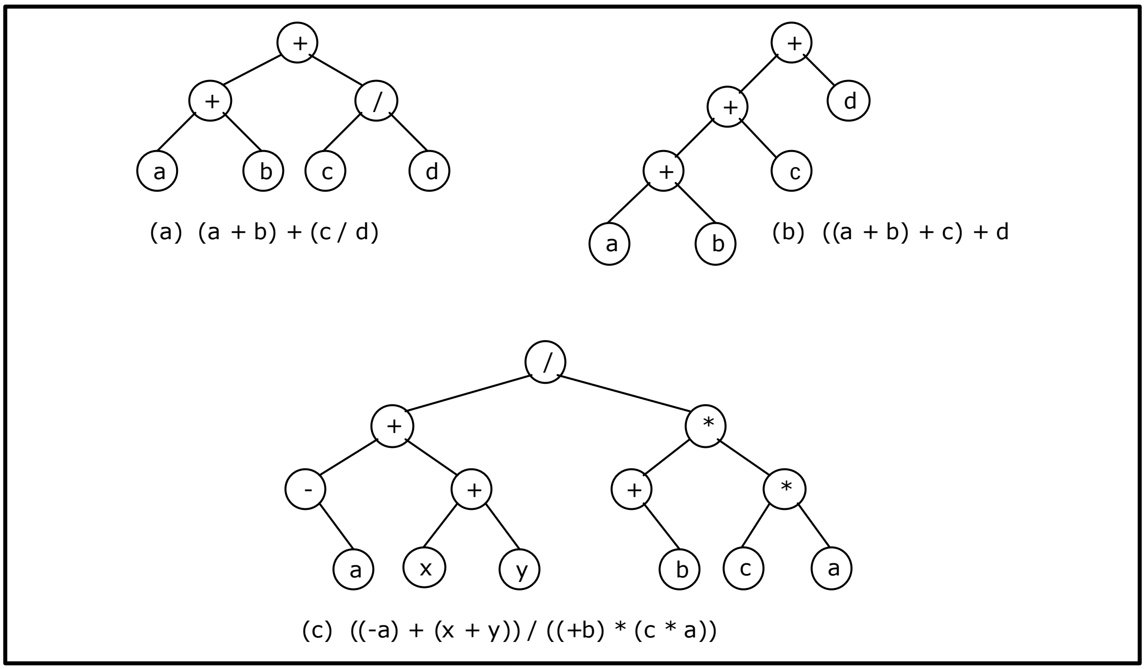


Figure 5.4.1 Expression Trees

An expression tree can be generated for the infix and postfix expressions.

An algorithm to convert a postfix expression into an expression tree is as follows:

1. Read the expression one symbol at a time.

2. If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack.

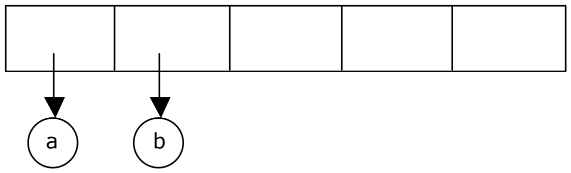
3. If the symbol is an operator, we pop pointers to two trees T1 and T2 from the stack (T1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T2 and T1 respectively. A pointer to this new tree is then pushed onto the stack.

Example 1:

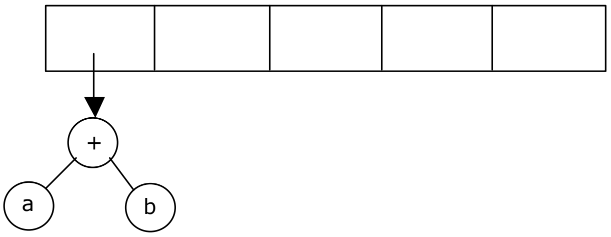
Construct an expression tree for the postfix expression: a b + c d e + * *

Solution:

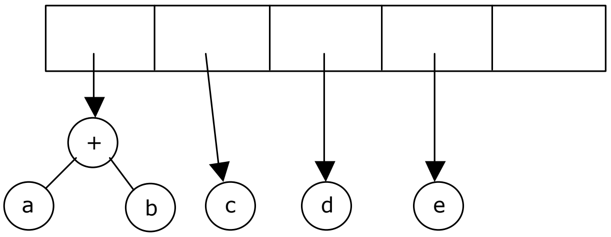
The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.



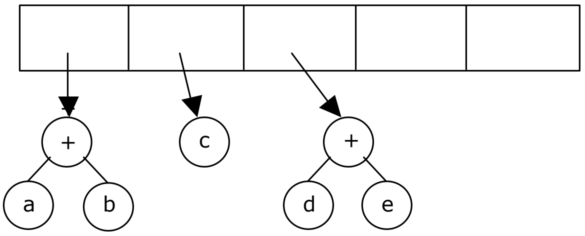
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



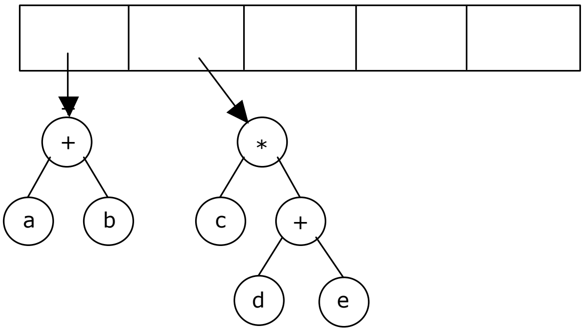
Next, c, d, and e are read, and for each one–node tree is created and a pointer to the corresponding tree is pushed onto the stack.



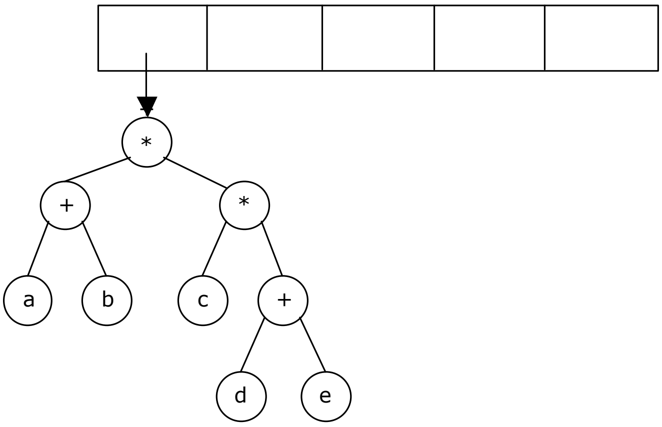
Now a '+' is read, so two trees are merged.



Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



For the above tree:

Inorder form of the expression: a + b * c * d + e

Preorder form of the expression: * + a b * c + d e

Postorder form of the expression: a b + c d e + * *

Example 2:

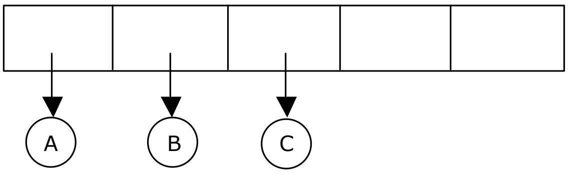
Construct an expression tree for the arithmetic expression:

(A + B * C) – ((D * E + F) / G)

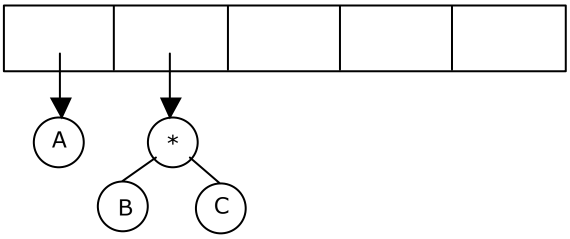
Solution:

First convert the infix expression into postfix notation. Postfix notation of the arithmetic expression is: A B C * + D E * F + G / -

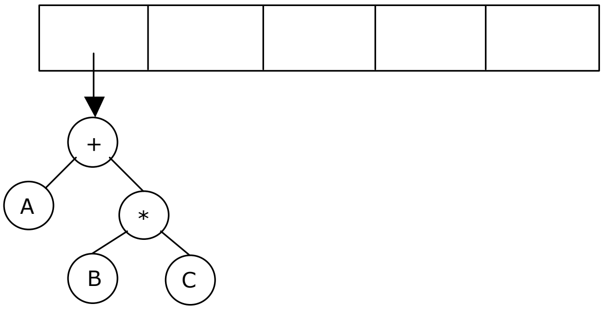
The first three symbols are operands, so we create one-node trees and pointers to three nodes pushed onto the stack.



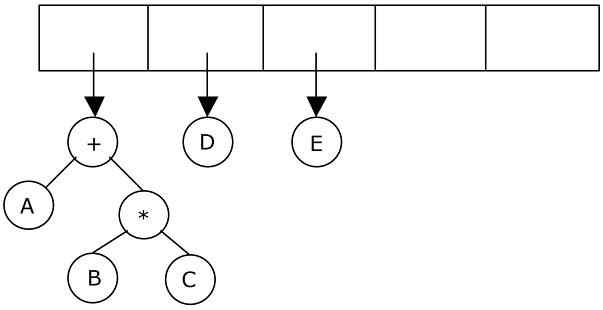
Next, a ‘*’ is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



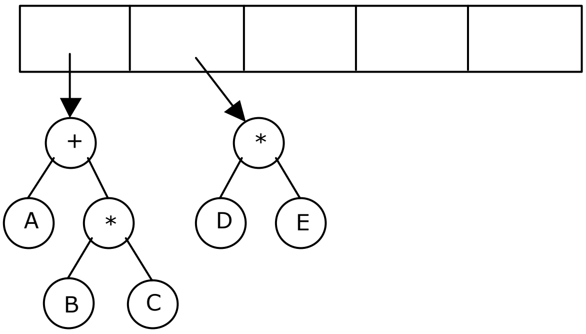
Next, a ‘+’ is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



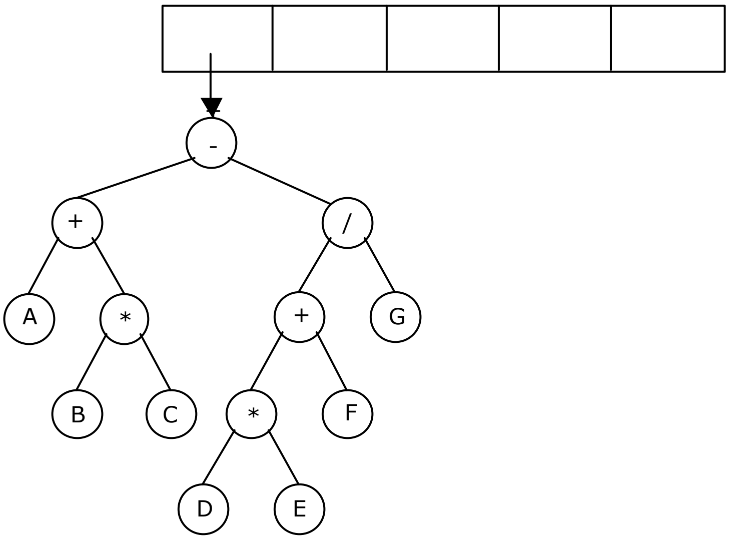
Next, D and E are read, and for each one–node tree is created and a pointer to the corresponding tree is pushed onto the stack.



Continuing, a ‘*’ is read, so we pop two tree pointers and form a new tree with a ‘*’ as root.



Proceeding similar to the previous steps, finally, when the last symbol is read, the expression tree is as follows:



5.4.1. Converting expressions with expression trees:

Let us convert the following expressions from one type to another. These can be as follows:

1. Postfix to infix
2. Postfix to prefix
3. Prefix to infix
4. Prefix to postfix

1. Postfix to Infix:

The following algorithm works for the expressions whose infix form does not require parenthesis to override conventional precedence of operators.

A. Create the expression tree from the postfix expression
B. Run inorder traversal on the tree.

2. Postfix to Prefix:

The following algorithm works for the expressions to convert postfix to prefix:

A. Create the expression tree from the postfix expression
B. Run preorder traversal on the tree.

3. Prefix to Infix:

The following algorithm works for the expressions whose infix form does not require parenthesis to override conventional precedence of operators.

A. Create the expression tree from the prefix expression
B. Run inorder traversal on the tree.

4. Prefix to postfix:

The following algorithm works for the expressions to convert postfix to prefix:

A. Create the expression tree from the prefix expression
B. Run postorder traversal on the tree.

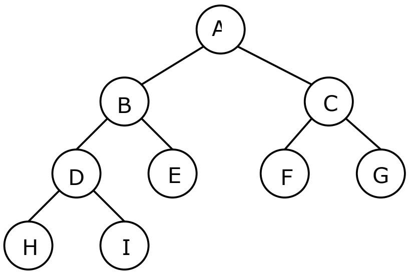
5.5. Threaded Binary Tree:

The linked representation of any binary tree has more null links than actual pointers. If there are 2n total links, there are n+1 null links. A clever way to make use of these null links has been devised by A.J. Perlis and C. Thornton.

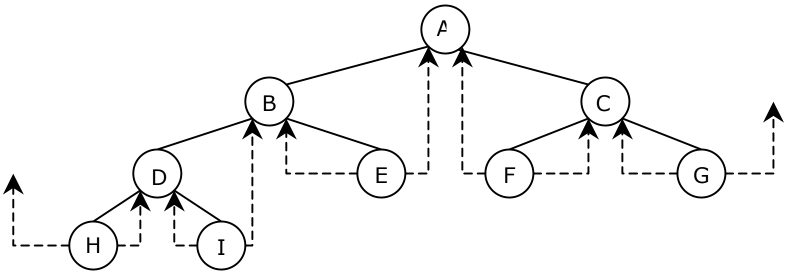
Their idea is to replace the null links by pointers called Threads to other nodes in the tree.

If the RCHILD(p) is normally equal to zero, we will replace it by a pointer to the node which would be printed after P when traversing the tree in inorder.

A null LCHILD link at node P is replaced by a pointer to the node which immediately precedes node P in inorder. For example, Let us consider the tree:



The Threaded Tree corresponding to the above tree is:



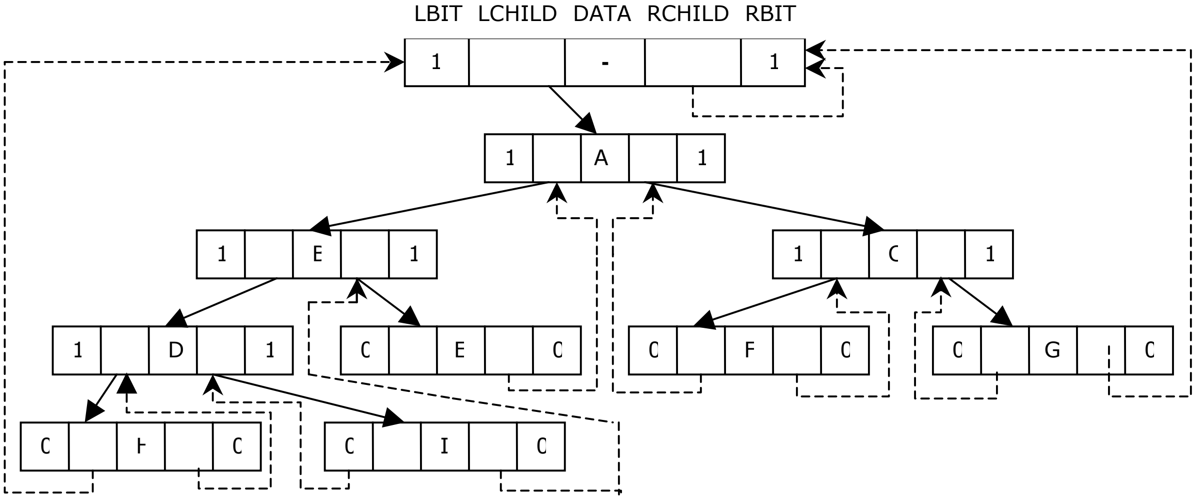
The tree has 9 nodes and 10 null links which have been replaced by Threads. If we traverse T in inorder the nodes will be visited in the order H D I B E A F C G.

For example, node 'E' has a predecessor Thread which points to 'B' and a successor Thread which points to 'A'. In memory representation Threads and normal pointers are distinguished between as by adding two extra one bit fields LBIT and RBIT.

LBIT(P) = 1 if LCHILD(P) is a normal pointer
LBIT(P) = 0 if LCHILD(P) is a Thread

RBIT(P) = 1 if RCHILD(P) is a normal pointer
RBIT(P) = 0 if RCHILD(P) is a Thread

In the above figure two threads have been left dangling in LCHILD(H) and RCHILD(G). In order to have no loose Threads we will assume a head node for all threaded binary trees. The Complete memory representation for the tree is as follows. The tree T is the left sub-tree of the head node.



5.6. Binary Search Tree:

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

1. Every element has a key and no two elements have the same key.
2. The keys in the left subtree are smaller than the key in the root.
3. The keys in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees.

Figure 5.2.5(a) is a binary search tree, whereas figure 5.2.5(b) is not a binary search tree.

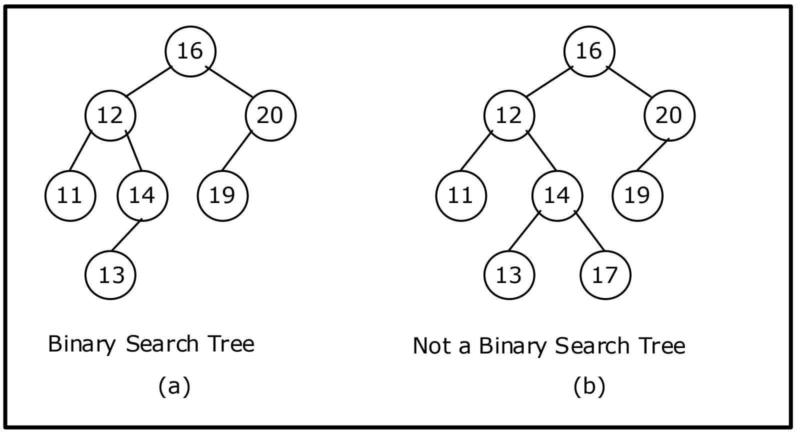


Figure 5.2.5. Examples of binary search trees

5.7. General Trees (m-ary tree):

If in a tree, the outdegree of every node is less than or equal to ***m***, the tree is called general tree. The general tree is also called as an *m-ary* tree. If the outdegree of every node is exactly equal to m or zero then the tree is called *a full or complete m-ary* tree. For m = 2, the trees are called *binary* and *full binary trees.*

Differences between trees and binary trees:

TREE	BINARY TREE
Each element in a tree can have any number of subtrees.	Each element in a binary tree has at most two subtrees.
The subtrees in a tree are unordered.	The subtrees of each element in a binary tree are ordered (i.e. we distinguish between left and right subtrees).

5.7.1. Converting a *m-ary* tree (general tree) to a binary tree:

There is a one-to-one mapping between general ordered trees and binary trees. So, every tree can be uniquely represented by a binary tree. Furthermore, a forest can also be represented by a binary tree.

Conversion from general tree to binary can be done in two stages.

Stage 1:

- As a first step, we delete all the branches originating in every node except the left most branch.

- We draw edges from a node to the node on the right, if any, which is situated at the same level.

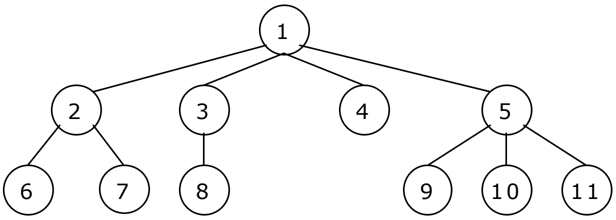
Stage 2:

- Once this is done then for any particular node, we choose its left and right sons in the following manner:

    - The left son is the node, which is immediately below the given node, and the right son is the node to the immediate right of the given node on the same horizontal line. Such a binary tree will not have a right subtree.

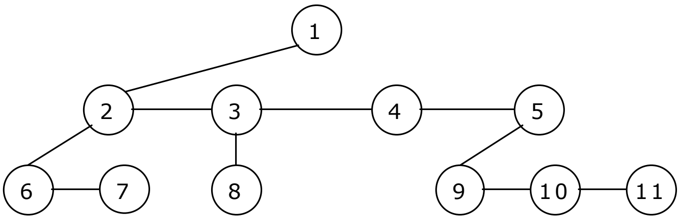
Example 1:

Convert the following ordered tree into a binary tree:

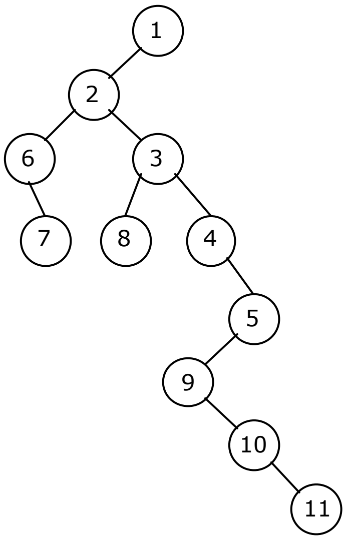


Solution:

Stage 1 tree by using the above mentioned procedure is as follows:

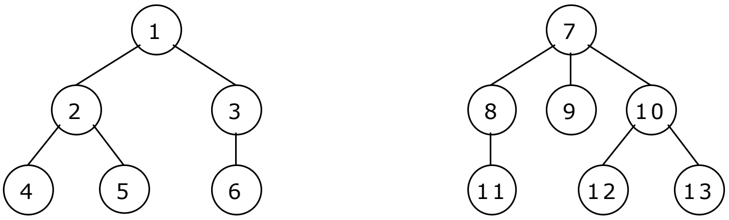


Stage 2 tree by using the above mentioned procedure is as follows:



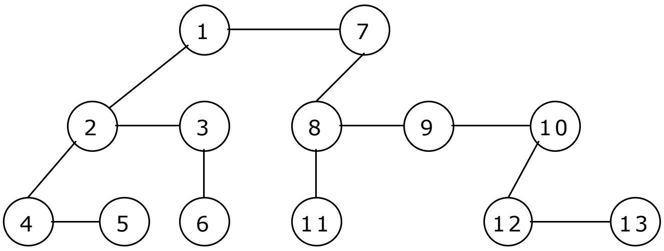
Example 2:

Construct a unique binary tree from the given forest.

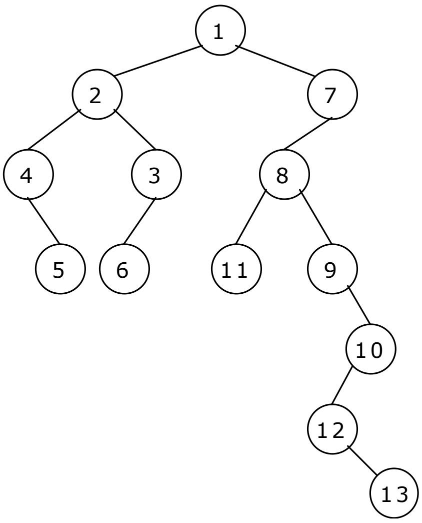


Solution:

Stage 1 tree by using the above mentioned procedure is as follows:



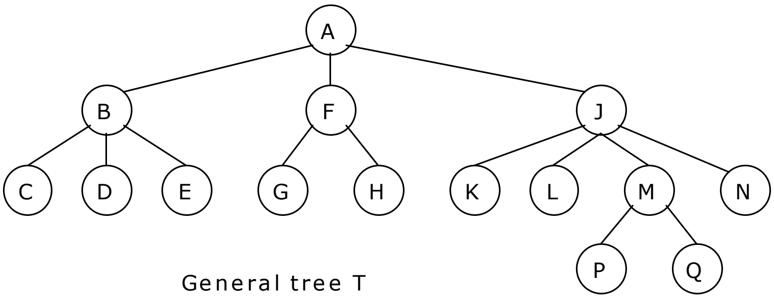
Stage 2 tree by using the above mentioned procedure is as follows (binary tree representation of forest):



Example 3:

For the general tree shown below:

1. Find the corresponding binary tree T′.
2. Find the preorder traversal and the postorder traversal of T.
3. Find the preorder, inorder and postorder traversals of T′.
4. Compare them with the preorder and postorder traversals obtained for T′ with the general tree T.

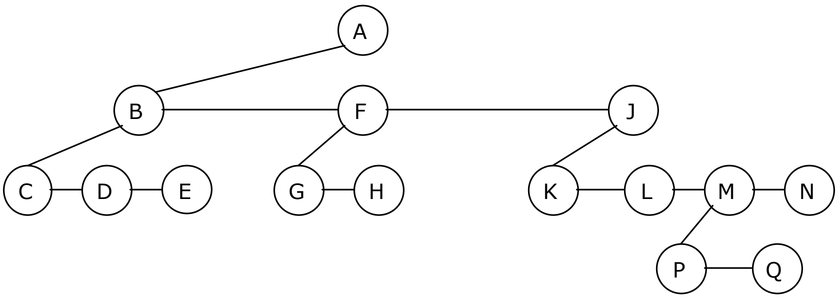


General tree T

Solution:

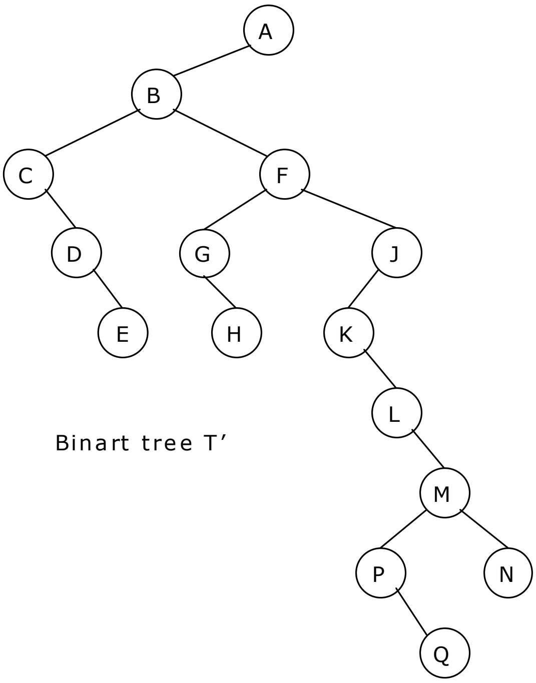
1. Stage 1:

The tree by using the above-mentioned procedure is as follows:



Stage 2:

The binary tree by using the above-mentioned procedure is as follows:



Binart tree T'

2. Suppose T is a general tree with root R and subtrees T_1, T_2, ………., T_M. The preorder traversal and the postorder traversal of T are:

Preorder: 1) Process the root R.
2) Traverse the subtree T_1, T_2, ……., T_M in preorder.

Postorder: 1) Traverse the subtree T_1, T_2, ……., T_M in postorder.
2) Process the root R.

The tree T has the root A and subtrees T_1, T_2 and T_3 such that:

T_1 consists of nodes B, C, D and E.

T_2 consists of nodes F, G and H.

T_3 consists of nodes J, K, L, M, N, P and Q.

A. The preorder traversal of T consists of the following steps:

(i) Process root A.

(ii) Traverse T_1 in preorder: Process nodes B, C, D, E.

(iii) Traverse T_2 in preorder: Process nodes F, G, H.

(iv) Traverse T_3 in preorder: Process nodes J, K, L, M, P, Q, N.

The preorder traversal of T is as follows:
A, B, C, D, E, F, G, H, J, K, L, M, P, Q, N

B. The postorder traversal of T consists of the following steps:

(i) Traverse T_1 in postorder: Process nodes C, D, E, B.

(ii) Traverse T_2 in postorder: Process nodes G, H, F.

(iii) Traverse T_3 in postorder: Process nodes K, L, P, Q, M, N, J.

(iv) Process root A.

The postorder traversal of T is as follows:

C, D, E, B, G, H, F, K, L, P, Q, M, N, J, A

3. The preorder, inorder and postorder traversals of the binary tree T' are as follows:

Preorder:	A, B, C, D, E, F, G, H, J, K, M, P, Q, N
Inorder:	C, D, E, B, G, H, F, K, L, P, Q, M, N, J, A
Postorder:	E, D, C, H, G, Q, P, N, M, L, K, J, F, B, A

4. Comparing the preorder and postorder traversals of T' with the general tree T:

We can observer that the preorder of the binary tree T' is identical to the preorder of the general T.

The inorder traversal of the binary tree T' is identical to the postorder traversal of the general tree T.

There is no natural traversal of the general tree T which corresponds to the postorder traversal of its corresponding binary tree T'.

5.8. Search and Traversal Techniques for m-ary trees:

Search involves visiting nodes in a tree in a systematic manner, and may or may not result into a visit to all nodes. When the search necessarily involved the examination of every vertex in the tree, it is called the traversal. Traversing of a tree can be done in two ways.

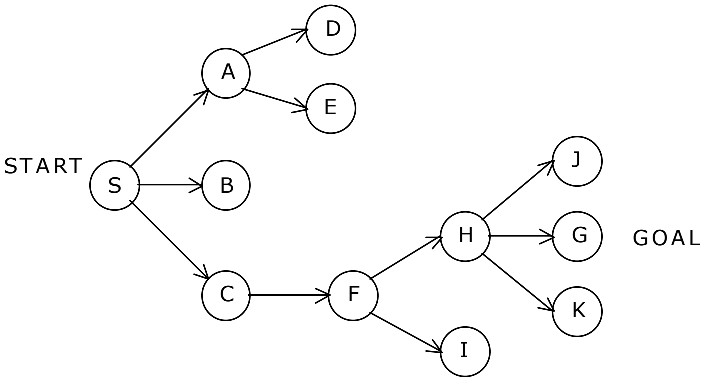
1. Depth first search or traversal.
2. Breadth first search or traversal.

5.8.1. Depth first search:

In Depth first search, we begin with root as a start state, then some successor of the start state, then some successor of that state, then some successor of that and so on, trying to reach a goal state. One simple way to implement depth first search is to use a stack data structure consisting of root node as a start state.

If depth first search reaches a state S without successors, or if all the successors of a state S have been chosen (visited) and a goal state has not get been found, then it "backs up" that means it goes to the immediately previous state or predecessor formally, the state whose successor was 'S' originally.

To illustrate this let us consider the tree shown below.



Suppose S is the start and G is the only goal state. Depth first search will first visit S, then A, then D. But D has no successors, so we must back up to A and try its second successor, E. But this doesn't have any successors either, so we back up to A again. But now we have tried all the successors of A and haven't found the goal state G so we must back to 'S'. Now 'S' has a second successor, B. But B has no successors, so we back up to S again and choose its third successor, C. C has one successor, F. The first successor of F is H, and the first of H is J. J doesn't have any successors, so we back up to H and try its second successor. And that's G, the only goal state.

So the solution path to the goal is S, C, F, H and G and the states considered were in order S, A, D, E, B, C, F, H, J, G.

Disadvantages:

1. It works very fine when search graphs are trees or lattices, but can get struck in an infinite loop on graphs. This is because depth first search can travel around a cycle in the graph forever.

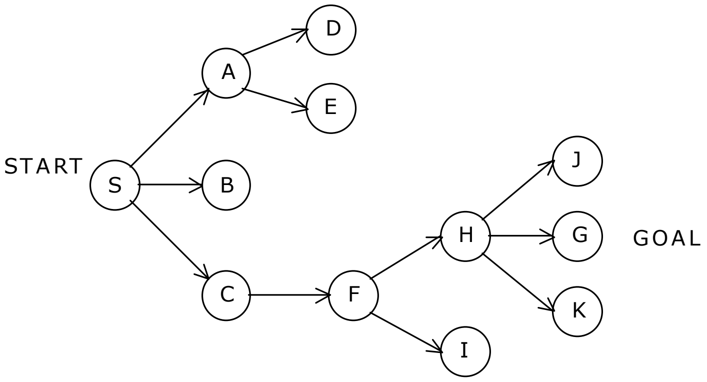
   To eliminate this keep a list of states previously visited, and never permit search to return to any of them.

2. We cannot come up with shortest solution to the problem.

5.8.2. Breadth first search:

Breadth-first search starts at root node S and "discovers" which vertices are reachable from S. Breadth-first search discovers vertices in increasing order of distance. Breadth-first search is named because it visits vertices across the entire breadth.

To illustrate this let us consider the following tree:



Breadth first search finds states level by level. Here we first check all the immediate successors of the start state. Then all the immediate successors of these, then all the immediate successors of these, and so on until we find a goal node. Suppose S is the start state and G is the goal state. In the figure, start state S is at level 0; A, B and C are at level 1; D, e and F at level 2; H and I at level 3; and J, G and K at level 4.

So breadth first search, will consider in order S, A, B, C, D, E, F, H, I, J and G and then stop because it has reached the goal node.

Breadth first search does not have the danger of infinite loops as we consider states in order of increasing number of branches (level) from the start state.

One simple way to implement breadth first search is to use a queue data structure consisting of just a start state.

5.9. Sparse Matrices:

A sparse matrix is a two–dimensional array having the value of majority elements as null. The density of the matrix is the number of non-zero elements divided by the total number of matrix elements. The matrices with very low density are often good for use of the sparse format. For example,

$$A = \begin{pmatrix} 0 & 0 & 0 & 5 \\ 0 & 2 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{pmatrix}$$

As far as the storage of a sparse matrix is concerned, storing of null elements is nothing but wastage of memory. So we should devise technique such that only non-null elements will be stored. The matrix A produces:

```
        (3, 1)  1
        (2, 2)  2
S =     (3, 2)  3
        (4, 3)  4
        (1, 4)  5
```

The printed output lists the non-zero elements of S, together with their row and column indices. The elements are sorted by columns, reflecting the internal data structure.
In large number of applications, sparse matrices are involved. One approach is to use the linked list.

The program to represent sparse matrix:

```
/*	Check whether the given matrix is sparse matrix or not, if so then print in
	alternative form for storage.		*/

# include <stdio.h>
# include <conio.h>

main()
{
	int matrix[20][20], m, n, total_elements, total_zeros = 0, i, j;
	clrscr();
	printf("\n Enter Number of rows and columns: ");
	scanf("%d %d",&m, &n);
	total_elements = m * n;
	printf("\n Enter data for sparse matrix: ");
	for(i = 0; i < m ; i++)
	{
		for( j = 0; j < n ; j++)
		{
			scanf("%d", &matrix[i][j]);
			if( matrix[i][j] == 0)
			{
				total_zeros++;
			}
		}
	}
	if(total_zeros > total_elements/2 )
	{
		printf("\n Given Matrix is Sparse Matrix..");
		printf("\n The Representaion of Sparse Matrix is: \n");
		printf("\n Row \t Col \t Value ");
		for(i = 0; i < m ; i++)
		{
			for( j = 0; j < n ; j++)
			{
				if( matrix[i][j] != 0)
				{
					printf("\n %d \t %d \t %d",i,j,matrix[i][j]);
				}
			}
		}
	}
	else
		printf("\n Given Matrix is Not a Sparse Matrix..");
}
```

EXCERCISES

1. How many different binary trees can be made from three nodes that contain the key value 1, 2, and 3?

2. a. Draw all the possible binary trees that have four leaves and all the nonleaf nodes have no children.

   b. Show what would be printed by each of the following.
   An inorder traversal of the tree
   A postorder traversal of the tree
   A preorder traversal of the tree

3. a. Draw the binary search tree whose elements are inserted in the following order:
   50 72 96 94 107 26 12 11 9 2 10 25 51 16 17 95

   b. What is the height of the tree?
   c. What nodes are on level?
   d. Which levels have the maximum number of nodes that they could contain?
   e. What is the maximum height of a binary search tree containing these nodes? Draw such a tree?
   f. What is the minimum height of a binary search tree containing these nodes? Draw such a tree?

   g. Show how the tree would look after the deletion of 29, 59 and 47?

   h. Show how the (original) tree would look after the insertion of nodes containing 63, 77, 76, 48, 9 and 10 (in that order).

4. Write a "C" function to determine the height of a binary tree.

5. Write a "C" function to count the number of leaf nodes in a binary tree.

6. Write a "C" function to swap a binary tree.

7. Write a "C" function to compute the maximum number of nodes in any level of a binary tree. The maximum number of nodes in any level of a binary tree is also called the width of the tree.

8. Construct two binary trees so that their postorder traversal sequences are the same.

9. Write a "C" function to compute the internal path length of a binary tree.

10. Write a "C" function to compute the external path length of a binary tree.

11. Prove that every node in a tree except the root node has a unique parent.

12. Write a "C" function to reconstruct a binary tree from its preorder and inorder traversal sequences.

13. Prove that the inorder and postorder traversal sequences of a binary tree uniquely characterize the binary tree. Write a "C" function to reconstruct a binary tree from its postorder and inorder traversal sequences.

14. Build the binary tree from the given traversal techniques:

A.	Inorder:	g d h b e i a f j c
	Preorder:	a b d g h e i c f j
B.	Inorder:	g d h b e i a f j c
	Postorder:	g h d i e b j f c a
C.	Inorder:	g d h b e i a f j c
	Level order:	a b c d e f g h i j

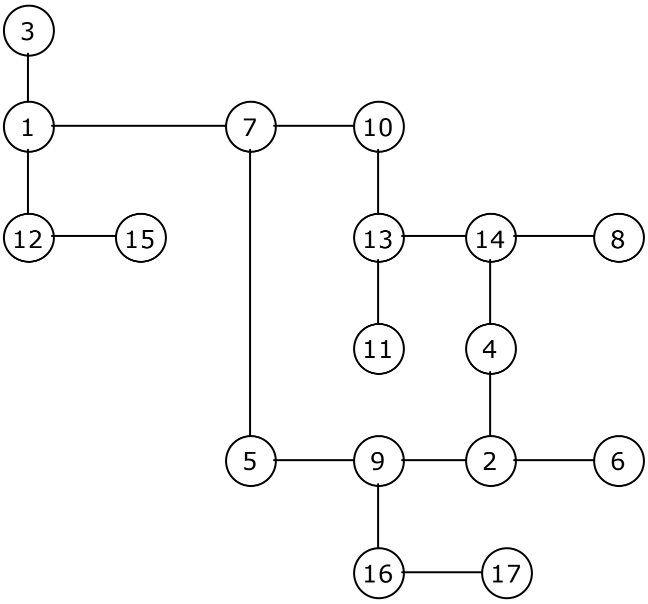
15. Build the binary tree from the given traversal techniques:

A.	Inorder:	n1 n2 n3 n4 n5 n6 n7 n8 n9
	Preorder:	n6 n2 n1 n4 n3 n5 n9 n7 n8
B.	Inorder:	n1 n2 n3 n4 n5 n6 n7 n8 n9
	Postorder:	n1 n3 n5 n4 n2 n8 n7 n9 n6
C.	Inorder:	n1 n2 n3 n4 n5 n6 n7 n8 n9
	Level order:	n6 n2 n9 n1 n4 n7 n3 n5 n8

16. Build the binary tree for the given inorder and preorder traversals:

Inorder:	E A C K F H D B G
Preorder:	F A E K C D H G B

17. Convert the following general tree represented as a binary tree:



Chapter 6

Graphs

6.1. Introduction to Graphs:

Graph G is a pair (V, E), where V is a finite set of vertices and E is a finite set of edges. We will often denote n = |V|, e = |E|.

A graph is generally displayed as figure 6.5.1, in which the vertices are represented by circles and the edges by lines.

An edge with an orientation (i.e., arrow head) is a directed edge, while an edge with no orientation is our undirected edge.

If all the edges in a graph are undirected, then the graph is an undirected graph. The graph in figure 6.5.1(a) is an undirected graph. If all the edges are directed; then the graph is a directed graph. The graph of figure 6.5.1(b) is a directed graph. A directed graph is also called as digraph. A graph G is connected if and only if there is a simple path between any two nodes in G.

A graph G is said to be complete if every node a in G is adjacent to every other node v in G. A complete graph with n nodes will have n(n-1)/2 edges. For example, Figure 6.5.1.(a) and figure 6.5.1.(d) are complete graphs.

A directed graph G is said to be connected, or strongly connected, if for each pair (u, v) for nodes in G there is a path from u to v and also a path from v to u. On the other hand, G is said to be unilaterally connected if for any pair (u, v) of nodes in G there is a path from u to v or a path from v to u. For example, the digraph shown in figure 6.5.1 (e) is strongly connected.

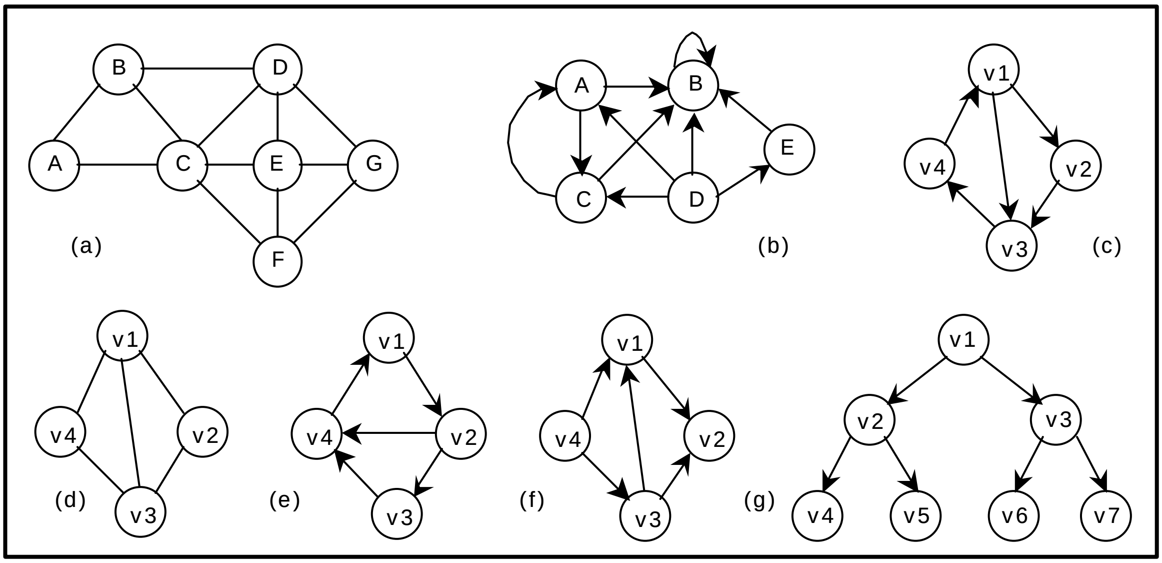


Figure 6.5.1 Various Graphs

We can assign weight function to the edges: $w_G(e)$ is a weight of edge e $\in$ E. The graph which has such function assigned is called weighted graph.

The number of incoming edges to a vertex v is called in–degree of the vertex (denote indeg(v)). The number of outgoing edges from a vertex is called out-degree (denote outdeg(v)). For example, let us consider the digraph shown in figure 6.5.1(f),

indegree(v_1) = 2 outdegree(v_1) = 1

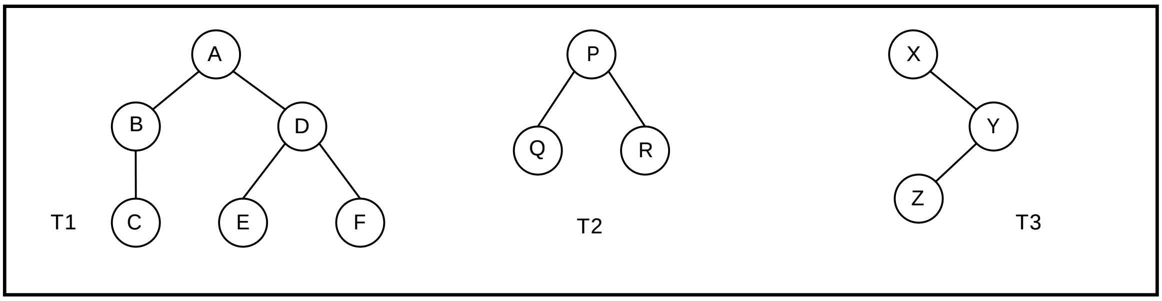
indegree(v_2) = 2 outdegree(v_2) = 0

A path is a sequence of vertices (v_1, v_2, ... , v_k), where for all i, (v_i, v_{i+1}) ε E. A path is simple if all vertices in the path are distinct. If there is a path containing one or more edges which starts from a vertex V_i and terminates into the same vertex then the path is known as a cycle. For example, there is a cycle in figure 6.5.1(a), figure 6.5.1(c) and figure 6.5.1(d).

If a graph (digraph) does not have any cycle then it is called **acyclic graph**. For example, the graphs of figure 6.5.1 (f) and figure 6.5.1 (g) are acyclic graphs.

A graph G' = (V', E') is a sub-graph of graph G = (V, E) iff V' $\subseteq$ V and E' $\subseteq$ E.

A **Forest** is a set of disjoint trees. If we remove the root node of a given tree then it becomes forest. The following figure shows a forest F that consists of three trees T1, T2 and T3.



A Forest F

A graph that has either self loop or parallel edges or both is called **multi-graph**.

Tree is a connected acyclic graph (there aren't any sequences of edges that go around in a loop). A spanning tree of a graph G = (V, E) is a tree that contains all vertices of V and is a subgraph of G. A single graph can have multiple spanning trees.

Let T be a spanning tree of a graph G. Then

1. *Any two vertices in T are connected by a unique simple path.*

2. *If any edge is removed from T, then T becomes disconnected.*

3. *If we add any edge into T, then the new graph will contain a cycle.*

4. *Number of edges in T is n-1.*

6.2. Representation of Graphs:

There are two ways of representing digraphs. They are:

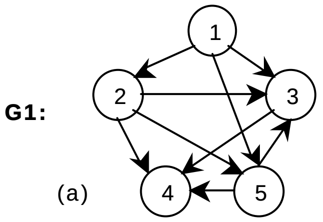
- Adjacency matrix.
- Adjacency List.
- Incidence matrix.

Adjacency matrix:

In this representation, the adjacency matrix of a graph G is a two dimensional n x n matrix, say A = ($a_{i,j}$), where

$$a_{i,\,j} = \begin{cases} 1 & \textit{if there is an edge from } v_i \textit{ to } v_j \\ 0 & \textit{otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed. This matrix is also called as Boolean matrix or bit matrix.



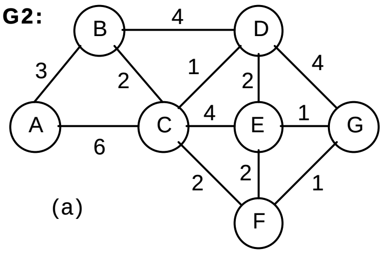
(b)

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

Figure 6.5.2. A graph and its Adjacency matrix

Figure 6.5.2(b) shows the adjacency matrix representation of the graph G1 shown in figure 6.5.2(a). The adjacency matrix is also useful to store multigraph as well as weighted graph. In case of multigraph representation, instead of entry 0 or 1, the entry will be between number of edges between two vertices.

In case of weighted graph, the entries are weights of the edges between the vertices. The adjacency matrix for a weighted graph is called as cost adjacency matrix. Figure 6.5.3(b) shows the cost adjacency matrix representation of the graph G2 shown in figure 6.5.3(a).



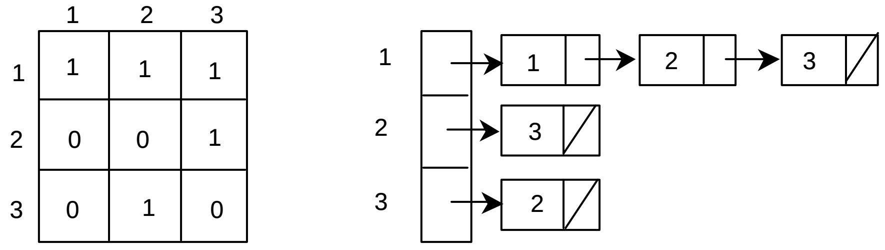
(b)

	A	B	C	D	E	F	G
A	0	3	6	∝	∝	∝	∝
B	3	0	2	4	∝	∝	∝
C	6	2	0	1	4	2	∝
D	∝	4	1	0	2	∝	4
E	∝	∝	4	2	0	2	1
F	∝	∝	2	∝	2	0	1
G	∝	∝	∝	4	1	1	0

Figure 6.5.3 Weighted graph and its Cost adjacency matrix

Adjacency List:

In this representation, the n rows of the adjacency matrix are represented as n linked lists. An array Adj[1, 2, ... n] of pointers where for 1 $\leq$ v $\leq$ n, Adj[v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements. For the graph G in figure 6.5.4(a), the adjacency list in shown in figure 6.5.4 (b).



	1	2	3
1	1	1	1
2	0	0	1
3	0	1	0

(a) Adjacency Matrix (b) Adjacency List

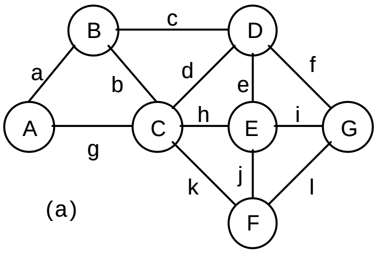
Figure 6.5.4 Adjacency matrix and adjacency list

Incidence Matrix:

In this representation, if G is a graph with n vertices, e edges and no self loops, then incidence matrix A is defined as an n by e matrix, say A = ($a_{i,j}$), where

$$a_{i,\ j} = \begin{cases} 1 & \textit{if there is an edge } j \textit{ incident to } v_i \\ 0 & \textit{otherwise} \end{cases}$$

Here, n rows correspond to n vertices and e columns correspond to e edges. Such a matrix is called as vertex-edge incidence matrix or simply incidence matrix.



(a)

(b)

	a	b	c	d	e	f	g	h	i	j	k	l
A	1	0	0	0	0	0	1	0	0	0	0	0
B	1	1	1	0	0	0	0	0	0	0	0	0
C	0	1	0	1	0	0	1	1	0	0	1	0
D	0	0	1	1	1	1	0	0	0	0	0	0
E	0	0	0	0	1	0	0	1	1	1	0	0
F	0	0	0	0	0	0	0	0	0	1	1	1
G	0	0	0	0	0	1	0	0	1	0	0	1

Figure 6.5.4 Graph and its incidence matrix

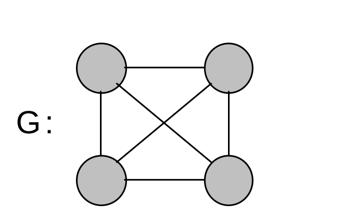
Figure 6.5.4(b) shows the incidence matrix representation of the graph G1 shown in figure 6.5.4(a).

6.3. Minimum Spanning Tree (MST):

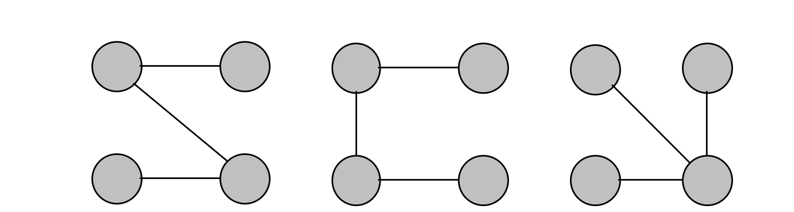
A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree w(T) is the sum of weights of all edges in T. Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.

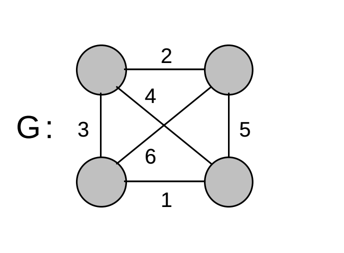
Example:



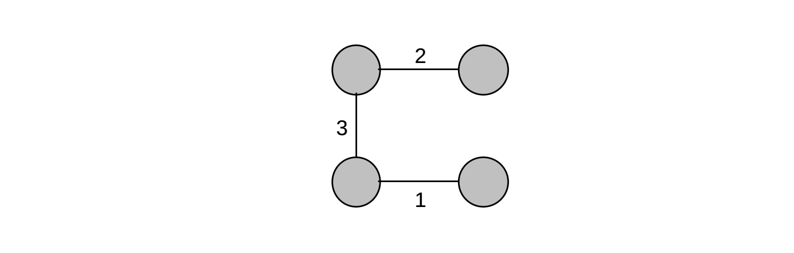
A graph G:



Three (of many possible) spanning trees from graph G:



A weighted graph G:



The minimal spanning tree from weighted graph G:

Let's consider a couple of real-world examples on minimum spanning tree:

- One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.

- Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

Minimum spanning tree, can be constructed using any of the following two algorithms:

1. Kruskal’s algorithm and
2. Prim’s algorithm.

Both algorithms differ in their methodology, but both eventually end up with the MST. *Kruskal's algorithm uses edges, and Prim’s algorithm uses vertex connections* in determining the MST. In *Prim’s algorithm at any instance of output it represents tree* whereas in *Kruskal’s algorithm at any instance of output it may represent tree or not*.

6.3.1. Kruskal’s Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost.

The order in which the edges are chosen, in this case, does not matter. Different MST’s may result, but they will all have the same total cost, which will always be the minimum cost.

Kruskal’s Algorithm for minimal spanning tree is as follows:

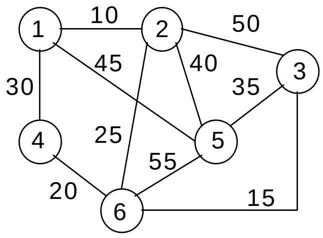
1. Make the tree T empty.
2. Repeat the steps 3, 4 and 5 as long as T contains less than n - 1 edges and E is not empty otherwise, proceed to step 6.
3. Choose an edge (v, w) from E of lowest cost.
4. Delete (v, w) from E.
5. If (v, w) does not create a cycle in T

   then Add (v, w) to T

   else discard (v, w)
6. If T contains fewer than n - 1 edges then print no spanning tree.

Example 1:

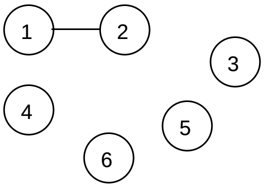
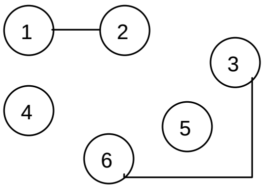
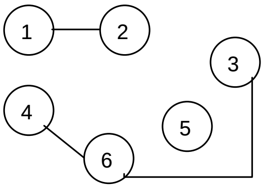
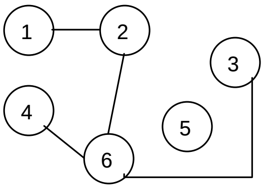
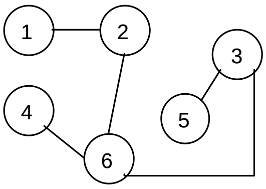
Construct the minimal spanning tree for the graph shown below:



Arrange all the edges in the increasing order of their costs:

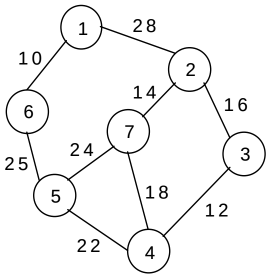
Cost	***10***	***15***	***20***	***25***	***30***	***35***	***40***	***45***	***50***	***55***
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.
(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.
(2, 6)	25		The edge between vertices 2 and 6 is considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 105.

Example 2:

Construct the minimal spanning tree for the graph shown below:

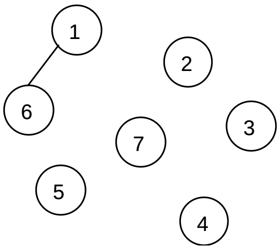
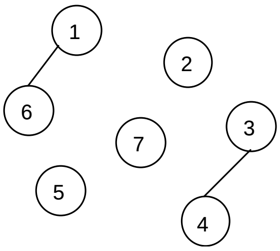
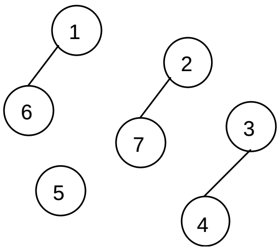


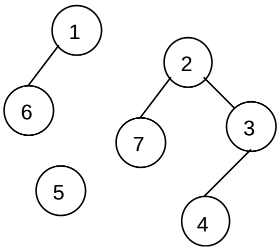
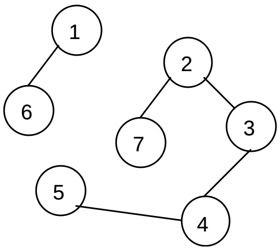
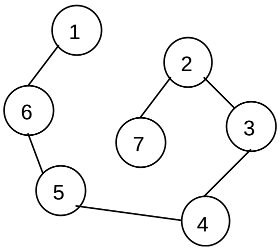
Solution:

Arrange all the edges in the increasing order of their costs:

Cost	***10***	***12***	***14***	***16***	***18***	***22***	***24***	***25***	***28***
Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 6)	10		The edge between vertices 1 and 6 is the first edge selected. It is included in the spanning tree.
(3, 4)	12		Next, the edge between vertices 3 and 4 is selected and included in the tree.
(2, 7)	14		The edge between vertices 2 and 7 is next included in the tree.

(2, 3)	16		The edge between vertices 2 and 3 is next included in the tree.
(4, 7)	18	Reject	The edge between the vertices 4 and 7 is discarded as its inclusion creates a cycle.
(4, 5)	22		The edge between vertices 4 and 7 is considered next and included in the tree.
(5, 7)	24	Reject	The edge between the vertices 5 and 7 is discarded as its inclusion creates a cycle.
(5, 6)	25		Finally, the edge between vertices 5 and 6 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 99.

6.3.2. MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree. Prim's algorithm is an example of a greedy algorithm.

Prim's Algorithm:

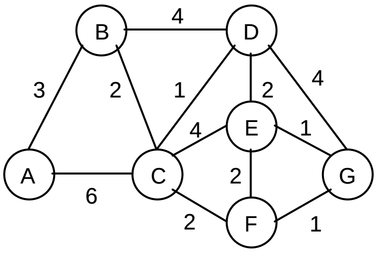
E is the set of edges in G. cost [1:n, 1:n] is the cost adjacency matrix of an n vertex graph such that cost [i, j] is either a positive real number or ∝ if no edge (i, j) exists. A minimum spanning tree is computed and stored as a set of edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in the minimum-cost spanning tree. The final cost is returned.

Algorithm Prim (E, cost, n, t)

```
{
        Let (k, l) be an edge of minimum cost in E;
        mincost := cost [k, l];
        t [1, 1] := k; t [1, 2] := l;
        for  i :=1 to n do                          // Initialize near
                if  (cost [i, l] < cost [i, k]) then near [i] := l;
                else near [i] := k;
        near [k] :=near [l] := 0;
        for  i:=2 to n - 1 do                       // Find n - 2 additional edges for t.
        {
                Let j be an index such that near [j] ≠ 0 and
                cost [j, near [j]] is minimum;
                t [i, 1] := j; t [i, 2] := near [j];
                mincost := mincost + cost [j, near [j]];
                near [j] := 0
                for  k:= 1 to n do                          // Update near[].
                        if ((near [k]  ≠ 0) and (cost [k, near [k]] > cost [k, j]))
                                then near [k] := j;
        }
        return mincost;
}
```

EXAMPLE:

Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.



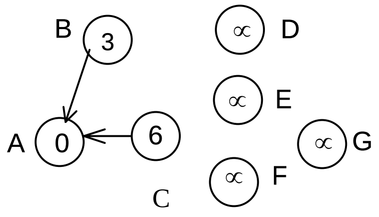
Solution:

The cost adjacency matrix is

$$\begin{pmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{pmatrix}$$

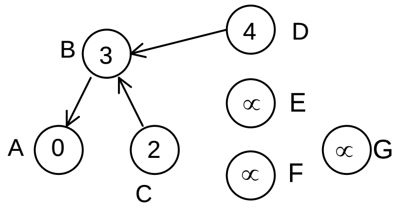
The stepwise progress of the prim's algorithm is as follows:

Step 1:



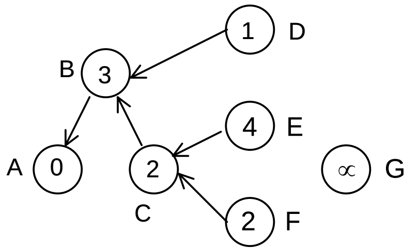
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∝	∝	∝	∝
Next	*	A	A	A	A	A	A

Step 2:



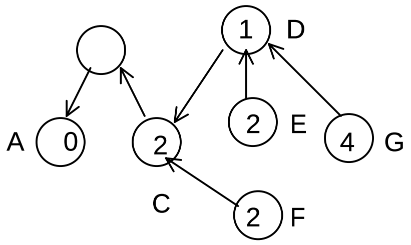
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	∝	∝	∝
Next	*	A	B	B	A	A	A

Step 3:



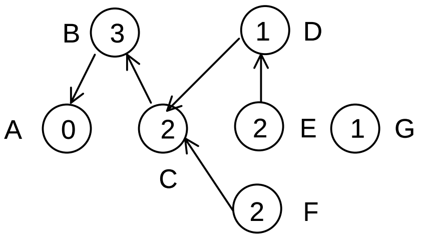
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	∝
Next	*	A	B	C	C	C	A

Step 4:



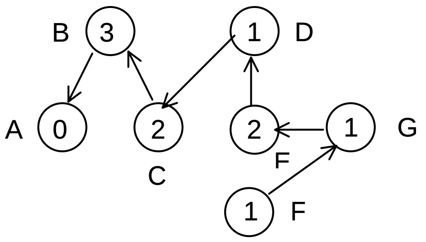
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step 5:



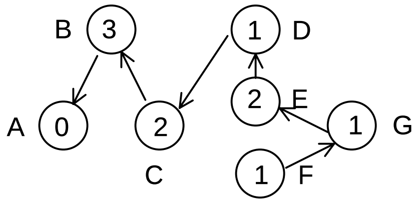
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

6.4. **Reachability Matrix** (Warshall's Algorithm):

Warshall's algorithm requires knowing which edges exist and which does not. It doesn't need to know the lengths of the edges in the given directed graph. This information is conveniently displayed by adjacency matrix for the graph, in which a '1' indicates the existence of an edge and '0' indicates non-existence.

Adjacency Matrix
Warshall's Algorithm
All Pairs Rechability Matrix

It begins with the adjacency matrix for the given graph, which is called A_0, and then updates the matrix 'n' times, producing matrices called A_1, A_2, ... , A_n and then stops.

In warshall's algorithm the matrix A_i contains information about the existence of i–paths. A one entry in the matrix A_i will correspond to the existence of i–paths and zero entry will correspond to non-existence. Thus when the algorithm stops, the final matrix A_n, contains the desired connectivity information.

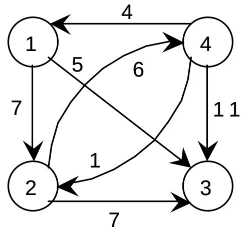
A one entry indicates a pair of vertices, which are connected and zero entry indicates a pair, which are not. This matrix is called a *reachability matrix or path matrix* for the graph. It is also called the *transitive closure* of the original adjacency matrix.

The update rule for computing A_i from A_{i-1} in warshall's algorithm is:

$$A_i [x, y] = A_{i-1} [x, y] \vee (A_{i-1} [x, i] \wedge A_{i-1} [i, y]) \qquad ---- \qquad (1)$$

Example 1:

Use warshall's algorithm to calculate the reachability matrix for the graph:



We begin with the adjacency matrix of the graph 'A_0'

$$A_0 = \begin{array}{c@{}c} \begin{array}{c}1\\2\\3\\4\end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

The first step is to compute 'A_1' matrix. To do so we will use the updating rule – (1).

Before doing so, we notice that only one entry in A_0 must remain one in A_1, since in Boolean algebra 1 + (anything) = 1. Since these are only nine zero entries in A_0, there are only nine entries in A_0 that need to be updated.

$A_1[1, 1] = A_0[1, 1] \vee (A_0[1, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0$

$A_1[1, 4] = A_0[1, 4] \vee (A_0[1, 1] \wedge A_0[1, 4]) = 0 \vee (0 \wedge 0) = 0$

$A_1[2, 1] = A_0[2, 1] \vee (A_0[2, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0$

$A_1[2, 2] = A_0[2, 2] \vee (A_0[2, 1] \wedge A_0[1, 2]) = 0 \vee (0 \wedge 1) = 0$

$A_1[3, 1] = A_0[3, 1] \vee (A_0[3, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0$

$A_1[3, 2] = A_0[3, 2] \vee (A_0[3, 1] \wedge A_0[1, 2]) = 0 \vee (0 \wedge 1) = 0$

$A_1[3, 3] = A_0[3, 3] \vee (A_0[3, 1] \wedge A_0[1, 3]) = 0 \vee (0 \wedge 1) = 0$

$A_1[3, 4] = A_0[3, 4] \vee (A_0[3, 1] \wedge A_0[1, 4]) = 0 \vee (0 \wedge 0) = 0$

$A_1[4, 4] = A_0[4, 4] \vee (A_0[4, 1] \wedge A_0[1, 4]) = 0 \vee (1 \wedge 0) = 0$

$$A_1 = \begin{array}{c@{}c} \begin{array}{c}1\\2\\3\\4\end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Next, A_2 must be calculated from A_1; but again we need to update the 0 entries,

$A_2[1, 1] = A_1[1, 1] \vee (A_1[1, 2] \wedge A_1[2, 1]) = 0 \vee (1 \wedge 0) = 0$

$A_2[1, 4] = A_1[1, 4] \vee (A_1[1, 2] \wedge A_1[2, 4]) = 0 \vee (1 \wedge 1) = 1$

$A_2[2, 1] = A_1[2, 1] \vee (A_1[2, 2] \wedge A_1[2, 1]) = 0 \vee (0 \wedge 0) = 0$

$A_2[2, 2] = A_1[2, 2] \vee (A_1[2, 2] \wedge A_1[2, 2]) = 0 \vee (0 \wedge 0) = 0$

$A_2[3, 1] = A_1[3, 1] \vee (A_1[3, 2] \wedge A_1[2, 1]) = 0 \vee (0 \wedge 0) = 0$

$A_2[3, 2] = A_1[3, 2] \vee (A_1[3, 2] \wedge A_1[2, 2]) = 0 \vee (0 \wedge 0) = 0$

$A_2[3, 3] = A_1[3, 3] \lor (A_1[3, 2] \land A_1[2, 3]) = 0 \lor (0 \land 1) = 0$

$A_2[3, 4] = A_1[3, 4] \lor (A_1[3, 2] \land A_1[2, 4]) = 0 \lor (0 \land 1) = 0$

$A_2[4, 4] = A_1[4, 4] \lor (A_1[4, 2] \land A_1[2, 4]) = 0 \lor (1 \land 1) = 1$

$$A_2 = \begin{array}{cc} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

This matrix has only seven 0 entries, and so to compute A_3, we need to do only seven computations.

$A_3[1, 1] = A_2[1, 1] \lor (A_2[1, 3] \land A_2[3, 1]) = 0 \lor (1 \land 0) = 0$

$A_3[2, 1] = A_2[2, 1] \lor (A_2[2, 3] \land A_2[3, 1]) = 0 \lor (1 \land 0) = 0$

$A_3[2, 2] = A_2[2, 2] \lor (A_2[2, 3] \land A_2[3, 2]) = 0 \lor (1 \land 0) = 0$

$A_3[3, 1] = A_2[3, 1] \lor (A_2[3, 3] \land A_2[3, 1]) = 0 \lor (0 \land 0) = 0$

$A_3[3, 2] = A_2[3, 2] \lor (A_2[3, 3] \land A_2[3, 2]) = 0 \lor (0 \land 0) = 0$

$A_3[3, 3] = A_2[3, 3] \lor (A_2[3, 3] \land A_2[3, 3]) = 0 \lor (0 \land 0) = 0$

$A_3[3, 4] = A_2[3, 4] \lor (A_2[3, 3] \land A_2[3, 4]) = 0 \lor (0 \land 0) = 0$

$$A_3 = \begin{array}{cc} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

Once A3 is calculated, we use the update rule to calculate A_4 and stop. This matrix is the reachability matrix for the graph.

$A_4[1, 1] = A_3 [1, 1] \lor (A_3 [1, 4] \land A_3 [4, 1]) = 0 \lor (1 \land 1) = 0 \lor 1 = 1$

$A_4[2, 1] = A_3 [2, 1] \lor (A_3 [2, 4] \land A_3 [4, 1]) = 0 \lor (1 \land 1) = 0 \lor 1 = 1$

$A_4[2, 2] = A_3 [2, 2] \lor (A_3 [2, 4] \land A_3 [4, 2]) = 0 \lor (1 \land 1) = 0 \lor 1 = 1$

$A_4[3, 1] = A_3 [3, 1] \lor (A_3 [3, 4] \land A_3 [4, 1]) = 0 \lor (0 \land 1) = 0 \lor 0 = 0$

$A_4[3, 2] = A_3 [3, 2] \lor (A_3 [3, 4] \land A_3 [4, 2]) = 0 \lor (0 \land 1) = 0 \lor 0 = 0$

$A_4[3, 3] = A_3 [3, 3] \lor (A_3 [3, 4] \land A_3 [4, 3]) = 0 \lor (0 \land 1) = 0 \lor 0 = 0$

$A_4[3, 4] = A_3 [3, 4] \lor (A_3 [3, 4] \land A_3 [4, 4]) = 0 \lor (0 \land 1) = 0 \lor 0 = 0$

$$A_4 = \begin{array}{cc} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

Note that according to the algorithm vertex 3 is not reachable from itself 1. This is because as can be seen in the graph, there is no path from vertex 3 back to itself.

6.5. Traversing a Graph

Many graph algorithms require one to systematically examine the nodes and edges of a graph G. There are two standard ways to do this. They are:

- Breadth first traversal (BFT)
- Depth first traversal (DFT)

The BFT will use a queue as an auxiliary structure to hold nodes for future processing and the DFT will use a STACK.

During the execution of these algorithms, each node N of G will be in one of three states, called the *status* of N, as follows:

1. STATUS = 1 (Ready state): The initial state of the node N.

2. STATUS = 2 (Waiting state): The node N is on the QUEUE or STACK, waiting to be processed.

3. STATUS = 3 (Processed state): The node N has been processed.

Both BFS and DFS impose a tree (the BFS/DFS tree) on the structure of graph. So, we can compute a spanning tree in a graph. The computed spanning tree is not a minimum spanning tree. The spanning trees obtained using depth first search are called depth first spanning trees. The spanning trees obtained using breadth first search are called Breadth first spanning trees.

6.5.1. Breadth first search and traversal:

The general idea behind a breadth first traversal beginning at a starting node A is as follows. First we examine the starting node A. Then we examine all the neighbors of A. Then we examine all the neighbors of neighbors of A. And so on. We need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is accomplished by using a QUEUE to hold nodes that are waiting to be processed, and by using a field STATUS that tells us the current status of any node. The spanning trees obtained using BFS are called Breadth first spanning trees.

Breadth first traversal algorithm on graph G is as follows:

This algorithm executes a BFT on graph G beginning at a starting node A.

```
Initialize all nodes to the ready state (STATUS = 1).

1.  Put the starting node A in QUEUE and change its status to the waiting
    state (STATUS = 2).

2.  Repeat the following steps until QUEUE is empty:

    a.  Remove the front node N of QUEUE. Process N and change the
        status of N to the processed state (STATUS = 3).

    b.  Add to the rear of QUEUE all the neighbors of N that are in the
        ready state (STATUS = 1), and change their status to the waiting
        state (STATUS = 2).

3.  Exit.
```

6.5.2. Depth first search and traversal:

Depth first search of undirected graph proceeds as follows: First we examine the starting node V. Next an unvisited vertex 'W' adjacent to 'V' is selected and a depth first search from 'W' is initiated. When a vertex 'U' is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited, which has an unvisited vertex 'W' adjacent to it and initiate a depth first search from W. The search terminates when no unvisited vertex can be reached from any of the visited ones.

This algorithm is similar to the inorder traversal of binary tree. DFT algorithm is similar to BFT except now use a STACK instead of the QUEUE. Again field STATUS is used to tell us the current status of a node.

The algorithm for depth first traversal on a graph G is as follows.

This algorithm executes a DFT on graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).

2. Push the starting node A into STACK and change its status to the waiting state (STATUS = 2).

3. Repeat the following steps until STACK is empty:

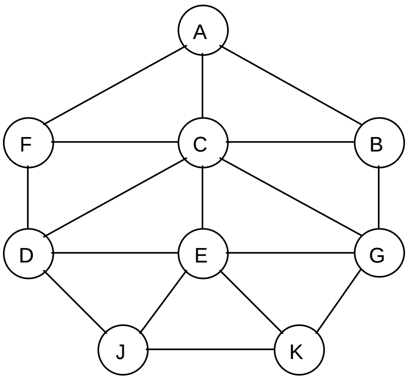
    a. Pop the top node N from STACK. Process N and change the status of N to the processed state (STATUS = 3).

    b. Push all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).

4. Exit.

Example 1:

Consider the graph shown below. Traverse the graph shown below in breadth first order and depth first order.



A Graph G

Node	*Adjacency List*
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

Breadth-first search and traversal:

The steps involved in breadth first traversal are as follows:

Current Node	QUEUE	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	F C B	A	3	2	2	1	1	2	1	1	1
F	C B D	A F	3	2	2	2	1	3	1	1	1
C	B D E G	A F C	3	2	3	2	2	3	2	1	1
B	D E G	A F C B	3	3	3	2	2	3	2	1	1
D	E G J	A F C B D	3	3	3	3	2	3	2	2	1
E	G J K	A F C B D E	3	3	3	3	3	3	2	2	2
G	J K	A F C B D E G	3	3	3	3	3	3	3	2	2
J	K	A F C B D E G J	3	3	3	3	3	3	3	3	2
K	EMPTY	A F C B D E G J K	3	3	3	3	3	3	3	3	3

For the above graph the breadth first traversal sequence is: ***A F C B D E G J K***.

Depth-first search and traversal:

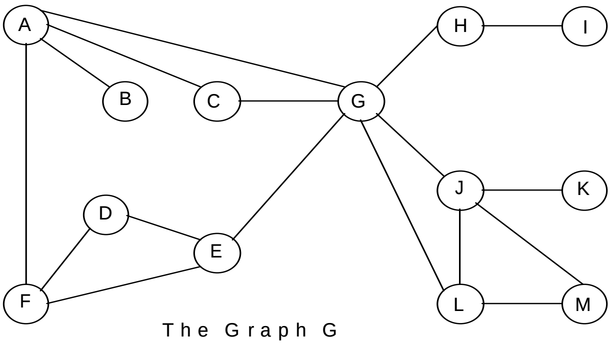
The steps involved in depth first traversal are as follows:

Current Node	Stack	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	B C F	A	3	2	2	1	1	2	1	1	1
F	B C D	A F	3	2	2	2	1	3	1	1	1
D	B C E J	A F D	3	2	2	3	2	3	1	2	1
J	B C E K	A F D J	3	2	2	3	2	3	1	3	2
K	B C E G	A F D J K	3	2	2	3	2	3	2	3	3
G	B C E	A F D J K G	3	2	2	3	2	3	3	3	3
E	B C	A F D J K G E	3	2	2	3	3	3	3	3	3
C	B	A F D J K G E C	3	2	3	3	3	3	3	3	3
B	EMPTY	A F D J K G E C B	3	3	3	3	3	3	3	3	3

For the above graph the depth first traversal sequence is: ***A F D J K G E C B***.

Example 2:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.

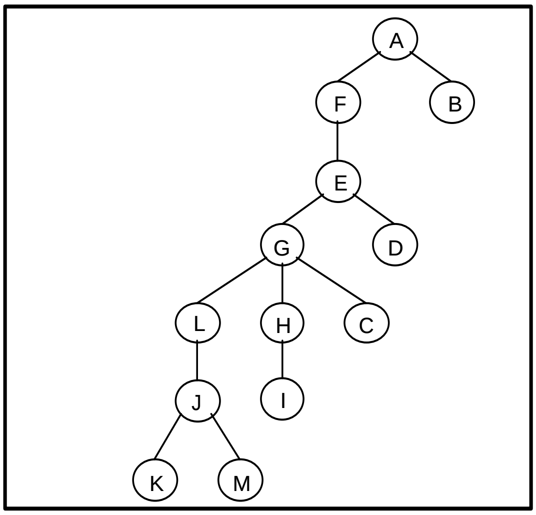


The Graph G

Node	***Adjacency List***
A	F, B, C, G
B	A
C	A, G
D	E, F
E	G, D, F
F	A, E, D
G	A, L, E, H, J, C
H	G, I
I	H
J	G, L, K, M
K	J
L	G, J, M
M	L, J

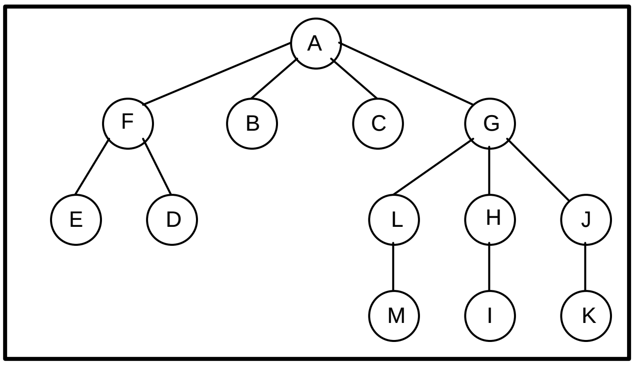
The adjacency list for the graph G

If the depth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: ***A F E G L J K M H I C D B***. The depth first spanning tree is shown in the figure given below:



Depth first Traversal

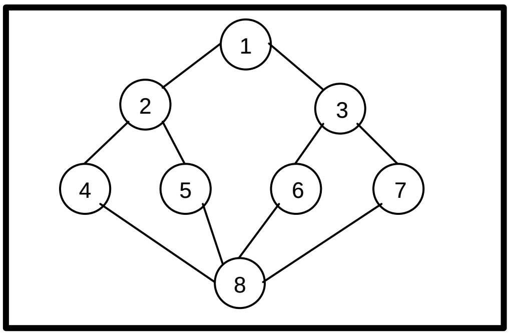
If the breadth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: ***A F B C G E D L H J M I K***. The breadth first spanning tree is shown in the figure given below:



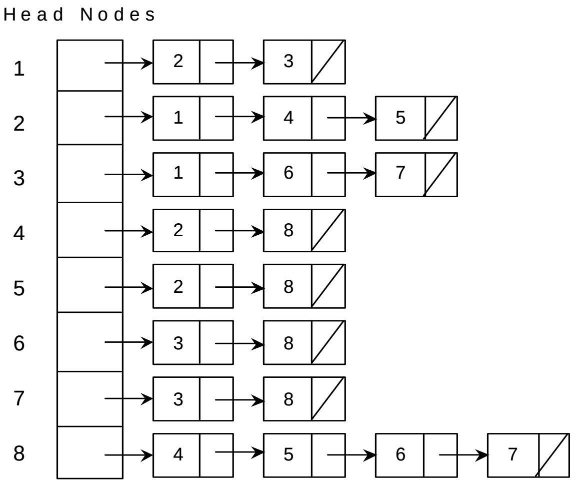
Breadth first traversal

Example 3:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.



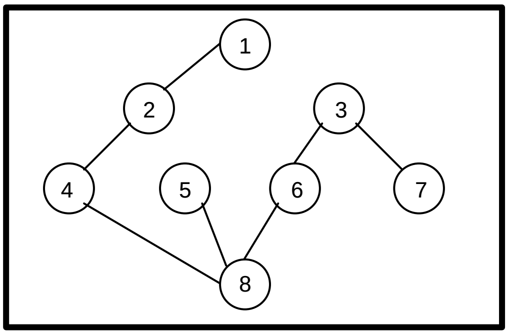
Graph G



Adjacency list for graph G

Depth first search and traversal:

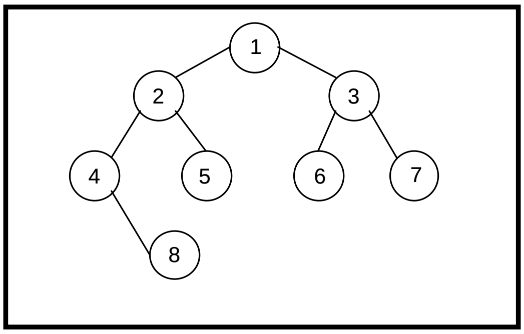
If the depth first is initiated from vertex 1, then the vertices of graph G are visited in the order: 1, 2, 4, 8, 5, 6, 3, 7. The depth first spanning tree is as follows:



Depth First Spanning Tree

Breadth first search and traversal:

If the breadth first search is initiated from vertex 1, then the vertices of G are visited in the order: 1, 2, 3, 4, 5, 6, 7, 8. The breadth first spanning tree is as follows:



Breadth First Spanning Tree

EXCERCISES

1. Show that the sum of degrees of all vertices in an undirected graph is twice the number of edges.

2. Show that the number of vertices of odd degree in a finite graph is even.

3. How many edges are contained in a complete graph of “n” vertices.

4. Show that the number of spanning trees in a complete graph of “n” vertices is $2^{n-1} – 1$.

5. Prove that the edges explored by a breadth first or depth first traversal of a connected graph from a tree.

6. Explain how existence of a cycle in an undirected graph may be detected by traversing the graph in a depth first manner.

7. Write a “C” function to generate the incidence matrix of a graph from its adjacency matrix.

8. Give an example of a connected directed graph so that a depth first traversal of that graph yields a forest and not a spanning tree of the graph.

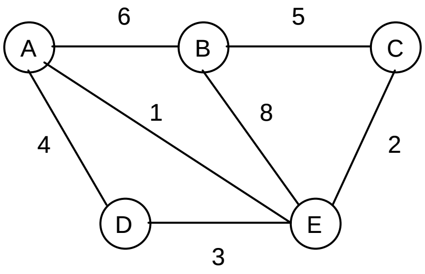
9. Rewrite the algorithms “BFSearch” and “DFSearch” so that it works on adjacency matrix representation of graphs.

10. Write a “C” function to find out whether there is a path between any two vertices in a graph (i.e. to compute the transitive closure matrix of a graph)

11. Write a “C” function to delete an existing edge from a graph represented by an adjacency list.

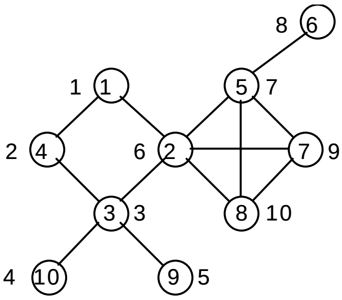
12. Construct a weighted graph for which the minimal spanning trees produced by Kruskal’s algorithm and Prim’s algorithm are different.

13. Describe the algorithm to find a minimum spanning tree T of a weighted graph G. Find the minimum spanning tree T of the graph shown below.



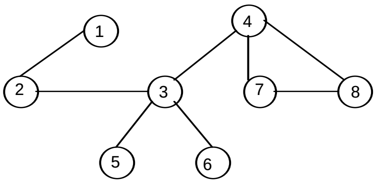
14. For the graph given below find the following:

a) Linked representation of the graph.
b) Adjacency list.
c) Depth first spanning tree.
d) Breadth first spanning tree.
e) Minimal spanning tree using Kruskal's and Prim's algorithms.



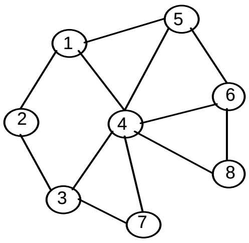
15. For the graph given below find the following:

f) Linked representation of the graph.
g) Adjacency list.
h) Depth first spanning tree.
i) Breadth first spanning tree.
j) Minimal spanning tree using Kruskal's and Prim's algorithms.



16. For the graph given below find the following:

k) Linked representation of the graph.
l) Adjacency list.
m) Depth first spanning tree.
n) Breadth first spanning tree.
o) Minimal spanning tree using Kruskal's and Prim's algorithms.



Chapter 7

Searching and Sorting

There are basically two aspects of computer programming. One is data organization also commonly called as data structures. Till now we have seen about data structures and the techniques and algorithms used to access them. The other part of computer programming involves choosing the appropriate algorithm to solve the problem. Data structures and algorithms are linked each other. After developing programming techniques to represent information, it is logical to proceed to manipulate it. This chapter introduces this important aspect of problem solving.

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

1. Linear or sequential search
2. Binary search

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words is arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

1. Bubble sort
2. Quick sort
3. Selection sort and
4. Heap sort

There are two types of sorting techniques:

1. Internal sorting
2. External sorting

If all the elements to be sorted are present in the main memory then such sorting is called **internal sorting** on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called **external sorting**. Here we study only internal sorting techniques.

7.1. Linear Search:

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are ‘n**’** elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison’s to search an element. If search is not successful, you would need ’n**’** comparisons.

The time complexity of linear search is ***O(n)***.

Algorithm:

Let array a[n] stores n elements. Determine whether element ‘x’ is present or not.

```
linsrch(a[n], x)
{
        index = 0;
        flag = 0;
        while (index < n) do
        {
                if (x == a[index])
                {
                        flag = 1;
                        break;
                }
                index ++;
        }
        if(flag == 1)
                printf(“Data found  at %d position“, index);
        else
                printf(“data not found”);

}
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for: 45, we’ll look at 1 element before success
39, we’ll look at 2 elements before success
8, we’ll look at 3 elements before success
54, we’ll look at 4 elements before success
77, we’ll look at 5 elements before success
38 we’ll look at 6 elements before success
24, we’ll look at 7 elements before success
16, we’ll look at 8 elements before success
4, we’ll look at 9 elements before success
7, we’ll look at 10 elements before success
9, we’ll look at 11 elements before success
20, we’ll look at 12 elements before success

For any element not in the list, we’ll look at 12 elements before failure.

Example 2:

Let us illustrate linear search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

Searching different elements is as follows:

1. Searching for x = 7 Search successful, data found at 3rd position.

2. Searching for x = 82 Search successful, data found at 7th position.

3. Searching for x = 42 Search un-successful, data not found.

7.1.1. A non-recursive program for Linear Search:

```
# include <stdio.h>
# include <conio.h>

main()
{
        int number[25], n, data, i, flag = 0;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i = 0; i < n; i++)
                scanf("%d", &number[i]);
        printf("\n Enter the element to be Searched: ");
        scanf("%d", &data);
        for( i = 0; i < n; i++)
        {
                if(number[i] == data)
                {
                        flag = 1;
                        break;
                }
        }
        if(flag == 1)
                printf("\n Data found at location: %d", i+1);
        else
                printf("\n Data not found ");
}
```

7.1.2. A Recursive program for linear search:

```
# include <stdio.h>
# include <conio.h>

void linear_search(int a[], int data, int position, int n)
{
        if(position < n)
        {
                if(a[position] == data)
                        printf("\n Data Found at %d ", position);
                else
                        linear_search(a, data, position + 1, n);
        }
        else
                printf("\n Data not found");
}

void main()
{
        int a[25], i, n, data;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i = 0; i < n; i++)
        {
                scanf("%d", &a[i]);
        }
        printf("\n Enter the element to be seached: ");
        scanf("%d", &data);
        linear_search(a, data, 0, n);
        getch();
}
```

7.2. BINARY SEARCH

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \dots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid]. Similarly, if a[mid] > x, then further search is only necessary in that part of the file which follows a[mid].

If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Algorithm:

Let array a[n] of elements in increasing order, $n \geq 0$, determine whether 'x' is present, and if so, set j such that x = a[j] else return 0.

```
binsrch(a[], n, x)
{
        low = 1; high = n;
        while (low ≤ high) do
        {
                mid = ⌊ (low + high)/2 ⌋
                if (x < a[mid])
                        high = mid – 1;
                else if (x > a[mid])
                        low = mid + 1;
                else return mid;
        }
        return 0;
}
```

low and *high* are integer variables such that each time through the loop either ‘x’ is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if ‘x’ is not present.

Example 1:

Let us illustrate binary search on the following 12 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for x = 4: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4, ***found***

If we are searching for x = 7: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4
low = 2, high = 2, mid = 4/2 = 2, check 7, ***found***

If we are searching for x = 8: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8, ***found***

If we are searching for x = 9: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9, ***found***

If we are searching for x = 16: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9
low = 5, high = 5, mid = 10/2 = 5, check 16, ***found***

If we are searching for x = 20: (This needs 1 comparison)
low = 1, high = 12, mid = 13/2 = 6, check 20, ***found***

If we are searching for x = 24: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24, ***found***

If we are searching for x = 38: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24
low = 8, high = 8, mid = 16/2 = 8, check 38, ***found***

If we are searching for x = 39: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39, ***found***

If we are searching for x = 45: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 10, high = 10, mid = 20/2 = 10, check 45, ***found***

If we are searching for x = 54: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54, ***found***

If we are searching for x = 77: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 12, high = 12, mid = 24/2 = 12, check 77, ***found***

The number of comparisons necessary by search element:

> 20 – requires 1 comparison;
> 8 and 39 – requires 2 comparisons;
> 4, 9, 24, 54 – requires 3 comparisons and
> 7, 16, 38, 45, 77 – requires 4 comparisons

Summing the comparisons, needed to find all twelve items and dividing by 12, yielding 37/12 or approximately 3.08 comparisons per successful search on the average.

Example 2:

Let us illustrate binary search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

Solution:

The number of comparisons required for searching different elements is as follows:

1. If we are searching for x = 101: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

2. Searching for x = 82: (Number of comparisons = 3)

low	high	mid
1	9	5
6	9	7
8	9	8
		found

3. Searching for x = 42: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

4. Searching for x = -14: (Number of comparisons = 3)

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If x < a(1), a(1) < x < a(2), a(2) < x < a(3), a(5) < x < a(6), a(6) < x < a(7) or a(7) < x < a(8) the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons.

Thus the average number of element comparisons for an unsuccessful search is:

(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4

Time Complexity:

The time complexity of binary search in a successful search is O(log n) and for an unsuccessful search is O(log n).

7.2.1. A non-recursive program for binary search:

```
# include <stdio.h>
# include <conio.h>

main()
{
        int number[25], n, data, i, flag = 0, low, high, mid;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements in ascending order: ");
        for(i = 0; i < n; i++)
                scanf("%d", &number[i]);
        printf("\n Enter the element to be searched: ");
        scanf("%d", &data);
        low = 0; high =  n-1;
        while(low <= high)
        {
                mid = (low + high)/2;
                if(number[mid] == data)
                {
                        flag = 1;
                        break;
                }
                else
                {
                        if(data < number[mid])
                                high = mid - 1;
                        else
                                low = mid + 1;
                }
        }
        if(flag == 1)
                printf("\n Data found at location: %d", mid + 1);
        else
                printf("\n Data Not Found ");
}
```

7.2.2. A recursive program for binary search:

```
# include <stdio.h>
# include <conio.h>

void bin_search(int a[], int data, int low, int high)
{
        int mid ;
        if( low <= high)
        {
                mid = (low + high)/2;
                if(a[mid] == data)
                        printf("\n Element found at location: %d ", mid + 1);
                else
                {
                        if(data < a[mid])
                                bin_search(a, data, low, mid-1);
                        else
                        bin_search(a, data, mid+1, high);
                }
        }
        else
                printf("\n Element not found");
}
void main()
{
        int a[25], i, n, data;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements in ascending order: ");
        for(i = 0; i < n; i++)
                scanf("%d", &a[i]);
        printf("\n Enter the element to be searched: ");
        scanf("%d", &data);
        bin_search(a, data, 0, n-1);
        getch();
}
```

7.3. Bubble Sort:

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., X[i] with X[i+1] and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Example:

Consider the array x[n] which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements).

We compare X[i] and X[i+1] for i = 0, 1, 2, 3, and 4, and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared).

We repeat the same process, but this time we don't include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared).

We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared).

We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number 33 to X[2].

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22	33	

Pass 5: (fifth element is compared).

We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.

For an array of size n, we required (n-1) passes.

7.3.1. Program for Bubble Sort:

```
#include <stdio.h>
#include <conio.h>
void bubblesort(int x[], int n)
{
        int i,  j,  temp;
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n–i-1 ; j++)
                {
                        if (x[j] > x[j+1])
                        {
                                temp = x[j];
                                x[j] = x[j+1];
                                x[j+1] = temp;
                        }
                }
        }
}

main()
{
        int i, n, x[25];
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter Data:");
        for(i = 0; i < n ; i++)
                scanf("%d", &x[i]);
        bubblesort(x, n);
        printf ("\n Array Elements after sorting: ");
        for (i = 0; i < n; i++)
                printf ("%5d", x[i]);
}
```

Time Complexity:

The bubble sort method of sorting an array of size n requires (n-1) passes and (n-1) comparisons on each pass. Thus the total number of comparisons is (n-1) * (n-1) = $n^2 – 2n + 1$, which is $O(n^2)$. Therefore bubble sort is very inefficient when there are more elements to sorting.

7.4. Selection Sort:

Selection sort will not require no more than n-1 interchanges. Suppose x is an array of size n stored in memory. The selection sort algorithm first selects the smallest element in the array x and place it at array position 0; then it selects the next smallest element in the array x and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array.

The array is passed through (n-1) times and the smallest element is placed in its respective position in the array as detailed below:

Pass 1: Find the location j of the smallest element in the array x [0], x[1], ... x[n-1], and then interchange x[j] with x[0]. Then x[0] is sorted.

Pass 2: Leave the first element and find the location j of the smallest element in the sub-array x[1], x[2], ... x[n-1], and then interchange x[1] with x[j]. Then x[0], x[1] are sorted.

Pass 3: Leave the first two elements and find the location j of the smallest element in the sub-array x[2], x[3], ... x[n-1], and then interchange x[2] with x[j]. Then x[0], x[1], x[2] are sorted.

Pass (n-1): Find the location j of the smaller of the elements x[n-2] and x[n-1], and then interchange x[j] and x[n-2]. Then x[0], x[1], ... x[n-2] are sorted. Of course, during this pass x[n-1] will be the biggest element and so the entire array is sorted.

Time Complexity:

In general we prefer selection sort in case where the insertion sort or the bubble sort requires exclusive swapping. In spite of superiority of the selection sort over bubble sort and the insertion sort (there is significant decrease in run time), its efficiency is also ***O(n^2)*** for n data items.

Example:

Let us consider the following example with 9 elements to analyze selection Sort:

1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***Remarks***
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	**50**	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	**50**	**55**	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	**50**	**55**	**60**	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	85	80	Find the eighth smallest element
							i	J	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	The outer loop ends.

7.4.1. Non-recursive Program for selection sort:

```
# include<stdio.h>
# include<conio.h>

void selectionSort( int low, int high );

int a[25];

int main()
{
        int num,  i= 0;
        clrscr();
        printf( "Enter the number of elements: " );
        scanf("%d", &num);
        printf( "\nEnter the elements:\n" );
        for(i=0; i < num; i++)
                scanf( "%d", &a[i] );
        selectionSort( 0, num - 1 );
        printf( "\nThe elements after sorting are: " );
        for( i=0; i< num; i++ )
                printf( "%d    ", a[i] );
        return 0;
}

void selectionSort( int low, int high )
{
        int i=0, j=0, temp=0, minindex;
        for( i=low; i <= high; i++ )
        {
                minindex = i;
                for( j=i+1; j <= high; j++ )
                {
                        if( a[j] < a[minindex] )
                                minindex = j;
                }
                temp = a[i];
                a[i] = a[minindex];
                a[minindex] = temp;
        }
}
```

7.4.2. Recursive Program for selection sort:

```
#include <stdio.h>
#include<conio.h>

int x[6] = {77, 33, 44, 11, 66};
selectionSort(int);

main()
{
        int i, n = 0;
        clrscr();
        printf (" Array Elements before sorting: ");
        for (i=0; i<5; i++)
```

```
                printf ("%d  ", x[i]);
        selectionSort(n);                       /* call selection sort */
        printf ("\n Array Elements after sorting: ");
        for (i=0; i<5; i++)
                printf ("%d  ", x[i]);
}

selectionSort( int n)
{
        int k, p, temp, min;
        if (n== 4)
                return (-1);
        min = x[n];
        p = n;
        for (k = n+1; k<5; k++)
        {
                if (x[k] <min)
                {
                        min = x[k];
                        p = k;
                }
        }
        temp = x[n];            /* interchange x[n] and x[p] */
        x[n] = x[p];
        x[p] = temp;
        n++ ;
        selectionSort(n);
}
```

7.5. Quick Sort:

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960’s. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by “divide and conquer” technique.

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the *pivot* element. Once the array has been rearranged in this way with respect to the *pivot*, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers up and down which are moved toward each other in the following fashion:

1. Repeatedly increase the pointer ‘up’ until a[up] >= pivot.
2. Repeatedly decrease the pointer ‘down’ until a[down] <= pivot.
3. If down > up, interchange a[down] with a[up]
4. Repeat the steps 1, 2 and 3 till the ‘up’ pointer crosses the ‘down’ pointer. If ‘up’ pointer crosses ‘down’ pointer, the position for pivot is found and place pivot element in ‘down’ pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array ‘a’ between positions ‘low’ and ‘high’.

1. It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

2. Here we choose the first element as the ‘pivot’. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . . x[high].

3. It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

4. It calls itself recursively to sort the right sub-array x[j+1], x[j+2], . . x[high] between positions j+1 and high.

The time complexity of quick sort algorithm is of ***O(n log n)***.

Algorithm

Sorts the elements a[p], ... ,a[q] which reside in the global array a[n] into ascending order. The a[n + 1] is considered to be defined and must be greater than all elements in a[n]; a[n + 1] = + ∝

```
quicksort (p, q)
{
     if ( p < q ) then
     {
          call j = PARTITION(a, p, q+1);     // j is the position of the partitioning element
          call quicksort(p, j – 1);
          call quicksort(j + 1 , q);
      }
}

partition(a, m, p)
{
      v = a[m]; up = m; down = p;          // a[m] is the partition element
      do
      {
             repeat
                    up = up + 1;
             until (a[up] ≥ v);

             repeat
                    down = down – 1;
             until (a[down] ≤ v);
             if (up < down) then call interchange(a, up, down);
       } while (up ≥ down);

      a[m] = a[down];
      a[down] = v;
      return (down);
}
```

```
interchange(a, up, down)
{
        p = a[up];
        a[up] = a[down];
        a[down] = p;
}
```

Example:

Select first element as the pivot element. Move ‘up’ pointer from left to right in search of an element larger than pivot. Move the ‘down’ pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.

This process continues till the ‘up’ pointer crosses the ‘down’ pointer. If ‘up’ pointer crosses ‘down’ pointer, the position for pivot is found and interchange pivot and element at ‘down’ position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	**24**								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	**08**	(16)									swap pivot & down
	pivot	down	up										
	(04)	**06**											swap pivot & down
	04 pivot, down, up												
				16 pivot, down, up									
(02	**04**	**06**	**08**	**16**	**24)**	38							

							(56	57	58	79	70	45**)**	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	**56**	(58	79	70	57)	
							45 pivot, down, up						swap pivot & down
									(58 pivot	79 up	70	57) down	swap up & down
										57		79	
										down	up		
									(57)	**58**	(70	79)	swap pivot & down
									57 pivot, down, up				
											(70	79)	
											pivot, down	up	swap pivot & down
											70		
												79 pivot, down, up	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

7.5.1. Recursive program for Quick Sort:

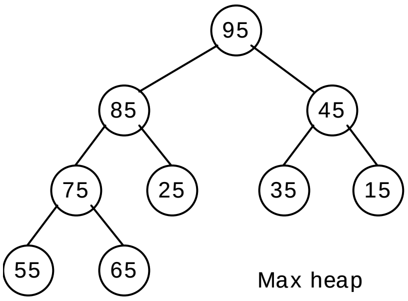
```
# include<stdio.h>
# include<conio.h>

void quicksort(int, int);
int partition(int, int);
void interchange(int, int);
int array[25];

int main()
{
        int num, i = 0;
        clrscr();
        printf( "Enter the number of elements: " );
        scanf( "%d", &num);
        printf( "Enter the elements: " );
        for(i=0; i < num; i++)
                scanf( "%d", &array[i] );
        quicksort(0, num -1);
        printf( "\nThe elements after sorting are: " );
        for(i=0; i < num; i++)
                printf("%d  ", array[i]);
        return 0;
}

void quicksort(int low, int high)
{
        int pivotpos;
        if( low < high )
        {
                pivotpos = partition(low, high + 1);
                quicksort(low, pivotpos - 1);
                quicksort(pivotpos + 1, high);
        }
}

int partition(int low, int high)
{
        int pivot = array[low];
        int up = low, down = high;

        do
        {
                do
                        up = up + 1;
                while(array[up] < pivot );

                do
                        down = down - 1;
                while(array[down] > pivot);

                if(up < down)
                        interchange(up, down);

        } while(up < down);
        array[low] = array[down];
        array[down] = pivot;
        return down;
}

void interchange(int i, int j)
{
        int temp;
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
}
```

7.6. Priority Queue, Heap and Heap Sort:

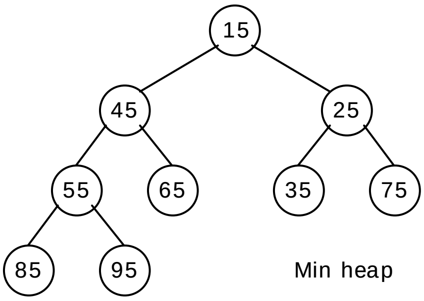
Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

7.6.1. Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.



Max heap



Min heap

A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children.

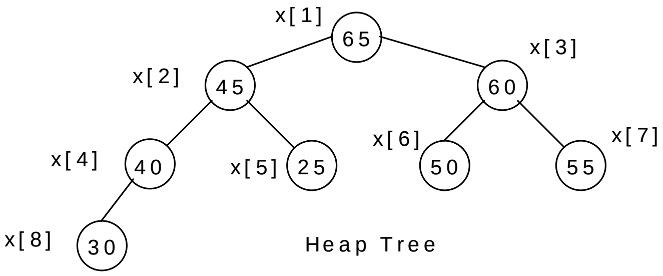
7.6.2. Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location *i* can be found in location 2**i*.
- The right child of an element stored at location *i* can be found in location 2**i*+1.
- The parent of an element stored at location *i* can be found at location floor(*i*/2).

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30



Heap Tree

7.6.3. Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent's value, and to be a max heap, parent's value greater than child's value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

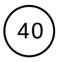
The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

```
Max_heap_insert (a, n)
{
        //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1]
        int i, n;
        i = n;
        item = a[n];
        while ( (i > 1) and (a[⌊ i/2 ⌋] < item ) do
        {
                a[i] = a[⌊ i/2 ⌋] ;                    // move the parent down
                i = ⌊ i/2 ⌋ ;
        }
        a[i] = item ;
        return true ;
}
```

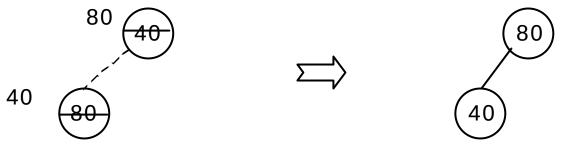
Example:

Form a heap using the above algorithm for the data: 40, 80, 35, 90, 45, 50, 70.

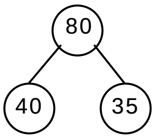
1. Insert 40:



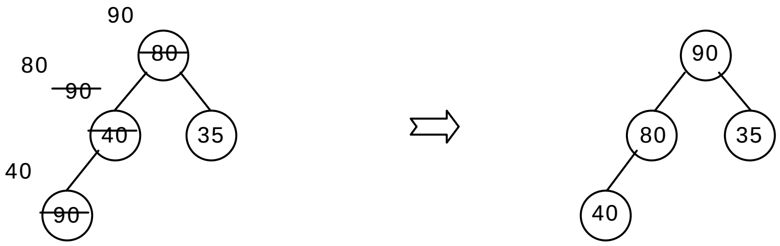
2. Insert 80:



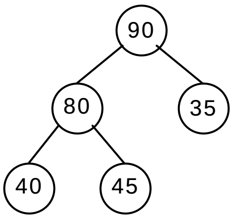
3. Insert 35:



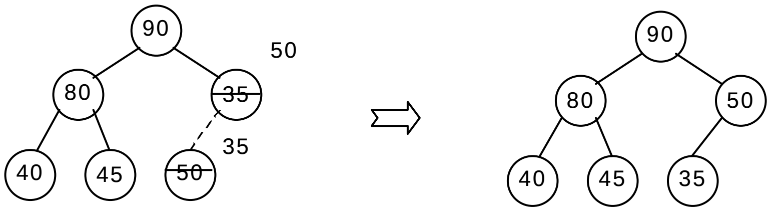
4. Insert 90:



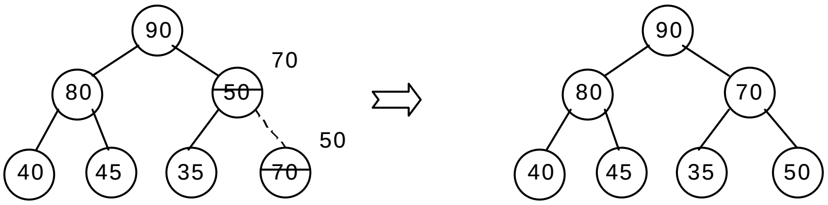
5. Insert 45:



6. Insert 50:



7. Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

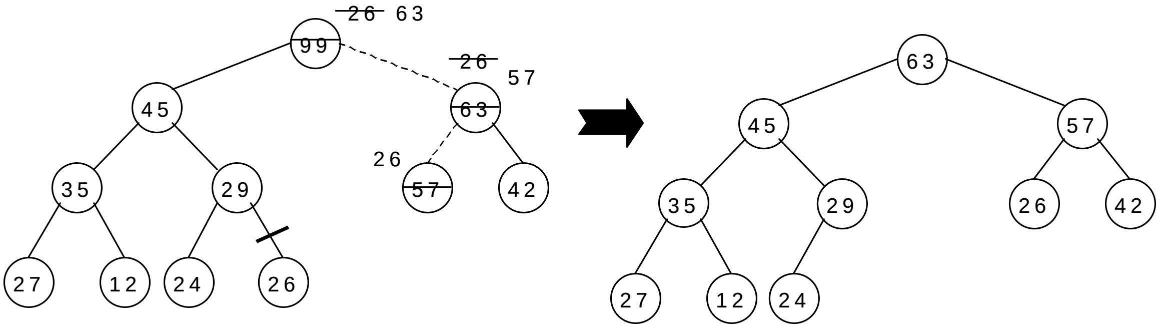
- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:
  - Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.
  - Make X as the current node.
  - Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

```
delmax (a, n, x)
// delete the maximum from the heap a[n] and store it in x
{
        if (n = 0) then
        {
                write ("heap is empty");
                return false;
        }
        x = a[1]; a[1] = a[n];
        adjust (a, 1, n-1);
        return true;
}
```

```
adjust (a, i, n)
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to
form a single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i  ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j ← j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j)) then break;
                                                // a position for item is found
                else a[⌊ j / 2 ⌋ ] = a[j]       // move the larger child up a level
                j = 2 * j;
        }
        a [⌊ j / 2 ⌋ ] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now, 26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appears as the leave node, hence re-heap is completed.



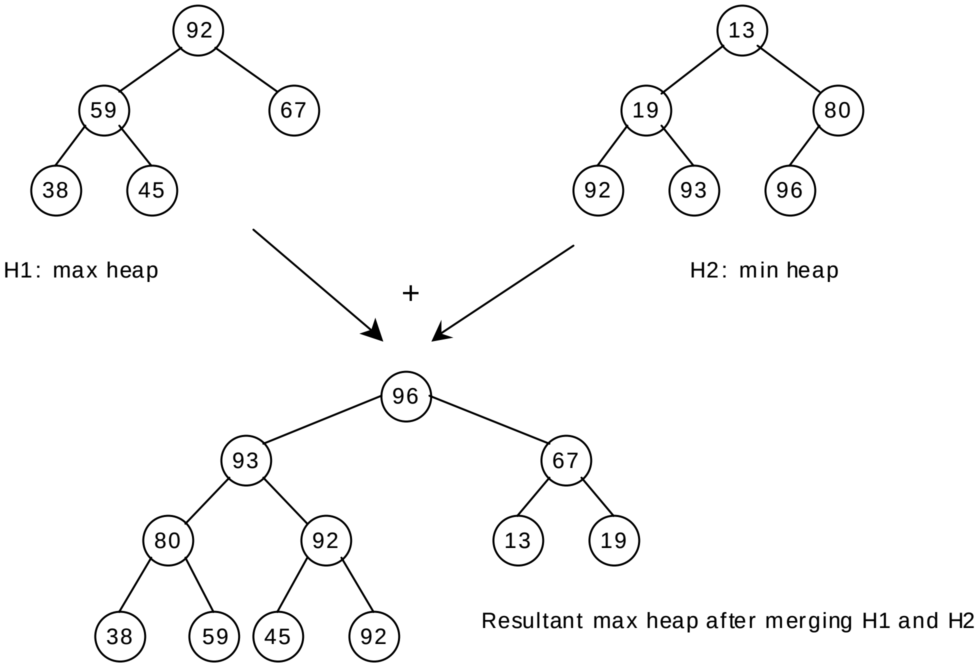
Deleting the node with data 99

After Deletion of node with data 99

7.6.4. Merging two heap trees:

Consider, two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap. Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

1. Delete the root node, say x, from H2. Re-heap H2.
2. Insert the node x into H1 satisfying the property of H1.



7.6.5. Application of heap tree:

They are two main applications of heap trees known are:

1. Sorting (Heap sort) and
2. Priority queue implementation.

7.7. HEAP SORT:

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

1. Build a heap tree with the given set of data.
2. a. Remove the top most item (the largest) and replace it with the last element in the heap.
   b. Re-heapify the complete binary tree.
   c. Place the deleted node in the output.
3. Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements a[n]. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

```
heapsort(a, n)
{
        heapify(a, n);
        for i = n to 2 by – 1 do
        {
                temp = a[i];
                a[i] =  a[1];
                a[1] = temp;
                adjust (a, 1, i – 1);
        }
}

heapify (a, n)
//Readjust the elements in a[n] to form a heap.
{
        for i ← ⌊ n/2 ⌋ to 1 by – 1 do adjust (a, i, n);
}

adjust (a, i, n)
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to
form a single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i  ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j ← j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j)) then break;
                                                // a position for item is found
                else a[⌊ j / 2 ⌋] = a[j]        // move the larger child up a level
                j = 2 * j;
        }
        a [⌊ j / 2 ⌋] = item;
}
```

Time Complexity:

Each ‘n’ insertion operations takes O(log k), where ‘k’ is the number of elements in the heap at the time. Likewise, each of the ‘n’ remove operations also runs in time O(log k), where ‘k’ is the number of elements in the heap at the time.

Since we always have k ≤ n, each such operation runs in O(log n) time in the worst case.

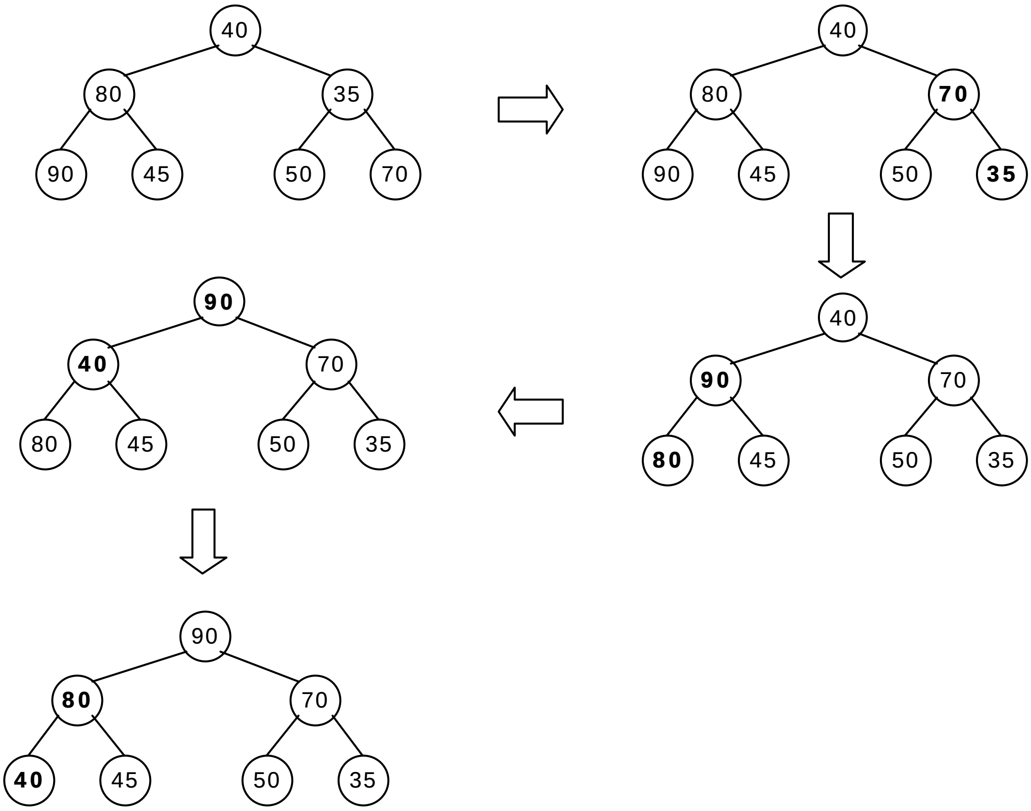
Thus, for ‘n’ elements it takes O(n log n) time, so the priority queue sorting algorithm runs in O(n log n) time when we use a heap to implement the priority queue.

Example 1:

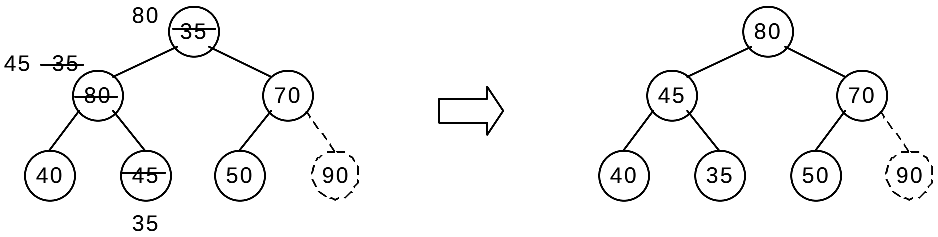
Form a heap from the set of elements (40, 80, 35, 90, 45, 50, 70) and sort the data using heap sort.

Solution:

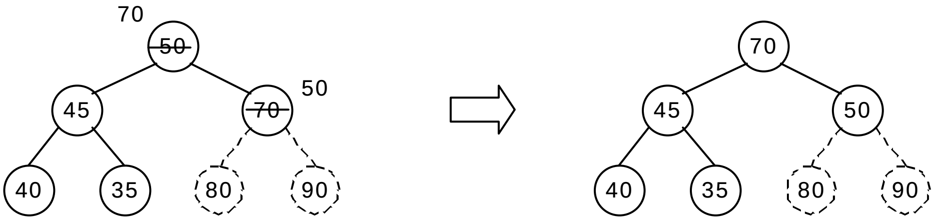
First form a heap tree from the given set of data and then sort by repeated deletion operation:



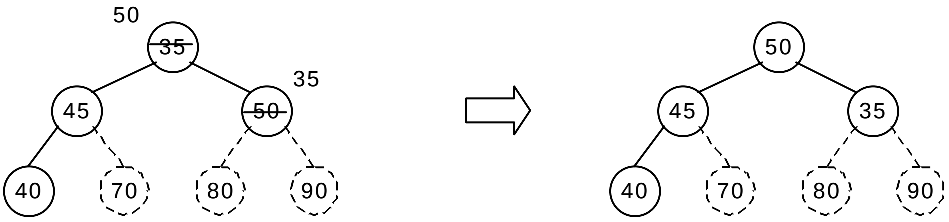
1. Exchange root 90 with the last element 35 of the array and re-heapify



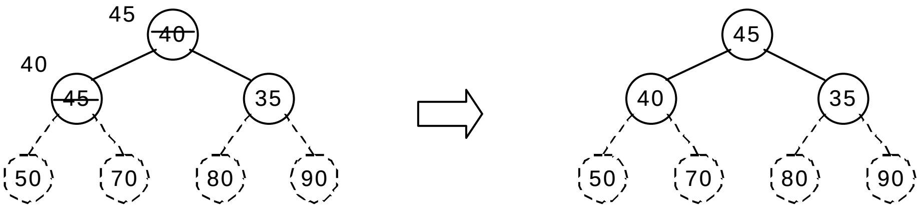
2. Exchange root 80 with the last element 50 of the array and re-heapify



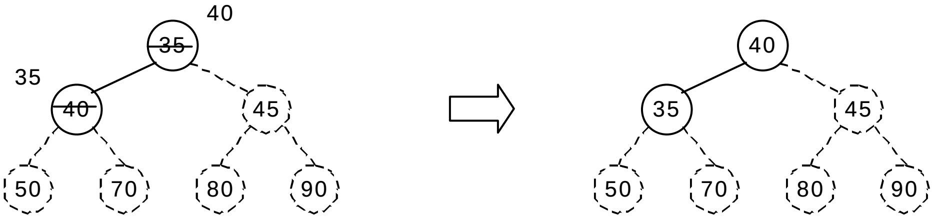
3. Exchange root 70 with the last element 35 of the array and re-heapify



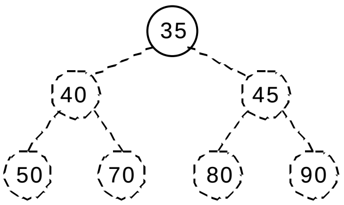
4. Exchange root 50 with the last element 40 of the array and re-heapify



5. Exchange root 45 with the last element 35 of the array and re-heapify



6. Exchange root 40 with the last element 35 of the array and re-heapify



The sorted tree

7.7.1. Program for Heap Sort:

```
void adjust(int i, int n, int a[])
{
        int j, item;
        j = 2 * i;
        item = a[i];
        while(j <= n)
        {
                if((j < n) && (a[j] < a[j+1]))
                        j++;
                if(item >= a[j])
                        break;
                else
                {
                        a[j/2] = a[j];
                        j = 2*j;
                }
        }
        a[j/2] = item;
}

void heapify(int n, int a[])
{
        int i;
        for(i = n/2; i > 0; i--)
                adjust(i, n, a);
}

void heapsort(int n,int a[])
{
        int temp, i;
        heapify(n, a);
        for(i = n; i > 0; i--)
        {
                temp = a[i];
                a[i] = a[1];
                a[1] = temp;
                adjust(1, i - 1, a);
        }
}

void main()
{
        int i, n, a[20];
        clrscr();
        printf("\n How many element you want: ");
        scanf("%d", &n);
        printf("Enter %d elements: ", n);
        for (i=1; i<=n; i++)
                scanf("%d", &a[i]);
        heapsort(n, a);
        printf("\n The sorted elements are: \n");
        for (i=1; i<=n; i++)
                printf("%5d", a[i]);
        getch();
}
```

7.8. Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on. As an illustration, consider the following processes with their priorities:

Process	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.

Exercises

1. Write a recursive "C" function to implement binary search and compute its time complexity.

2. Find the expected number of passes, comparisons and exchanges for bubble sort when the number of elements is equal to "10". Compare these results with the actual number of operations when the given sequence is as follows: 7, 1, 3, 4, 10, 9, 8, 6, 5, 2.

3. An array contains "n" elements of numbers. The several elements of this array may contain the same number "x". Write an algorithm to find the total number of elements which are equal to "x" and also indicate the position of the first such element in the array.

4. When a "C" function to sort a matrix row-wise and column-wise. Assume that the matrix is represented by a two dimensional array.

5. A very large array of elements is to be sorted. The program is to be run on a personal computer with limited memory. Which sort would be a better choice: Heap sort or Quick sort? Why?

6. Here is an array of ten integers: 5 3 8 9 1 7 0 2 6 4
Suppose we partition this array using quicksort's partition function and using 5 for the pivot. Draw the resulting array after the partition finishes.

7. Here is an array which has just been partitioned by the first step of quicksort: 3, 0, 2, 4, 5, 8, 7, 6, 9. Which of these elements could be the pivot? (There may be more than one possibility!)

8. Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap.

9. Sort the sequence 3, 1, 4, 5, 9, 2, 6, 5 using insertion sort.

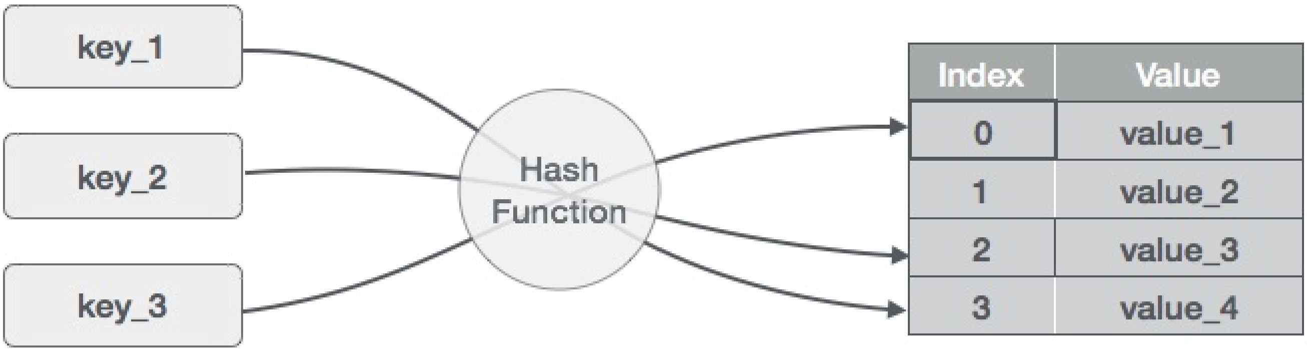
DATA STRUCTURE - HASH TABLE

Hash Table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data values has its own unique index value. Access of data becomes very fast if we know the index of desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of size of data. Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hashtable of size 20, and following items are to be stored. Item are in *key, value* format.



- 1, 20
- 2, 70
- 42, 80
- 4, 25
- 12, 44
- 14, 32
- 17, 11
- 13, 78
- 37, 98

S.n.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13

9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique used create already used index of the array. In such case, we can search the next empty location in the array by looking into the next cell until we found an empty cell. This technique is called linear probing.

S.n.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are basic primary operations of a hashtable which are following.

- **Search** − search an element in a hashtable.
- **Insert** − insert an element in a hashtable.
- **delete** − delete an element from a hashtable.

DataItem

Define a data item having some data, and key based on which search is to be conducted in hashtable.

```
struct DataItem {
   int data;
   int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
   return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched. Compute the hash code of the key passed and locate the element using that hashcode as index in the array. Use linear probing to get element ahead if element not found at computed hash code.

```
struct DataItem *search(int key){
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL){

      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}
```

Insert Operation

Whenever an element is to be inserted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing for empty location if an element is found at computed hash code.

```
void insert(int key,int data){
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
   item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1){
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing to get element ahead if an element is not found at computed hash code. When found, store a dummy item there to keep performance of hashtable intact.

```
struct DataItem* delete(struct DataItem* item){
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] !=NULL){

      if(hashArray[hashIndex]->key == key){
         struct DataItem* temp = hashArray[hashIndex];

         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
```

```
            return temp;
        }

        //go to next cell
        ++hashIndex;

        //wrap around the table
        hashIndex %= SIZE;
    }

    return NULL;
}
```

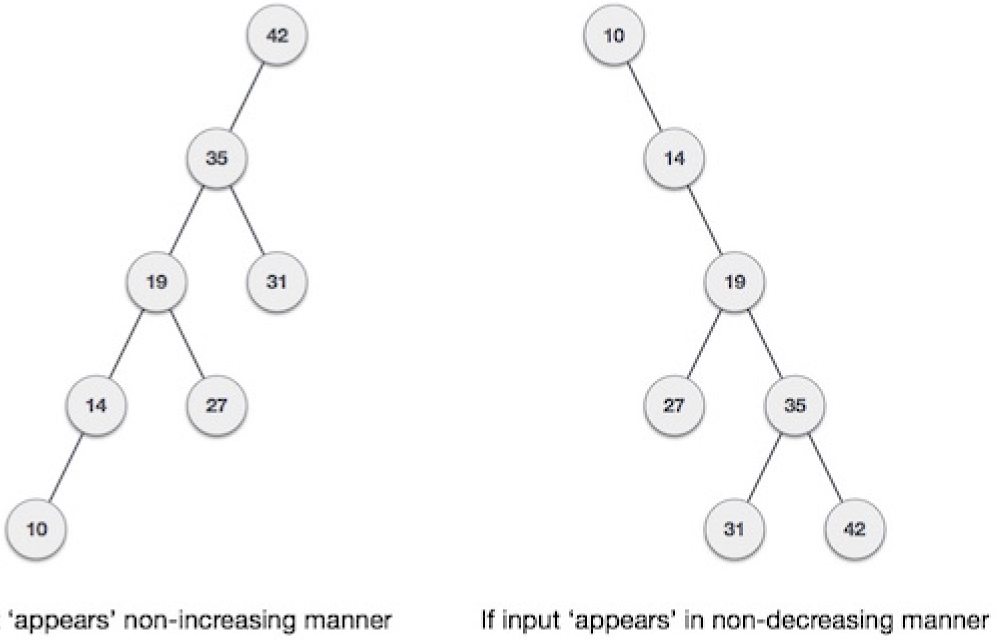
To see hash implementation in C programming language, please click here.

Loading [MathJax]/jax/output/HTML-CSS/jax.js

DATA STRUCTURE - AVL TREES

http://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm

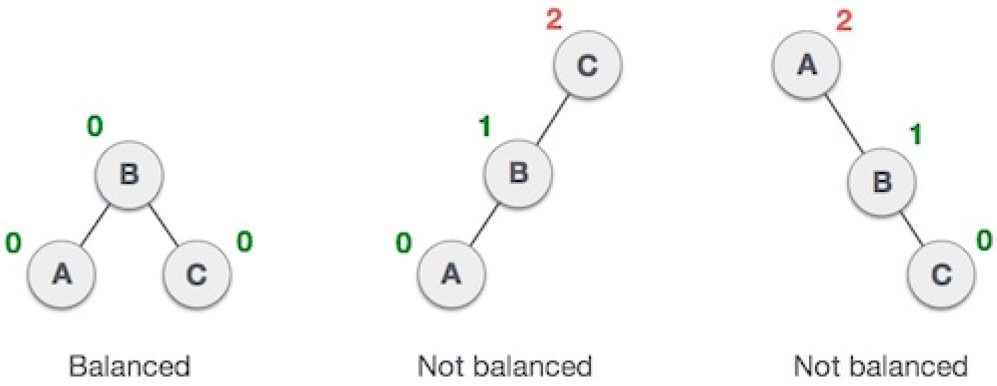
What if the input to binary search tree comes in sorted *ascendingordescending* manner? It will then look like this −


42
35
19
31
14
27
10
If input 'appears' non-increasing manner
10
14
19
27
35
31
42
If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance closes to linear search algorithms, that is O*n*. In real time data we cannot predict data pattern and their frequencies. So a need arises to balance out existing BST.

Named after their inventor ***Adelson***, ***Velski*** & ***Landis***, **AVL** trees are height balancing binary search tree. AVL tree checks the height of left and right sub-trees and assures that the difference is not more than 1. This difference is called ***Balance Factor***.

Here we see that the first tree is balanced and next two trees are not balanced −



In second tree, the left subtree of **C** has height 2 and right subtree has height 0, so the difference is 2. In third tree, the right subtree of **A** has height 2 and left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference *balancefactor* to be only 1.

```
BalanceFactor = height(left-sutree) − height(right-sutree)
```

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

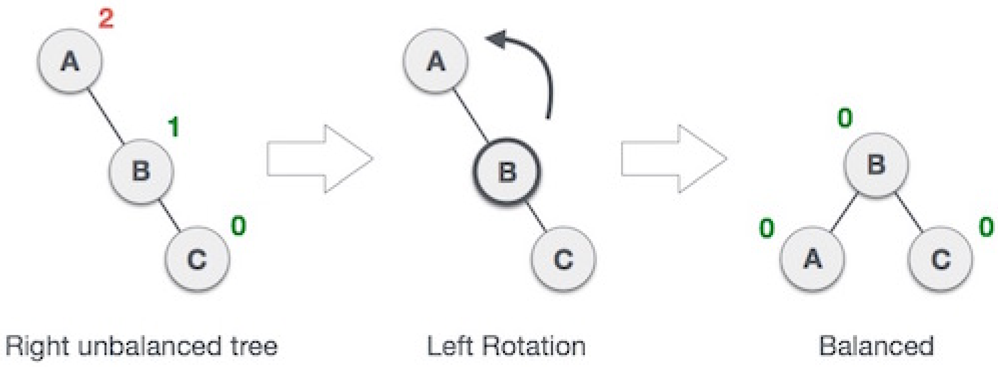
To make itself balanced, an AVL tree may perform four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

First two rotations are single rotations and next two rotations are double rotations. Two have an unbalanced tree we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

If a tree become unbalanced, when a node is inserted into the right subtree of right subtree, then we perform single left rotation −



Right unbalanced tree

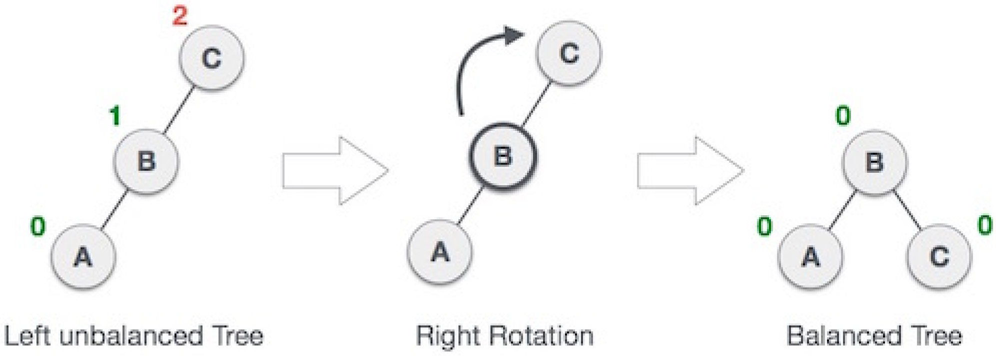
Left Rotation

Balanced

In our example, node **A** has become unbalanced as a node is inserted in right subtree of A's right subtree. We perform left rotation by making **A** left-subtree of B.

Right Rotation

AVL tree may become unbalanced if a node is inserted in the left subtree of left subtree. The tree then needs a right rotation.



Left unbalanced Tree

Right Rotation

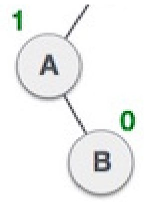
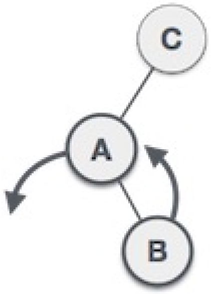
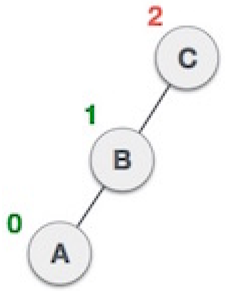
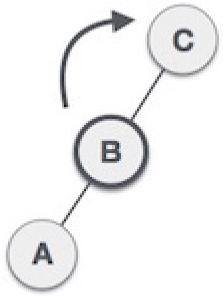
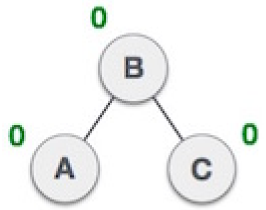
Balanced Tree

As depicted, the unbalanced node becomes right child of its left child by performing a right rotation.

Left-Right Rotation

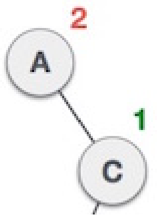
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is combination of left rotation followed by right rotation.

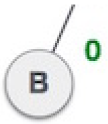
State	Action

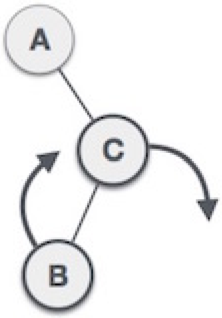
	A node has been inserted into right subtree of left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform left rotation on left subtree of **C**. This makes **A**, left subtree of **B**.
	Node **C** is still unbalanced but now, it is because of left-subtree of left-subtree.
	We shall now right-rotate the tree making **B** new root node of this subtree. **C** now becomes right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation

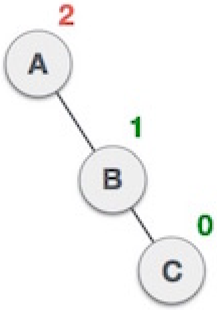
Second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into left subtree of right subtree. This makes **A** an unbalanced node, with balance factor 2.

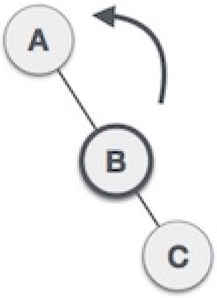




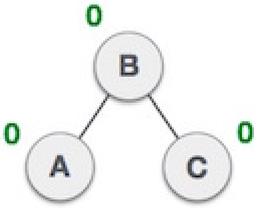
First, we perform right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes right subtree of **A**.



Node **A** is still unbalanced because of right subtree of its right subtree and requires a left rotation.



A left rotation is performed by making **B** the new root node of the subtree. **A** becomes left subtree of its right subtree **B**.



The tree is now balanced.

Loading [MathJax]/jax/output/HTML-CSS/jax.js

Lecture Notes on
Red/Black Trees

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 17
October 21, 2010

1 Introduction

In this lecture we discuss an ingenious way to maintain the balance invariant for binary search trees. The resulting data structure of *red/black trees* is used in a number of standard library implementations in C, C++, and Java.

2 Three Invariants

A red/black tree is a binary search tree in which each node is colored either red or black. At the interface, we maintain three invariants:

Ordering Invariant This is the same as for binary search trees: all the keys to left of a node are smaller, and all the keys to the right of a node are larger than the key at the node itself.

Height Invariant The number of *black* nodes on every path from the root to each leaf is the same. We call this the *black height* of the tree.

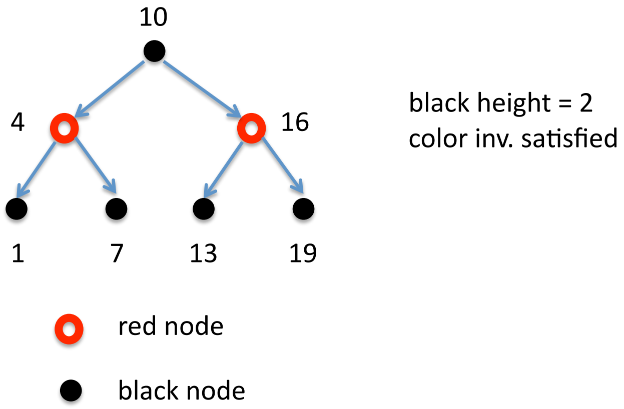
Color Invariant No two consecutive nodes are red.

The balance and color invariants together imply that the longest path from the root to a leaf is at most twice as long as the shortest path. Since insert and search in a binary search tree have time proportional to the length of the path from the root to the leaf, this guarantees $O(\log(n))$ times for these operations, even if the tree is not perfectly balanced. We therefore refer to the height and color invariants collectively as the *balance invariant*.

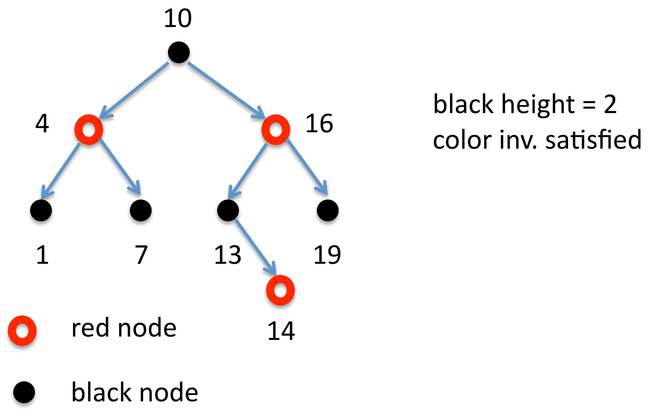
3 Insertion

The trick is, of course, to maintain all three invariants while sticking to the logarithmic time bound for each insert and search operation. Search is easy, since the search algorithm does not require the node colors: it works exactly as for the general case of balanced binary trees.

Insertion, however, is not obvious. Let's try just following the usual algorithm for insertion. We compare the key of the new element with the key at the root. If it is equal, we replace the current element. If it is less we recursively insert into the left subtree. If it is greater, we recursively insert into the right subtree. Below is a concrete example.

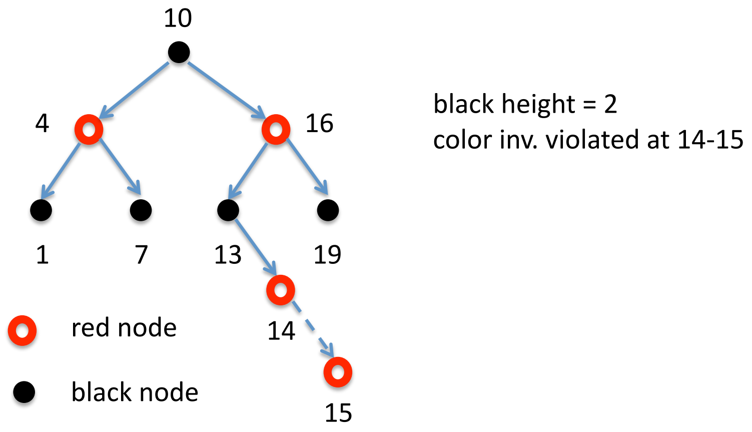


If we want to insert a new element with a key of 14, the insertion algorithm sketched above would put it to the right of 13. In order to preverse the height invariant, we must color the new node red.

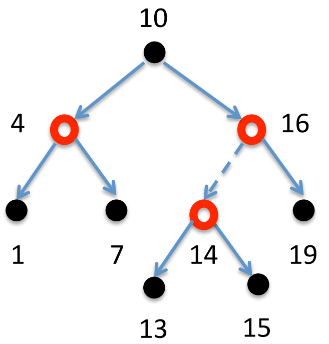


Both color and height invariants are still satisfied, as is the ordering invariant (which we always preserve).

Now consider another insertion, this time of an element with key 15. This is inserted to the right of the node with key 14. Again, to preserve the height invariant we must color it red, but this violates the color invariant since we have to adjacent red nodes.

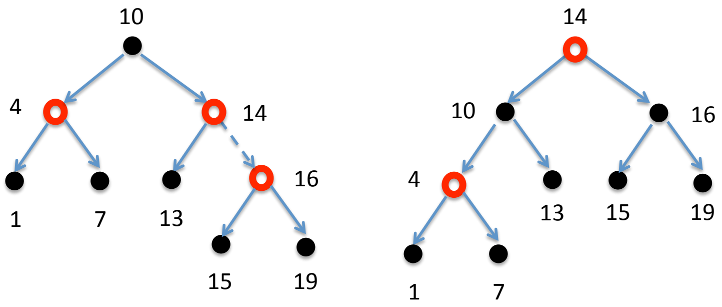


In order to restore the color invariant between 14 and 15 we can apply a *left rotation*, moving 14 up in the tree and 13 to the left. At the same time we recolor 15 to be black, so as to remove the color conflict while preserving the black height invariant. However, we introduce a new color conflict between 14 and its parent.



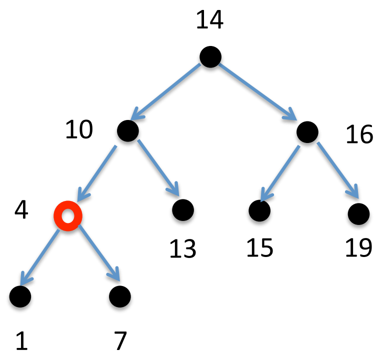
black height = 2
color inv. restored at 14-15
color inv. violated at 16-14

Now we can apply a right rotation, moving 14 up one level, immediately followed by a left rotation, moving 14 up to the root.



Such a sequence of two rotations is called a *double rotation*. The result now satisfies the height invariant (still with height 2) and the color invariant.

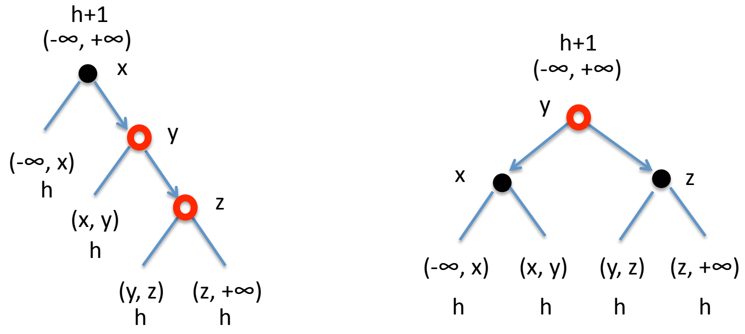
We can apply one further simple step, which is to recolor the root to black. This increases the black height on every path from the root by one, so it preserves the height invariant. Generally speaking, the fewer red nodes in the tree the fewer rotations will be forced, which is why we perform this recoloring. The final result is



black height = 3
color inv. satisfied

4 Rotations

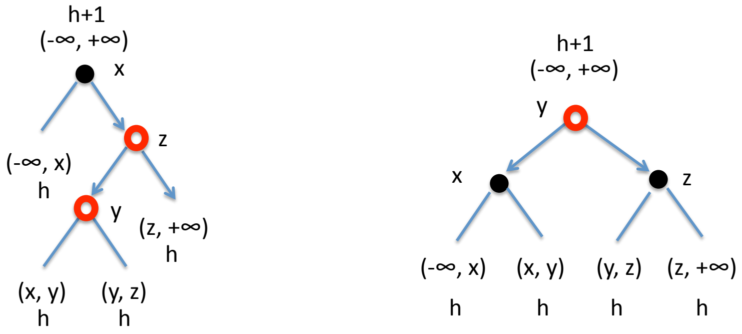
The general shape of a *left rotation* is shown in the following diagram. With each potential subtree and its interval we also show the black height of the the tree. The whole tree has black height $h + 1$ which means each elided subtree must have height h.



We see that all invariants are preserved, and the color invariant is restored. Of course, if this is a subtree below a red node, the tree on the left would satisfy the color invariant at the connection to its parent, while the tree on the right would not. We also saw this in the example.

The right rotation is entirely symmetric, so we don't show it here.

The double rotation is best thought of as a single operation, so we can examine its shape directly to verify that it preserves the invariants.



Observe that all invariants are preserved and, in fact, the tree on the right does observe the color invariant. As with a single rotation, however, we may still violate the color invariant at the interface of the node y at the top and its parent.

5 Abstract Data Types Revisited

We revisit some of the ideas underlying abstract types, as they are exhibited in this implementation. One of them is the fact that sometimes the client has to provide some operations to the implementation of an abstract type. Exactly which operations have to be supplied varies from case to case. Here, the client needs to provide the type of `elem` of elements (which must be a pointer type so it can be null), the function `elem_key` to extract keys from elements, and `compare` which compares two keys and returns their order (< 0 for *less*, $= 0$ for *equal* and > 0 for *greater*). First, the type definitions and the functions themselves.

```
/* elements */
struct elem {
  string word; /* key */
  int count;   /* information */
};

/* key comparison */
int compare(string s1, string s2) {
  return string_compare(s1,s2);
}

/* extracting keys from elements */
string elem_key(struct elem* e)
//@requires e != NULL;
{
  return e->word;
}
```

Next the interface, using the functions above.

```
/* interface definitions, implementation by client */
typedef string key;
typedef struct elem* elem;  /* NULL must be an elem */
key elem_key(elem e);
int compare(key k1, key k2);
```

Next comes the interface to the data structure itself.

```
typedef struct bst* bst;
bst bst_new();
void bst_insert(bst B, elem x);
elem bst_search(bst B, key k);  /* return NULL if not in tree */
```

This is uniform across all binary search tree implementations, so that we can experiment with different implementations without having to change the client code.

6 Defining Invariants

First, the implementation of the trees themselves. We add a boolean field `red` to the trees from the last two lectures. If it is `true`, the node is red, otherwise it should be considered black. As usual, we also have a header.

```
typedef struct tree* tree;
struct tree {
  elem data;                    /* data element */
  bool red;                     /* is node red? */
  tree left;                    /* left subtree */
  tree right;                   /* right subtree */
};
```

```
struct bst {
  tree root;                    /* root of the tree */
};

typedef bst rbt;
```

The last definition allows us to write `rbt` internally to refer to the particular implementation of binary search trees that are red/black trees.

The check that the tree is ordered does not change from the last two lectures, so we do not replicate the code here. To check that the color invariant is satisfied we just recurse over the tree, passing down the information if the parent is red. This is a common pattern of recursion.

```
/* sat_colorinv(T, red_parent) iff T satisfies the color invariant */
bool sat_colorinv(tree T, bool red_parent)
{ if (T == NULL) return true;
  if (red_parent && T->red) return false;
  return sat_colorinv(T->left, T->red)
      && sat_colorinv(T->right, T->red);
}
```

For the height invariant we calculate the black height of the two subtrees and compare. If the height invariant is violated, we return -1, otherwise the valid height invariant (which should be zero or positive).

```
/* black_height(T) >= 0 is the black height of T if it exists,
 * -1 otherwise
 */
int black_height(tree T) {
  if (T == NULL) return 0;
  {
    int hleft = black_height(T->left);
    int hright = black_height(T->right);
    if (hleft < 0 || hright < 0 || hleft != hright) return -1;
    //@assert hleft == hright;
    if (T->red) return hleft;
    else return hleft+1;
  }
}
```

A balanced tree satisfies the height invariant and the color invariant.

```
bool is_baltree(tree T) {
  int h = black_height(T);
  return h >= 0 && sat_colorinv(T, false);
}
```

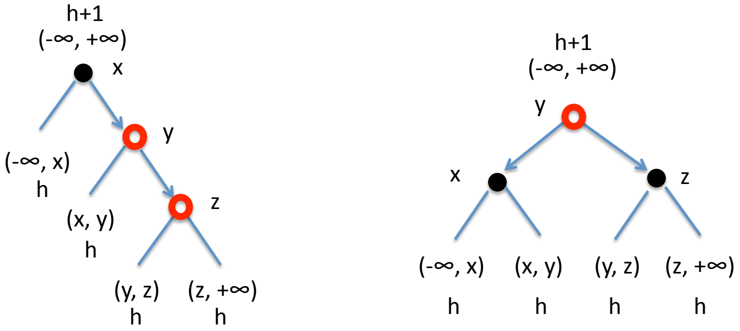
For insertion, we need to be able to check if we have a valid red/black tree with all invariants *except* that the color invariant might be violated between the root and its left child or the root and its right child. Care should be taken to make sure that we don't dereference a pointer to a tree that is potentially null.

```
bool is_rbtree_except_root(tree T) {
  bool ok = is_ordtree(T);
  ok = ok && black_height(T) >= 0;
  if (T->red
      && T->left != NULL && T->left->red
      && (T->right == NULL || !T->right->red))
    ok = ok && sat_colorinv(T->left, false)
           && sat_colorinv(T->right, false);
  else if (T->red
          && T->right != NULL && T->right->red
          && (T->left == NULL || !T->left->red))
    ok = ok && sat_colorinv(T->left, false)
           && sat_colorinv(T->right, false);
  else
    ok = ok && sat_colorinv(T, false);
  return ok;
}
```

In the first four calls to `sat_colorinv` we pass `false` as the second argument, pretending that the parent is *not* red, in order to allow the exception.

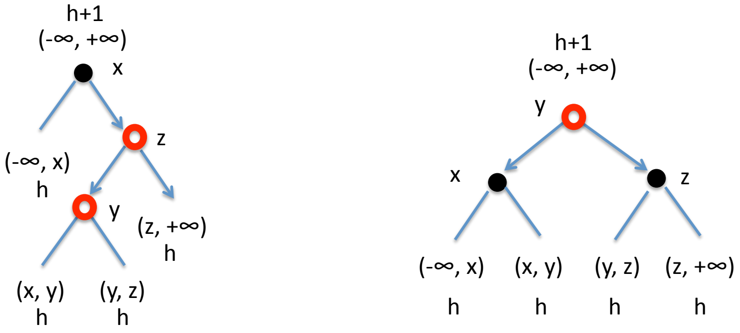
7 Implementing Insertion

Recall the picture for, say, the left rotation.



The right subtree, here labeled with key y is the result of a recursive insertion, which is allowed to violate the color invariant at its root. However, the tree we generate by adding the black node x now would violate the color invariant below x: we need to apply the rotation.

Similarly, if we have the situation on the left, with the black root



then we can apply the double rotation and restore the invariant.

So far, we get the following specification for a function that is called on the tree *after* insertion into the right subtree.

```
tree balance_right(tree T)
//@requires T != NULL;
//@requires !T->red ? is_rbtree_except_root(T->right) : ??;
//@ensures \old(!T->red) ? is_rbtree(\result) : ??;
  ;
```

In words: If the root of the tree we have to rebalance is black, then the right subtree must be a valid red/black tree except that the color invariant might be violated at its root. In turn, we guarantee that the result will be a valid red/black tree, as the diagrams above show.

But what happens if the root is red? Turns out in this case we can arrange that the right subtree is valid, without exception. This in turns means that we can immediately return, but we may violate the invariant at the root (which is permitted).

```
tree balance_right(tree T)
//@requires T != NULL;
/*@requires !T->red ? is_rbtree_except_root(T->right)
                    : is_rbtree(T->left); @*/
/*@ensures \old(!T->red) ? is_rbtree(\result)
                         : is_rbtree_except_root(\result); @*/
  ;
```

How can this work? It works because we can assert the following refined postcondition for `tree_insert`.

```
tree tree_insert(tree T, elem e)
//@requires is_rbtree(T);
/*@ensures \old(T != NULL && T->red)
           ? is_rbtree_except_root(\result)
           : is_rbtree(\result); @*/
  ;
```

If we insert into a tree with a red root, then the result may violate the invariant at the root. However, if we insert into a tree with a black or null root, then the result will be valid without exception.

Here is the complete code for insert.

```
tree tree_insert(tree T, elem e)
//@requires is_rbtree(T);
/*@ensures \old(T != NULL && T->red)
           ? is_rbtree_except_root(\result)
           : is_rbtree(\result); @*/
//@ensures tree_search(\result, elem_key(e)) == e;
{
  if (T == NULL) {  /* create new node */
    T = alloc(struct tree);
    T->data = e; T->red = true; /* new nodes are red */
    T->left = NULL; T->right = NULL;
  } else {
    key kt = elem_key(T->data);
    key k = elem_key(e);
    if (compare(k, kt) == 0) {
      T->data = e;
    } else if (compare(k, kt) < 0) {
      //@assert T->red ? (T->left == NULL || !T->left->red) : true;
      T->left = tree_insert(T->left, e);
      /*@assert T->red ? is_rbtree(T->left)
                       : is_rbtree_except_root(T->left); @*/
      T = balance_left(T);
    } else {
      //@assert compare(k, kt) > 0;
      //@assert T->red ? (T->right == NULL || !T->right->red) : true;
      T->right = tree_insert(T->right, e);
      /*@assert T->red ? is_rbtree(T->right)
                       : is_rbtree_except_root(T->right); @*/
      T = balance_right(T);
    }
  }
  return T;
}
```

Let's reason through the crucial section, when we insert into the right subtree. Inserting into the left subtree is symmetric.

```
//@assert T->red ? (T->right == NULL || !T->right->red) : true;
T->right = tree_insert(T->right, e);    /* (2) */
/*@assert T->red ? is_rbtree(T->right)
                 : is_rbtree_except_root(T->right); @*/
T = balance_right(T);                   /* (5) */
...
return T;
```

We have the following cases.

T **is red:** T is a valid red/black tree, so when its root is red, its right subtree must be null or black. Therefore, the postcondition for the recursive call in line (2) tells us that `is_rbtree(T->right)` after the assignment in line (2). This establishes the precondition for `balance_right(T)`.

From the postcondition of the call to `balance_right(T)` in line (5) we conclude that `is_rbtree_except_root(T)` after the assignment in line (5). If the original `T` is red, this is what we needed to establish.

T **is black:** In this case the postcondition of the recursive insertion only guarantees that `is_rbtree_except_root(T->right)` after the assignment in line (2). This is establishes the precondition for `balance_right(T)` in line (5). The postcondition established `is_rbtree(T)` after the assignment in line (5). If the original `T` was black, this is what we neede to establish.

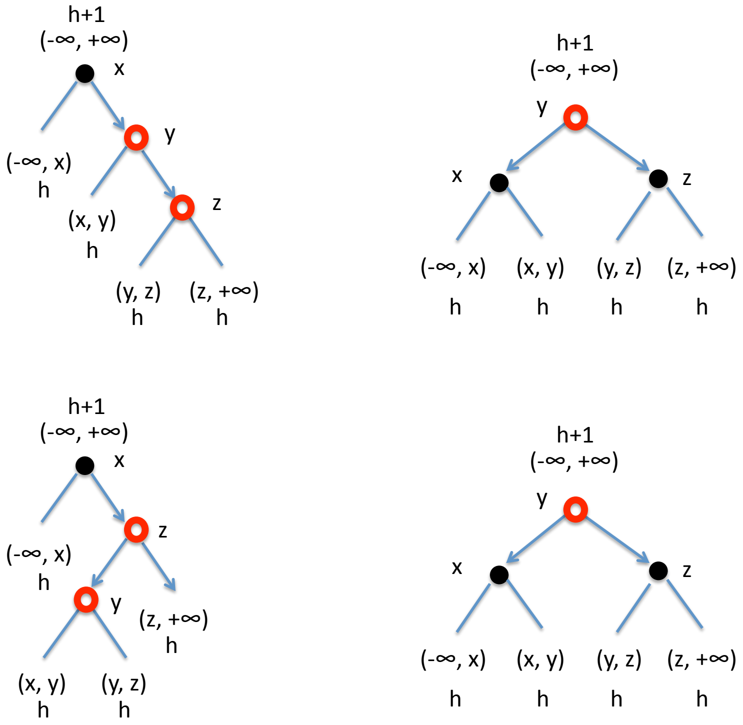
T **is null:** In this case we need to establish `is_rbtree(T)`, where T is the value returned by the insertion. But this is evident from its construction: T consists of a single red node with black height 0.

Finally, we have a wrapper function which might need to restore the invariant at the root, because the recursive insertion may leave with with the color invariant violated at the root. Fortunately, recoloring the root black restores the color invariant and maintains the black height invariant by uniformly increasing the number of black nodes on all paths to the leaves by one.

```
void bst_insert(rbt RBT, elem e)
//@requires is_rbt(RBT);
//@ensures is_rbt(RBT);
{
  // wrapper function to start the process at root
  RBT->root = tree_insert(RBT->root, e);
  //@assert is_rbtree_except_root(RBT->root);
  RBT->root->red = false; /* color root black; might already be black */
  return;
}
```

8 Implementing Rebalancing

Now that we have a precise specification, the rebalancing code practically implements itself. Here is the code for rebalancing after an insertion into the right subtree; the other one is completely symmetric.



The picture should help understand the code.

```
tree balance_right(tree T)
//@requires T != NULL;
/*@requires T->red ? is_rbtree(T->right)
                    : is_rbtree_except_root(T->right); @*/
/*@ensures \old(T->red) ? is_rbtree_except_root(\result)
                         : is_rbtree(\result); @*/
{ if (T->red) return T;
  //@assert !T->red;
  if (T->right->red
     && T->right->right != NULL && T->right->right->red)
    // rotate left
    { tree root = T->right; T->right = root->left; root->left = T;
      root->right->red = false;
      //@assert root->red == true;
      //@assert root->left->red == false;
      return root;
    } else if (T->right->red
       && T->right->left != NULL && T->right->left->red)
    // double rotate left
    { tree root = T->right->left; T->right->left = root->right;
      root->right = T->right; T->right = root->left; root->left = T;
      root->right->red = false;
      //@assert root->red == true;
      //@assert root->left->red == false;
      return root;
    } else return T;
}
```

Splay Trees

Programming II - Elixir Version

Johan Montelius

Spring Term 2018

Introduction

A splay tree is an ordered binary tree with the advantage that the last key we looked for is found in the root of the tree. We will rearrange the tree in every access, moving the key to the top and trying to keep the rest of the tree balanced.

The amortized cost of operations (search, insert, delete) are all $O(lg(n))$. There are worst case scenarios, since there is no guarantees that the tree is balanced, but we accept this since the data structure has its advantages. Frequently used keys will be found higher up in the tree so it is very good to use in an application where we expect temporal locality i.e. if you have used one key it is very likely that you will use this key again with in a short period of time.

In this assignment you're going to learn how to implement a quite tricky algorithm using pattern matching. You should look up some tutorial on splay trees so that you have a basic understanding of the algorithm but we will explain the algorithm as we go along. We will first look at the general idea of these operations using a graphical representation before going in to how to implement it in Elixir.

1 The splay tree

In this implementation we will create a splay tree that keeps key-value pairs. The key-values will reside in all of the nodes besides the leafs which are empty branches. The tree is ordered with smaller values in left branches.

Doing search in this tree is of course trivial but we will do a search where we also change the structure of the tree. This is the key operation in all splay tree operations; updating, lookup and insertion are essentially the same algorithms since they all perform the same transformation.

In this description we will first look at the operation to update or insert a value given a key. If the key is present we will update the value, if not we will add the key-value pair. Once we understand how this is done it will be easy to implement the other operations.

The splay operation is slightly different depending on if we are in the root of the tree or doing a recursive operation further down in the tree. Before explaining the general rules let's look at the rules for the root.

1.1 Splay of the root

Assume that we want to update a value for a key in a tree. We will have two very easy special cases when the tree is empty or if the key is found in the root of the tree. In these cases we simply return a tree with one node or update the existing root with the new value.

The splay operation comes in when we find the key in one of the branches. We should then rearrange the tree in a way that moves the key to root of the tree. The operation is quite straight forward as seen in the graph shown in figure 1.

Note that the lefter-most sub-tree, A, of course contains keys that are less than the key (K) that we're looking for and it is safe to have it as the left branch of the new tree. The sub-tree B contains keys that are greater than the key we are looking for but smaller than the key of the root (R) . Its position in the transformed tree is therefore sound. The same thing goes for the sub-tree C that must contain keys that are greater than the key of the root.

There is s corresponding splay operations when we find the key in the left branch. Write it down using the same naming scheme as in in figure 1. This will make the implementation step easier.

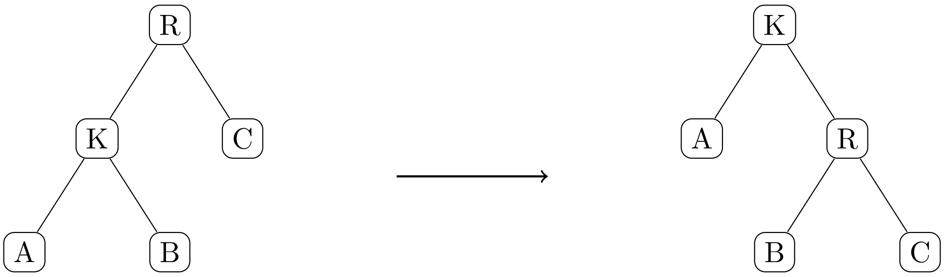


Figure 1: Zig: splay operation of the root when the key (K) is found in left branch.

1.2 Splay further down the tree

In general the splay operation on a tree will result in a tree where the key that we're updating becomes the root of the tree. The operation looks very similar to the operation that we have seen for the root of the tree, the only difference is that we now look further down the tree before determining what to do. We first describe the general rules before describing the special cases.

We have four cases and we could call them left-left, left-right etc but for historic reasons we call them: zig-zig, zig-zag, zag-zig and zag-zag. We only show the two first since the two other are mirror images of the two first.

When describing these rules we use the naming 'K' for the node where the key is found, 'P' for the parent node and 'G' for the grandparent. Sub-trees that are moved around are called 'A', 'B' etc and we write them in an order so that we know that all keys in for example sub-tree 'A' are smaller than keys in sub-tree 'B'. We will later use these names for variables in our implementation so let's try to be consistent.

Zig-zig

The zig-zig rule is used if we find the key in the lefter-left node. In figure 2 we see how we move the node holding the key we're looking for (K) to the root and rearrange the parent (P), and grandparent (G), to form the new tree. Note the order of the sub-trees A, B, C and D. Make sure that you understand why it is safe to do the transformation of the tree and why the tree is still ordered.

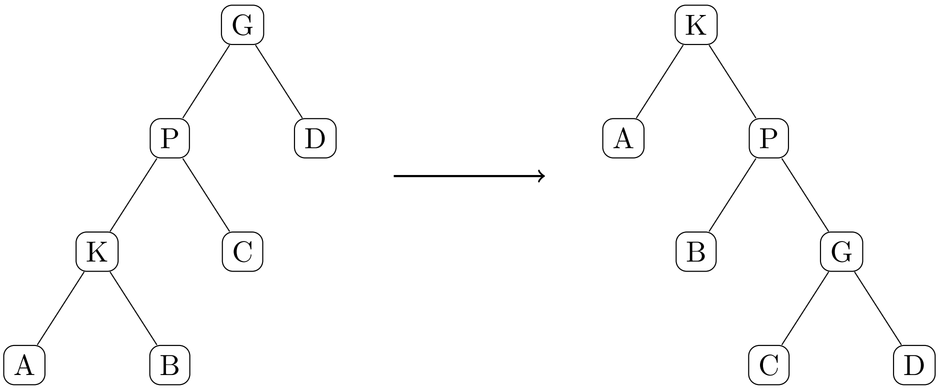


Figure 2: Zig-Zig: splay operation when key is found in left-left branch.

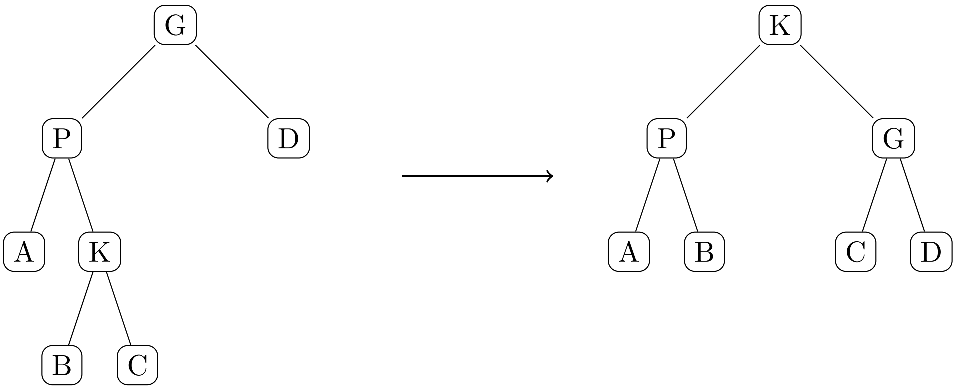


Figure 3: Zig-Zag: splay operation when the key (K) is found in left-right branch.

Zig-zag

The second rule covers the case where we find the key in the left-right node. The transformation is a little different but the aim is the same; move the key to the root and rearrange the sub-trees to keep the tree ordered. In figure 3 we see how the transformation is done.

1.3 Zag-zig and zag-zag

The two described rules of course have their mirror rules when the key is found in either the right-right or right-left node. The idea is the same, move the found key to the root and the parent node one level down. The grandparent node becomes the child of either the parent node or the key node.

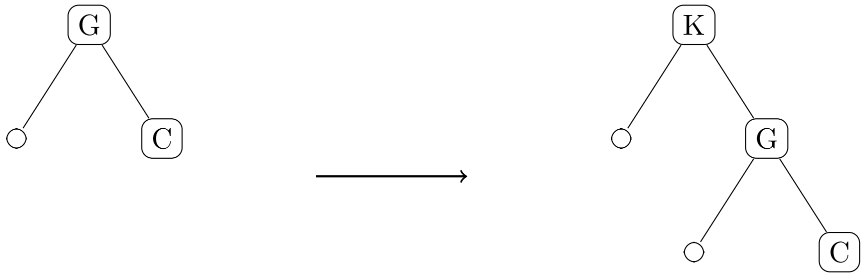


Figure 4: Zig: splay operation when key should go into left branch.

You're strongly advised to draw the graphs that describes these zag-zig and zag-zag rules. Keep the naming of nodes: G for grandparent, P for parent and K for the node where we find the key. Sub-trees are called: A, B etc and are named in order.

These are the complicated rules, what we have left are the very simple rules for some base cases.

1.4 Zig or zag or nil

We have several base cases that cover the situations where the tree is not very deep or when we find the key that we're looking for in one of the two first levels. In the same way as for the root we could have an empty tree or a tree where the key is in the root; these cases are trivial. If the key is found in the root of either sub-tree we do a transformation that is exactly the same transformation that we would do for the root.

If the sub-tree where the key should be found is empty we do the same operation as we would do if we found the key in the root of the sub-tree. An example of what this looks like is seen in figure 4.

The rule is equivalent to the zig rule that we used in the root of the tree. It of course has its corresponding zag versions where the key should go into the right branch.

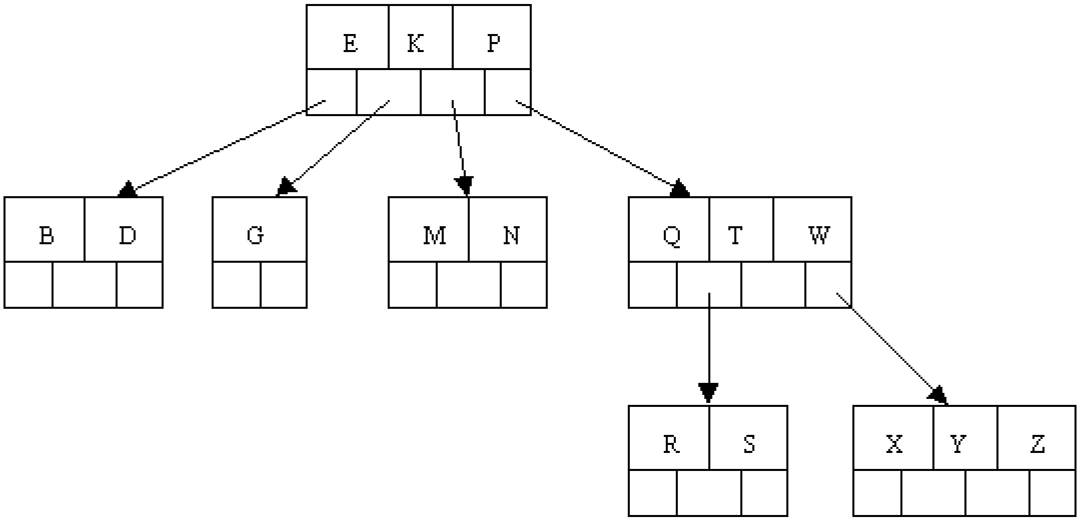
B-Trees

Introduction

A B-tree is a specialized multiway tree designed especially for use on disk. In a B-tree each node may contain a large number of keys. The number of subtrees of each node, then, may also be large. A B-tree is designed to branch out in this large number of directions and to contain a lot of keys in each node so that the height of the tree is relatively small. This means that only a small number of nodes must be read from disk to retrieve an item. The goal is to get fast access to the data, and with disk drives this means reading a very small number of records. Note that a large node size (with lots of keys in the node) also fits with the fact that with a disk drive one can usually read a fairly large amount of data at once (perhaps 1024 bytes).

Definitions

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



What does it mean to say that the keys and subtrees are "arranged in the fashion of a search tree"?

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

- The keys in each node are in ascending order.
- At every given node (call it Node) the following is true:
  - The subtree starting at record Node.Branch[0] has only keys that are less than Node.Key[0].
  - The subtree starting at record Node.Branch[1] has only keys that are greater than Node.Key[0] and

at the same time less than Node.Key[1].
    - The subtree starting at record Node.Branch[2] has only keys that are greater than Node.Key[1] and at the same time less than Node.Key[2].
    - The subtree starting at record Node.Branch[3] has only keys that are greater than Node.Key[2].
- Note that if less than the full number of keys are in the Node, these 4 conditions are truncated so that they speak of the appropriate number of keys and branches.

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

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

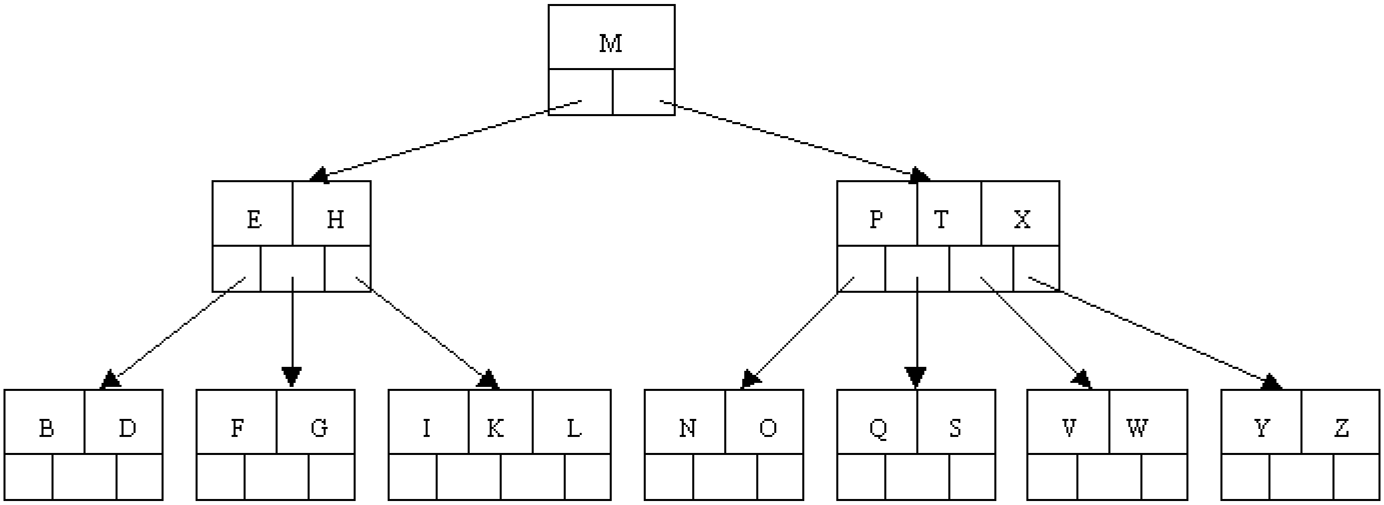
- All leaves are on the bottom level.
- All internal nodes (except the root node) have at least ceil(m / 2) (nonempty) children.
- The root node can have as few as 2 children if it is an internal node, and can obviously have no children if the root node is a leaf (that is, the whole tree consists only of the root node).
- Each leaf node must contain at least ceil(m / 2) - 1 keys.

Note that ceil(x) is the so-called ceiling function. It's value is the smallest integer that is greater than or equal to x. Thus ceil(3) = 3, ceil(3.35) = 4, ceil(1.98) = 2, ceil(5.01) = 6, ceil(7) = 7, etc.

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

Example B-Tree

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



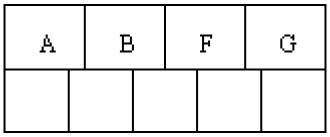
Operations on a B-Tree

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

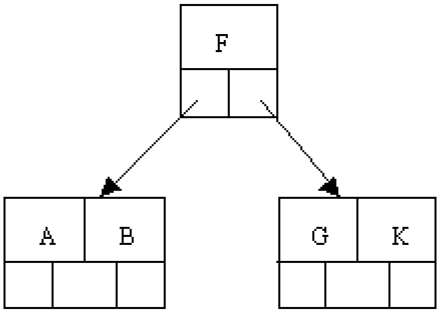
Inserting a New Item

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

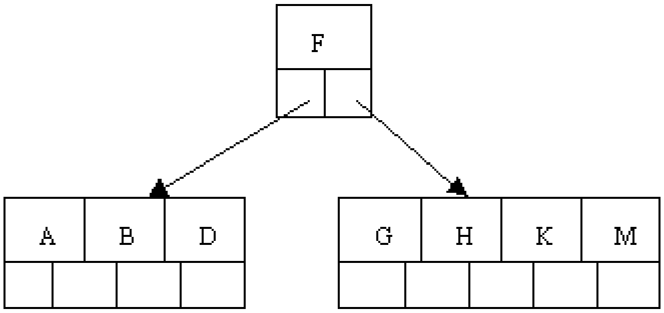
An example: Insert the following letters into what is originally an empty B-tree of order 5: A G F B K D H M J E S I R X C L N T U P. Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:



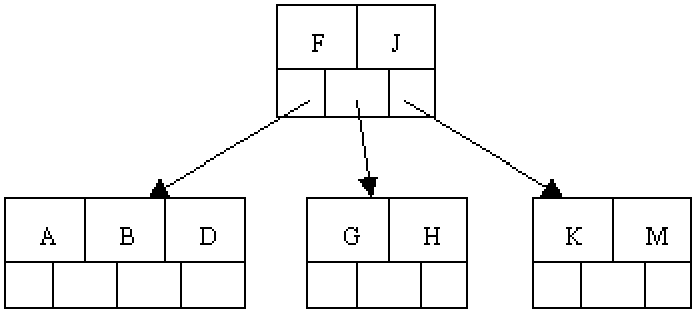
When we try to insert the K, we find no room in this node, so we split it into 2 nodes, moving the median item F up into a new root node. Note that in practice we just leave the A and B in the current node and place the G and K into a new node to the right of the old one.



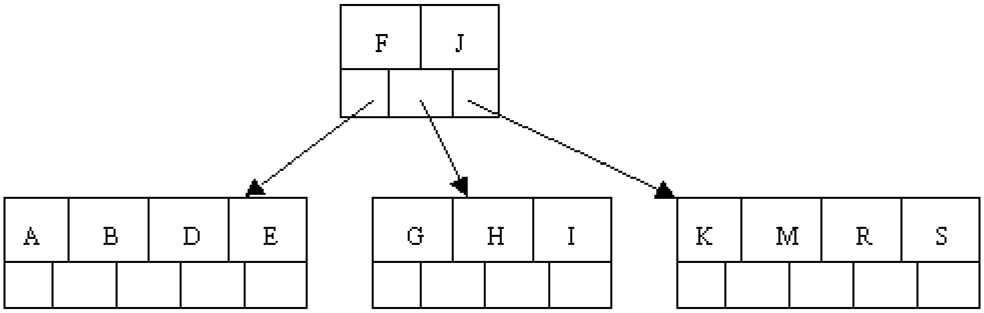
Inserting D, H, and M proceeds without requiring any splits:



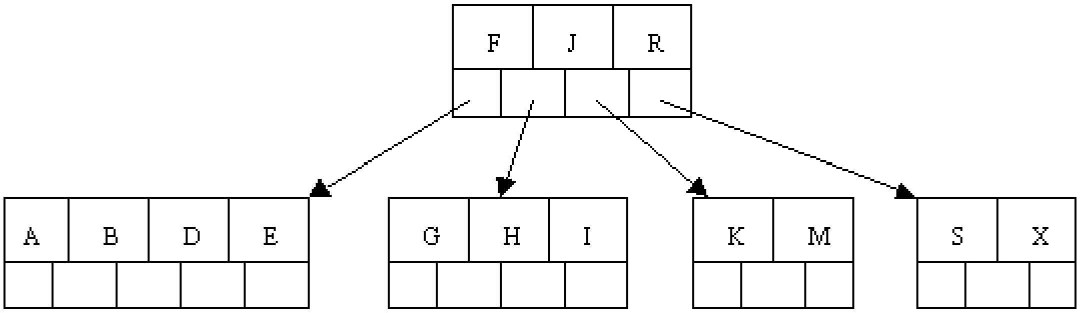
Inserting J requires a split. Note that J happens to be the median key and so is moved up into the parent node.



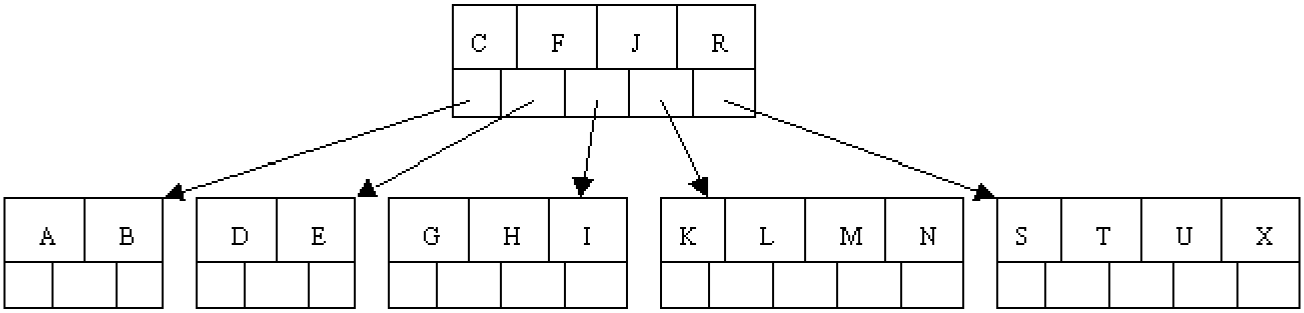
The letters E, S, I, and R are then added without needing any split.



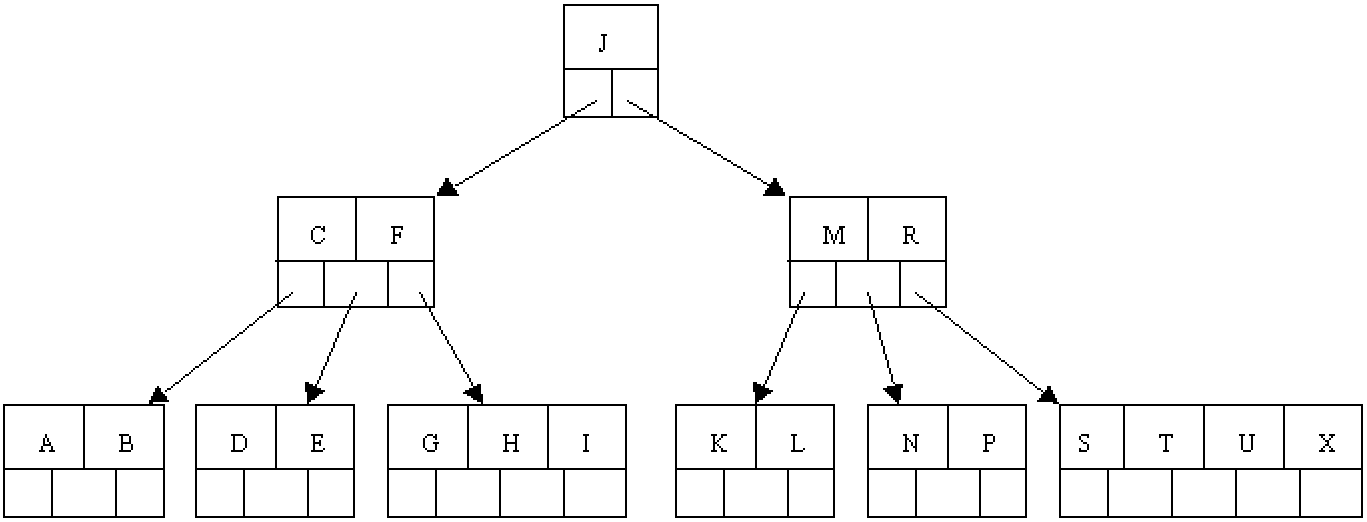
When X is added, the rightmost leaf must be split. The median item R is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.



The insertion of C causes the leftmost leaf to be split. C happens to be the median key and so is the one moved up into the parent node. The letters L, N, T, and U are then added without any need of splitting:

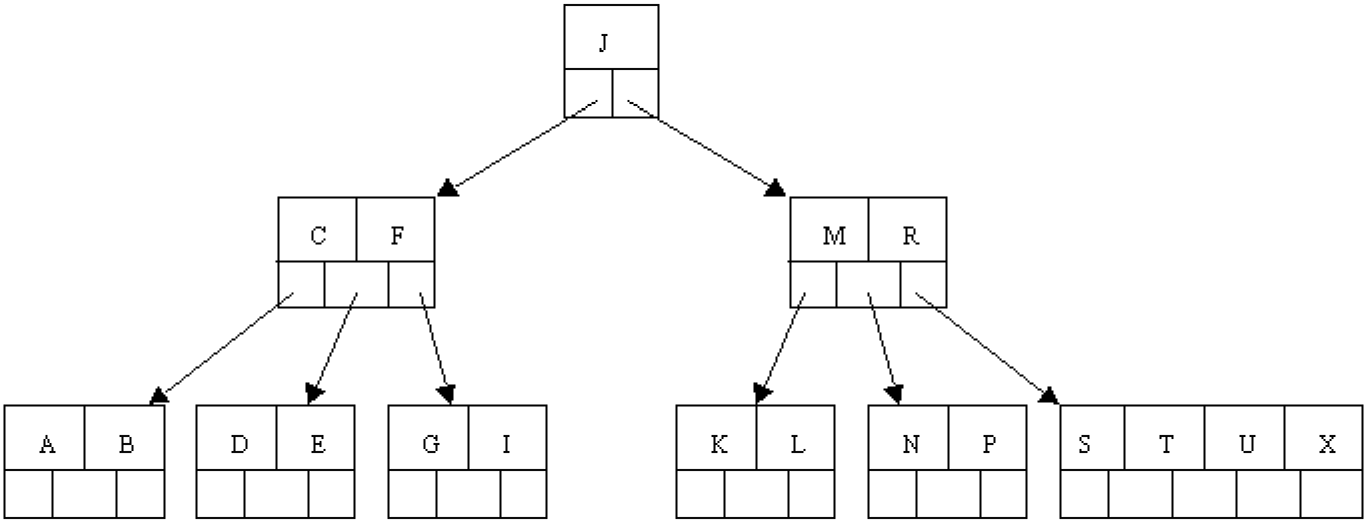


Finally, when P is added, the node with K, L, M, and N splits sending the median M up to the parent. However, the parent node is full, so it splits, sending the median J up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains C and F.

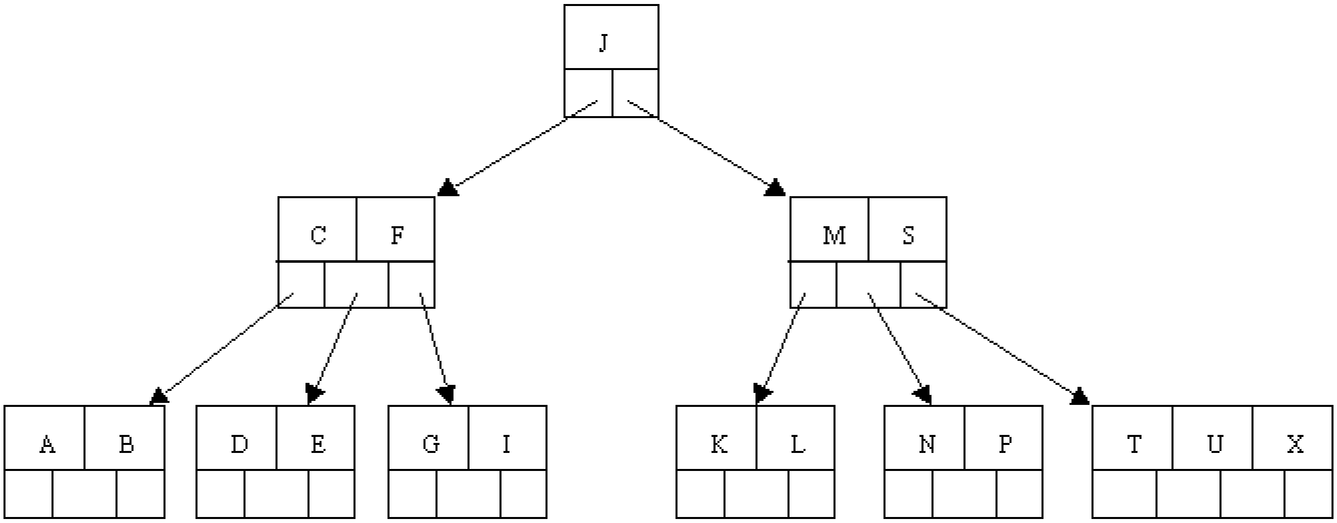


Deleting an Item

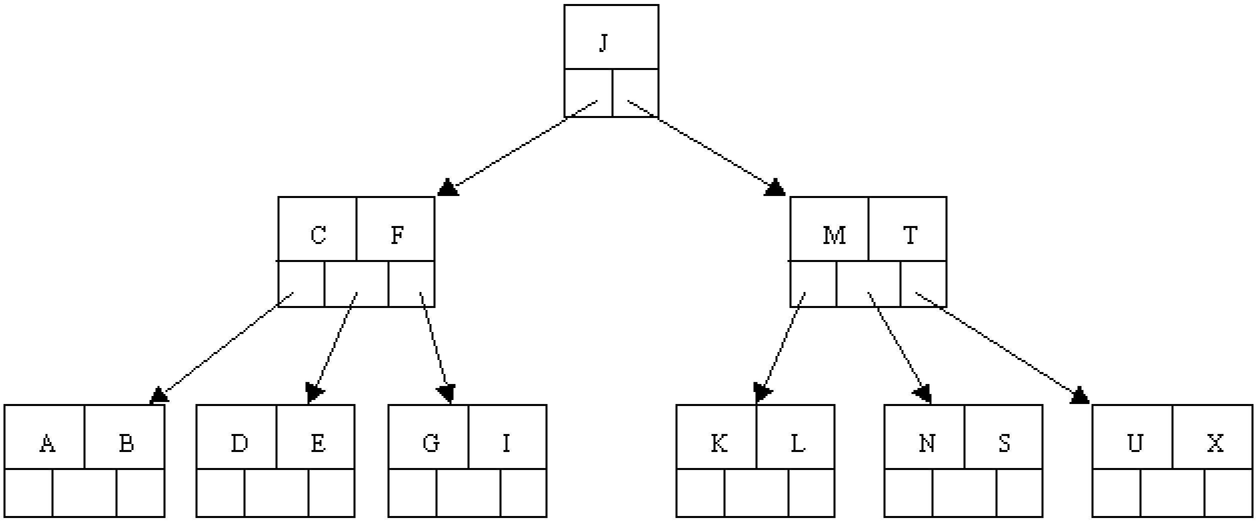
Let's again follow an example. In the B-tree as we left it at the end of the last section, delete H. Of course, we first do a lookup to find H. Since H is in a leaf and the leaf has more than the minimum number of keys, this is easy. We move the I over where the H had been. This gives:



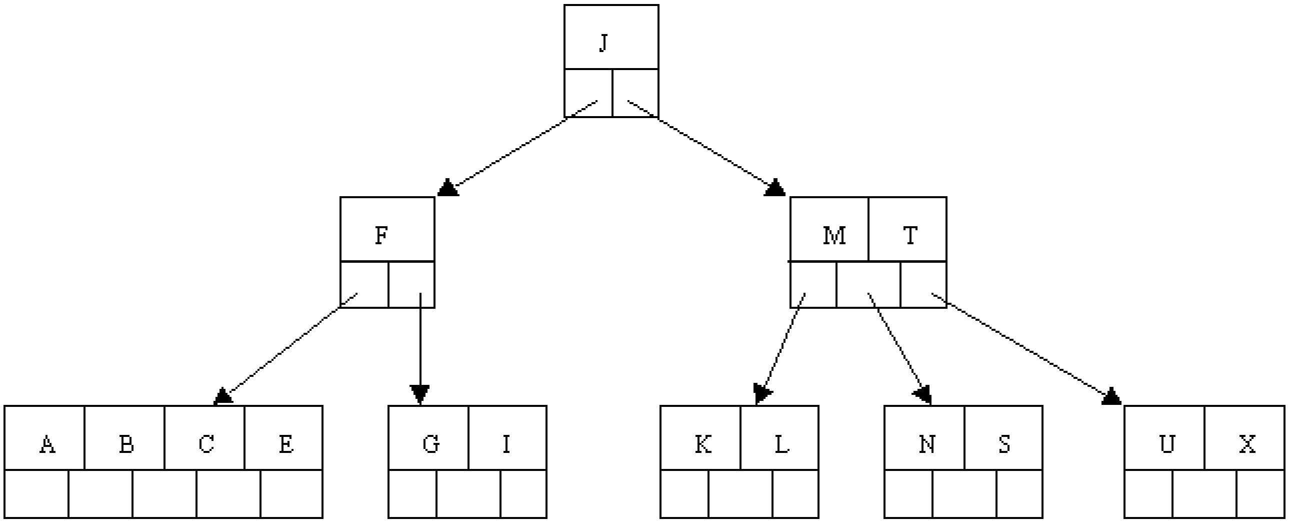
Next, delete the R. Since R is not in a leaf, we find its successor (the next item in ascending order), which happens to be S, and move S up to replace the R. That way, what we really have to do is to delete S from the leaf, which we already know how to do, since this leaf has extra keys. In ALL cases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



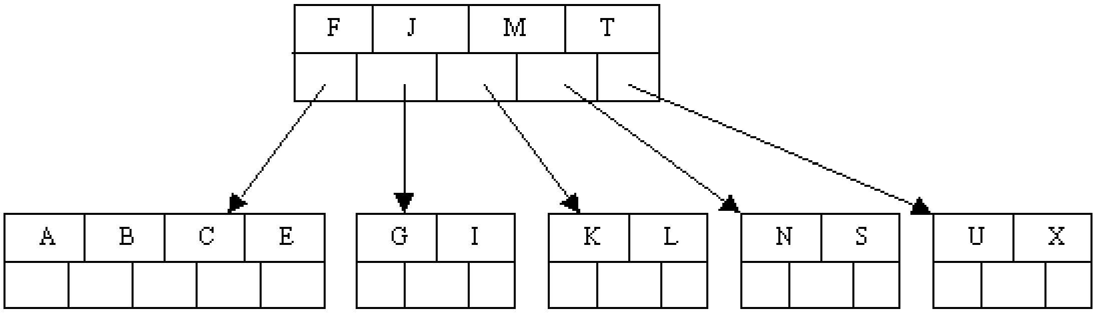
Next, delete P. Although P is in a leaf, this leaf does not have an extra key; the deletion results in a node with only one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we can then borrow a key from the parent and move a key up from this sibling. In our specific case, the sibling to the right has an extra key. So, the successor of P, which is S, is moved down from the parent, and the T is moved up.



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

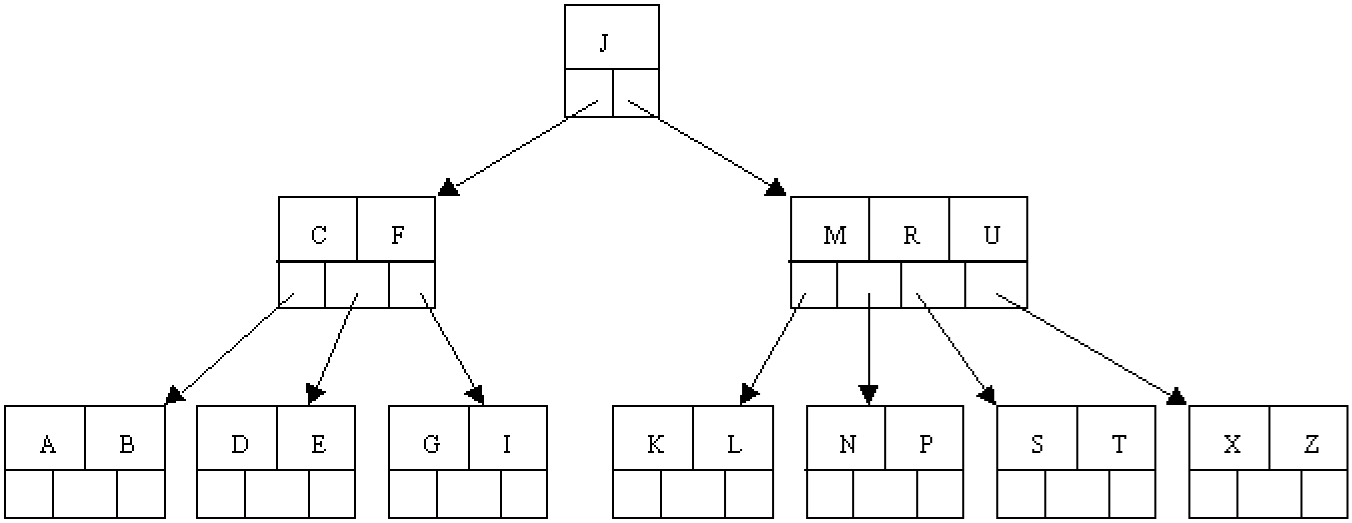


Of course, you immediately see that the parent node now contains only one key, F. This is not acceptable. If this problem node had a sibling to its immediate left or right that had a spare key, then we would again "borrow" a key. Suppose for the moment that the right sibling (the node with M T) had one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

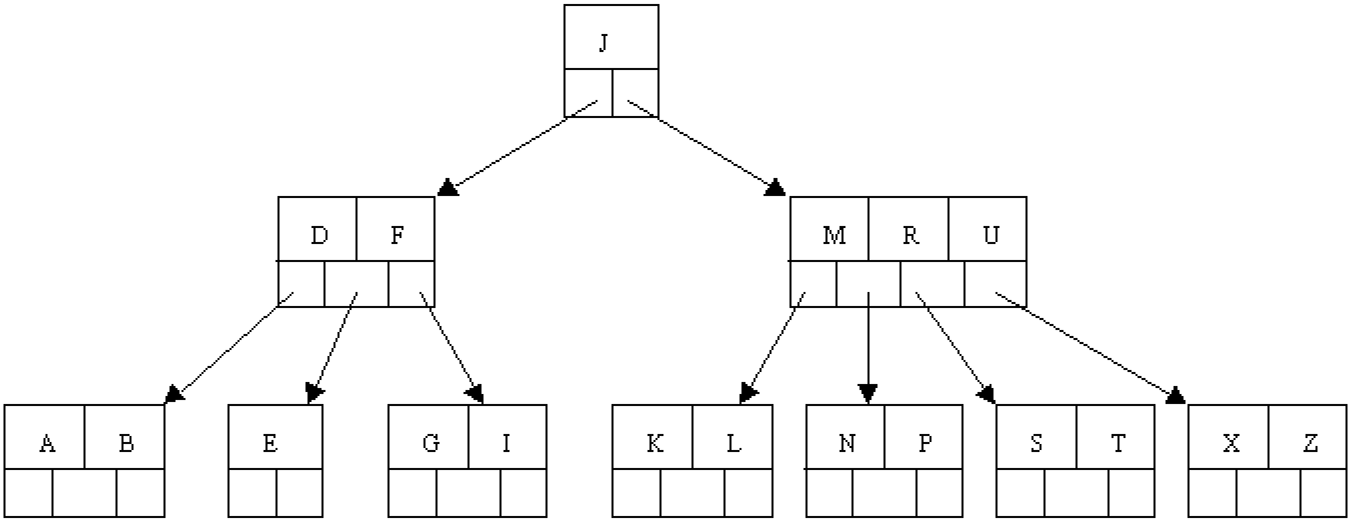


Another Example

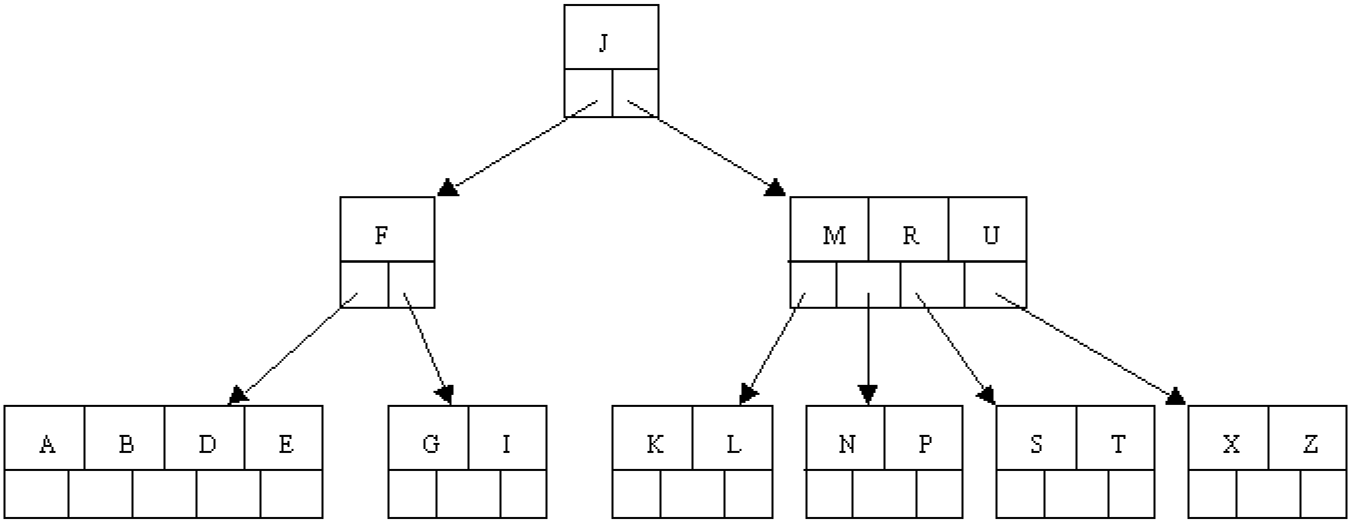
Here is a different B-tree of order 5. Let's try to delete C from it.



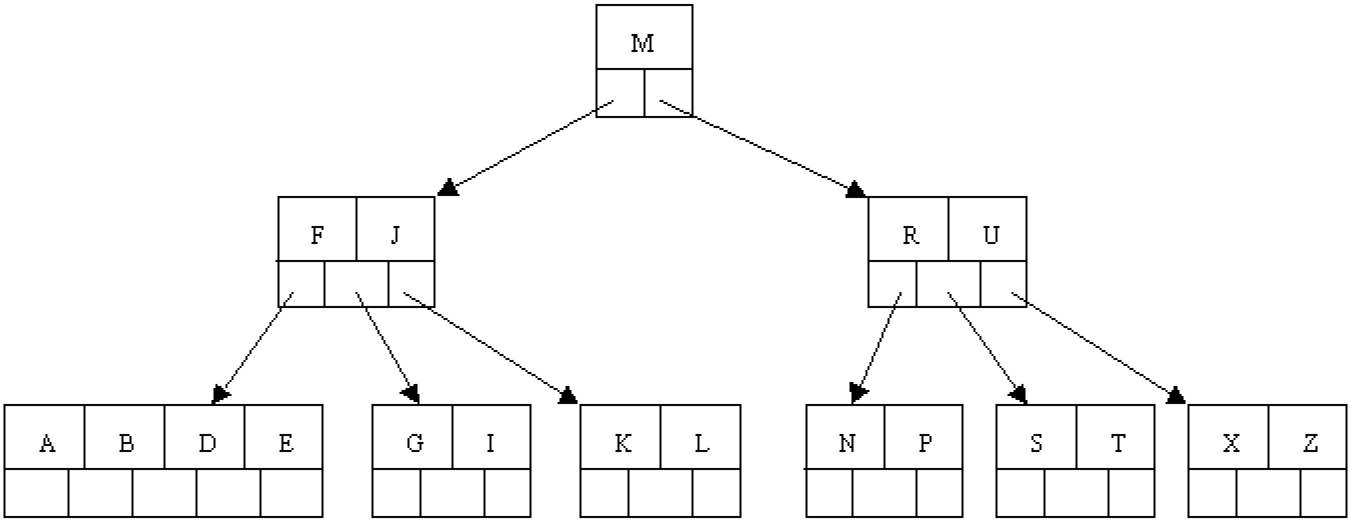
We begin by finding the immediate succesor, which would be D, and move the D up to replace the C. However, this leaves us with a node with too few keys.



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



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



Variations

The B-tree and variations on it are commonly used in large commercial databases to provide quick access to the data. In fact, he says that they are "the standard file organization for applications requiring insertion, deletion, and key range searches". The variant called the B+ tree is the usual one. Another variant is the B* tree, which is very similar to the B+ tree, but tries to keep the nodes about two-thirds full at a minimum.

In a B+ tree, data records are only stored in the leaves. Internal nodes store just keys. These are used for directing a search to the proper leaf. If a target key is less than a key in an internal node, then the pointer just to its left is followed. If a target key is greater or equal to the key in the internal node, then the pointer just to its right is followed. The leaves are also linked together so that all of the data records in the B+ tree can be traversed in ascending order, just by going through all of the nodes in this linked list along the bottom level of the tree.

As an example, consider a B+ tree of order 100, whose leaves can contain up to 100 records. A 2 level B+ tree can store up to 10,000 records. A 3 level B+ tree can store up to 1 million records. A 4 level B+ tree can store up to 100 million records. To get faster access to the data, the root node is commonly kept in main memory. Maybe even the child nodes of the root can fit in main memory. Thus one can find one of 100 million records with only 2 or 3 disk reads.

Note that a B-tree (or B+ tree, etc.) can be used as an index file. The actual data records are stored elsewhere on disk. When one looks up and finds a target key, what one finds is the target key and an associated pointer (disk block number or whatever) to the appropriate data record. This means that one data file could have several such indices, each giving an ordering by a different key field, something highly desirable to have.

References:

Data Structures & Program Design, 2nd ed. Robert L. Kruse. Prentice-Hall (1987). There is also a third edition (1994) as well as a new text, Data Structures and Program Design in C++, authored by Kruse and Alexander J. Ryba (1999).
A Practical Introduction to Data Structures and Algorithm Analysis Clifford A. Shaffer. Prentice-Hall (1997).

Adapted from: http://cis.stvincent.edu/carlsond/swdesign/btree/btree.html

(appeared in GInfo 15/7, November 2005)

Suffix arrays – a programming contest approach

Adrian Vladu and Cosmin Negruşeri

Summary

An important algorithmic area often used in practical tasks is that of algorithms on character strings. Many programming contests have used problems that could have been easily solved if one efficiently managed to determine if one string was a subsequence of another string, or had found an order relation within a string's suffixes. We shall present a versatile structure that allows this along with other useful operations on a given string.

Keywords: suffix sorting, suffix arrays, suffix trees

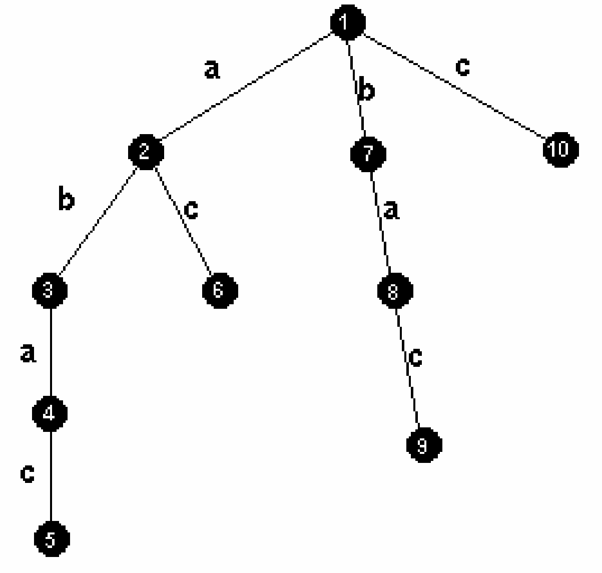
1 Introduction

What are suffix arrays?

In order to let the reader gain a better vista on suffix arrays, we shall make a short presentation of two data structures called **trie**, respectively suffix tree [1] – which is a special case of a trie. A trie is a tree meant to store strings. Each of its nodes will have a number of sons equal to the size of the alphabet used by the strings that are needed to be stored. In our case, with strings containing only small letters of the English alphabet, each node will have at most 26 sons. Every edge going from the father toward its sons is labeled with a different letter of the alphabet. The labels on a path starting from the root and ending in a leaf will form a string stored in that tree. As it can be easily seen, finding whether a string is contained in this data structure is very efficient and is done in O(M) time, where M is the string's length. Therefore, the searching time does not depend on the number of words stored in the structure, this making it an ideal structure for implementing dictionaries.

Let's see what a suffix trie is:
Given a string A = $a_0a_1…a_{n-1,}$ denote by $A_i = a_ia_{i+1}…a_{n-1}$ the suffix of A that begins at position i. Let n = length of A. The suffix trie is made by compressing all the suffixes of $A_1…A_{n-1}$ into a trie, as in the figure below.

The suffix trie corresponding to the string "abac" is:



Operations on this structure are very easily done:

- checking whether a string W is a substring of A – it is enough to traverse the nodes starting from the root and going through the edges labeled correspondingly to the characters in W (complexity O(|W|))
- searching the longest common prefix for two suffixes of A – choose nodes u and v in the trie, corresponding to the ends of the two suffixes, then, with a LCA algorithm (least common ancestor), find the node corresponding to the end of the searched prefix. For example, for “abac” and “ac”, the corresponding nodes are 5 and 6. Their least common ancestor is 2, that gives the prefix “a”. The authors are strongly recommending [2] for an $O(\sqrt{n})$ solution, [3] for an accessible presentation of a solution in O(lg n) or O(1), and [4] for a state of the art algorithm.
- finding the k-th suffix in lexicographic order - (complexity O(n), with a corresponding preprocessing). For example, the 3rd suffix of “abac” is represented in the trie by the 3rd leaf.

Even if the idea of a suffix trie would be very pleasing at first sight, the simplist implementation, where at every step one of the strings suffixes is inserted into the structure leads to an $O(n^2)$ complexity algorithm.There is a structure called suffix tree[1] that can be built in linear time, which is a suffix trie where the chains containing only nodes with the out-degree equal to 1 were compressed into a single edge (in the example above, these are represented by the chains 2 -3 – 4 – 5 and 1 – 7 – 8 – 9). Implementing the linear algorithm is scarcely possible in a short time, such as during a contest, this determining us to search another structure, easier to implement.

Let’s see which are the suffixes of A, by a depth first traversal of the trie. Noting that during the depth first search we have to consider the nodes in the ascending lexicographic order of the edges linking them to their father, we gain the following suffix array:

abac	**= A_0**
ac	**= A_2**
bac	**= A_1**
c	**= A_3**

It is easy to see that these are sorted ascending. To store them, it is not necessary to keep a vector of strings; it is enough to maintain the index of every suffix in the sorted array. For the example above, we get the array **P = (0, 2, 1, 3),** this being the suffix array for the string “abac”.

Introduction to Algorithms March 18, 2004
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik Demaine and Shafi Goldwasser Handout 14

Lecture Notes on Skip Lists

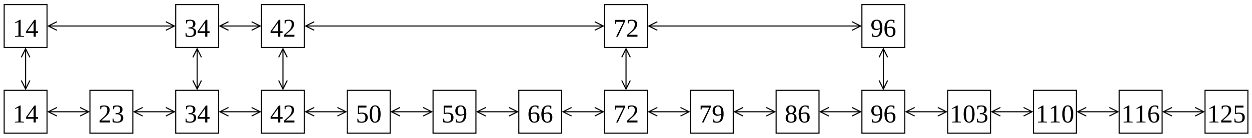
Lecture 12 — March 18, 2004

Erik Demaine

- Balanced tree structures we know at this point: B-trees, red-black trees, treaps.
- Could you implement them right now? Probably, with time... but without looking up any details in a book?
- Skip lists are a simple randomized structure you'll never forget.

Starting from scratch

- Initial goal: *just searches* — ignore updates (Insert/Delete) for now
- Simplest data structure: linked list
- Sorted linked list: $\Theta(n)$ time
- 2 sorted linked lists:
  - Each element can appear in 1 or both lists
  - How to speed up search?
  - **Idea:** Express and local subway lines
  - **Example:** $\boxed{14}$, 23, $\boxed{34}$, $\boxed{42}$, 50, 59, 66, $\boxed{72}$, 79, 86, $\boxed{96}$, 103, 110, 116, 125 (What is this sequence?)
  - $\boxed{\text{Boxed}}$ values are "express" stops; others are normal stops
  - Can quickly jump from express stop to next express stop, or from any stop to next normal stop
  - Represented as two linked lists, one for express stops and one for all stops:



  - Every element is in linked list 2 (LL2); some elements also in linked list 1 (LL1)
  - Link equal elements between the two levels
  - To search, first search in LL1 until about to go too far, then go down and search in LL2

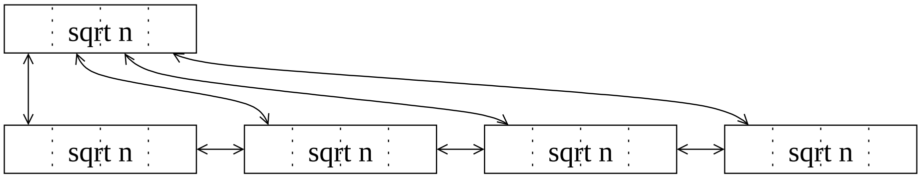
- Cost:

$$\text{len}(\text{LL1}) + \frac{\text{len}(\text{LL2})}{\text{len}(\text{LL1})} = \text{len}(\text{LL1}) + \frac{n}{\text{len}(\text{LL1})}$$

- Minimized when

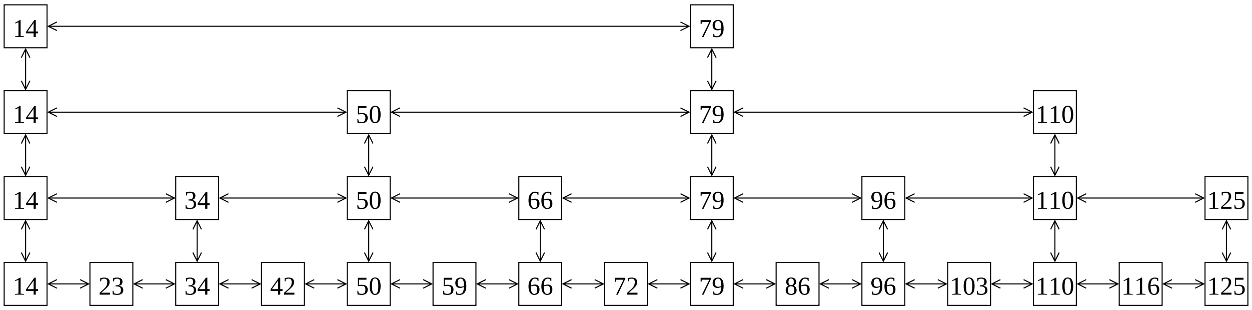
$$\begin{aligned} & \text{len}(\text{LL1}) = \frac{n}{\text{len}(\text{LL1})} \\ \Rightarrow\ & \text{len}(\text{LL1})^2 = n \\ \Rightarrow\ & \text{len}(\text{LL1}) = \sqrt{n} \\ \Rightarrow\ & \text{search cost} = 2\sqrt{n} \end{aligned}$$

- Resulting 2-level structure:



- 3 linked lists: $3 \cdot \sqrt[3]{n}$
- k linked lists: $k \cdot \sqrt[k]{n}$
- $\lg n$ linked lists: $\lg n \cdot \sqrt[\lg n]{n} = \lg n \cdot \underbrace{n^{1/\lg n}}_{=2} = \Theta(\lg n)$

  - Becomes like a binary tree:



  - **Example:** Search for 72
    - ∗ Level 1: 14 too small, 79 too big; go down 14
    - ∗ Level 2: 14 too small, 50 too small, 79 too big; go down 50
    - ∗ Level 3: 50 too small, 66 too small, 79 too big; go down 66
    - ∗ Level 4: 66 too small, 72 spot on

Insert

- New element should certainly be added to bottommost level
  (Invariant: Bottommost list contains all elements)

- Which other lists should it be added to?
  (Is this the entire balance issue all over again?)

- **Idea:** Flip a coin
  - With what probability should it go to the next level?
  - To mimic a balanced binary tree, we'd like half of the elements to advance to the next-to-bottommost level
  - So, when you insert an element, flip a fair coin
  - If heads: add element to next level up, and flip another coin (repeat)

- Thus, on average:
  - $1/2$ the elements go up 1 level
  - $1/4$ the elements go up 2 levels
  - $1/8$ the elements go up 3 levels
  - Etc.

- Thus, "approximately even"

Example

- Get out a real coin and try an example

- You should put a special value $-\infty$ at the beginning of each list, and always promote this special value to the highest level of promotion

- This forces the leftmost element to be present in every list, which is necessary for searching

... many coins are flipped ...
(Isn't this easy?)

- The result is a skip list.

- It probably isn't as balanced as the ideal configurations drawn above.

- It's clearly good on average.

- Claim it's really really good, almost always.

Analysis: Claim of With High Probability

- **Theorem:** *With high probability, every* search costs $\Theta(\lg n)$ in a skip list with n elements

- What do we need to do to prove this? [Calculate the probability, and show that it's high!]

- We need to define the notion of "with high probability"; this is a powerful technical notion, used throughout randomized algorithms

- **Informal definition:** An event occurs ***with high probability*** if, for any $\alpha \geq 1$, there is an appropriate choice of constants for which E occurs with probability at least $1 - O(1/n^\alpha)$

- In reality, the constant hidden within $\Theta(\lg n)$ in the theorem statement actually depends on c.

- **Precise definition:** A (parameterized) event E_α occurs ***with high probability*** if, for any $\alpha \geq 1$, E_α occurs with probability at least $1 - c_\alpha/n^\alpha$, where c_α is a "constant" depending only on α.

- The term $O(1/n^\alpha)$ or more precisely c_α/n^α is called the ***error probability***

- The idea is that the error probability can be made very very very small by setting α to something big, e.g., 100

Analysis: Warmup

- **Lemma:** With high probability, skip list with n elements has $O(\lg n)$ levels

- (In fact, the number of levels is $\Theta(\log n)$, but we only need an upper bound.)

- **Proof:**

  - Pr[element x is in more than $c \lg n$ levels] $= 1/2^{c \lg n} = 1/n^c$
  - Recall Boole's inequality / union bound:

    $$\Pr[E_1 \cup E_2 \cup \cdots \cup E_n] \leq \Pr[E_1] + \Pr[E_2] + \cdots + \Pr[E_n]$$

  - Applying this inequality:
    Pr[any element is in more than $c \lg n$ levels] $\leq n \cdot 1/n^c = 1/n^{c-1}$
  - Thus, error probability is polynomially small and exponent ($\alpha = c - 1$) can be made arbitrarily large by appropriate choice of constant in level bound of $O(\lg n)$

Analysis: Proof of Theorem

- **Cool idea:** Analyze search backwards—from leaf to root
  - Search starts at leaf (element in bottommost level)
  - At each node visited:
    - If node wasn't promoted higher (got TAILS here), then we go [came from] left
    - If node wasn't promoted higher (got HEADS here), then we go [came from] top
  - Search stops at root of tree
- Know height is $O(\lg n)$ with high probability; say it's $c \lg n$
- Thus, the number of "up" moves is at most $c \lg n$ with high probability
- Thus, search cost is at most the following quantity:

  How many times do we need to flip a coin to get $c \lg n$ heads?
- Intuitively, $\Theta(\lg n)$

Analysis: Coin Flipping

- **Claim:** Number of flips till $c \lg n$ heads is $\Theta(\lg n)$ with high probability
- Again, constant in $\Theta(\lg n)$ bound will depend on α
- **Proof of claim:**
  - Say we make $10c \lg n$ flips
  - When are there at least $c \lg n$ heads?
  - Pr[exactly $c \lg n$ heads] $= \underbrace{\binom{10c\lg n}{c\lg n}}_{\substack{\text{orders}\\ \text{HHHTTT vs. HTHTHT}}} \cdot \underbrace{\left(\frac{1}{2}\right)^{c\lg n}}_{\text{heads}} \cdot \underbrace{\left(\frac{1}{2}\right)^{9c\lg n}}_{\text{tails}}$
  - Pr[at most $c \lg n$ heads] $\le \underbrace{\binom{10c\lg n}{c\lg n}}_{\substack{\text{overestimate}\\ \text{on orders}}} \cdot \underbrace{\left(\frac{1}{2}\right)^{9c\lg n}}_{\text{tails}}$
  - Recall bounds on $\binom{y}{x}$:

$$\left(\frac{y}{x}\right)^x \le \binom{y}{x} \le \left(e\,\frac{y}{x}\right)^x$$

  [Michael's "deathbed" formula: even on your deathbed, if someone gives you a binomial and says "simplify", you should know this!]

  - Applying this formula to the previous equation:

$$\begin{aligned}
\text{Pr[at most } c\lg n \text{ heads]} &\le \binom{10c\lg n}{c\lg n}\left(\frac{1}{2}\right)^{9c\lg n} \\
&\le \left(\frac{e\cdot 10c\lg n}{c\lg n}\right)^{c\lg n}\cdot\left(\frac{1}{2}\right)^{9c\lg n} \\
&= (10e)^{c\lg n}\cdot\left(\frac{1}{2}\right)^{9c\lg n} \\
&= 2^{\lg(10e)\cdot c\lg n}\cdot\left(\frac{1}{2}\right)^{9c\lg n} \\
&= 2^{(\lg(10e)-9)c\lg n} \\
&= 2^{-\alpha\lg n} \\
&= 1/n^{\alpha}
\end{aligned}$$

  - The point here is that, as $10 \to \infty$, $\alpha = 9 - \lg(10e) \to \infty$, independent of (for all) c

- End of proof of claim and theorem

Acknowledgments

The mysterious “Michael” is Michael Bender at SUNY Stony Brook. This lecture is based on discussions with him.