CodeHype

DSA Notes

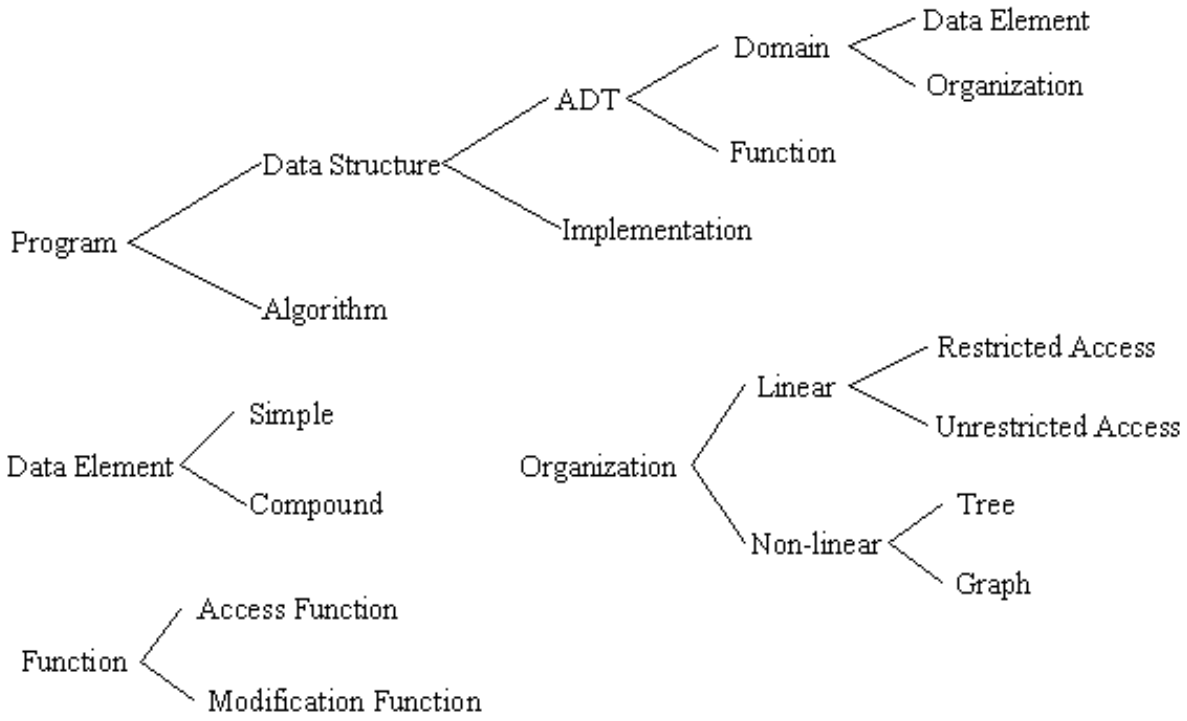
@codehype_

@codehype

Chapter 1

Introduction

Overview (Data Structures, ADTs, and Algorithms)



Why data structures?

Ultimate goal of data structures is to write efficient programs. In order to do that, one needs to organize the data in such a way that it can be accessed and manipulated efficiently.

Data structure

A data structure is used for the storage of data in computer so that data can be used efficiently. For the organization of mathematical and logical concepts data structure provides a methodology. With the proper selection of data structure you can also get efficient algorithm. With very few resources like memory space and time critical operations can be carried out with a well designed data structure. The major use of data structure is its implementation in the programming language.

Moreover, there are different kinds of data structures and they have different uses. Some data structures are used for specialized tasks like B-trees are used for the implementation of databases and networks of machines use routing tables.

A **data structure** is a way of storing data in a computer so that it can be used efficiently. Often a carefully chosen data structure will allow the most efficient algorithm to be used. The choice of the data structure often begins from the choice of an abstract data type. A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible. Data structures are implemented by a programming language as data types and the references and operations they provide.

"Collection of data *elements* organized in a specified manner and a set of functions to store, retrieve and manipulate the individual data elements."

"The way of representing data internally in the memory is called data structure" Or "A data structure is a way of *store* data in a computer so that it can be used efficiently"

A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms.

Mathematcial Model	⟹	Abstract Dat Type	⟹	Data Structures
Informal Algorithm	⟹	Pseudo Language Program	⟹	Program in C or Java or .....

The Problem Solving Process in Computer Science

We use data structure because without it, the use and storage of data would be impossible. Whenever you view or process data on a computer system, there is always a data structure behind what you are doing. Different types of data structures are used for varying kinds of data - for instance, word documents will have a different data structure than spreadsheets. Data structures act as the fundamental foundations of all computer software processes.

- Data structures have a number of great advantages, including those listed below:

- Data structures allow information to be securely stored on a computer system. Most data structures require only a small proportion of a computer's memory capacity. Storing data is

convenient and allows it to be accessed at any time. It is also impossible to lose the information, as you might if it was on paper.

- Data structures provide you with the capacity to use and process your data on a software system. For instance, if you wished to log your work hours and produce a report, you could do so on a computer through an automated process. This process would make wide use of data structures.
- Data structures make all processes involving them very quick.

- On the other hand, data structures do have some disadvantages

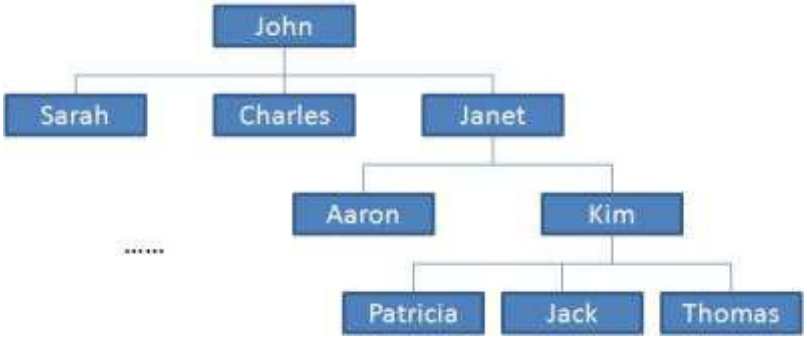
- To alter data structures, you must be a very advanced IT technician with a vast experience base. It is almost impossible to change most data structures.

- If you have a problem relating to data structures, it is highly unlikely you will be able to solve it without the expertise of a professional.

Example: Suppose you are hired to create a database of names with all company's management and employees.

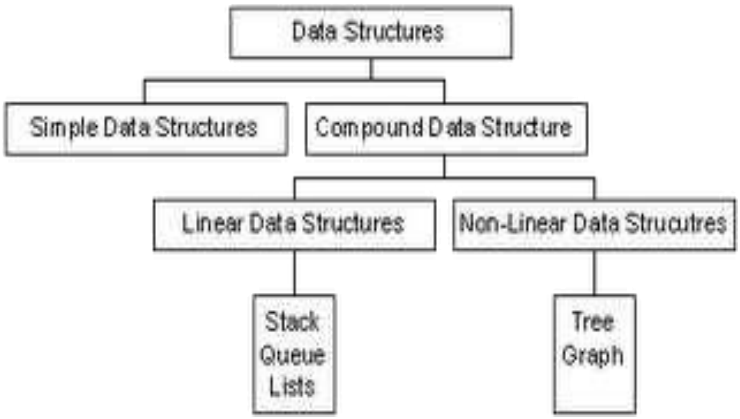
You can make a list. You can also make a tree.

name	position
Aaron	Manager
Charles	VP
George	Employee
Jack	Employee
Janet	VP
John	President
...	...



Types of data structure

In all, it can be seen that data structures are highly beneficial, not least because they allow the most basic pieces of computer software to function correctly. Without data structures, we would not have the modern computer of today.



Data structures can be classified as

- Simple data structure
- Compound data structure
- Linear data structure
- Non linear data structure

Simple Data Structure:

Simple data structure can be constructed with the help of primitive data structure. A primitive data structure used to represent the standard data types of any one of the computer languages. Variables, arrays, pointers, structures, unions, etc. are examples of primitive data structures.

Compound Data structure

Compound data structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user. It can be classified as

- Linear data structure
- Non-linear data structure

Linear data structure:

Collection of nodes which are logically adjacent in which logical adjacency is maintained by pointers

Or

Linear data structures can be constructed as a continuous arrangement of data elements in the memory. It can be constructed by using array data type. In the linear Data Structures the relationship of adjacency is maintained between the Data elements.

Operations applied on linear data structure:

The following list of operations applied on linear data structures

- Add an element
- Delete an element
- Traverse
- Sort the list of elements
- Search for a data element

By applying one or more functionalities to create different types of data structures

For example: *Stack, Queue, Tables, List, and Linked Lists.*

Non-linear data structure:

Non-linear data structure can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear Data structure the relationship of adjacency is not maintained between the Data items.

Operations applied on non-linear data structures:

The following list of operations applied on non-linear data structures.

- Add elements
- Delete elements
- Display the elements
- Sort the list of elements
- Search for a data element

By applying one or more functionalities and different ways of joining randomly distributed data items to create different types of data structures. *For example Tree, Decision tree, Graph and Forest*

Some Definitions

We provide below informal definitions of a few important, common notions that we will frequently use in this lecture notes.

Algorithm

A finite sequence of instructions, each of which have a clear meaning and can be executed with a finite amount of effort in finite time is called algorithm. Whatever the input values, an algorithm will definitely terminate after executing a finite number of instructions.

Data Type

Data type of a variable is the set of values that the variable may assume.

Basic data types in C:

- **int**
- **char**
- **float**
- **double**

Abstraction

The first thing that we need to consider when writing programs is the *problem*. However, the problems that we asked to solve in real life are often nebulous or complicated. Thus we need to distill such as system down to its most fundamental parts and describe these parts in a simple,

precise language. This process is called abstraction. Through abstraction, we can generate a model of the problem, which defines an abstract view of the problem. Normally, the model includes the data that are affected and the operations that are required to access or modify.

As an example, consider a program that manages the student records. The head of the Bronco Direct comes to you and asks you to create a program which allows administering the students in a class. Well, this is too vague a problem. We need to think about, for example, what student information is needed for the record? What tasks should be allowed?

There are many properties we could think of about a student, such as name, DOB, SSN, ID, major, email, mailing address, transcripts, hair color, hobbies, etc. Not all these properties are necessary to solve the problem. To keep it simple, we assume that a student's record includes the following fields: (1) the student's name and (2) ID. The three simplest operations performed by this program include (1) adding a new student to the class, (2) searching the class for a student, given some information of the student, and (3) deleting a student who has dropped the class. These three operations can be furthermore defined as below:

- **ADD (stu_record)**: This operation adds the given student record to the collection of student records.
- **SEARCH (stu_record_id)**: This operation searches the collection of student records for the student whose ID has been given.
- **DELETE (stu_record_id)**: This operation deletes the student record with the given ID from the collection.

Now, we have modeled the problem in its most abstract form: listing the types of data we are interested in and the operations that we would like to perform on the data. We have not discussed anything about how these student records will be stored in memory and how these operations will be implemented. This kind of abstraction defines an ***abstract data type*** (**ADT**).

Abstract Data Type (ADT):

An ADT is a set of elements with a collection of well defined operations.

- The operations can take as operands not only instances of the ADT but other types of operands or instances of other ADTs.
- Similarly results need not be instances of the ADT
- At least one operand or the result is of the ADT type in question.

An ADT is a mathematical model of a data structure that specifies the type of **data** stored, the **operations** supported on them, and the types of **parameters of the operations**. An ADT specifies what each operation does, but not how it does it. Typically, an ADT can be implemented using one of many different data structures. A useful first step in deciding what data structure to use in a program is to specify an ADT for the program.

In general, the steps of building ADT to data structures are:

1. Understand and clarify the nature of the target information unit.
2. Identify and determine which data objects and operations to include in the models.
3. Express this property somewhat formally so that it can be understood and communicate well.
4. Translate this formal specification into proper language. In C++, this becomes a .h file. In Java, this is called "user interface".

Upon finalized specification, write necessary implementation. This includes storage scheme and operational detail. Operational detail is expressed as separate functions (methods

Object-oriented languages such as C++ and Java provide explicit support for expressing ADTs by means of *classes*.

Examples of ADTs include list, stack, queue, set, tree, graph, etc.

Data Structures:

An implementation of an ADT is a translation into statements of a programming language,

- the declarations that define a variable to be of that ADT type

- the operations defined on the ADT (using procedures of the programming language)
  An ADT implementation chooses a *data structure* to represent the ADT. Each data structure is built up from the basic data types of the underlying programming language using the available data structuring facilities, such as arrays, records (structures in C), pointers, files, sets, etc.

Example:

A ``Queue" is an ADT which can be defined as a sequence of elements with operations such as null (Q), empty (Q), enqueue(x, Q), and dequeue (Q). This can be implemented using data structures such as

- array
- singly linked list
- doubly linked list
- circular array

Chapter 2

STACK

Stacks are the subclass of lists that permits the insertion and deletion operation to be performed at only one end. They are LIFO (last in first out) list. An example of a stack is a railway system for shunting cars in this system, the last railway car to be placed on the stack is the first to leave.

Operation on stacks

A pointer TOP keeps track of the top element in the stack. Initially when the stack is empty, TOP has a value of zero and when the stack contains a single element; TOP has a value of one and so on. Each time a new element is inserted in the stack, the pointer is incremented by one before the element is placed in the stack. The pointer is decremented by one each time a deletion is made from the stack.

PUSH Operation

This is the insertion operation.when the value is entered it is inserted into an array.

- In this algorithm, stack S is being used which can store maximum n element. The stack element are S [0], S [1]… S [n-1].
- Top keeps track of top of the stack. An item 'VAL' is added to the stack provided it is already not full i.e. TOP<n
- When the stack is full, it is called stack-full condition. This condition is also called stack overflow.

Algorithm:

Step 1: If (TOP=n-1)

Write "STACK FULL".
goto step 4.

Step 2: Set TOP = TOP + 1
Step 3: Set S [TOP] = VAL
Step 4: Stop.

POP Operation

- This operation is used to remove a data item form the stack.
- In this algorithm, stack S is being used which can store maximum n elements. The stack element are S [0], S [1]… S [n-1].
- Top keeps track of top of stack. In this algorithm an item is removed from the top of the stack.

- When there is no element in the stack and the pop is performed it is called Stack-empty condition. This condition is also called Stack underflow condition.

Algorithm:

Step 1: if (TOP = - 1)
a. write "STACK EMPTY".
b. goto step 4.
Step 2: Set val s[TOP]
Step 3: Set TOP TOP-1
Step 4: Stop.

An underflow condition is checked in the first step .If there is an underflow then appropriate actions should take place.

Peep

This operation is to read the value on the top of the stack without removing it.

Algorithm

Peep(S,TOP,I).Given a vector S of N elements representing a sequentially allocated stack, and a pointer TOP denoting the top element of the stack, this function returns the value of the i^{th} element from the top of the stack. The element is not deleted by this function.

1. [Check for stack underflow]

   if TOP-I+1<=0

   then Write ("stack underflow on peep")

   action in response to underflow

   Exit

2. [Return i^{th} element from the top of the stack]

   Return(s[TOP-I+1])

Applications

The linear data structure stack can be used in the following situations.
1. It can be used to process function calls.
2. Implementing recursive functions in high level languages
3. Converting and evaluating expressions.

Function calls:

A stack is useful for the compiler/operating system to store local variables used inside a function block, so that they can be discarded once the control comes out of the function block.

Recursive functions:

The stack is very much useful while implementing recursive functions. The return values and addresses of the function will be pushed into the stack and the lastly invoked function will first return the value by popping the stack.

Representation of expressions:-

In general there are 3 kinds of expressions available depending on the placement of the operators & operands.

- Infix expression
  It is the general notation used for representing expressions."In this expression the operator is fixed in between the operands"
  Ex: a + bc

- Post fix expression :- (Reverse polish notation)
  "In this expression the operator is placed after the operands".
  Ex : abc+

- Prefix expression :- (Polish notation)

  "In this expression the operators are followed by operands i.e the operators are fixed before the operands"
  Ex : +abc

All the infix expression will be converted into post fix notation with the help of stack in any program. The stack will be useful in evaluating the postfix expressions also.

The simplest application of a stack is to reverse a word. You push a given word to stack -letter by letter - and then pop letters from the stack.

Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.

Backtracking

This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?

Once you reach a dead end, you must backtrack. But backtrack to where? To the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then, backtracking simply means popping a next choice from the stack.

Language processing

- Space for parameters and local variables is created internally using a stack.
- Compiler's syntax check for matching braces is implemented by using stack.
- support for recursion

Last in, first out stack	
COMMAND	**STACK CONTENTS**
PUSH 7 Add value 7 to the stack and top of stack is 7.	7
PUSH 8 Add value 8 to the stack and top of stack is 8.	8 7
PUSH 1	1 8 7
PUSH 4	4 1 8 7
POP 4 "4" is removed and top of stack is 1.	1 8 7
POP 1 "1" is removed and top of stack is 8.	8 7

Program to implement stack using array

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

class stack
{
        int stk[5];
        int top;
    public:
        stack()
         {
           top=-1;
         }
        void push(int x)
         {
           if(top >  4)
                {
                   cout <<"stack over flow";
                   return;
                }
           stk[++top]=x;
           cout <<"inserted" <<x;
         }
        void pop()
          {
            if(top <0)
             {
                 cout <<"stack under flow";
                 return;
             }
            cout <<"deleted" <<stk[top--];
          }
        void display()
          {
             if(top<0)
              {
                    cout <<" stack empty";
                    return;
              }
              for(int i=top;i>=0;i--)
              cout <<stk[i] <<" ";
           }
};
```

```
main()
{
    int ch;
    stack st;
    while(1)
      {
          cout <<"\n1.push  2.pop  3.display  4.exit\nEnter your choice";
          cin >> ch;
          switch(ch)
           {
            case 1:  cout <<"enter the element";
                     cin >> ch;
                     st.push(ch);
                     break;
            case 2:  st.pop();  break;
            case 3:  st.display();break;
            case 4:  exit(0);
           }
       }
return (0);
}
```

Infix transformation to Postfix

This process uses a stack as well. We have to hold information that's expressed inside parentheses while scanning to find the closing ')'. We also have to hold information on operations that are of lower precedence on the stack. The algorithm is:

Infix Expression:

Any expression in the standard form like "2*3-4/5" is an Infix (Inorder) expression.

Postfix Expression:

The Postfix (Postorder) form of the above expression is "23*45/-".

Infix to Postfix Conversion:

- Scan the Infix string from left to right.
- Initialize an empty stack.
- If the scanned character is an operand, add it to the Postfix string. If the scanned character is an operator and if the stack is empty push the character topStack.

- If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack (topStack). If topStack has higher precedence over the scanned character, pop the stack else push the scanned character to stack. Repeat this step as long as stack is not empty and topStack has precedence over the character.

- Repeat this step till all the characters are scanned.

- If the current input token is '(', push it onto the stack
- If the current input token is ')', pop off all operators and append them to the output string until a '(' is popped; discard the '('.
- If the end of the input string is found, pop all operators and append them to the output string.
- Return the Postfix string.

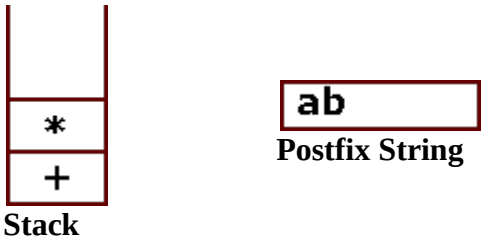
Example:

Let us see how the above algorithm will be implemented using an example.

Infix String: a+b*c-d

Initially the Stack is empty and our Postfix string has no characters. Now, the first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. It being an operator, it is pushed to the stack.

Next character scanned is 'b' which will be placed in the Postfix string. Next character is '*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '*', so '*' will be pushed to the stack.



The next character is 'c' which is placed in the Postfix string. Next character scanned is '-'. The topmost character in the stack is '*' which has a higher precedence than '-'. Thus '*' will be popped

out from the stack and added to the Postfix string. Even now the stack is not empty. Now the topmost element of the stack is '+' which has equal priority to '-'. So pop the '+' from the stack and add it to the Postfix string. The '-' will be pushed to the stack.

Stack

Postfix String

Next character is'd' which is added to Postfix string. Now all characters have been scanned so we must pop the remaining elements from the stack and add it to the Postfix string. At this stage we have only a '-' in the stack. It is popped out and added to the Postfix string. So, after all characters are scanned, this is how the stack and Postfix string will be:

Stack

Postfix String

End result:

- Infix String : a+b*c-d

Postfix String: abc*+d-

Postfix Evaluation

Infix Expression:

Any expression in the standard form like "2*3-4/5" is an Infix (Inorder) expression.

Postfix Expression:

The Postfix (Postorder) form of the above expression is "23*45/-".

Postfix Evaluation:

In normal algebra we use the infix notation like a+b*c. The corresponding postfix notation is abc*+. The algorithm for the conversion is as follows:

- Scan the Postfix string from left to right.

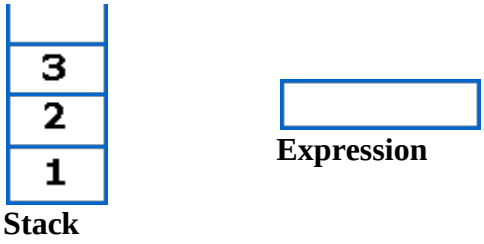
- Initialize an empty stack.
- If the scanned character is an operand, add it to the stack. If the scanned character is an operator, there will be at least two operands in the stack.
    - If the scanned character is an Operator, then we store the top most element of the stack (topStack) in a variable temp. Pop the stack. Now evaluate topStack (Operator) temp. Let the result of this operation be retVal. Pop the stack and Push retVal into the stack.
    - Repeat this step till all the characters are scanned.
- After all characters are scanned, we will have only one element in the stack. Return topStack.

Example:

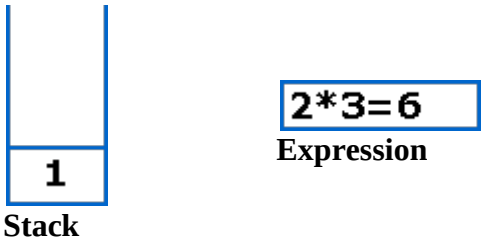
Let us see how the above algorithm will be implemented using an example.

Postfix String: 123*+4-

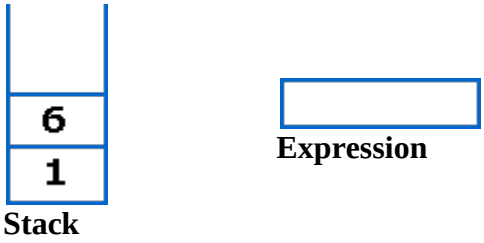
Initially the Stack is empty. Now, the first three characters scanned are 1, 2 and 3, which are operands. Thus they will be pushed into the stack in that order.



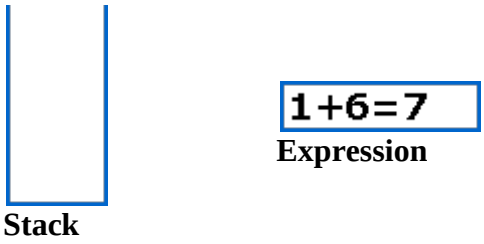
Next character scanned is "*", which is an operator. Thus, we pop the top two elements from the stack and perform the "*" operation with the two operands. The second operand will be the first element that is popped.



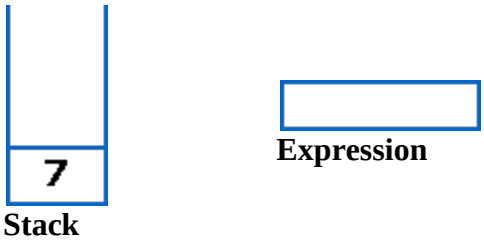
The value of the expression (2*3) that has been evaluated (6) is pushed into the stack.



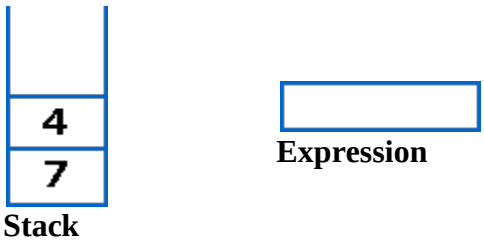
Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.



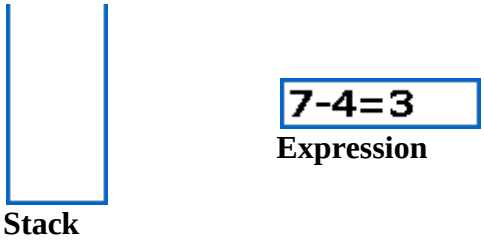
The value of the expression (1+6) that has been evaluated (7) is pushed into the stack.



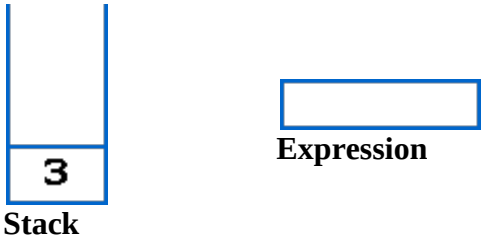
Next character scanned is "4", which is added to the stack.



Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression (7-4) that has been evaluated (3) is pushed into the stack.



Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned.

End result:

- Postfix String : 123*+4-

Result: 3

Abstract data types vs Data structures

Before starting that, we need to be clear about the logical and implementation level of data. Let us take an example of a simple built-in data type integer. At the logic level, we application programmers know only about what are the operations an integer data type can perform, ie. Addition, subtraction etc., But we are no way aware of how the data type is actually implemented. At the implementation level, it is about how the data type integer is implemented in the machine level, i.e., it could either of binary-coded decimal, unsigned binary, sign-and-magnitude binary, One's complement and Two's complement notation.

Now, for the understanding the ADT and data structure, we need to assume a higher level abstraction where we have the built-in types at the implementation level.

To put it simple, ADT is a logical description and data structure is concrete. ADT is the logical picture of the data and the operations to manipulate the component elements of the data. Data structure is the actual representation of the data during the implementation and the algorithms to manipulate the data elements. ADT is in the logical level and data structure is in the implementation level.

As you can see, ADT is implementation independent. For example, it only describes what a data type List consists (data) and what are the operations it can perform, but it has no information about how the List is actually implemented.

Whereas data structure is implementation dependent, as in the same example, it is about how the List implemented i.e., using array or linked list. Ultimately, data structure is how we implement the data in an abstract data type.

Chapter 3

Queue

Queue is the concept used in data structures. It is also referred to the linear data structure as the object or item can be added in one terminal and the item can be retrieved from the other end. Item, which is added to one end, is called as REAR and the item that is retrieved from the other end is called as FRONT. Queue is used in array as well as in linked list of the computer programs mainly used by the programmers. Queue follows the concept of **FIFO (First-In-First-Out).** It means that the items, which are enters first will be accessed or retrieved first.

Best Example

The best example to represent the queue is the print jobs. Consider the systems are connected in the network with the common print using the print share methodology. So, whenever the users give their documents for printing, the jobs will be stored in a queue and the job which first enters the print queue will be printed first, which compiles the concept of queue FIFO.

Queue Operations

The following are operations performed by queue in data structures

- Enqueue (Add operation)
- Dequeue (Remove operation)

Enqueue

This operation is used to add an item to the queue at the rear end. So, the head of the queue will be now occupied with an item currently added in the queue. Head count will be incremented by one after addition of each item until the queue reaches the tail point. This operation will be performed at the rear end of the queue.

Dequeue

This operation is used to remove an item from the queue at the front end. Now the tail count will be decremented by one each time when an item is removed from the queue until the queue reaches the head point. This operation will be performed at the front end of the queue.

Let us now consider the following example of using the operation Enqueue and Dequeue. A queue of an array A [3] is initialized and now the size of the queue is 3 which represents the queue can hold a maximum of 3 items.

The enqueue operation is used to add an item to the queue.

Enqueue (3) – This will add an item called 3 to the queue in the front end. (rear count will be incremented by 1)

Again we are adding another item to the queue

Enqueue (5) – This will add an item called 5 to the queue (rear count will be incremented by 1)

Again we are adding the last item to the queue

Enqueue (8) – This will add an item called 8 to the queue (Now the queue has reached its maximum count and hence no more addition of an item is possible in the queue)

Now dequeue operation is performed to remove an item from the queue.

Dequeue () – This will remove the item that has been added first in the queue, i.e., the item called 3 by following the concept of FIFO. Now the queue consists of the remaining items 5 and 8 in which the object 5 will be in the front end. The dequeue operation continues until the queue is reaching its last element.

Queue Applications

In the computer environment, the below areas where the queue concept will be implemented:

- Print queue – Jobs sent to the printer
- Operating system maintain queue in allocating the process to each unit by storing them in buffer
- The interrupts that occurs during the system operation are also maintained in queue
- The allocation of memory for a particular resource (Printer, Scanner, any hand held devices) are also maintained in queue.

Program to implement queue using array

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

# define MAX_SIZE 10

class queues
{
        int front,rear;
        int queue[MAX_SIZE];
        public:
    queues()        // Constructor
                    {
                            front=0;
```

```
                        rear= -1;
                }
                void insert_rear(int);
                void delete_front();
                void display();
};

void queues::insert_rear(int item)
{
        If(rear>=MAX_SIZE-1)
        {
                cout<<"Queue is full!\nOverflow";

        }
        else
        {

                rear=rear+1;
                queue[rear]=item;
        }
}

void queues::delete_front()
{
        if(front> rear)
        {
                cout<<"Queue is empty!\nUnderflow";

        }
        else
        {
                cout<<"\nItem deleted."<<queue[front];
                front=front+1;
        }

}

void queues:: display()
{
        int ptr;


        for(ptr=rear;ptr>=front;ptr--)
                        cout<<queue[ptr];

}
```

```
void main()
{
	clrscr();
	int length,i,element,choice;
	queues q1;
	while(1)
	{
		clrscr();
		cout<<"1: Insert an item.\n2: Delete an item.";
		cout<<"\n3: Display elements\n4: Exit";
		cout<<"\nEnter your choice: ";
		cin>>choice;
		switch(choice)
		{
			case 1:
				cout<<"How many elements are in the queue: ";
				cin>>length;
				cout<<"Enter "<<length<<" elements: ";
				for(i=0;i<length;i++)
				{
					cin>>element;
					q1.insert_rear(element);
				}
				q1.display();
				getch();
				break;
			case 2:
				q1.delete_front();
				q1.display();
				getch();
				break;
			case 3:
				q1.display();
				getch();
				break;
			case 4:
				exit(0);
				break;
			default:
				cout<<"Please re-enter tour choice.";
				getch();
				break;
		 }
	}
	getch();
}
```

Circular queue

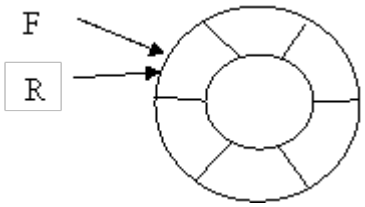
In a standard queue data structure re-buffering problem occurs for each dequeue operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue Circular queue is a linear data structure. It follows FIFO principle.

- In circular queue the last node is connected back to the first node to make a circle.
- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as "Ring buffer".
- Items can inserted and deleted from a queue in O(1) time.

Using array

In arrays the range of a subscript is 0 to n-1 where n is the maximum size. To make the array as a circular array by making the subscript 0 as the next address of the subscript n-1 by using the formula subscript = (subscript +1) % maximum size. In circular queue the front and rear pointer are updated by using the above formula.

The following figure shows circular array:



Algorithm for Enqueue operation using array

Step 1. start
Step 2. if (front == (rear+1)%max)
Print error "circular queue overflow "
Step 3. else
{ rear = (rear+1)%max
Q[rear] = element;
If (front == -1) f = 0;
}
Step 4. stop

Algorithm for Dequeue operation using array

Step 1. start
Step 2. if ((front == rear) && (rear == -1))
Print error "circular queue underflow "
Step 3. else
{ element = Q[front]
If (front == rear) front=rear = -1
Else
Front = (front + 1) % max
}
Step 4. stop

Program to implement Circular queue using array

```
#include <iostream.h>
class cqueue
{
        private :
                int *arr ;
                int front, rear ;
                int MAX;
        public :
                cqueue( int maxsize = 10 ) ;
                void addq ( int item ) ;
                int delq( ) ;
                void display( ) ;
} ;
cqueue :: cqueue( int maxsize )
{
        MAX = maxsize ;
        arr = new int [ MAX ];
        front = rear = -1 ;
        for ( int i = 0 ; i < MAX ; i++ )
                arr[i] = 0 ;
}
void cqueue :: addq ( int item )
{
        if ( ( rear + 1 ) % MAX == front )
        {
                cout << "\nQueue is full" ;
                return ;
        }
        rear = ( rear + 1 ) % MAX;
        arr[rear] = item ;
```

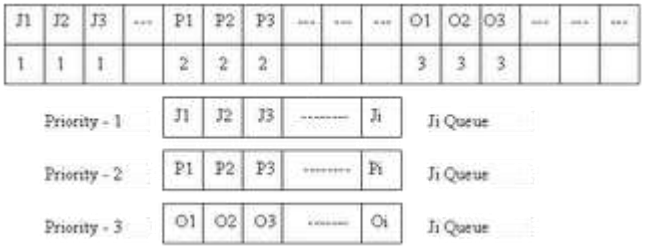
```
        if ( front == -1 )
                front = 0 ;
}
int cqueue :: delq( )
{
        int data ;
        if ( front == -1 )
        {
                cout << "\nQueue is empty" ;
                return NULL ;
        }

        data = arr[front] ;
        arr[front] = 0 ;
        if ( front == rear )
        {
                front = -1 ;
                rear = -1 ;
        }
        else
                front = ( front + 1 ) % MAX;
        return data ;
}
void cqueue :: display( )
{
        cout << endl ;
        for ( int i = 0 ; i < MAX ; i++ )
                cout << arr[i] << "  " ;
        cout << endl ;
}
void main( )
{
        cqueue a ( 10 ) ;
        a.addq ( 14 ) ;
        a.addq ( 22 ) ;
        a.addq ( 13 ) ;
        a.addq ( -6 ) ;
        a.addq ( 25 ) ;
        cout << "\nElements in the circular queue: " ;
        a.display( ) ;
        int i = a.delq( ) ;
        cout << "Item deleted: " << i ;
        i = a.delq( ) ;
        cout << "\nItem deleted: " << i ;
        cout << "\nElements in the circular queue after deletion: " ;
        a.display( ) ;
```

```
        a.addq ( 21 ) ;
        a.addq ( 17 ) ;
        a.addq ( 18 ) ;
        a.addq ( 9 ) ;
        a.addq ( 20 ) ;
        cout << "Elements in the circular queue after addition: " ;
        a.display( ) ;
        a.addq ( 32 ) ;
        cout << "Elements in the circular queue after addition: " ;
        a.display( ) ;
}
```

Priority queue

Priority queue is a linear data structure. It is having a list of items in which each item has associated *priority*. It works on a principle *add an element to the queue with an associated priority and remove the element from the queue that has the highest priority.* In general different items may have different priorities. In this queue highest or the lowest priority item are inserted in random order. It is possible to delete an element from a priority queue in order of their priorities starting with the highest priority.



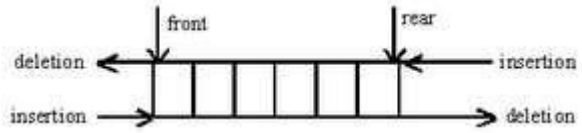
The above figure represents 3 separated Queues each following FIFO behavior. Elements in the 2nd Queue are removed only, when the first Queue is empty and elements from the 3rd Queue are removed only when the 2nd Queue is empty and soon.

Double Ended Queue

DeQueue (or) Deque (Double ended Queue) :-

DeQueue is a data structure in which elements may be added to or deleted from the front or the rear. Like an ordinary queue, a double-ended queue is a data structure it supports the following operations: enq_front, enq_back, deq_front, deq_back, and empty. Dequeue can be behave like a queue by using only enq_front and deq_front, and behaves like a stack by using only enq_front and deq_rear.

The DeQueue is represented as follows.



DeQueue can be represented in two ways they are

1) Input restricted DeQueue 2) output restricted DeQueue

The output restricted Dequeue allows deletions from only one end and input restricted Dequeue allows insertions at only one end.

The DeQueue can be constructed in two ways they are

1) Using array

2. Using linked list

Algorithm to add an element into DeQueue:

Assumptions: pointer f, r and initial values are -1,-1, Q[] is an array max represent the size of a queue.

enq_front

step1. Start
step2. Check the queue is full or not
step3. If false update the pointer f as f= f-1
step4. Insert the element at pointer f as Q[f] = element
step5. Stop

enq_back

step1. Start
step2. Check the queue is full or not as if (r == max-1) if yes queue is full
step3. If false update the pointer r as r= r+1
step4. Insert the element at pointer r as Q[r] = element
step5. Stop

Algorithm to delete an element from the DeQueue

deq_front

step1. Start
step2. Check the queue is empty or not as if (f == r) if yes queue is empty
step3. If false update pointer f as f = f+1 and delete element at position f as element = Q[f]
step4. If (f== r) reset pointer f and r as f=r=-1
step5. Stop

deq_back

step1. Start
step2. Check the queue is empty or not as if (f == r) if yes queue is empty
step3. If false delete element at position r as element = Q[r]
step4. Update pointer r as r = r-1
step5. If (f == r) reset pointer f and r as f = r= -1
step6. Stop

Chapter 4

Static and dynamic List

LIST

A list is a series of linearly arranged finite elements (numbers) of same type. The data elements are called nodes. The list can be of two types, i.e. basic data type or custom data type. The elements are positioned one after the other and their position numbers appear in sequence. The first element of the list is known as head and the last element is known as tail.

IMPLEMENTATION OF LIST

There are two methods of implementation of the list: they are static and dynamic.

Static Implementation

Static implementation can be implemented using arrays. It is a very simple method but it has few limitations. Once a size is declared, it cannot be changed during the program. It is also not efficient for memory. When array is declared, memory allocated is equal to the size of the array. The vacant space of array also occupies the memory space. In both the cases, if we store less arguments than declared, the memory is wasted and if more elements are stored than declared, array cannot be expanded. It is suitable only when exact number of elements is to be stored.

Insertion and Deletion of elements in a Static list

Suppose you are storing temperature data for a few months and you forgot to store the temperature of a particular day (say 5th day), then you need to INSERT that temperature after the 4th element of the array and in the other case if you accidentally stored duplicate data then you need to DELETE the duplicate element.

Apart from these simple examples, there are many other uses of insertion and deletion

The array to which the element is to be inserted or deleted can be of two types unordered (unsorted) and ordered (sorted). Here we will be discussing about the insertion and deletion of element in an unordered or unsorted array.

For insertion in these types of arrays, we need to have two information, the element to be inserted and the position to which it will be inserted. For deletion, we only need the position.

Suppose we have the following array:

arr[5]={5,7,2,1,3}

And we need to insert the element 6 at the 2nd position, after insertion:

arr[5]={5,6,7,2,1}

Notice how the last element of the array (i.e. 3) has gone out to compensate for the insertion.

Now, suppose we wish to delete the element at position 3rd, after deletion:

arr[5]={5,6,2,1,0}

We see, all the elements after the 3rd have shifted to their left and the vacant space is filled with 0.

That's exactly how insertion and deletion are done!

Algorithm for Insertion of an element in static list

Suppose the array to be arr [max], **pos** to be the position at which the element **num** has to be inserted. For insertion, all the elements starting from the position **pos** are shifted towards their right to make a vacant space where the element **num** is inserted.

1. FOR I = (max-1) TO pos
2. arr[I] = arr[I-1]
3. arr[I] = num

Algorithm for Deletion of an element in a static list

Suppose the array to be arr [max], **pos** to be the position from which the element has to be deleted. For deletion, all the elements to the right of the element at position **pos** are shifted to their left and the last vacant space is filled with 0.

1. FOR I = pos TO (max-1)
2. arr[I-1] = arr[I]
3. arr[I-1] = 0

```
// Example Program to illustrate insertion and deletion of elements in an array

#include<iostream.h>

// array has been declared as global so that other functions can also have access to it
int arr[5];

// function prototype
void a_insert(int, int);
void a_delete(int);

void main(void)
{
 int ch;
```

```
 int num,pos;

 while(ch!=4)
 {
  cout<<"1> Insert";
  cout<<"\n2> Delete";
  cout<<"\n3> Show";
  cout<<"\n4> Quit\n";

  cin>>ch;

  switch(ch)
  {
   case 1:
   cout<<"enter element:";
   cin>>num;
   cout<<"enter pos.:";
   cin>>pos;

   a_insert(num,pos);
   break;

   case 2:
   cout<<"enter pos.:";
   cin>>pos;
   a_delete(pos);
   break;

   case 3:
   cout<<"\nArray:";
   for(int i=0;i<5;i++)
     cout<<arr[i]<<" ";
   break;
  }
  cout<<"\n";
 }
}

// insertion function
void a_insert(int num, int pos)
{
 for(int i=4; i>=pos;i--)
   arr[i]=arr[i-1];
 arr[i]=num;
}
// deletion function
```

```
void a_delete(int pos)
{
 for(int i=pos; i<=4;i++)
 arr[i-1]=arr[i];
 arr[i-1]=0;
}
```

Linked list

A linked list is a dynamic data structure. It is an ideal technique to store data when the user is not aware of the number of elements to be stored. The dynamic implementation of list using pointers is also known as *linked list*. Each element of the list is called as *node*. Each element points to the next element. In the linked list a node can be inserted or deleted at any position. Each node of linked list has two components. The first component contains the information or any data field and second part the address of the next node. In other words, the second part holds address of the next element. This pointer points to the next data item. The pointer variable member of the last record of the list is generally assigned a NULL value to indicate the end of the list

Linked lists can be represented in memory by two ways they are

- Using array method
- Using pointer method

Array method:

In array method all the elements are stored in the continuous memory locations. It is having following disadvantages they are

- It follows static memory allocation.
- It is not possible to extend the size of the array at runtime
- Due to static memory allocation some memory space will be wasted.

Pointer method:

In pointer method all the data elements are represented using nodes. Each node is having data item and pointer to the next node. The elements in the list need not occupy continuous memory locations. The advantages of this method are

- Efficient memory management is possible, i.e., due to dynamic memory allocation the memory is not wasted
- It is possible to add or delete an element any ware in the list
- It is dynamic in nature
- It is possible to handle a list of any size
- It is possible to extend the size of a list at runtime

Various operations performed on lists:

The operations performed on lists are.

1. Inserting a new element at the given position.
2. Delete the element from the given position
3. Find the length of the list
4. Read the list from left to right
5. Retrieve the ith element
6. Copy a list
7. Sort the elements in either ascending or descending order
8. Combine 2 or more list

Single Linked list

Single Linked list is a linear data structure. It is used as a fundamental data structure in computer programming. It contains a collection of nodes which are connected by only one pointer in one direction. Each node is having two parts the first part contains the data and second part contains the address of the next node. The first part is called data field or information field and the second part is called as link field or next address field.

The graphical representation of linked list is

Hear the pointer **start** always points to the first node of a list and the end node is represented by special value called NULL.

Need for linked representation:-

The following problem arises when it is implemented by using arrays

- Wastage of storage
- It is not possible to add or delete in the middle of a list without rearrangement
- It is not possible to extend the size of a list
- To overcome all the above problem by implementing the list using linked representation. In linear lists insertion and deletion operations are inexpensive.

Operations applied on single linked list:

The basic types of operations applied on single linked list are

- Inserting a new element at the given position
- Delete the element from the given position
- Find the length of the list
- Read the list from left to right
- Retrieve the ith element
- Copy a list

- Sort the elements in either ascending or descending order
- Combine 2 or more list

Algorithm to add an element to an existing list:

It is possible to add an element to an existing list in three ways they are

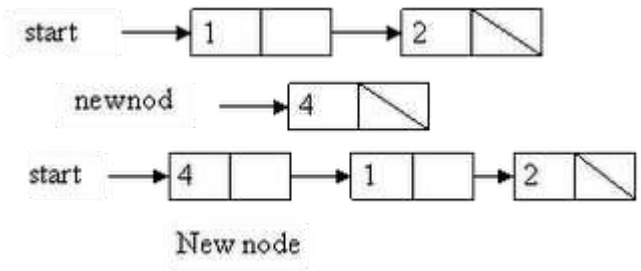
- Add at the front
- Add at end of list
- Add at position

Assumptions:

- *start* is a pointer which always points to the first node of a list
- ->next stands for next node address

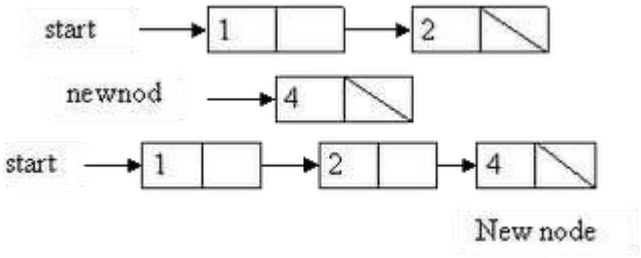
Add at the front of a list

Step1: create a new node
Step2: newnode->next = start
Step3: start = newnode



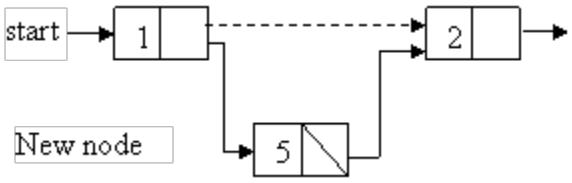
Add at the end of a list

Step1: create a new node (newnode)
Step2: t = start
Step3: while (t->next != NULL) t=t->next;
Step3: t->next = newnode



Add after a node of a list (position)

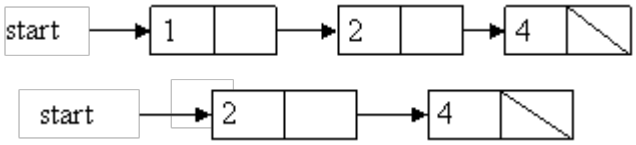
Step1: create a new node (newnode)
Step2: read the position of the node p;
Step3: t = start
Step4: traverse the list up to position as
for (i=1; inext;
Step4: newnode->next = t->next;
Step5: t->next = newnode



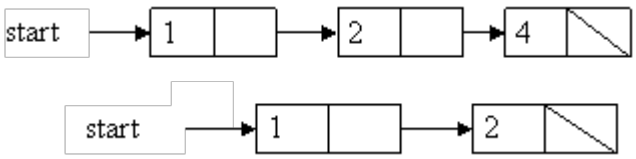
Algorithm for delete an element from the list
It is possible to delete an element from an existing list in three ways

- Delete at front
- Delete at end
- Delete at a position

Delete at front:
Step1: t1 = start
Step2: start = start->next
Step3: delete (t1)



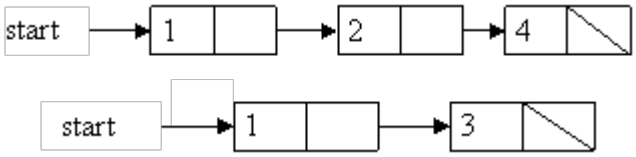
Delete at end:
Step1: t = start;
Step2: traverse the list up to n-1 nodes as
For (I =1; I < t =" t-">next
Step3: t1= t->next;
Step4: t->next = t->next->next;
Step5: delete (t1)



Delete at position:

Step1: read the position of the deleted node p
Step2: traverse the list upto p-1as
For (i=1;inext;
Step3: t1 = t->next;
Step4: t->next= t->next->next
Step5: delete (t1)

delete a node at position 2

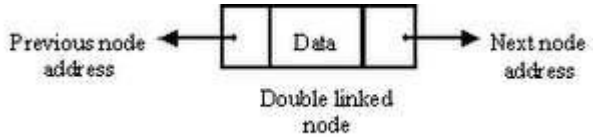


Drawbacks of a single linked list:

It is having only one pointer so that it is possible to traverse in only one way

Double Linked list

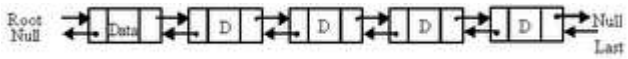
Double Linked list is a fundamental data structure used in computer programming. It consists of a sequence of nodes connected by two pointers in both directions. Each node is having three parts the first part contains the address of previous node and second part contains data and third part contains address of next node. The first and third part is called address fields and the second part is data field. The start pointer always points to the first node of a list. The graphical representation of a node structure shown in the fig



From the above fig the node is containing 2 pointers.

- Pointer1 is pointing to the previous node.

- Pointer2 is pointing to the next node.



Doubly linked list can be represented as above fig. in doubly linked list it is possible to traverse both forward & backward directions. It is also called two way lists.

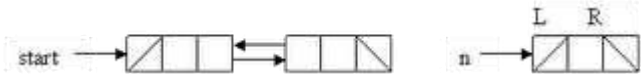
Operation applied on double linked list:

- Inserting an element
- Delete element from the list
- Find the length of the list
- Read the list from left to right or right to left
- Copy a list
- Sort the elements in either ascending or descending order
- Combine 2 or more list
- Search a list

Algorithm to add an element to an existing list:

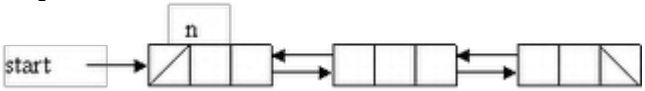
It is possible to add element to an existing list in three ways they are add at the *front*, add at the *end* of the list, add *after* a node of the existing list.
Assumption: start always contains the current list start node address.

>next stands for next node address



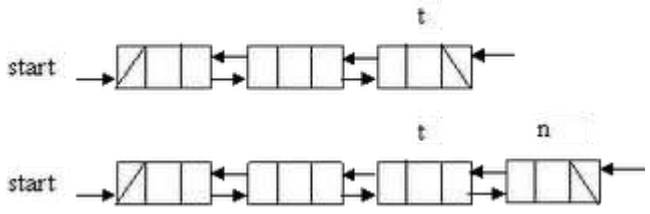
Add node at the front of a list

Step1. Create a new node
Step2. n->right = start
Step3. start->left = n
Step4. start = n



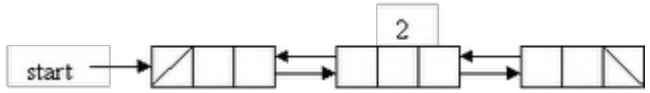
Add node at the end of a list

Step1. Create a new node
Step2. t = start where t is a temporary variable.
Step3. Traverse the tree until t->right = null
Step4. t->right = n
Step5. n->left = t

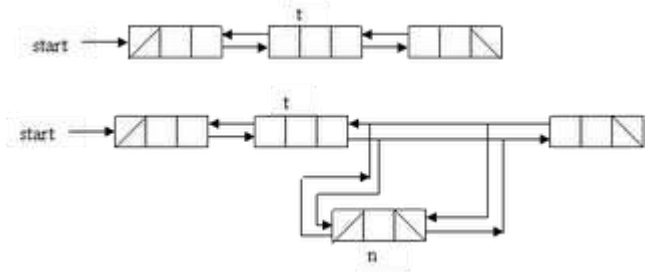


Add node after a particular position or a node

Step1. Create a new node.
Step2. t = start where t is a temporary pointer variable
Step3. Read position p from console.
Step4. Traverse the list until the position p
Step5. if not found the position error "no such node exist ": go to step 9
Step6. n->right = p->right
Step7. n->left = p
Step8. p->right = n
Step9. End



Insert node after second (2) node



Program to implement singly linked list

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

class list
{
  struct node
          {
          int data;
          struct node *next;

          }*p;
  public:

   list();
```

```
  void append(int);
  void addatbeg(int);
  void addatanyloc(int, int);
  void addatend(int);
  void delatbeg();
  void delatend();
  void delatanyloc(int);
 void display();
};

list::list()
{
p=NULL;
}
void list::append(int item)
{
          struct node *q, *t;
  if(p==NULL)
  {
   p=new node;
   p->data =item;
   p->next=NULL;
  }
  else
  {
   q=p;
   while(q->next!=NULL)
    {
          q=q->next;
    }
    t=new node;
    t->data=item;
    t->next=NULL;
    q->next=t;
  }
 }
 void list::addatbeg(int item)
 {
 struct node *q;
 q=new node;
 q->data=item;
 q->next=p;
 p=q;
}
void list:: addatanyloc(int item, int loc)
{
```

```
struct node *q,*t;
q=p;
for(int i=1; i<loc; i++)
{
q=q->next;
}
t=new node;
t->data=item;
t->next=q->next;
q->next=t;
}

void list::addatend(int item)
{
        struct node *q, *t;
 q=p;
 while(q->next!=NULL)
        q=q->next;
 t=new node;
 t->data=item;
 t->next=NULL;
 q->next=t;
}

 void list::delatbeg()
 {
  struct node *q;
  q=p;
  p=q->next;
  delete q;

 }
 void list::delatend()
 {
  struct node *q,*t;
  q=p;
  while(q->next!=NULL)
  {
   t=q;
   q=q->next;
  }
  t->next=NULL;
  delete q;

 }
void list::delatanyloc(int loc)
```

```
{
 struct node *q,*t;
 q=p;
 for(int i=1; i<loc; i++)
 {
  t=q;
  q=q->next;
 }
 t->next=q->next;
 delete q;

}
void list::display()
{

struct node *q;
q=p;
while(q!=NULL)
{

cout<<q->data<<"  ";
q=q->next;
}
}

void main()
{
 list l;
 cout<<"first node"<<endl;
 l.append(10);

 l.display();
 cout<<endl;
 cout<<"complete list"<<endl;
 l.append(20);
 l.append(30);
 l.append(40);
 l.append(50);
 l.display();
 l.addatbeg(60);
 cout<<endl<<"complete list"<<endl;
 l.display();
 l.addatanyloc(70,3);
 cout<<endl<<"complete list"<<endl;
 l.display();
 l.addatend(80);
```

```
 cout<<endl<<"complete list"<<endl;
 l.display();
 l.delatbeg();
 cout<<endl<<"complete list"<<endl;
 l.display();
 l.delatend();
 cout<<endl<<"complete list"<<endl;
 l.display();
 l.delatanyloc(3);
 cout<<endl<<"complete list"<<endl;
 l.display();
 getch();
}
```

Program to implement doubly linked list

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

class list
{
 struct node
        {
        int data;
        struct node *next, *prev;

        }*p;
 public:

  list();
        void append(int);
  void addatbeg(int);
  void addatanyloc(int, int);
  void addatend(int);
  void delatbeg();
  void delatend();
  void delatanyloc(int);
        void display();
};

list::list()
{
p=NULL;
}
void list::append(int item)
```

```
{
        struct node *q, *t;
 if(p==NULL)
 {
  p=new node;
  p->data =item;
  p->next=NULL;
  p->prev=NULL;
 }
 else
 {
  q=p;
  while(q->next!=NULL)
   {
        q=q->next;
   }
   t=new node;
   t->data=item;
   t->next=NULL;
   t->prev=q;
   q->next=t;
 }
}
void list::addatbeg(int item)
{
struct node *q;
q=new node;
q->data=item;
q->next=p;
p=q;
}
void list:: addatanyloc(int item, int loc)
{
struct node *q,*t;
q=p;
for(int i=1; i<loc; i++)
{
q=q->next;
}
t=new node;
t->data=item;
t->next=q->next;
t->prev=q;
q->next->prev=t;
q->next=t;
}
```

```
void list::addatend(int item)
{
        struct node *q, *t;
 q=p;
 while(q->next!=NULL)
        q=q->next;
 t=new node;
 t->data=item;
 t->next=NULL;
 t->prev=q;
 q->next=t;
}

 void list::delatbeg()
 {
  struct node *q;
  q=p;
  p=q->next;
  p->prev=NULL;
  delete q;

 }
 void list::delatend()
 {
  struct node *q,*t;
  q=p;
  while(q->next!=NULL)
  {
   t=q;
   q=q->next;
  }
  t->next=NULL;
  delete q;

 }
void list::delatanyloc(int loc)
 {
  struct node *q,*t;
  q=p;
  for(int i=1; i<loc; i++)
  {
   t=q;
   q=q->next;
  }
  t->next=q->next;
```

```
 q->next->prev=q->prev;
 delete q;

 }
 void list::display()
 {

 struct node *q;
 q=p;
 while(q!=NULL)
 {

 cout<<q->data<<"-->";
 q=q->next;
 }
 cout<<"NULL";
}

void main()
{
 list l;
 cout<<"first node"<<endl;
 l.append(10);

 l.display();
 cout<<endl;
 cout<<"complete list"<<endl;
 l.append(20);
 l.append(30);
 l.append(40);
 l.append(50);
 l.display();
 l.addatbeg(60);
 cout<<endl<<"complete list"<<endl;
 l.display();
 l.addatanyloc(70,3);
 cout<<endl<<"complete list"<<endl;
 l.display();
 l.addatend(80);
 cout<<endl<<"complete list"<<endl;
 l.display();
 l.delatbeg();
 cout<<endl<<"ater deleting first node complete list"<<endl;
 l.display();
 l.delatend();
 cout<<endl<<"after deleting last node complete list"<<endl;
```

```
 l.display();
 l.delatanyloc(3);
 cout<<endl<<"after deleting 3rd node complete list"<<endl;
 l.display();
 getch();
}
```

Program to implement stack using linked list

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

class stack
{
 struct node
        {
        int data;
  struct node *next;
  }*tos;
 public:

  stack();
        void push(int);
  void pop();
        void display();
};

stack::stack()
{
tos=NULL;
}
void stack::push(int item)
{
  struct node *p;
        p = new node;
  p->data = item;
  p->next = NULL;
  if(tos!=NULL)
  {
     p->next = tos;
  }
  tos = p;

}

void stack::pop()
```

```
{
   struct node *q;
   if(tos==NULL)
   {
      cout<<"\nThe stack is Empty"<<endl;
   }
   else
   {
      q= tos;
      tos = tos->next;
      cout<<"\nThe value popped is "<<q->data<<endl;
      delete q;
   }
}
void stack::display()
{
   struct node *p;
   p= tos;
   if(tos==NULL)
   {
      cout<<"\nNothing to Display\n";
   }
   else
   {
      cout<<"\nThe contents of Stack\n";
      while(p!=NULL)
      {
         cout<<p->data<<endl;
         p = p->next;
      }
   }
}
void main()
{
         stack s;

  s.push(10);

  s.display();
  cout<<endl;

 s.push(20);
 s.push(30);
 s.push(40);
 s.push(50);
```

```
 s.display();
 s.pop();

 s.display();
 s.pop();

 s.display();
 getch();
 }
```

Program to implement queue using linked list

```
#include<conio.h>
#include<iostream.h>
class queue
{
 struct node
        {
        int data;
  struct node *next;
  }*front,*rear;
 public:

  queue();
        void enqueue(int);
  void dequeue();
        void display();
};

queue::queue()
{
front=rear=NULL;
}
void queue::enqueue(int item)
{
   struct node *p;
         p = new node;
```

```
    p->data = item;
    p->next = NULL;
    if(front==NULL)
    {
          front=p;
    }
    if(rear!=NULL)
    {
       rear->next = p;
    }
    rear = p;

}
void queue::dequeue()
{
    struct node *q;
    if(front==NULL)
    {
       cout<<"\nThe queue is Empty"<<endl;
    }
    else
    {
q= front;
       front = front->next;
       cout<<"\nThe value popped is "<<q->data<<endl;
       delete q;
    }
}
void queue::display()
{
struct node *p;
    p= front;
    if(front==NULL)
```

```
    {
        cout<<"\nNothing to Display\n";
    }
    else
    {
        cout<<"\nThe contents of Queue\n";
        while(p!=NULL)
        {
            cout<<p->data<<"  ";
            p = p->next;
        }
    }
}
void main()
{
queue q;
q.enqueue(10);
q.display();
cout<<endl;
q.enqueue(20);
q.enqueue(30);
q.enqueue(40);
 q.enqueue(50);
q.display();
q.dequeue();
q.display();
q.dequeue();
 q.display();
q.dequeue();
q.display();
   getch();
  }
```

Chapter 5
Recursion

A recursive method is a method that calls itself either directly or indirectly

There are two key requirements to make sure that the recursion is successful:

- Every recursive call must simplify the computation in some way.
- There must be special cases to handle the simplest computations.

Iteration Vs. Recursion

- If a recursive method is called with a base case, the method returns a result. If a method is called with a more complex problem, the method divides the problem into two or more conceptual pieces: a piece that the method knows how to do and a slightly smaller version of the original problem. Because this new problem looks like the original problem, the method launches a recursive call to work on the smaller problem.
- For recursion to terminate, each time the recursion method calls itself with a slightly simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case. When the method recognizes the base case, the result is returned to the previous method call and a sequence of returns ensures all the way up the line until the original call of the method eventually returns the final result.
- Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure.
- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated method calls.
- Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- Iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base case.
- Recursion repeatedly invokes the mechanism, and consequently the overhead, of method calls. This can be expensive in both processor time and memory space.

The advantages and disadvantages of the two are not always obvious, and you should really take it on a case-by-case basis. When in doubt: test it. But generally:

1. Recursion may be slower, and use greater resources, because of the extra function calls.
2. Recursion may lead to simpler, shorter, easier-to-understand functions, especially for mathematicians who are comfortable with induction formulae.

Either way, one of the most critical aspects of writing either a recursive or an iterative function is to have an exit clause (or "base case" for recursions) that is checked at every recursion/iteration and is guaranteed to be reached at some finite point, given any input. Otherwise you will get either:
Infinite Recursion: More and more functions being spawned, never closing, using up resources until there are none left, possibly crashing the system.

Infinite loop: A loop that cycle forever, burning CPU and never completing.

Recursive Fibonacci

Fibonacci can be defined recursively as follows. Each number is the sum of the previous two numbers in the sequence. The n^{th} Fibonacci number is

Fib(*n*) = Fib(*n*-1) + Fib(*n*-2)
Fib(0) = 1,
Fib(1) = 1.

That is, the n^{th} number is formed by adding the two previous numbers together. The equation for Fib(*n*) is said to be the *recurrence relation* and the terms for Fib(0) and Fib(1) are the *base cases*.

The code for the recursive Fibonacci sequence is provided in Figure 3 below.

```
int Fib (int n) {
  if (n<=1)
    return 0;
  else
    return ( Fib(n-1) + Fib(n-2) );
}
```

Fibonacci in C

To find Fib(*n*) you need to call Fib two more times: once for Fib(*n*-1) and once more for Fib(*n*-2); then you add the results. But to find Fib(*n*-1), you need to call Fib two more times: once for Fib(*n*-2) and once more for Fib(*n*-3); then you need to add the results. And so on! You can see the recursion tree for Fib(*n*) in Figure 4. The recursion tree shows each call to Fib() with different values for *n*.

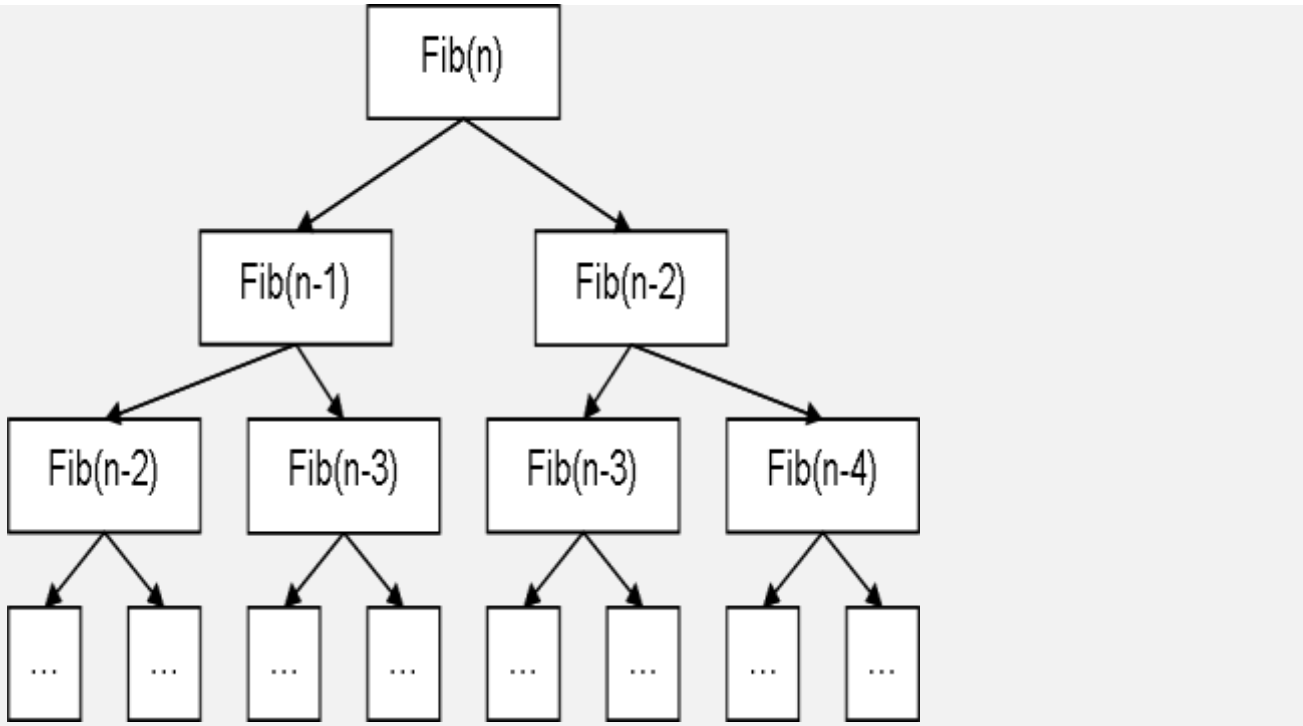
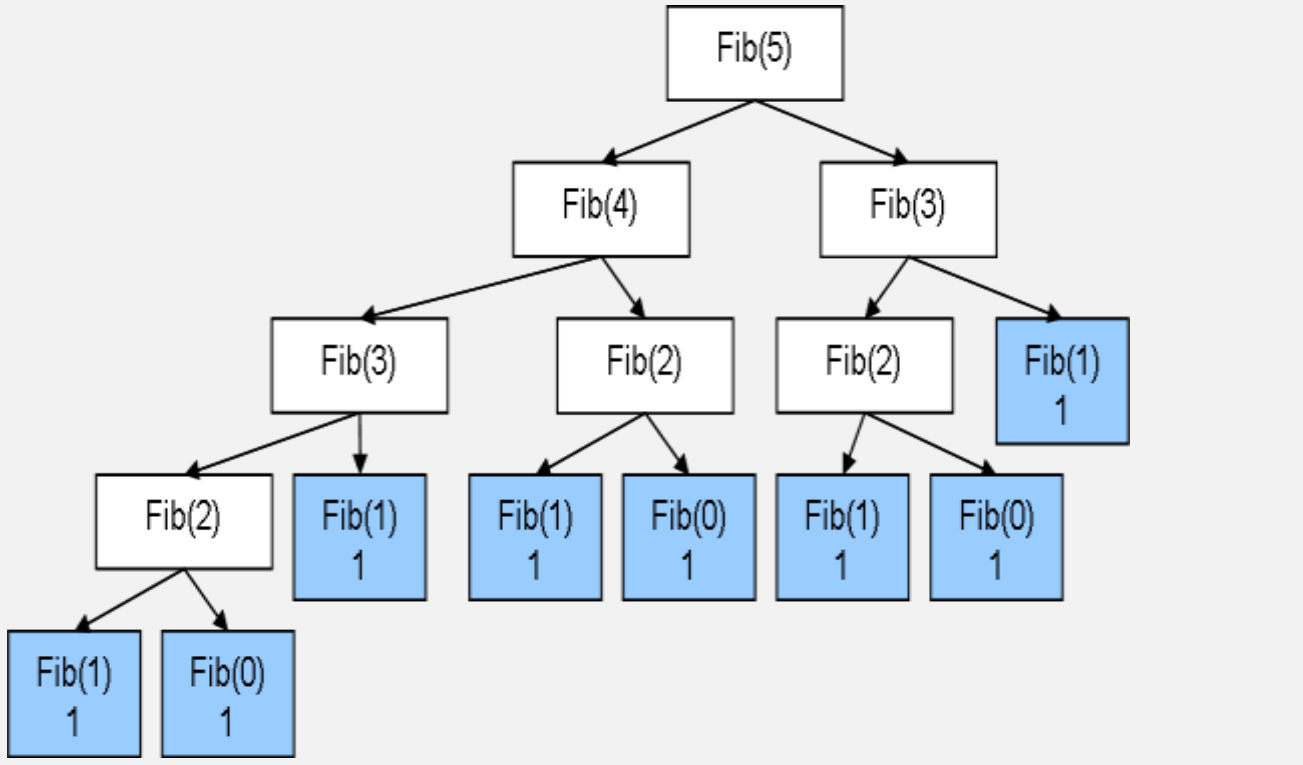


Figure 4: The recursion tree for Fib(*n*). Click to enlarge

For example, consider Fib(5), the fifth term in the sequence. To find Fib(5), you would need to find Fib(4) + Fib(3). But to find Fib(4), you would need to find Fib(3) + Fib(2). Luckily, Fib(2) is a base case; the answer to Fib(2) is 1. And so on. Look at Figure 5 for an example recursion tree for Fib(5) -- it lists all of the computations needed to carry out the calculation recursively.



The recursion tree for Fib(5). Base cases are highlighted in blue.

It may help to visualize the nested instances of the series of recursive calls for Fib(5) in a sort of table. Figure 6 shows the recursive calls for Fib(5), with each call generating two subtables. The call to Fib() returns when both subtables have been worked down to their respective base cases, and the return value is propagated up the tree (down the table).

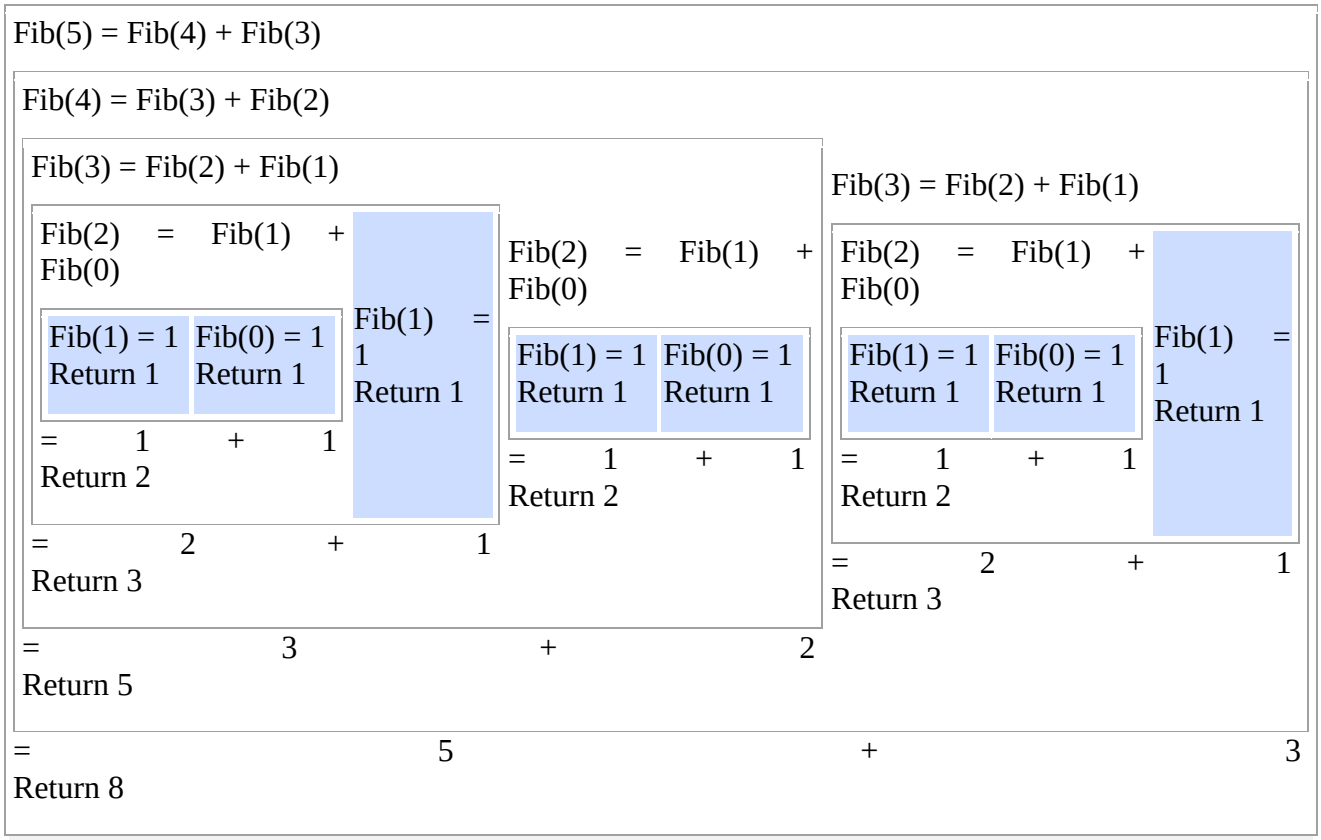


Figure Nested calls to Fib for Fib (5)

Iterative Fibonacci

The Fibonacci sequence can be defined iteratively by describing how to obtain the next value in the sequence. That is, if you start at the base cases of Fib (0) = Fib (1) = 1, you can build the sequence up to Fib (*n*) without using recursive calls.

For example, calculating Fib (5) may go as follows.

- Fib(0) = 1
- Fib(1) = 1
- Fib(2) = Fib(1) + Fib(0) = 1 + 1 = 2
- Fib(3) = Fib(2) + Fib(1) = 2 + 1 = 3
- Fib(4) = Fib(3) + Fib(2) = 3 + 2 = 5

- Fib(5) = Fib(4) + Fib(3) = 5 + 3 = 8

All that is needed for this calculation is a while loop!

RECURSION VS. ITERATION

We have studied both recursion and iteration. They can be applied to a program depending upon the situation. **Following table** explains the differences between recursion and iteration.

Recursion Vs. Iteration	
Recursion	***Iteration***
Recursion is the term given to the mechanism of defining a set or procedure in terms of itself.	The block of statement executed repeatedly using loops.
A conditional statement is required in the body of the function for stopping the function execution.	The iteration control statement itself contains statement for stopping the iteration. At every execution, the condition is checked.
At some places, use of recursion generates extra overhead. Hence, better to skip when easy solution is available with iteration.	All problems can be solved with iteration.
Recursion is expensive in terms of speed and memory.	Iteration does not create any overhead. All the programming languages support iteration.

ADVANTAGES & DISADVANTAGES OF RECURSION

Advantages of recursion,

1. Sometimes, in programming a problem can be solved without recursion, but at some situations in programming it is must to use recursion. For example, a program to display the list of all files of the system cannot be solved without recursion.

2. The recursion is very flexible in data structure like stacks, queues, linked list and quick sort.

3. Using recursion, the length of the program can be reduced.

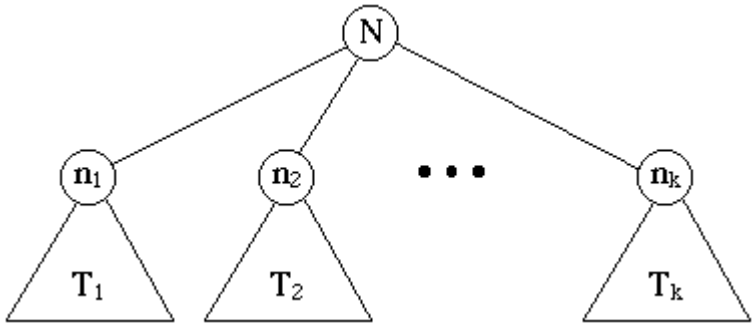
Chapter 6

Trees

Basics

Some basic terminology for trees:

- Trees are formed from *nodes* and *edges*. Nodes are sometimes called *vertices*. Edges are sometimes called *branches*.
- Nodes may have a number of properties including *value* and *label*.
- Edges are used to relate nodes to each other. In a tree, this relation is called "parenthood."
- An edge {a,b} between nodes a and b establishes a as the *parent* of b. Also, b is called a *child* of a.
- Although edges are usually drawn as simple lines, they are really directed from parent to child. In tree drawings, this is top-to-bottom.
- **Informal Definition**: a *tree* is a collection of nodes, one of which is distinguished as "root," along with a relation ("parenthood") that is shown by edges.
- **Formal Definition**: This definition is "recursive" in that it defines tree in terms of itself. The definition is also "constructive" in that it describes how to construct a tree.
    1. A single node is a tree. It is "root."
    2. Suppose N is a node and T_1, T_2, ..., T_k are trees with roots n_1, n_2, ...,n_k, respectively. We can construct a new tree T by making N the parent of the nodes n_1, n_2, ..., n_k. Then, N is the root of T and T_1, T_2, ..., T_k are subtrees.



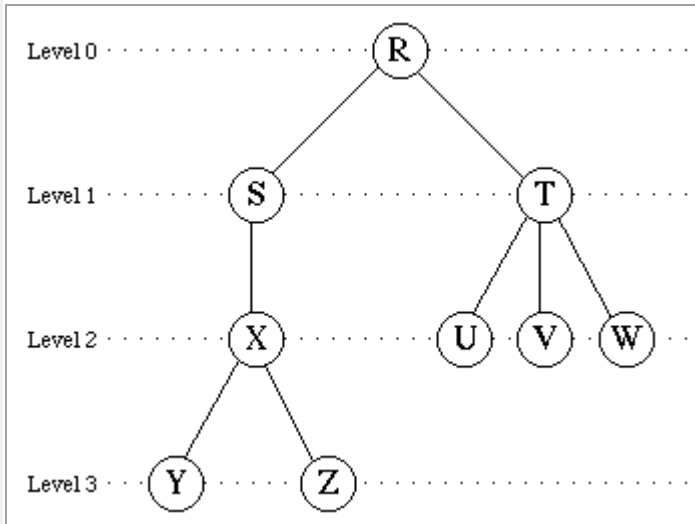
The tree T, constructed using k subtrees

More terminology

- A node is either *internal* or it is a *leaf*.
- A *leaf* is a node that has no children.
- Every node in a tree (except root) has exactly one parent.
- The *degree of a node* is the number of children it has.
- The *degree of a tree* is the maximum degree of all of its nodes.

Paths and Levels

- **Definition**: A *path* is a sequence of nodes n_1, n_2, ..., n_k such that node n_i is the parent of node n_{i+1} for all 1 <= i <= k.
- **Definition**: The *length* of a path is the number of edges on the path (one less than the number of nodes).
- **Definition**: The *descendents* of a node are all the nodes that are on some path from the node to any leaf.
- **Definition**: The *ancestors* of a node are all the nodes that are on the path from the node to the root.
- **Definition**: The *depth* of a node is the length of the path from root to the node. The depth of a node is sometimes called its *level*.
- **Definition**: The *height of a node* is the length of the longest path from the node to a leaf.
- **Definition**: the *height of a tree* is the height of its root.



A general tree, showing node depths (levels)

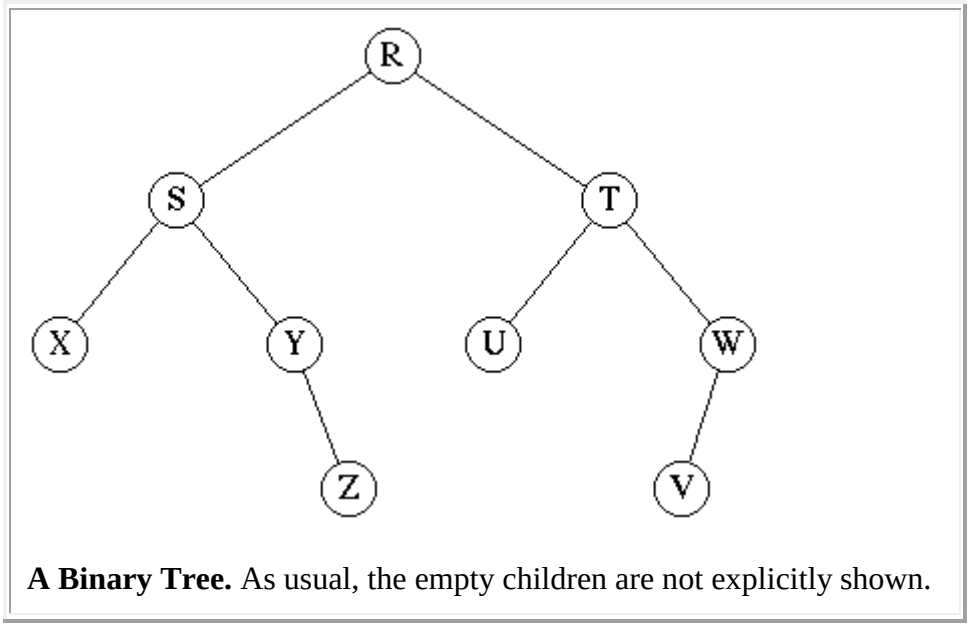
In the example above:

- The nodes Y, Z, U, V, and W are leaf nodes.
- The nodes R, S, T, and X are internal nodes.
- The degree of node T is 3. The degree of node S is 1.
- The depth of node X is 2. The depth of node Z is 3.
- The height of node Z is zero. The height of node S is 2. The height of node R is 3.
- The height of the tree is the same as the height of its root R. Therefore the height of the tree is 3.
- The sequence of nodes R,S,X is a path.
- The sequence of nodes R,X,Y is not a path because the sequence does not satisfy the "parenthood" property (R is not the parent of X).

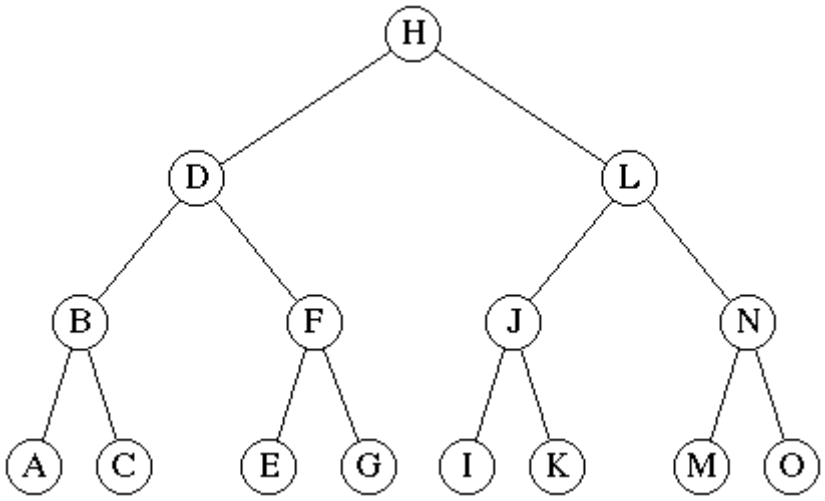
Binary Trees

- **Definition**: A binary tree is a tree in which each node has degree of exactly 2 and the children of each node are distinguished as "left" and "right." Some of the children of a node may be empty.
- **Formal Definition**: A binary tree is:
    1. either empty, or
    2. it is a node that has a left and a right subtree, each of which is a binary tree.
- **Definition**: A *full binary tree* (FBT) is a binary tree in which each node has exactly 2 non-empty children or exactly two empty children, and all the leaves are on the same level. (Note that this definition differs from the text definition).
- **Definition**: A *complete binary tree* (CBT) is a FBT except, perhaps, that the deepest level may not be completely filled. If not completely filled, it is filled from left-to-right.
- A FBT is a CBT, but not vice-versa.

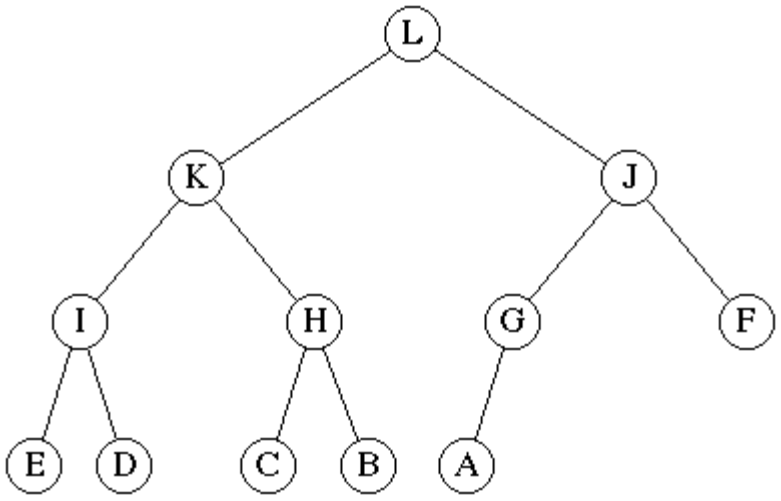
Examples of Binary Trees



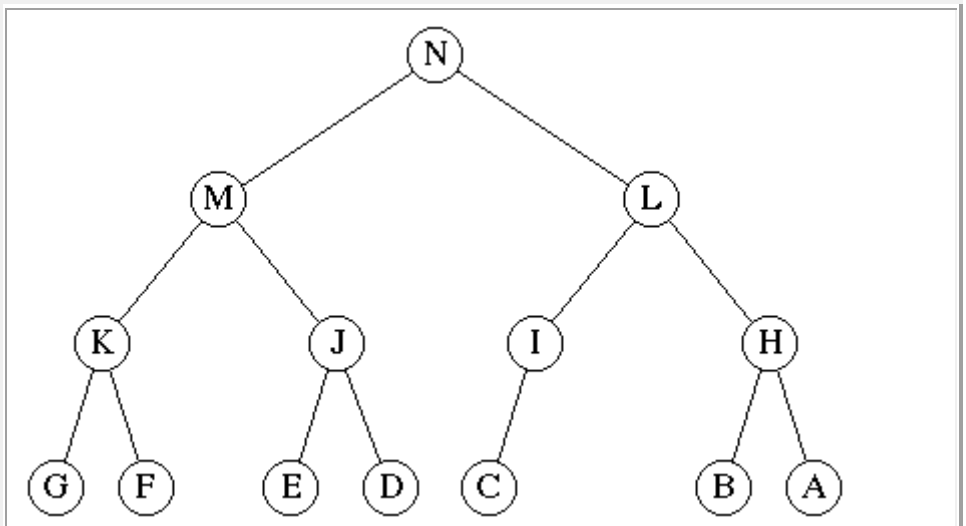
A Binary Tree. As usual, the empty children are not explicitly shown.



A Full Binary Tree. In a FBT, the number of nodes at level **i** is 2^i.



A Complete Binary Tree.



Not a Complete Binary Tree. The tree above is **not** a CBT because the deepest level is not filled from left-to-right.

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move subtrees around with minumum effort

Traversals

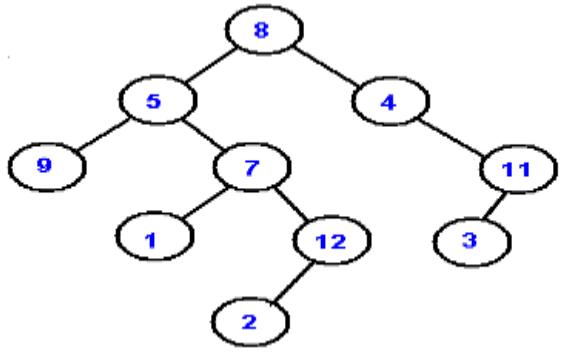
A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds

- depth-first traversal
- breadth-first traversal

There are three different types of depth-first traversals:

- PreOrder traversal - visit the parent first and then left and right children;
- InOrder traversal - visit the left child, then the parent and the right child;
- PostOrder traversal - visit left child, then the right child and then the parent;

There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right.



As an example consider the following tree and its four traversals:

PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

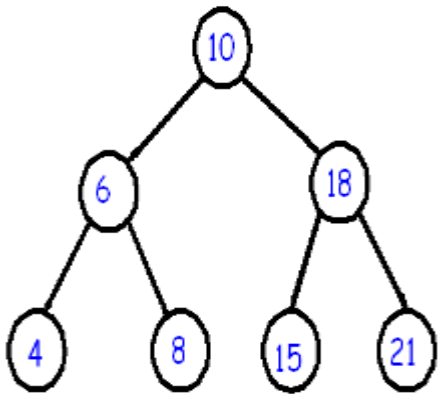
Binary Search Trees

We consider a particular kind of a binary tree called a Binary Search Tree (BST). The basic idea behind this data structure is to have such a storing repository that provides the efficient way of data sorting, searching and retriving.

A BST is a binary tree where nodes are ordered in the following way:

- each node contains one key (also known as data)
- the keys in the left subtree are less then the key in its parent node, in short L < P;
- the keys in the right subtree are greater the key in its parent node, in short P < R;
- duplicate keys are not allowed.

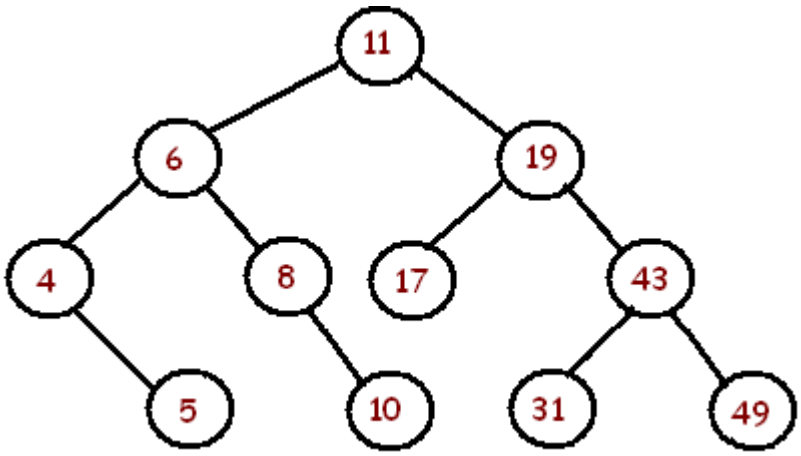
In the following tree all nodes in the left subtree of 10 have keys < 10 while all nodes in the right subtree > 10. Because both the left and right subtrees of a BST are again search trees; the above definition is recursively applied to all internal nodes:



Exercise. Given a sequence of numbers:

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31

Draw a binary search tree by inserting the above numbers from left to right.



Searching

Searching in a BST always starts at the root. We compare a data stored at the root with the key we are searching for (let us call it as toSearch). If the node does not contain the key we proceed either to the left or right child depending upon comparison. If the result of comparison is negative we go to the left child, otherwise - to the right child. The recursive structure of a BST yields a recursive algorithm.

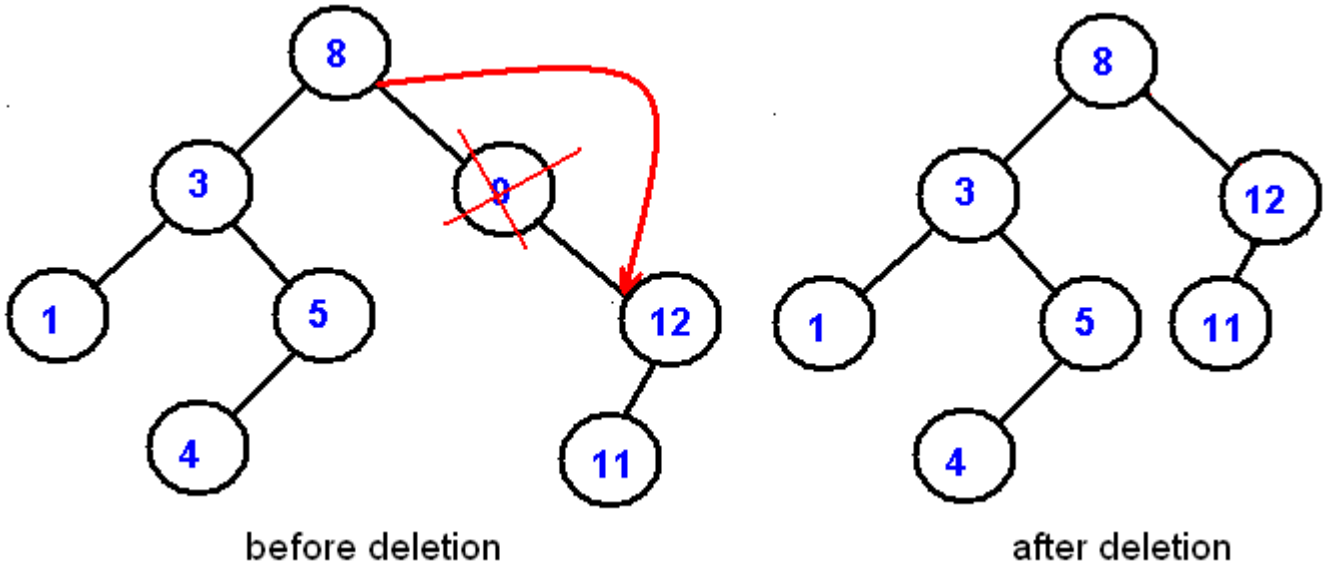
Searching in a BST has O(h) worst-case runtime complexity, where h is the height of the tree. Since s binary search tree with n nodes has a minimum of O(log n) levels, it takes at least O(log n) comparisons to find a particular node. Unfortunately, a binary serch tree can degenerate to a linked list, reducing the search time to O(n).

Deletion

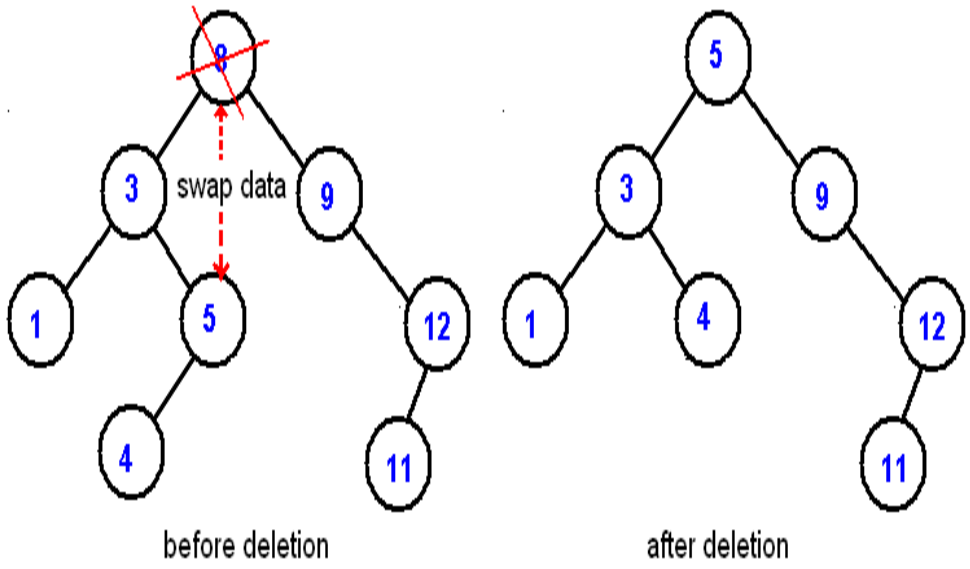
Deletion is somewhat more tricky than insertion. There are several cases to consider. A node to be deleted (let us call it as toDelete)

- is not in a tree;
- is a leaf;
- has only one child;
- has two children.

If toDelete is not in the tree, there is nothing to delete. If toDelete node has only one child the procedure of deletion is identical to deleting a node from a linked list - we just bypass that node being deleted



Deletion of an internal node with two children is less straightforward. If we delete such a node, we split a tree into two sub-trees and therefore, some children of the internal node won't be accessible after deletion. In the picture below we delete 8:

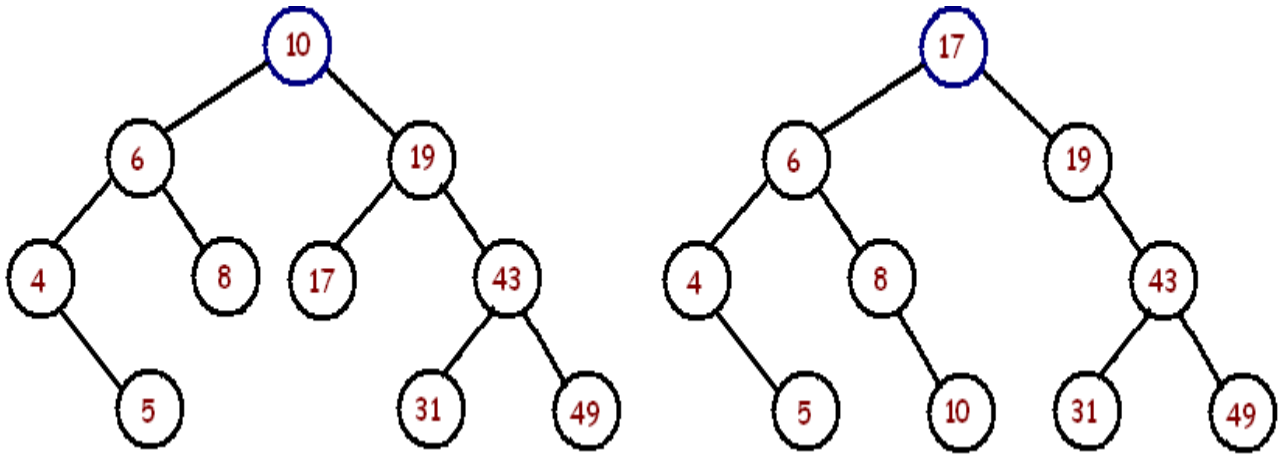


Deletion starategy is the following: replace the node being deleted with the largest node in the left subtree and then delete that largest node. By symmetry, the node being deleted can be swapped with the smallest node is the right subtree.

Given a sequence of numbers:

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31

Draw a binary search tree by inserting the above numbers from left to right and then show the two trees that can be the result after the removal of 11.



Binary search tree

Adding a value

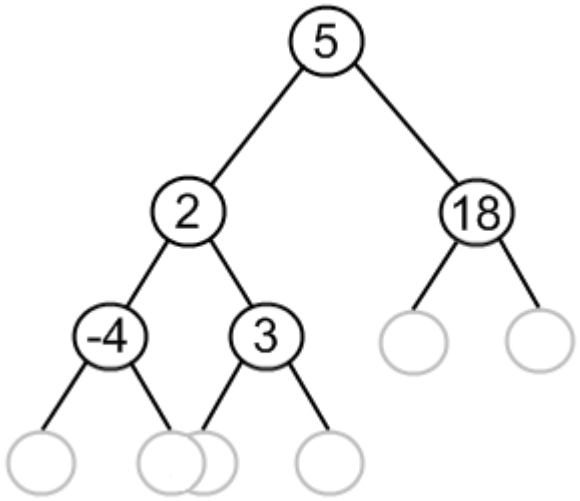
Adding a value to BST can be divided into two stages:

- search for a place to put a new element;
- insert the new element to this place.

Let us see these stages in more detail.

Search for a place

At this stage an algorithm should follow binary search tree property. If a new value is less, than the current node's value, go to the left sub-tree, else go to the right su-btree. Following this simple rule, the algorithm reaches a node, which has no left or right subtree. By the moment a place for insertion is found, we can say for sure, that a new value has no duplicate in the tree. Initially, a new node has no children, so it is a leaf. Let us see it at the picture. Gray circles indicate possible places for a new node.



Now, let's go down to algorithm itself. Here and in almost every operation on BST recursion is utilized. Starting from the root,

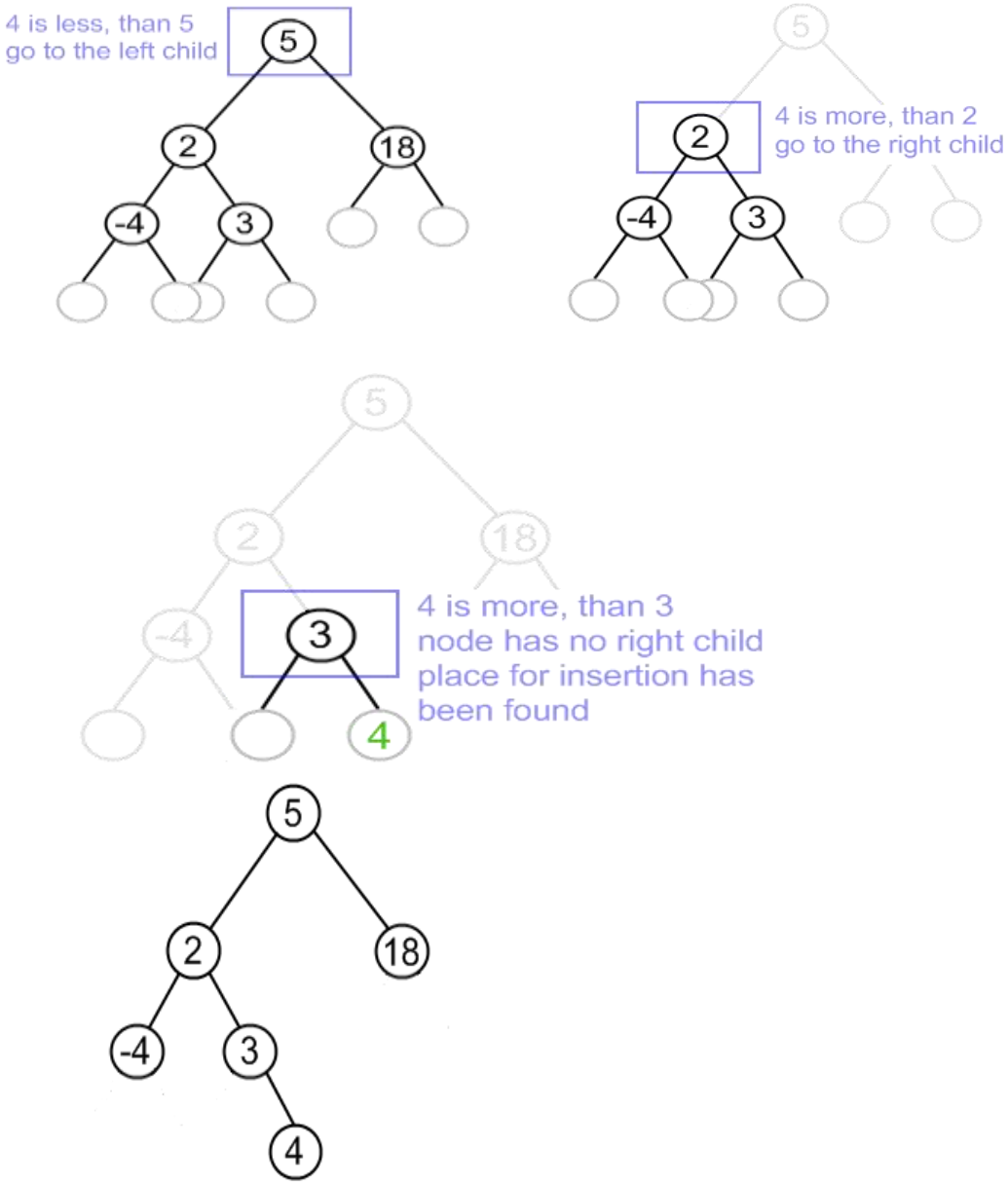
1. check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise,
2. if a new value is less, than the node's value:
    - if a current node has no left child, place for insertion has been found;
    - otherwise, handle the left child with the same algorithm.
3. if a new value is greater, than the node's value:
    - if a current node has no right child, place for insertion has been found;

- otherwise, handle the right child with the same algorithm.

Just before code snippets, let us have a look on the example, demonstrating a case of insertion in the binary search tree.

Example

Insert 4 to the tree, shown above.



Binary search tree. Removing a node

Remove operation on binary search tree is more complicated, than add and search. Basically, in can be divided into two stages:

- search for a node to remove;
- if the node is found, run remove algorithm.

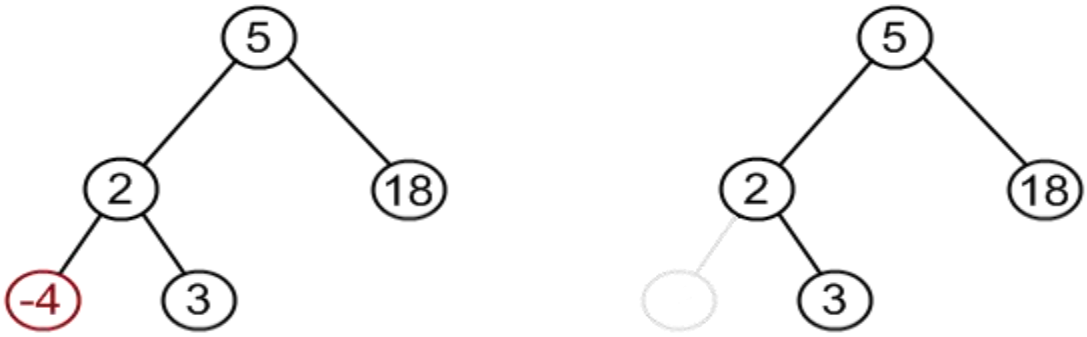
Remove algorithm in detail

Now, let's see more detailed description of a remove algorithm. First stage is identical to algorithm for lookup, except we should track the parent of the current node. Second part is more tricky. There are three cases, which are described below.

1. Node to be removed has no children.

   This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

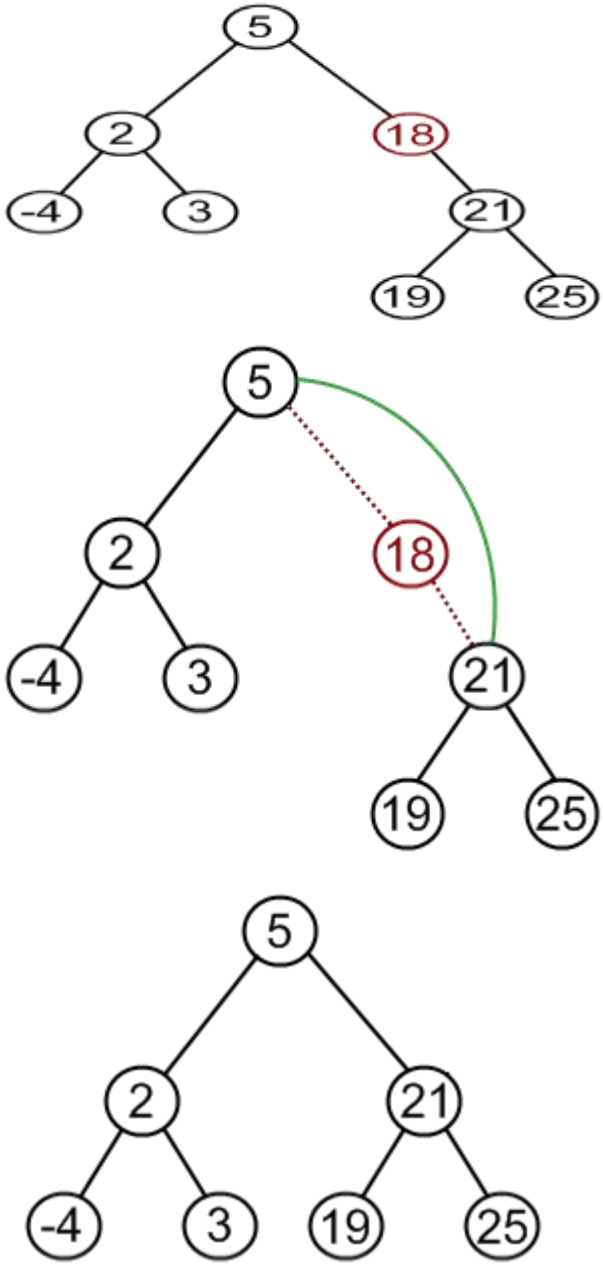
   Example. Remove -4 from a BST.



2. Node to be removed has one child.

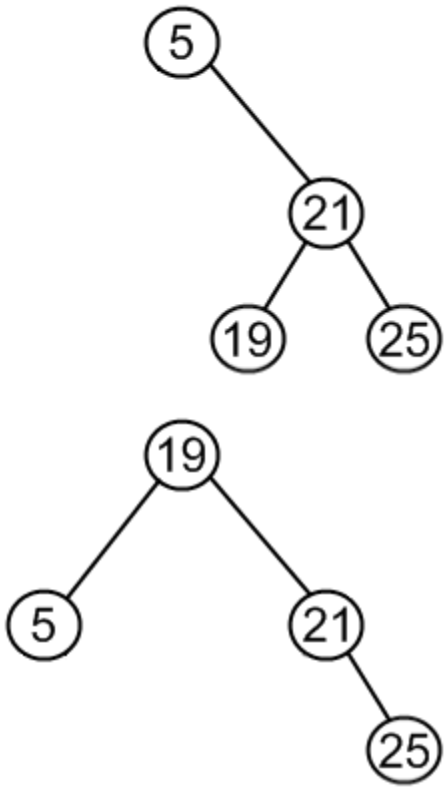
   It this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.

   Example. Remove 18 from a BST.



3. Node to be removed has two children.

   This is the most complex case. To solve it, let us see one useful BST property first. We are going to use the idea, that the same set of values may be represented as different binary-search trees. For example those BSTs:



contains the same values {5, 19, 21, 25}. To transform first tree into second one, we can do following:

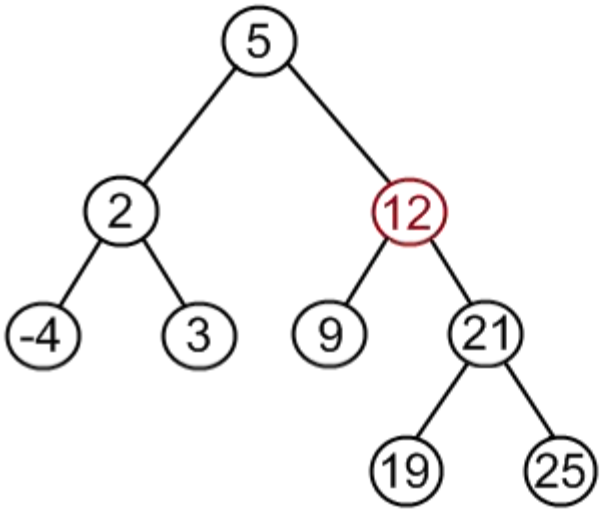
- choose minimum element from the right sub-tree (19 in the example);
- replace 5 by 19;
- hang 5 as a left child.

The same approach can be utilized to remove a node, which has two children:

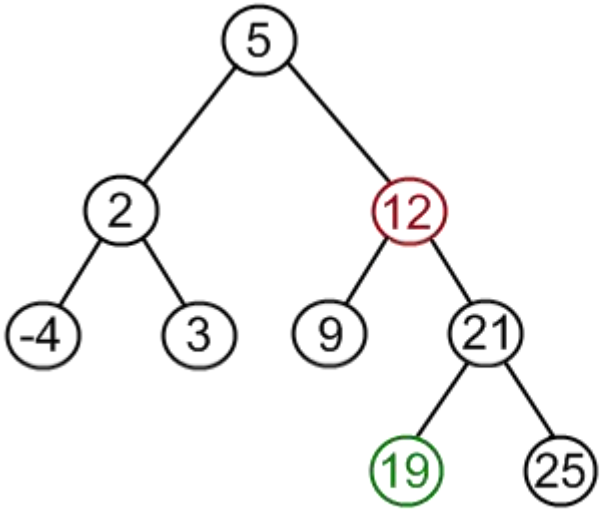
- find a minimum value in the right sub-tree;
- replace value of the node to be removed with found minimum. Now, right sub-tree contains a duplicate!
- apply remove to the right sub-tree to remove a duplicate.

Notice, that the node with minimum value has no left child and, therefore, it's removal may result in first or second cases only.

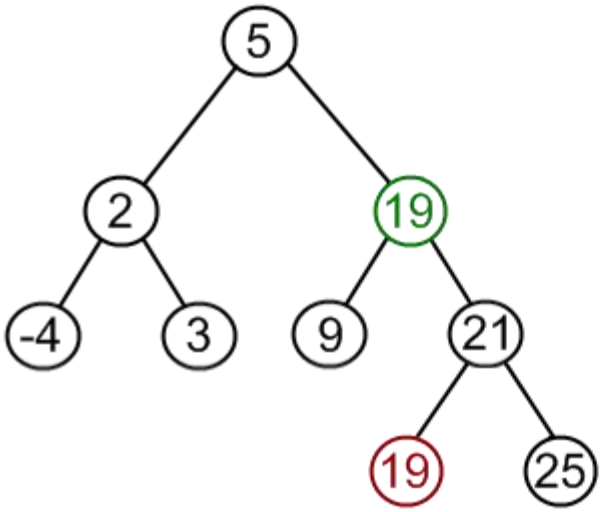
Example. Remove 12 from a BST.



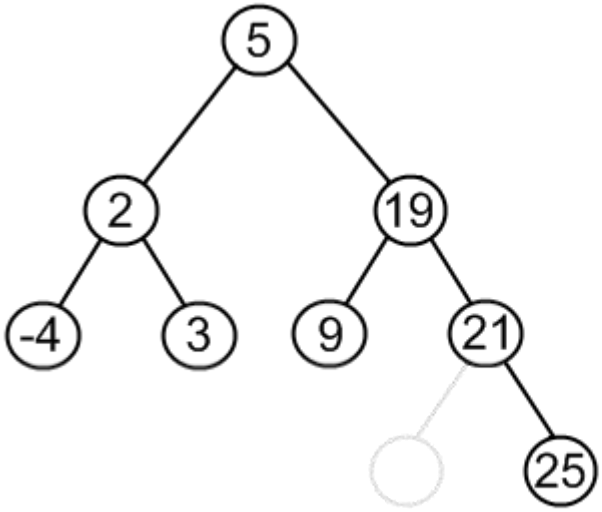
Find minimum element in the right sub-tree of the node to be removed. In current example it is 19.



Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.



Remove 19 from the left sub-tree.




```
//Binary Search Tree Program

#include <iostream>
#include <cstdlib>
using namespace std;

class BinarySearchTree
{
   private:
      struct tree_node
```

```
    {
      tree_node* left;
      tree_node* right;
      int data;
    };
    tree_node* root;

  public:
    BinarySearchTree()
    {
      root = NULL;
    }

    bool isEmpty() const { return root==NULL; }
    void print_inorder();
    void inorder(tree_node*);
    void print_preorder();
    void preorder(tree_node*);
    void print_postorder();
    void postorder(tree_node*);
    void insert(int);
    void remove(int);
};

// Smaller elements go left
// larger elements go right
void BinarySearchTree::insert(int d)
{
  tree_node* t = new tree_node;
  tree_node* parent;
  t->data = d;
  t->left = NULL;
  t->right = NULL;
  parent = NULL;

  // is this a new tree?
  if(isEmpty()) root = t;
  else
  {
    //Note: ALL insertions are as leaf nodes
    tree_node* curr;
    curr = root;
    // Find the Node's parent
    while(curr)
    {
      parent = curr;
```

```
            if(t->data > curr->data) curr = curr->right;
            else curr = curr->left;
        }

        if(t->data < parent->data)
          parent->left = t;
        else
          parent->right = t;
    }
}

void BinarySearchTree::remove(int d)
{
    //Locate the element
    bool found = false;
    if(isEmpty())
    {
        cout<<" This Tree is empty! "<<endl;
        return;
    }

    tree_node* curr;
    tree_node* parent;
    curr = root;

    while(curr != NULL)
    {
        if(curr->data == d)
        {
          found = true;
          break;
        }
        else
        {
            parent = curr;
            if(d>curr->data) curr = curr->right;
            else curr = curr->left;
        }
    }
    if(!found)
                    {
        cout<<" Data not found! "<<endl;
        return;
    }
```

```
                    // 3 cases :
   // 1. We're removing a leaf node
   // 2. We're removing a node with a single child
   // 3. we're removing a node with 2 children

   // Node with single child
   if((curr->left == NULL && curr->right != NULL)|| (curr->left != NULL
&& curr->right == NULL))
   {
     if(curr->left == NULL && curr->right != NULL)
     {
        if(parent->left == curr)
        {
          parent->left = curr->right;
          delete curr;
        }
        else
        {
          parent->right = curr->right;
          delete curr;
        }
     }
     else // left child present, no right child
     {
       if(parent->left == curr)
        {
          parent->left = curr->left;
          delete curr;
        }
        else
        {
          parent->right = curr->left;
          delete curr;
        }
     }
    return;
   }

                    //We're looking at a leaf node
                    if( curr->left == NULL && curr->right == NULL)
   {
      if(parent->left == curr) parent->left = NULL;
      else parent->right = NULL;
                                    delete curr;
                                    return;
   }
```

```
  //Node with 2 children
  // replace node with smallest value in right subtree
  if (curr->left != NULL && curr->right != NULL)
  {
     tree_node* chkr;
     chkr = curr->right;
     if((chkr->left == NULL) && (chkr->right == NULL))
     {
        curr = chkr;
        delete chkr;
        curr->right = NULL;
     }
     else // right child has children
     {
        //if the node's right child has a left child
        // Move all the way down left to locate smallest element

        if((curr->right)->left != NULL)
        {
           tree_node* lcurr;
           tree_node* lcurrp;
           lcurrp = curr->right;
           lcurr = (curr->right)->left;
           while(lcurr->left != NULL)
           {
              lcurrp = lcurr;
              lcurr = lcurr->left;
           }
                 curr->data = lcurr->data;
           delete lcurr;
           lcurrp->left = NULL;
        }
        else
        {
           tree_node* tmp;
           tmp = curr->right;
           curr->data = tmp->data;
              curr->right = tmp->right;
           delete tmp;
        }

     }
                  return;
  }
```

```
}

void BinarySearchTree::print_inorder()
{
  inorder(root);
}

void BinarySearchTree::inorder(tree_node* p)
{
    if(p != NULL)
    {
        if(p->left) inorder(p->left);
        cout<<" "<<p->data<<" ";
        if(p->right) inorder(p->right);
    }
    else return;
}

void BinarySearchTree::print_preorder()
{
    preorder(root);
}

void BinarySearchTree::preorder(tree_node* p)
{
    if(p != NULL)
    {
        cout<<" "<<p->data<<" ";
        if(p->left) preorder(p->left);
        if(p->right) preorder(p->right);
    }
    else return;
}

void BinarySearchTree::print_postorder()
{
    postorder(root);
}

void BinarySearchTree::postorder(tree_node* p)
{
    if(p != NULL)
    {
        if(p->left) postorder(p->left);
        if(p->right) postorder(p->right);
```

```
        cout<<" "<<p->data<<" ";
    }
    else return;
}

int main()
{
    BinarySearchTree b;
    int ch,tmp,tmp1;
    while(1)
    {
      cout<<endl<<endl;
      cout<<" Binary Search Tree Operations "<<endl;
      cout<<" ---------------------------- "<<endl;
      cout<<" 1. Insertion/Creation "<<endl;
      cout<<" 2. In-Order Traversal "<<endl;
      cout<<" 3. Pre-Order Traversal "<<endl;
      cout<<" 4. Post-Order Traversal "<<endl;
      cout<<" 5. Removal "<<endl;
      cout<<" 6. Exit "<<endl;
      cout<<" Enter your choice : ";
      cin>>ch;
      switch(ch)
      {
         case 1 : cout<<" Enter Number to be inserted : ";
                  cin>>tmp;
                  b.insert(tmp);
                  break;
         case 2 : cout<<endl;
                  cout<<" In-Order Traversal "<<endl;
                  cout<<" -------------------"<<endl;
                  b.print_inorder();
                  break;
         case 3 : cout<<endl;
                  cout<<" Pre-Order Traversal "<<endl;
                  cout<<" -------------------"<<endl;
                  b.print_preorder();
                  break;
         case 4 : cout<<endl;
                  cout<<" Post-Order Traversal "<<endl;
                  cout<<" --------------------"<<endl;
                  b.print_postorder();
                  break;
         case 5 : cout<<" Enter data to be deleted : ";
                  cin>>tmp1;
                  b.remove(tmp1);
```

```
                break;
        case 6 :
                return 0;

    }
  }
}
```

Rebuild a binary tree from Inorder and Preorder traversals

This is a well known problem where given any two traversals of a tree such as inorder & preorder or inorder & postorder or inorder & levelorder traversals we need to rebuild the tree.

The following procedure demonstrates on how to rebuild tree from given inorder and preorder traversals of a binary tree:

- Preorder traversal visits Node, left subtree, right subtree recursively
- Inorder traversal visits left subtree, node, right subtree recursively
- Since we know that the first node in Preorder is its root, we can easily locate the root node in the inorder traversal and hence we can obtain left subtree and right subtree from the inorder traversal recursively

Consider the following example:

Preorder Traversal: 1 2 4 8 9 10 11 5 3 6 7

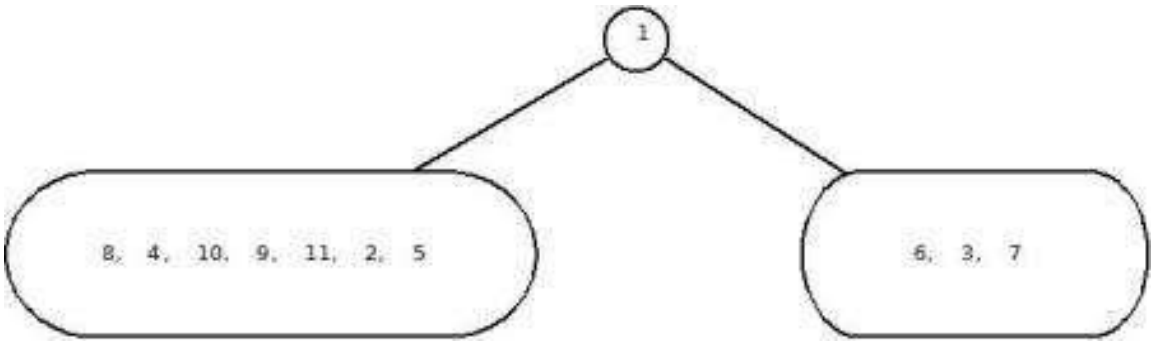
Inorder Traversal: 8 4 10 9 11 2 5 1 6 3 7

Iteration 1:

Root – {1}

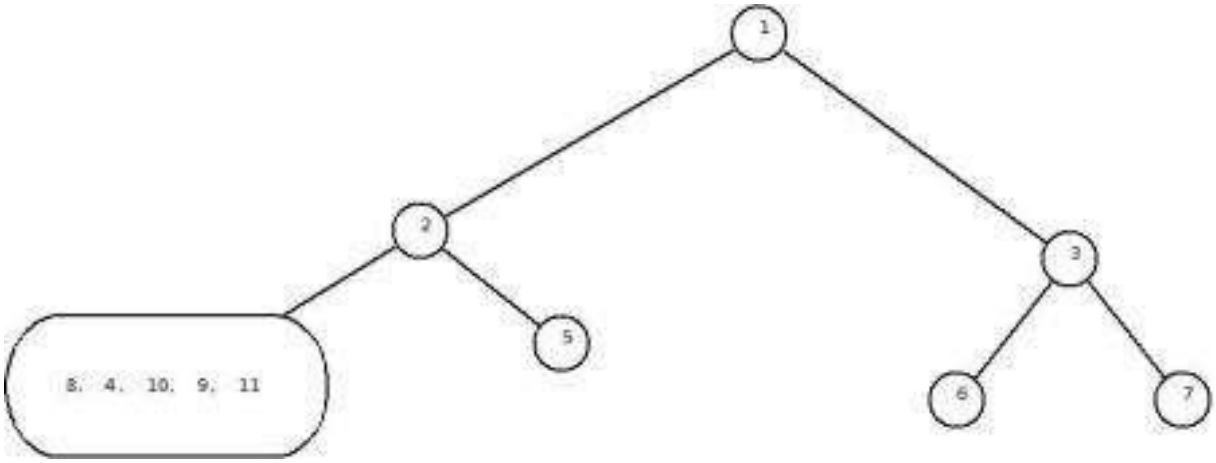
Left Subtree – {8,4,10,9,11,2,5}

Right Subtree – {6,3,7}



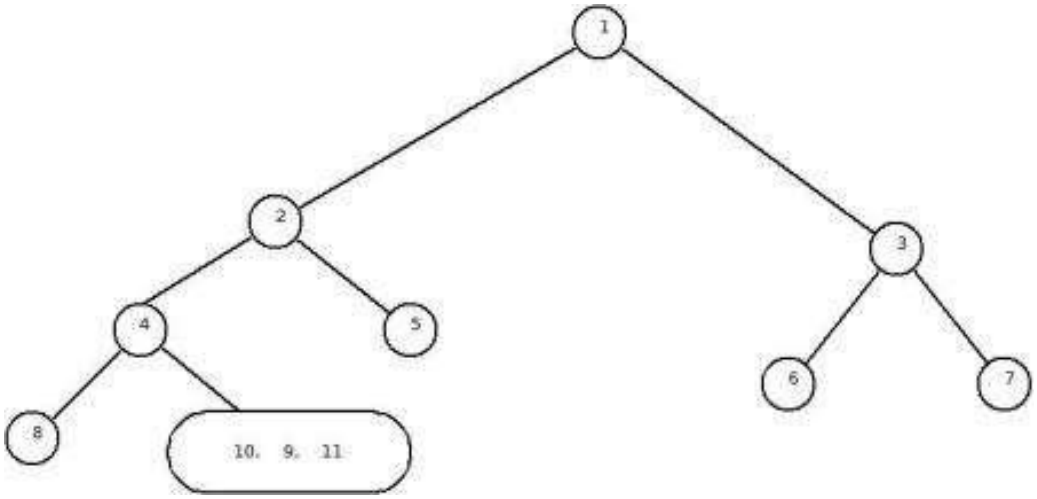
Iteration 2:

Root – {2} Left Subtree – {8,4,10,9,11} Right Subtree – {5}	Root – {3} Left Subtree – {6} Right Subtree – {7}



Iteration 3:

Root – {2} Left Subtree – {8,4,10,9,11} Right Subtree – {5}		Root – {3} Left Subtree – {6} Right Subtree – {7}
Root – {4} Left Subtree – {8} Right Subtree – {10,9,11}	Done	Done



Iteration 4:

<table>
<tr><td colspan="2">Root – {2}

Left Subtree – {8,4,10,9,11}

Right Subtree – {5}</td><td>Root – {3}

Left Subtree – {6}

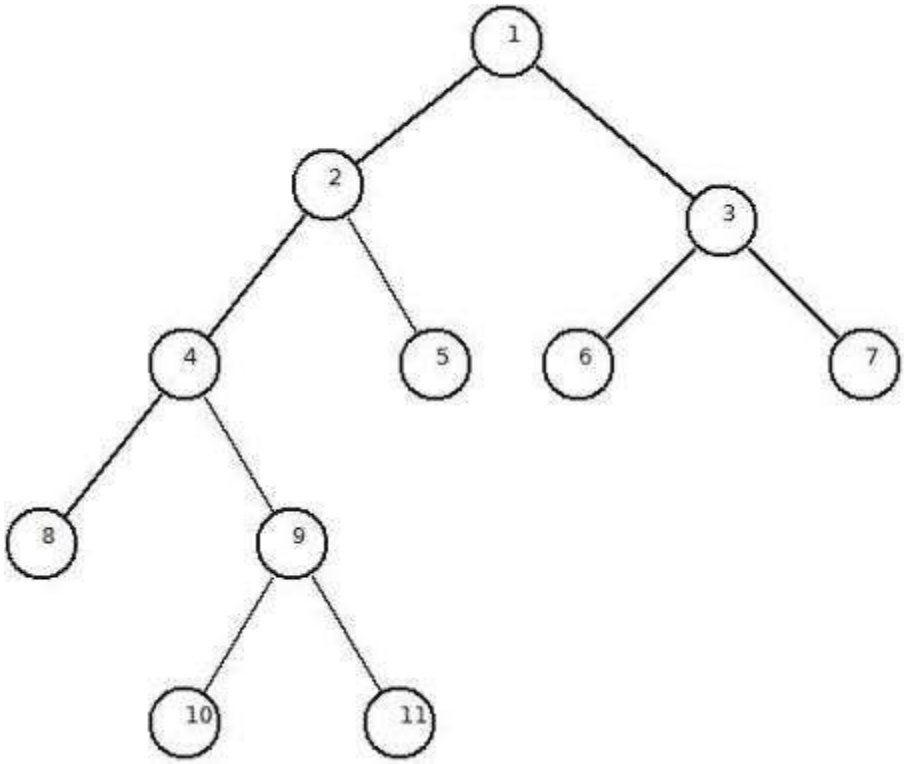
Right Subtree – {7}</td></tr>
<tr><td colspan="2">Root – {4}

Left Subtree – {8}

Right Subtree – {10,9,11}</td><td>Done</td><td>Done</td></tr>
<tr><td>Done</td><td>R – {9}

Left ST – {10}

Right ST- {11}</td><td>Done</td><td>Done</td></tr>
</table>



Given inorder and postorder traversals construct a binary tree

Let us consider the below traversals

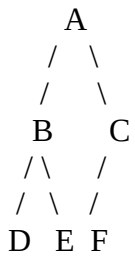
Inorder sequence: D B E A F C
Postorder sequence: D E B F C A
In a Postorder sequence, rightmost element is the root of the tree. So we know A is root.

Search for A in Inorder sequence. Once we know position of A (or index of A) in Inorder sequence, we also know that all elements on left side of A are in left subtree and elements on right are in right subtree.

```
          A
        /   \
      /       \
  D B E     F C
```

Let i be the index of root (i is 3 in the above case) in Indorder sequence.
In Inorder sequence, everything from 0 to i-1 is in left subtree and (i-1)th element in postorder traversal is root of the left subtree. If i-1 < 0 then left child of root is NULL
In Inorder sequence, everything from (i+1) to (n-1) is in right subtree and (n-1)th element is the root of right subtree. If n-1 is equal to i then right child of root is NULL.

Recursively follow above steps, and we get the tree shown below.

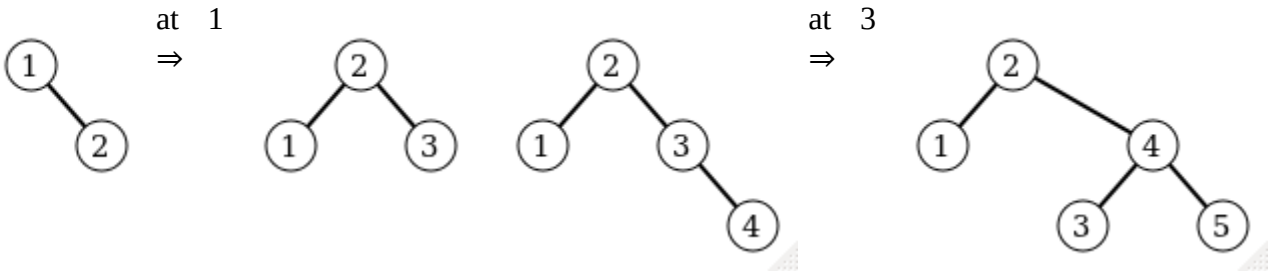


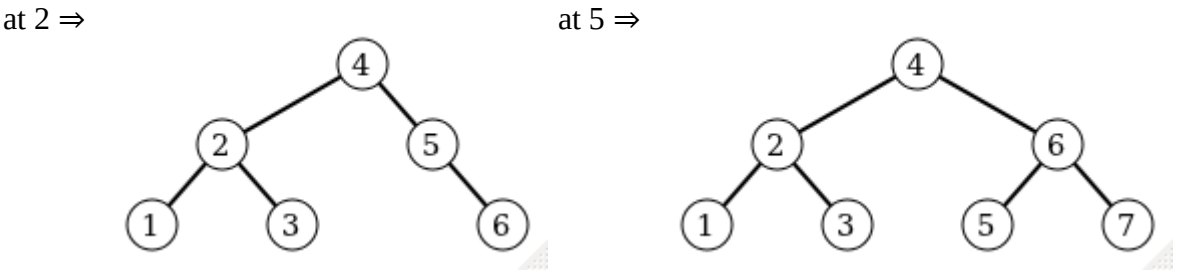
Examples

An important example of AVL trees is the behaviors on a worst-case add sequence for regular binary trees:

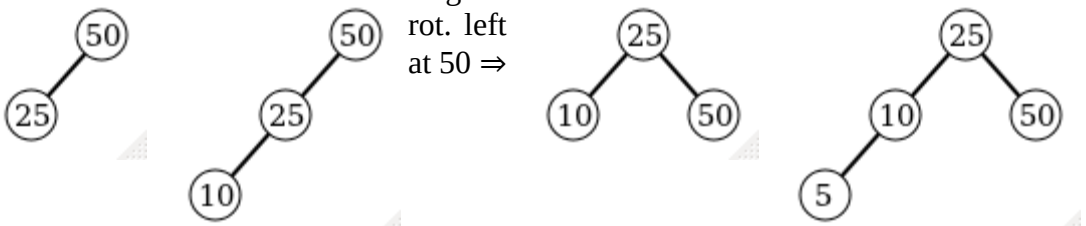
1, 2, 3, 4, 5, 6, 7

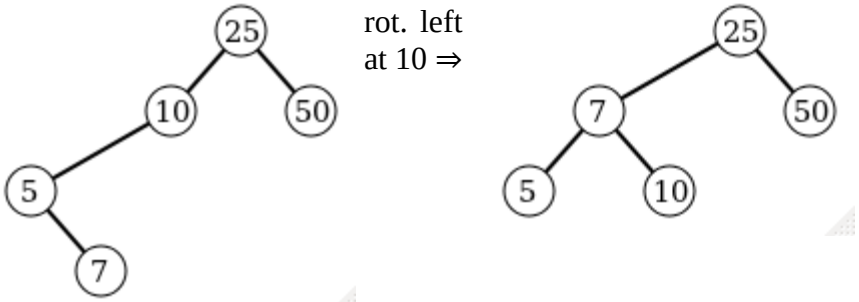
All insertions are **right-right** and so rotations are all **single rotate** from the **right**. All but two insertions require rebalancing:

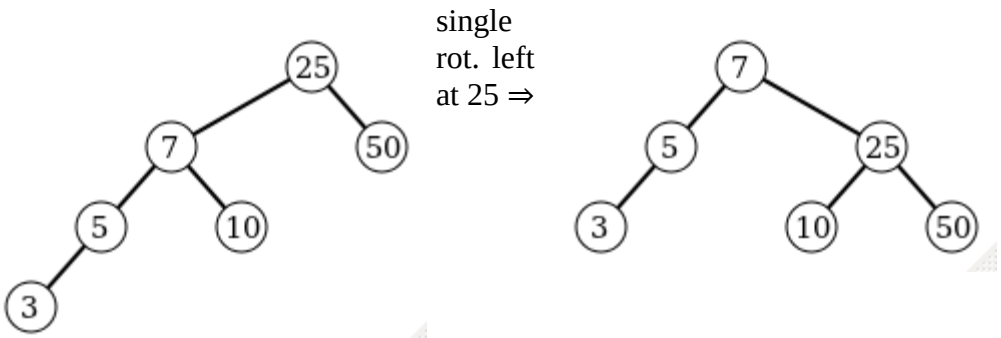




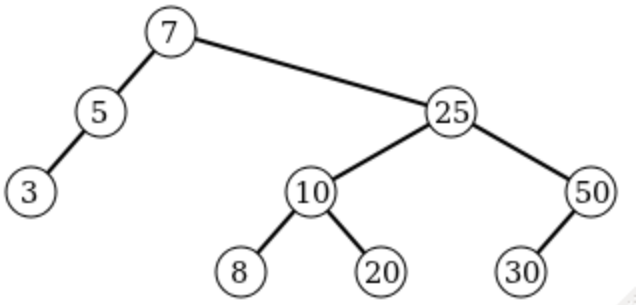
It can be shown that inserting the sequence 1,...,2^{n+1}-1 will make a perfect tree of height n.
Here is another example. The insertion sequence is: 50, 25, 10, 5, 7, 3, 30, 20, 8, 15

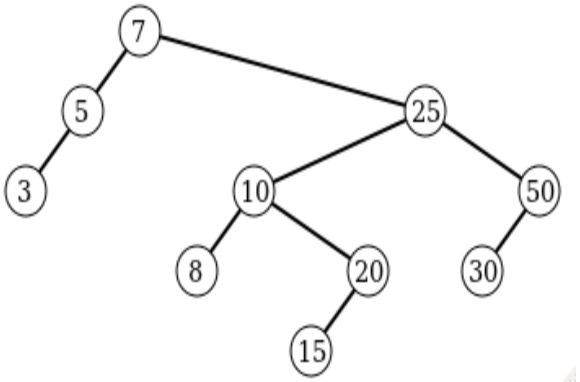




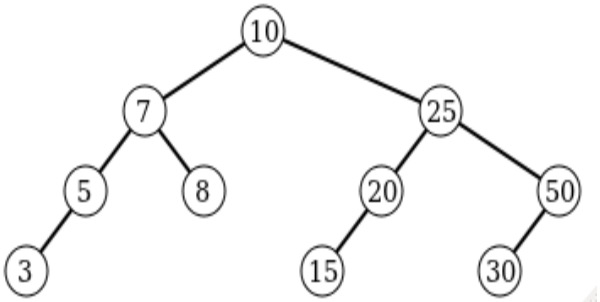


add(30), add(20), add(8) need no rebalancing





double rot. right at 7 ⇒



The AVL Tree Rotations

1. Rotations: How they work
A tree rotation can be an intimidating concept at first. You end up in a situation where you're juggling nodes, and these nodes have trees attached to them, and it can all become confusing very fast. I find it helps to block out what's going on with any of the sub-trees which are attached to the nodes you're fumbling with, but that can be hard.

Left Rotation (LL)

Imagine we have this situation:

Figure 1-1

```
a
 \
  b
   \
    c
```

To fix this, we must perform a left rotation, rooted at A. This is done in the following steps:

b becomes the new root.
a takes ownership of b's left child as its right child, or in this case, null.
b takes ownership of a as its left child.

The tree now looks like this:

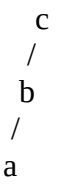
Figure 1-2

```
  b
 / \
a   c
```

Right Rotation (RR)

A right rotation is a mirror of the left rotation operation described above. Imagine we have this situation:

Figure 1-3



To fix this, we will perform a single right rotation, rooted at C. This is done in the following steps:

b becomes the new root.
c takes ownership of b's right child, as its left child. In this case, that value is null.
b takes ownership of c, as it's right child.

The resulting tree:

Figure 1-4

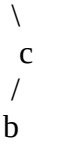
Left-Right Rotation (LR) or "Double left"

Sometimes a single left rotation is not sufficient to balance an unbalanced tree. Take this situation:

Figure 1-5

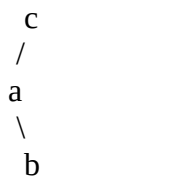
Perfect. It's balanced. Let's insert 'b'.

Figure 1-6



Our initial reaction here is to do a single left rotation. Let's try that.

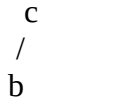
Figure 1-7



Our left rotation has completed, and we're stuck in the same situation. If we were to do a single right rotation in this situation, we would be right back where we started. What's causing this? The answer is that this is a result of the right subtree having a negative balance. In other words, because the right subtree was left heavy, our rotation was not sufficient. What can we do? The answer is to perform a right rotation on the right subtree. Read that again. We will perform a right rotation on the *right subtree*. We are not rotating on our current root. We are rotating on our right child. Think of our right subtree, isolated from our main tree, and perform a right rotation on it:

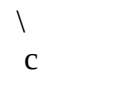
Before:

Figure 1-8



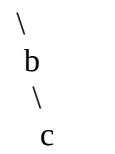
After:

Figure 1-9



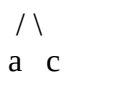
After performing a rotation on our right subtree, we have prepared our root to be rotated left. Here is our tree now:

Figure 1-10



Looks like we're ready for a left rotation. Let's do that:

Figure 1-11



Voila. Problem solved.

Right-Left Rotiation (RL) or "Double right"

A double right rotation, or right-left rotation, or simply RL, is a rotation that must be performed when attempting to balance a tree which has a left subtree, that is right heavy. This is a mirror operation of what was illustrated in the section on Left-Right Rotations, or double left rotations. Let's look at an example of a situation where we need to perform a Right-Left rotation.

Figure 1-12

```
  c
 /
a
 \
  b
```

In this situation, we have a tree that is unbalanced. The left subtree has a height of 2, and the right subtree has a height of 0. This makes the balance factor of our root node, c, equal to -2. What do we do? Some kind of right rotation is clearly necessary, but a single right rotation will not solve our problem. Let's try it:

Figure 1-13

```
a
 \
  c
 /
b
```

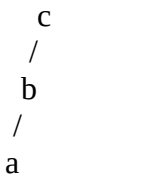
Looks like that didn't work. Now we have a tree that has a balance of 2. It would appear that we did not accomplish much. That is true. What do we do? Well, let's go back to the original tree, before we did our pointless right rotation:

Figure 1-14

```
  c
 /
a
 \
  b
```

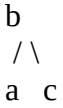
The reason our right rotation did not work, is because the left subtree, or 'a', has a positive balance factor, and is thus right heavy. Performing a right rotation on a tree that has a left subtree that is right heavy will result in the problem we just witnessed. What do we do? The answer is to make our left subtree left-heavy. We do this by performing a left rotation our left subtree. Doing so leaves us with this situation:

Figure 1-15



This is a tree which can now be balanced using a single right rotation. We can now perform our right rotation rooted at C. The result:

Figure 1-16



Balance at last.

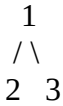
2. Rotations, When to Use Them and Why

How to decide when you need a tree rotation is usually easy, but determining which type of rotation you need requires a little thought.

A tree rotation is necessary when you have inserted or deleted a node which leaves the tree in an unbalanced state. An unbalanced state is defined as a state in which any subtree has a balance factor of greater than 1, or less than -1. That is, any tree with a difference between the heights of its two sub-trees greater than 1, is considered unbalanced.

This is a balanced tree:

Figure 2-1



This is an unbalanced tree:

Figure 2-2

```
  \
   3
```

This tree is considered unbalanced because the root node has a balance factor of 2. That is, the right subtree of 1 has a height of 2, and the height of 1's left subtree is 0. Remember that balance factor of a tree with a left subtree A and a right subtree B is

B - A

Simple.

In figure 2-2, we see that the tree has a balance of 2. This means that the tree is considered "right heavy". We can correct this by performing what is called a "left rotation". How we determine which rotation to use follows a few basic rules. See psuedo code:

```
IF tree is right heavy
{
  IF tree's right subtree is left heavy
  {
    Perform Double Left rotation
  }
  ELSE
  {
    Perform Single Left rotation
  }
}
ELSE IF tree is left heavy
{
  IF tree's left subtree is right heavy
  {
    Perform Double Right rotation
  }
  ELSE
  {
    Perform Single Right rotation
  }
}
```

As you can see, there is a situation where we need to perform a "double rotation". A single rotation in the situations described in the pseudo code leave the tree in an unbalanced state. Follow these rules, and you should be able to balance an AVL tree following an insert or delete every time.

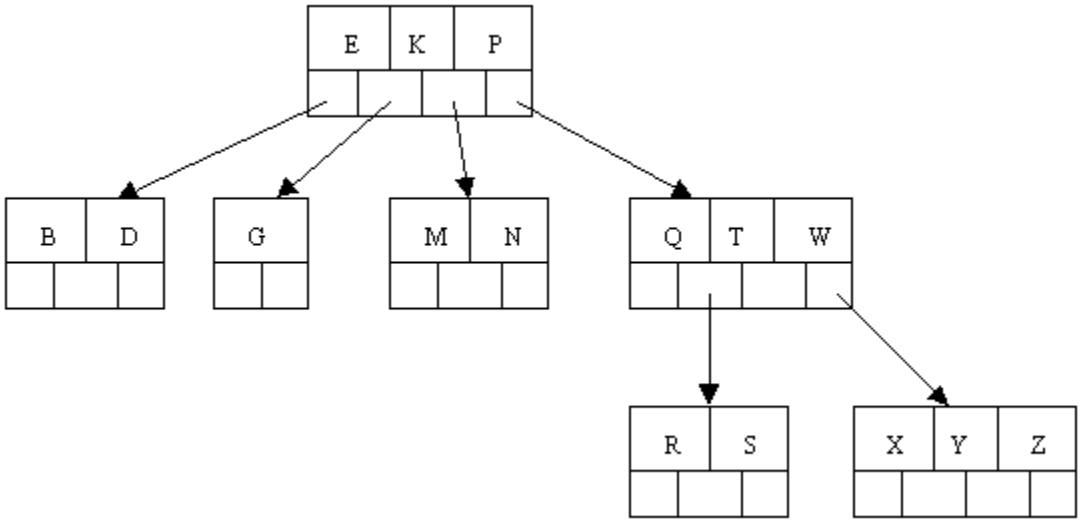
B-Trees

Introduction

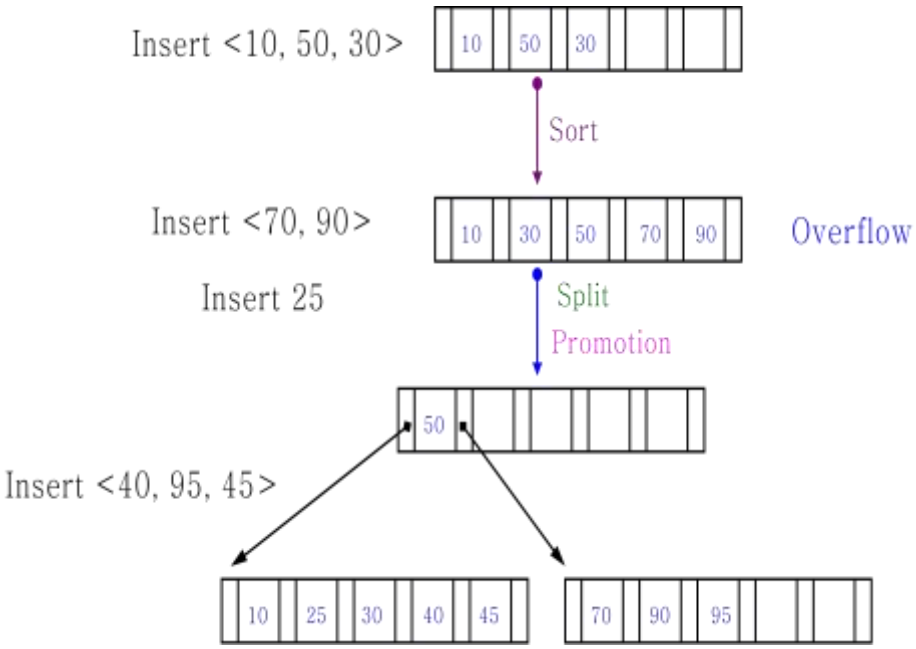
A B-tree is a specialized multiway tree designed especially for use on disk. In a B-tree each node may contain a large number of keys. The number of subtrees of each node, then, may also be large. A B-tree is designed to branch out in this large number of directions and to contain a lot of keys in each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

Definitions

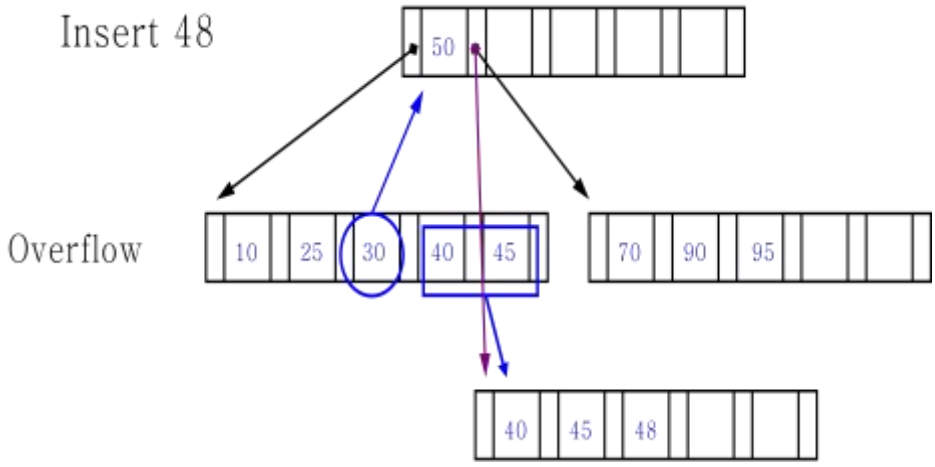
A *multiway tree of order m* is an ordered tree where each node has at most m children. For each node, if k is the actual number of children in the node, then k - 1 is the number of keys in the node. If the keys and subtrees are arranged in the fashion of a search tree, then this is called a *multiway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



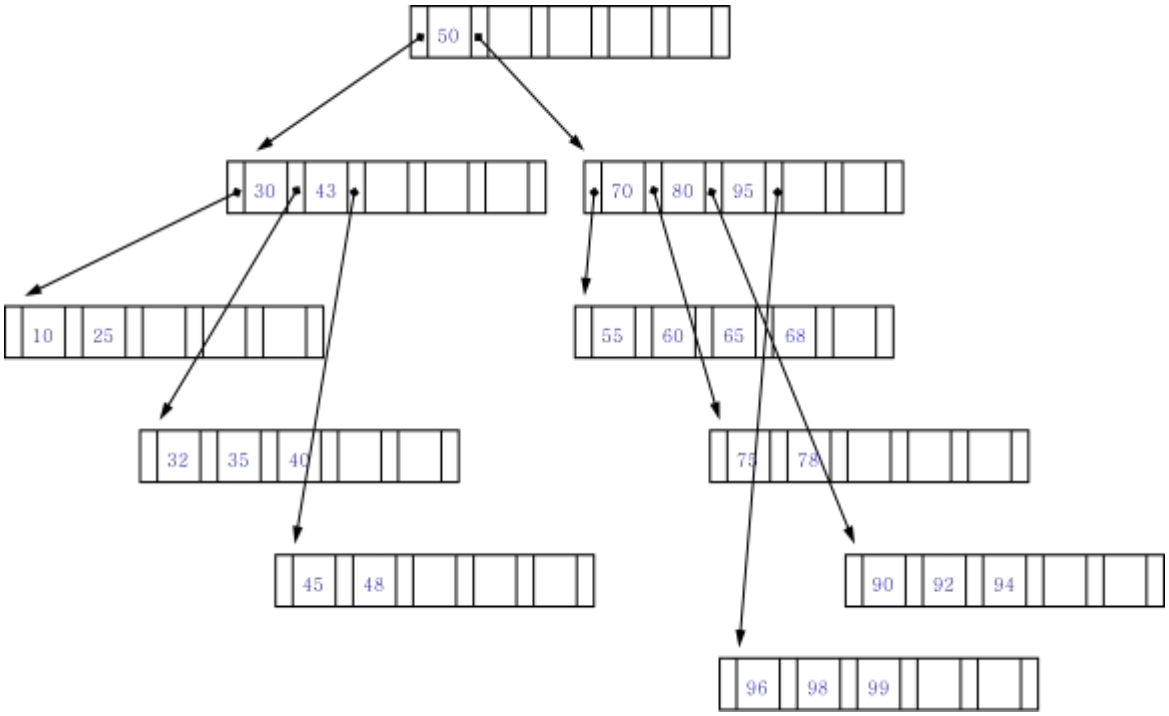
Example of Insertion in B-Tree (1)



Example of Insertion in B-Tree (2)



Example of Insertion in B-Tree (3)

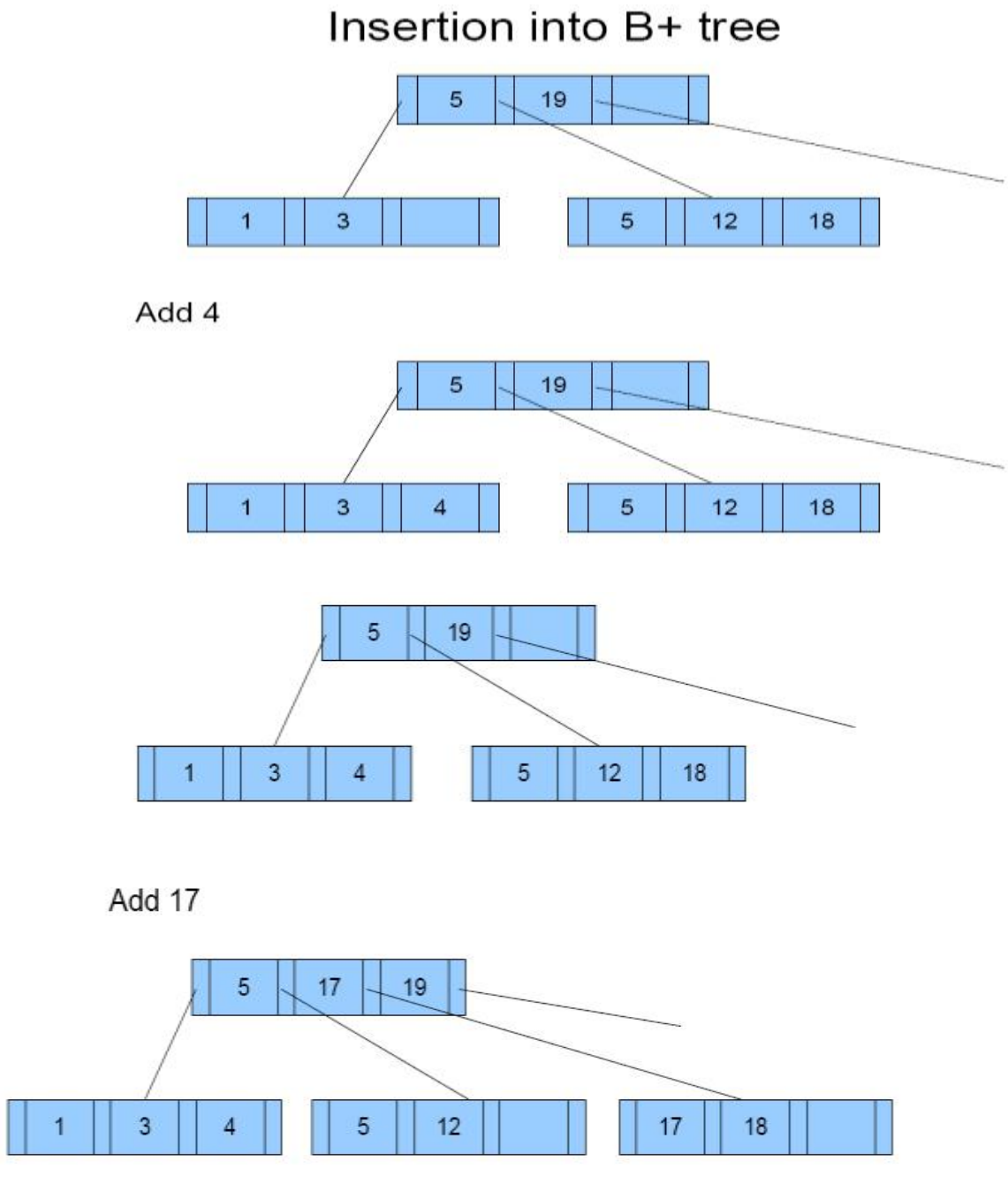


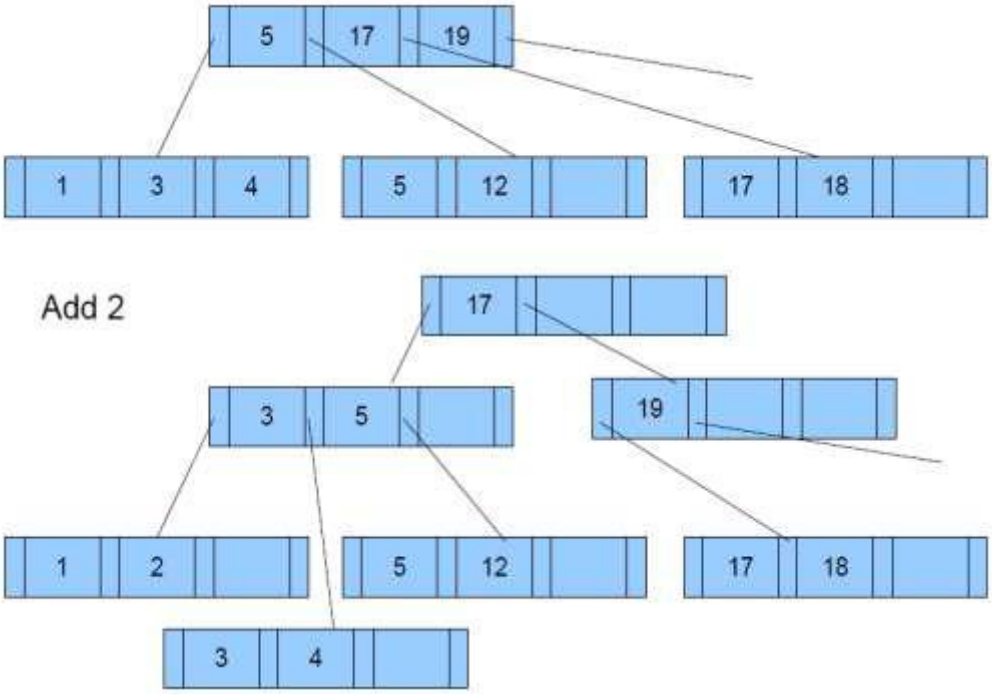
B tree insertion

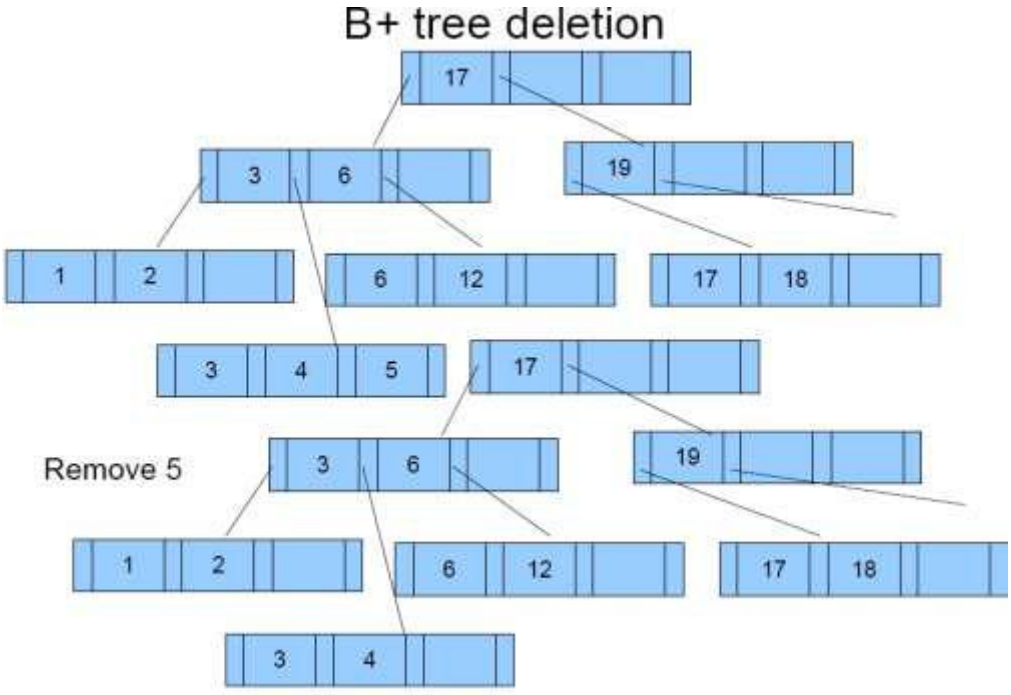
• When splitting leaf
- lowest value in right part gets inserted into parent
- value also stays in leaf

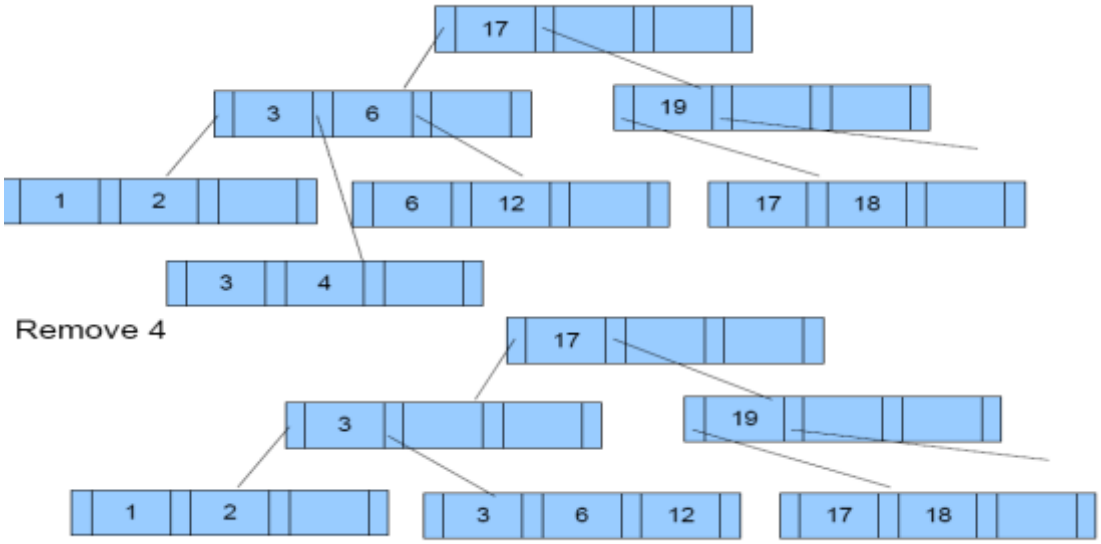
• When splitting internal node
- lowest value in right part gets inserted into parent
- value is removed from the right part

• Trick...
– to avoid cascading insertions, simply redistribute values among neighboring leaves

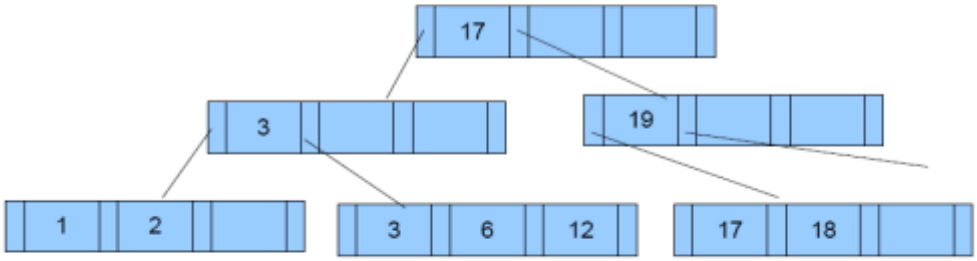

Insertion into B+ tree
5
19
1
3
5
12
18
Add 4
5
19
1
3
4
5
12
18
5
19
1
3
4
5
12
18
Add 17
5
17
19
1
3
4
5
12
17
18


5
17
19
1
3
4
5
12
17
18
Add 2
17
3
5
19
1
2
5
12
17
18
3
4

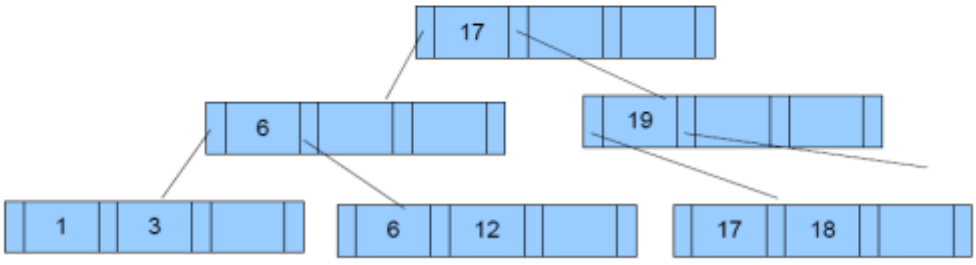

B+ tree deletion
17
3
6
19
1
2
6
12
17
18
3
4
5
17
Remove 5
3
6
19
1
2
6
12
17
18
3
4

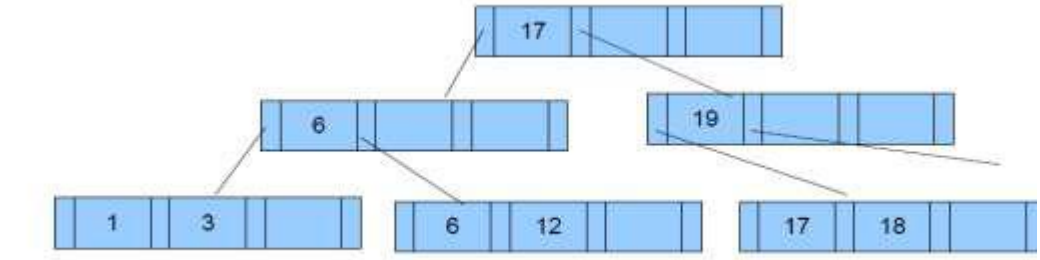


Remove 4

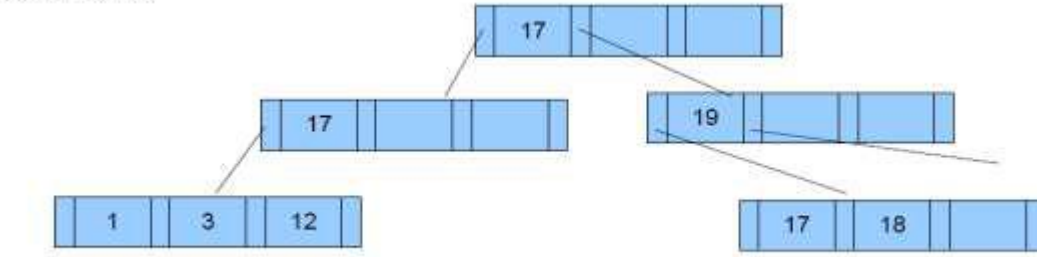


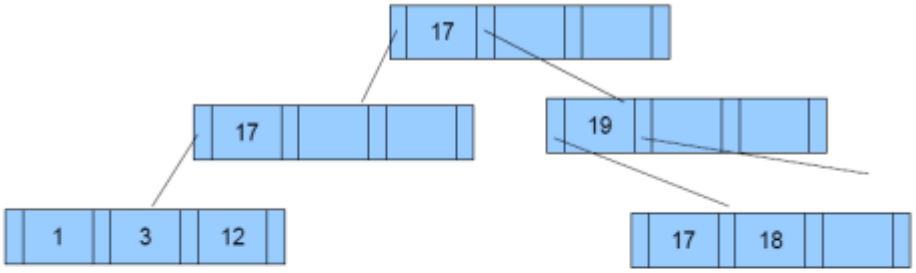
Remove 2

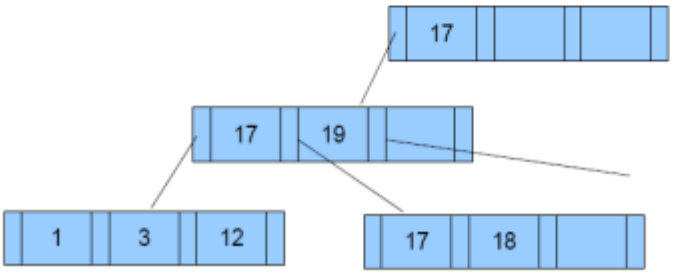


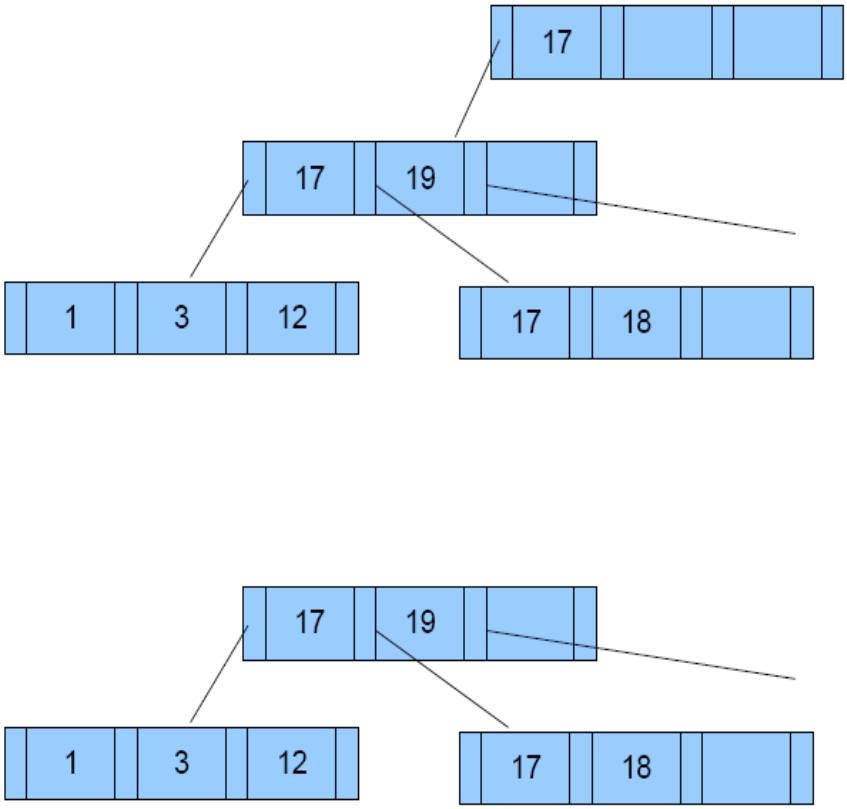

17
6
19
1
3
6
12
17
18

Remove 6


17
17
19
1
3
12
17
18


17
17
19
1
3
12
17
18


17
17
19
1
3
12
17
18



B+ tree deletion

• Operation on leaves and internal nodes the same
• Need to think a bit about the correct value for the 'middle' separating key

Application of Binary Trees:

Huffman Code Construction

Suppose we have messages consisting of sequences of characters. In each message, the characters are independent and appear with a known probability in any given position; the probabilities are the same for all positions.

e.g. - Suppose we have a message made from the five characters a, b, c, d, e, with probabilities 0.12, 0.40, 0.15, 0.08, 0.25, respectively.

We wish to encode each character into a sequence of 0s and 1s so that no code for a character is the prefix of the code for any other character. This prefix property allows us to decode a string of 0s and 1s by repeatedly deleting prefixes of the string that are codes for characters.

Symbol	Prob	code 1	code 2
a	0.12	000	000
b	0.40	001	11
c	0.15	010	01
d	0.08	011	001
e	0.25	100	10

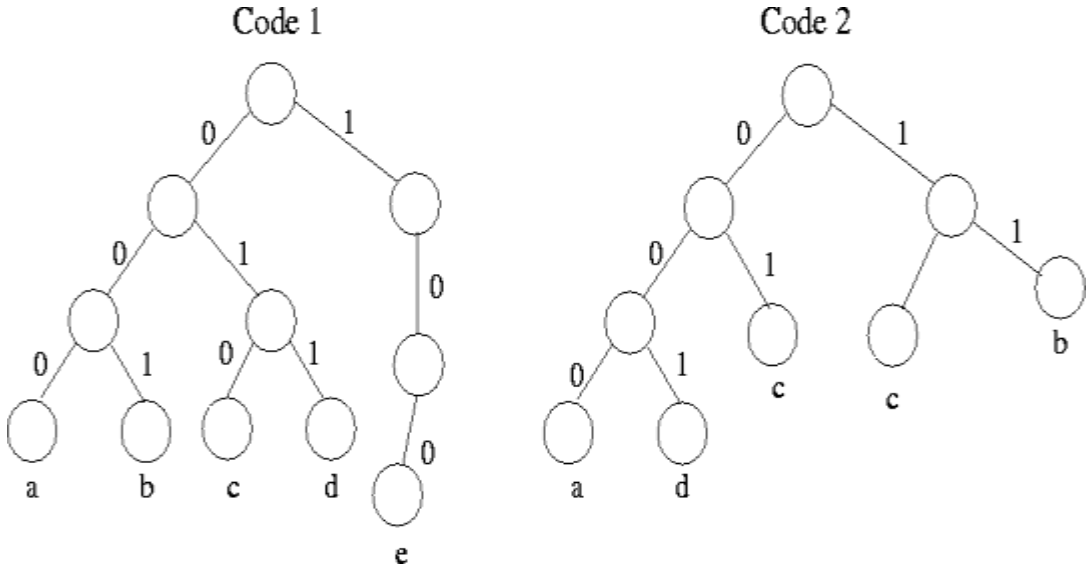
In the above, Code 1 has the prefix property. Code 2 also has the prefix property.

- The problem: given a set of characters and their probabilities, find a code with the *prefix property* such that the average length of a code for a character is a minimum.

  Code 1: (0.12)(3) + (0.4)(3) + (0.15)(3) + (0.08)(3) + (0.25)(3) = 3
  Code 2: (0.12)(3) + (0.4)(2) + (0.15)(2) + (0.08)(3) + (0.25)(2) = 2.2

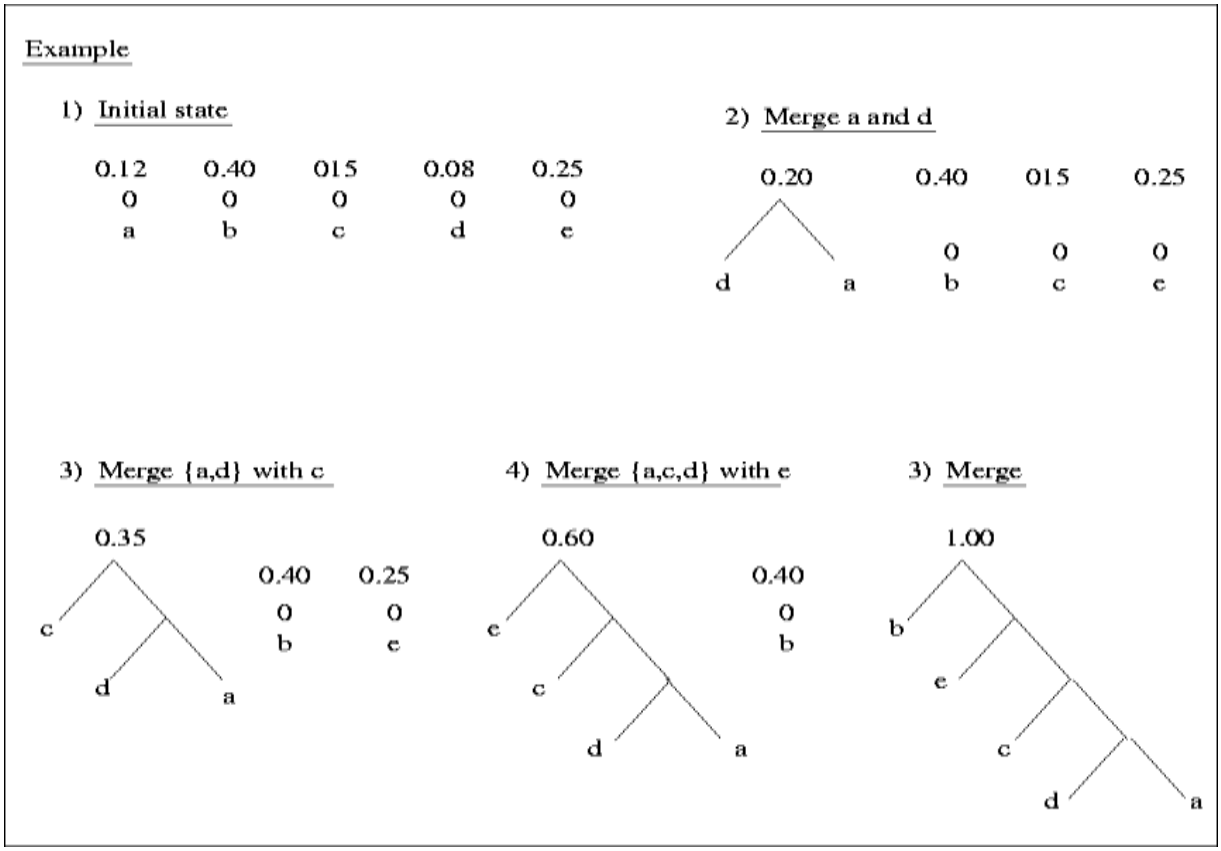
- Huffman's algorithm is one technique for finding optional prefix codes. The algorithm works by selecting two characters *a* and *b* having the lowest probabilities and replacing them with a single (imaginary) character, say *x*, whose probability of occurrence is the sum of probabilities for *a* and *b*. We then find an optimal prefix code for this smaller set of characters, using this procedure recursively. The code for the original character set is obtained by using the code for *x* with a 0appended for *"a"* and a 1 appended for *"b"*.
- We can think of prefix codes as paths in binary trees. For example, see Figure.

Figure : Code 1 and Code 2



- The prefix property guarantees that no character can have a code that is an interior node, and conversely, labeling the leaves of any binary tree with characters gives us a code with the prefix property for these characters.
- Huffman's algorithm is implemented using a *forest* (disjoint collection of trees), each of which has its leaves labeled by characters whose codes we desire to select and whose roots are labeled by the sum of the probabilities of all the leaf labels. We call this sum the *weight* of the tree.
- Initially each character is in a one-node tree by itself and when the algorithm ends, there will be only one tree, with all the characters as its leaves. In this final tree, the path from the root to any leaf represents the code for the label of that leaf.
- The essential step of the algorithm is to select the two trees in the forest that have the smallest weights (break ties arbitrarily). Combine these two trees into one, whose weight is the sum of the weights of the two trees. To combine the trees, we create a new node, which becomes the root and has the roots of the two given trees as left and right children (which is which doesn't matter). This process continues until only one tree remains.
- An example of Huffman algorithm is shown in Figure for five alphabets.

Figure An example of Huffman algorithm



Huffman Code: Example

The following example bases on a data source using a set of five different symbols. The symbol's frequencies are:

Symbol Frequency

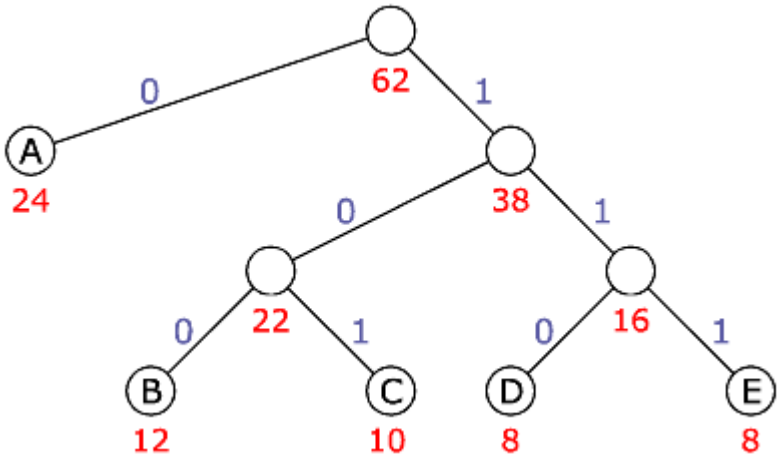
A	24
B	12
C	10
D	8
E	8

----> total 186 bit

(with 3 bit per code word)

The two rarest symbols 'E' and 'D' are connected first, followed by 'C' and 'D'. The new parent nodes have the frequency 16 and 22 respectively and are brought together in the next step. The resulting node and the remaining symbol 'A' are subordinated to the root node that is created in a final step.

Code Tree according to Huffman



Symbol	Frequency	Code	Code Length	total Length
A	24	0	1	24
B	12	100	3	36
C	10	101	3	30
D	8	110	3	24
E	8	111	3	24

---

ges. 186 bit tot. 138 bit

(3 bit code)

Chapter 7
Searching

INTRODUCTION

In our day-to-day life there are various applications, in which the process of searching is to be carried. Searching a name of a person from the given list, searching a specific card from the set of cards, etc., are few examples of searching.

A typical example where searching technique is applied is personal telephone diary. This diary is used to store phone / mobile numbers of friends, relatives, etc. For searching the phone number of a person one can directly search the number by using indexing in that diary. Quick search methods are to be adopted to search the data from the heap or large records. Two processes such as searching and sorting are essential for database applications.

SEARCHING

Searching is a technique of finding an element from the given data list or set of the elements like an array, list, or trees. It is a technique to find out an element in a sorted or unsorted list. For example, consider an array of 10 elements. These data elements are stored in successive memory locations. We need to search an element from the array. In the searching operation, assume a particular element n is to be searched. The element n is compared with all the elements in a list starting from the first element of an array till the last element. In other words, the process of searching is continued till the element is found or list is completely exhausted. When the exact match is found then the search process is terminated. In case, no such element exists in the array, the process of searching should be abandoned.

In case the given element is in the set of elements or array then the search process is said to be successful. It means that the given element belongs to the array. The search is said to be unsuccessful if the given element does not exist in the array. It means that the given element does not belong to the array or collection of the items.

Linear Search

The linear search is a conventional method of searching data. The linear search is a method of searching a target element in a list sequence. The expected element is to be searched in the entire data structure in a sequential method from starting to last element. Though, it is simple and straightforward, it has serious limitations. It consumes more time and reduces the retrieval rate of the system. The linear or sequential name implies that the items are stored in systematic manner. The linear search can be applied on sorted or unsorted linear data structure.

Figure %: The array we're searching

Lets search for the number 3. We start at the beginning and check the first element in the array. Is it 3?

Figure %: Is the first value 3?
No, not it. Is it the next element?

Figure %: Is the second value 3?
Not there either. The next element?

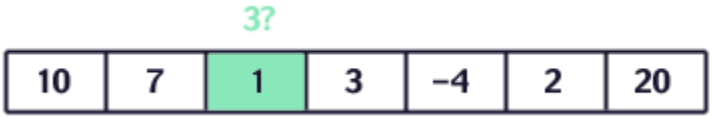


Figure %: Is the third value 3?
Not there either. Next?

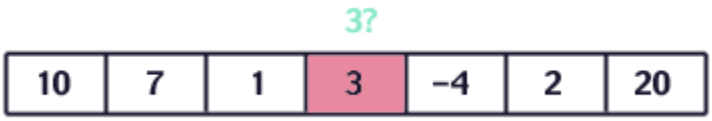


Figure %: Is the fourth value 3? Yes!

We found it!!! Now you understand the idea of linear searching; we go through each element, in order, until we find the correct value

Binary Search

The binary search approach is different from the linear search. Here, search of an element is not passed in sequence as in the case of linear search. Instead, two partitions of lists are made and then the given element is searched. Hence, it creates two parts of the lists known as binary search.

Binary search is quicker than the linear search. It is an efficient searching technique and works with sorted lists. However, it cannot be applied on unsorted data structure. Before applying binary search, the linear data structure must be sorted in either ascending or descending order. The binary search is unsuccessful if the elements are unordered. The binary search is based on the approach divide-and-conquer. In binary search, the element is compared with the middle element. If the expected element falls before the middle element, the left portion is searched otherwise right portion is searched.

```
/*   File: sortsrch.c - continued */
     /*   Function uses binary search to search for item in the array y[]. */
     int binsrch(int y[], int lim, int key)
     {    int low, mid, high = lim - 1;

          low = 0;

          while (low <= high) {          /* Is the array exhausted?   */
               mid = (low + high) / 2;   /* If not, find middle index */

               if (key == y[mid])        /* Is the key here?          */
                    return(mid);         /* If so, return index.      */

               else if (key < y[mid])    /* else if key is smaller,   */
                    high = mid - 1;      /* reduce the high end;      */

               else
                    low = mid + 1;       /* otherwise, increase low   */
          }
          return(-1);                    /* Not found, return -1      */
     }
```

Figure 10.14: Code for Binary Search

We use the binsrch() function in an example program which repeatedly searches for numbers input by the user. For each number, it either gives the index where it is found or prints a message if it is not found. An array in sorted form is initialized in the declaration. The code for this driver is shown in Figure.

```
/*   File: bsrcharay.c
          Other Source Files: sortsrch.c
          Header Files: sortsrch.h
          Program uses binary search to search a sorted array of numbers.
     */

     #include <stdio.h>
     #define MAX 100
     #define DEBUG
     #include "sortsrch.h"

     main()
     {    int i, x, y[MAX] = {12, 29, 30, 32, 35, 49};
          int k = 6;          /* no. of items in the array y□ */

          printf("***Binary Search***\n\n");
          printf("The array is:\n");
          pr_aray_line(y, k);
          printf("Type a number, EOF to quit: ");

          while (scanf("%d", &x) != EOF) {
               i = binsrch(y, k, x);

               if (i >= 0)
                    printf("%d found at array index %d\n", x, i);
               else
                    printf("%d not found in array\n", x);

               printf("Type a number, EOF to quit: ");
          }
     }
```

Figure 10.15: Test Driver for Binary Search

Sample Session:

- ***Binary Search***
-
- The array is:
- 12 29 30 32 35 49
-
- Type a number, EOF to quit: *12*
- 12 found at array index 0
-
- Type a number, EOF to quit: *23*
- 23 not found in array
-
- Type a number, EOF to quit: *34*
- 34 not found in array

-
- Type a number, EOF to quit: *45*
- 45 not found in array
-
- Type a number, EOF to quit: *30*
- 30 found at array index 2
-
- Type a number, EOF to quit: *29*
- 29 found at array index 1

Hashing is a method for storing and retrieving records from a database. It lets you insert, delete, and search for records based on a search key value. When properly implemented, these operations can be performed in constant time. In fact, a properly tuned hash system typically looks at only one or two records for each search, insert, or delete operation. This is far better than the O(log *n*) average cost required to do a binary search on a sorted array of *n* records, or the O(log *n*) average cost required to do an operation on a binary search tree. However, even though hashing is based on a very simple idea, it is surprisingly difficult to implement properly. Designers need to pay careful attention to all of the details involved with implementing a hash system.

A hash system stores records in an array called a **hash table**, which we will call **HT**. Hashing works by performing a computation on a search key *K* in a way that is intended to identify the position in **HT** that contains the record with key *K*. The function that does this calculation is called the **hash function**, and will be denoted by the letter **h**. Since hashing schemes place records in the table in whatever order satisfies the needs of the address calculation, records are not ordered by value. A position in the hash table is also known as a **slot**. The number of slots in hash table **HT** will be denoted by the variable *M* with slots numbered from 0 to *M* - 1.

Hashing

Definition: The mapping of a value (often a key value) to a position in a linear space (hash space) by means of a hash function **h**.

h: key => position (a mapping from a key space to a hash space)

Issues:

- main advantage: **search time is O(1)** (i.e. constant)
- **collision handling** (i.e. if h(x) maps to the same position as h(y))
- the **distribution of values** in the key space
  if this is known, some form of optimisation may be applied
- the **load factor** in the hash space (see the animation below)
- the **number of probes** required (increases as the load factor increases)

Hashing Techniques:

- Division Method: often h(k) = k mod M (M is usually prime)
- Multiplication Method: h(k) = floor(M (kA - floor(kA))) (0 < A < 1)

Collision Handling Techniques:
On the nth collision, the probe tries position h(k) + f(n) (n = 1, 2, 3....)

- Open addressing (Closed Hashing) [f(n) = n]
  -ve: primary clustering / search time O(n)
- Quadratic Probing [f(n) = n * n]
  -ve: secondary clustering
- Double Hashing [f(n) = n * h2(k) - h2 is another hashing function]
  -ve: h2 may increase calculation time
- Separate Chaining (Open Hashing) [a linked list is created]
  -ve: search time O(n)
- Overflow Slots (see the animation below)
  -ve: search time O(n)

Applications:

- In compilers to map to the symbol table
- Historically was used in database systems (B-trees are now used)

Collision resolution techniques

Method 1: Chaining or Open hashing

Input sequence: 1, 27, 6, 87, 47, 7, 8, 17, 37, 67

Hash Function: key mod 10

Index	Hash Table	Collision chain
0		
1	1	
2		
3		
4		
5		
6	6	
7	27	87, 47, 7, 17, 37, 67
8	8	
9		

Method 2: Closed hashing - linear probing

Input sequence: 1, 27, 6, 87, 47, 7, 8, 17, 37, 67

Hash Function: key mod 10

Collision Handling: H(key) + f(i) where f(i) = i

Index	**Hash Table**									
add:	1	27	6	87	47	7	8	17	37	67
0						**7**	7	7	7	7
1	**1**	1	1	1	1	1	1	1	1	1
2							**8**	8	8	8
3								**17**	17	17
4									**37**	37
5										**67**
6			**6**	6	6	6	6	6	6	6
7		**27**	27	27	27	27	27	27	27	27
8				**87**	87	87	87	87	87	87
9					**47**	47	47	47	47	47

Calculation Sequence

1. **add 1** : no collision ==> index 1
2. **add 27** : no collision ==> index 7
3. **add 6** : no collision ==> index 6
4. **add 87** : **collision** ==> index 7
    1. collision 1: new try: 7 + 1 ==> 8 (OK)
5. **add 47** : **collision** ==> index 7
    1. collision 1: new try: 7 + 1 ==> 8 (NOT OK)
    2. collision 2: new try: 7 + 2 ==> 9 (OK)
6. **add 7** : **collision** ==> index 7
    1. collision 1: new try: 7 + 1 ==> 8 (NOT OK)
    2. collision 2: new try: 7 + 2 ==> 9 (NOT OK)
    3. collision 3: new try: 7 + 3 ==> 10 (mod 10) ==> 0 (OK)
7. **add 8** :**collision** ==> index 8
    1. collision 1: new try: 8 + 1 ==> 9 (NOT OK)
    2. collision 2: new try: 8 + 2 ==> 10 (mod 10) ==> 0 (NOT OK)
    3. collision 3: new try: 8 + 3 ==> 11 (mod 10) ==> 1 (NOT OK)
    4. collision 4: new try: 8 + 4 ==> 12 (mod 10) ==> 2 (OK)
8. **add 17** : **collision** ==> index 7
    1. collision 1: new try: 7 + 1 ==> 8 (NOT OK)

2. collision 2: new try: 7 + 2 ==> 9 (NOT OK)
3. collision 3: new try: 7 + 3 ==> 10 (mod 10) ==> 0 (NOT OK)
4. collision 4: new try: 7 + 4 ==> 11 (mod 10) ==> 1 (NOT OK)
5. collision 5: new try: 7 + 5 ==> 12 (mod 10) ==> 2 (NOT OK)
6. collision 6: new try: 7 + 6 ==> 13 (mod 10) ==> 3 (OK)

9. **add 37** : **collision** ==> index 7
    1. collision 1: new try: 7 + 1 ==> 8 (NOT OK)
    2. collision 2: new try: 7 + 2 ==> 9 (NOT OK)
    3. collision 3: new try: 7 + 3 ==> 10 (mod 10) ==> 0 (NOT OK)
    4. collision 4: new try: 7 + 4 ==> 11 (mod 10) ==> 1 (NOT OK)
    5. collision 5: new try: 7 + 5 ==> 12 (mod 10) ==> 2 (NOT OK)
    6. collision 5: new try: 7 + 6 ==> 13 (mod 10) ==> 3 (NOT OK)
    7. collision 6: new try: 7 + 7 ==> 14 (mod 10) ==> 4 (OK)
10. **add 67** : **collision** ==> index 7
    1. collision 1: new try: 7 + 1 ==> 8 (NOT OK)
    2. collision 2: new try: 7 + 2 ==> 9 (NOT OK)
    3. collision 3: new try: 7 + 3 ==> 10 (mod 10) ==> 0 (NOT OK)
    4. collision 4: new try: 7 + 4 ==> 11 (mod 10) ==> 1 (NOT OK)
    5. collision 5: new try: 7 + 5 ==> 12 (mod 10) ==> 2 (NOT OK)
    6. collision 5: new try: 7 + 6 ==> 13 (mod 10) ==> 3 (NOT OK)
    7. collision 5: new try: 7 + 7 ==> 14 (mod 10) ==> 4 (NOT OK)
    8. collision 6: new try: 7 + 8 ==> 15 (mod 10) ==> 6 (OK)

Method 3: Closed hashing - quadratic probing

Input sequence: 1, 27, 6, 87, 47, 7, 8, 17, 37, 67

Hash Function: key mod 10

Collision Handling: H(key) + f(i) where f(i) = i*i

Index	**Hash Table**									
add:	1	27	6	87	47	7	8	17	37	67
0										
1	**1**	1	1	1	1	1	1			
2						**7**	7			
3					**47**	47	47			
4										
5										
6			**6**	6	6	6	6			
7		**27**	27	27	27	27	27			
8				**87**	87	87	87			

9							**8**			

Calculation Sequence

1. **add 1** : no collision ==> index 1
2. **add 27** : no collision ==> index 7
3. **add 6** : no collision ==> index 6
4. **add 87** : **collision** ==> index 7
    1. collision 1: new try: 7 + 1 ==> 8 (OK)
5. **add 47** : **collision** ==> index 7
    1. collision 1: new try: 7 + 1 ==> 8 (NOT OK)
    2. collision 2: new try: 7 + 4 ==> 11 (mod 10) ==> 1 (NOT OK)
    3. collision 3: new try: 7 + 9 ==> 16 (mod 10) ==> 6 (NOT OK)
    4. collision 4: new try: 7 + 16 ==> 23 (mod 10) ==> 3 (OK)
6. **add 7** : **collision** ==> index 7
    1. collision 1: new try: 7 + 1 ==> 8 (NOT OK)
    2. collision 2: new try: 7 + 4 ==> 11 (mod 10) ==> 1 (NOT OK)
    3. collision 3: new try: 7 + 9 ==> 16 (mod 10) ==> 6 (NOT OK)
    4. collision 4: new try: 7 + 16 ==> 23 (mod 10) ==> 3 (NOT OK)
    5. collision 5: new try: 7 + 25 ==> 32 (mod 10) ==> 2 (OK)
7. **add 8** :**collision** ==> index 8
    1. collision 1: new try: 8 + 1 ==> 9 (OK)
8. **add 17** : **collision** ==> index 7
    1. collision 1: new try: 7 + 1 ==> 8 (NOT OK)
    2. collision 2: new try: 7 + 4 ==> 11 (mod 10) ==> 1 (NOT OK)
    3. collision 3: new try: 7 + 9 ==> 16 (mod 10) ==> 6 (NOT OK)
    4. collision 4: new try: 7 + 16 ==> 23 (mod 10) ==> 3 (NOT OK)
    5. collision 5: new try: 7 + 25 ==> 32 (mod 10) ==> 2 (NOT OK)
    6. collision 6: new try: 7 + 36 ==> 43 (mod 10) ==> 3 (NOT OK)
    7. collision 7: new try: 7 + 49 ==> 56 (mod 10) ==> 6 (NOT OK)
    8. collision 8: new try: 7 + 64 ==> 71 (mod 10) ==> 1 (NOT OK)
    9. collision 9: new try: 7 + 81 ==> 88 (mod 10) ==> 8 (NOT OK)
    10. collision 10: new try: 7 + 100 ==> 107 (mod 10) ==> 7 (NOT OK)
        need we go further? The sequence is repeating (8,1,6,3,2,3,6,1,8,7)
    11. collision 11: new try: 7 + xx1 ==> xx8 (mod 10) ==> 8 (NOT OK)
    12. collision 12: new try: 7 + xx4 ==> xx1 (mod 10) ==> 1 (NOT OK)
    13. collision 13: new try: 7 + xx9 ==> xx6 (mod 10) ==> 6 (NOT OK)
    14. collision 14: new try: 7 + xx6 ==> xx3 (mod 10) ==> 3 (NOT OK)
    15. collision 15: new try: 7 + xx5 ==> xx2 (mod 10) ==> 2 (NOT OK)
    16. collision 16: new try: 7 + xx6 ==> xx3 (mod 10) ==> 3 (NOT OK)
    17. collision 17: new try: 7 + xx9 ==> xx6 (mod 10) ==> 6 (NOT OK)
    18. collision 18: new try: 7 + xx4 ==> xx1 (mod 10) ==> 1 (NOT OK)
    19. collision 19: new try: 7 + xx1 ==> xx8 (mod 10) ==> 8 (NOT OK)
    20. collision 20: new try: 7 + xx0 ==> xx7 (mod 10) ==> 7 (NOT OK)
        Time to give up!

COMMENT 1: We know that for quadratic probing, with a greater that 50% load, there is a risk that the collision handling technique will be unable to find a free position in the hash table.

COMMENT 2: If we looked at the input sequence we see that this reqquires 100% load in the hash table **AND** that there were 7 entries ending in a 7!

COMMENT 3: With this input sequence, we at least made it to 70% full in the table!

Method 4: Double hashing

Input sequence: 1, 27, 6, 87, 47, 7, 8, 17, 37, 67

Hash Function: key mod 10

Collision Handling: H1(key) + f(i) where f(i) = i * H2(key)

and H2(key) = 7 - (key mod 7)

Index	**Hash Table**									
add:	1	27	6	87	47	7	8	17	37	67
0										**67**
1	**1**	1	1	1	1	1	1	1	1	1
2									**37**	37
3								**17**	17	17
4						**7**	7	7	7	7
5				**87**	87	87	87	87	87	87
6			**6**	6	6	6	6	6	6	6
7		**27**	27	27	27	27	27	27	27	27
8							**8**	8	8	8
9					**47**	47	47	47	47	47

Calculation Sequence

1. **add 1** : no collision ==> index 1
2. **add 27** : no collision ==> index 7
3. **add 6** : no collision ==> index 6
4. **add 87** : **collision** ==> index 7
    1. collision 1: new try: H2 = (7 - 87 mod 7) = (7 - 3) = 4;
       7 + 1*4 ==> 11 (mod 10) ==> 1 (NOT OK)
    2. collision 2: new try: 7 + 2*4 ==> 15 (mod 10) ==> 5 (OK)
5. **add 47** : **collision** ==> index 7

    1. collision 1: new try: H2 = (7 - 47 mod 7) = (7 - 5) = 2;
       7 + 1*2 ==> 9 (mod 10) ==> 9 (OK)
6. **add 7** : **collision** ==> index 7
    1. collision 1: new try: H2 = (7 - 7 mod 7) = (7 - 0) = 7;
       7 + 1*7 ==> 14 (mod 10) ==> 4 (OK)
7. **add 8** : no collision ==> index 8
8. **add 17** : **collision** ==> index 7
    1. collision 1: new try: H2 = (7 - 17 mod 7) = (7 - 3) = 4;
       7 + 1*4 ==> 11 (mod 10) ==> 1 (NOT OK)
    2. collision 2: new try: 7 + 2*4 ==> 15 (mod 10) ==> 5 (NOT OK)
    3. collision 3: new try: 7 + 3*4 ==> 19 (mod 10) ==> 9 (NOT OK)
    4. collision 4: new try: 7 + 4*4 ==> 23 (mod 10) ==> 3 (OK)
9. **add 37** : **collision** ==> index 7
    1. collision 1: new try: H2 = (7 - 37 mod 7) = (7 - 2) = 5;
       7 + 1*5 ==> 12 (mod 10) ==> 2 (OK)
10. **add 67** : **collision** ==> index 7
    1. collision 1: new try: H2 = (7 - 67 mod 7) = (7 - 4) = 3;
       7 + 1*3 ==> 10 (mod 10) ==> 0 (OK)

Chapter 8
Sorting

INTRODUCTION

In our daily life, we keep the information in particular order. This helps in accessing any part of it easily. Likewise, in database programs, large data is maintained in the form of records. It would be very difficult if data or records were unsorted. It is very boring to find a particular record if data or records are randomly stored. The process of sorting is essential for database applications.

Sorting is a process in which records are arranged in ascending or descending order. A group of fields is called record.

SORTING

As already defined in the previous section sorting is a process in which records are arranged in ascending or descending order. In real life we come across several examples of sorted information. For example, in telephone directory the names of the subscribers and their phone numbers are written in alphabetical order. The records of the list of these telephone holders are to be sorted by their names. By using this directory, we can access the telephone number and address of the subscriber very easily.

Bankers or businesspersons sort the currency denomination notes received from customers in the appropriate form. Currency denominations of Rs 1000, 500, 100, 50, 20, 10, 5, and 1 are separated first and then separate bundles are prepared.

INSERTION SORT

In insertion sort the element is inserted at appropriate place. For example, consider an array of n elements. In this type, swapping of elements is done without taking any temporary variable. The greater numbers are shifted towards the end of the array and smaller are shifted to beginning. Here, a real life example of playing cards can be cited. We keep the cards in increasing order. The card having least value is placed at the extreme left and the largest one at the other side. In between them the other cards are managed in ascending order.

Example

[37 | 32, 70, 15, 93, 40, 63, 3, 40, 63]

[32, 37 | 70, 15, 93, 40, 63, 3, 40, 63]

[32, 37, 70 | 15, 93, 40, 63, 3, 40, 63]

[15, 32, 37, 70 | 93, 40, 63, 3, 40, 63]

[15, 32, 37, 70, 93 | 40, 63, 3, 40, 63]

[15, 32, 37, 40, 70, 93 | 63, 3, 40, 63]
[15, 32, 37, 40, 63, 70, 93 | 3, 40, 63]

[3, 15, 32, 37, 40, 63, 70, 93 | 40, 63]

[3, 15, 32, 37, 40, 40, 63, 70, 93 | 63]

[3, 15, 32, 37, 40, 40, 63, 63, 70, 93]

SELECTION SORT

The selection sort is nearly the same as exchange sort. Assume, we have a list containing elements. By applying selection sort, the first element is compared with all remaining (n-1) elements. The smallest element is placed at the first location. Again, the second element is compared with remaining (n-2) elements. If the item found is lesser than the compared elements in the remaining (n-2) list then the swap operation is done. In this type, entire array is checked for the smallest element and then swapped.

In each pass, one element is sorted and kept at the left. Initially, the elements are temporarily sorted and after next pass, they are permanently sorted and kept at the left. Permanently sorted elements are covered with squares and temporarily with circles. Element inside the circle "O" is chosen for comparing with the other elements marked in a circle and sorted temporarily

Example

64 25 12 22 11

Step 2: 11 25 12 22 64

Step 3: 11 12 25 22 64

Step 4: 11 12 22 25 64

Step 5: 11 12 22 25 64

BUBBLE SORT

Bubble sort is a commonly used sorting algorithm. In this type, two successive elements are compared and swapping is done if the first element is greater than the second one. The elements are sorted in ascending order. It is easy to understand but it is time consuming.

Bubble sort is an example of exchange sort. In this method repetitively comparison is performed between the two successive elements and if essential swapping of elements is done. Thus, step-by-step entire array elements are checked. It is different from the selection sort. Instead of searching the

minimum element and then applying swapping, two records are swapped instantly upon noticing that they are not in order.

Example

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

First Pass:
(**5 1** 4 2 8) →(**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) →(1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) →(1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) →(1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(**1 4** 2 5 8) →(**1 4** 2 5 8)
(1 **4 2** 5 8) →(1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) →(1 2 **4 5** 8)
(1 2 4 **5 8**) →(1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
Third Pass:
(**1 2** 4 5 8) →(**1 2** 4 5 8)
(1 **2 4** 5 8) →(1 **2 4** 5 8)
(1 2 **4 5** 8) →(1 2 **4 5** 8)
(1 2 4 **5 8**) →(1 2 4 **5 8**)

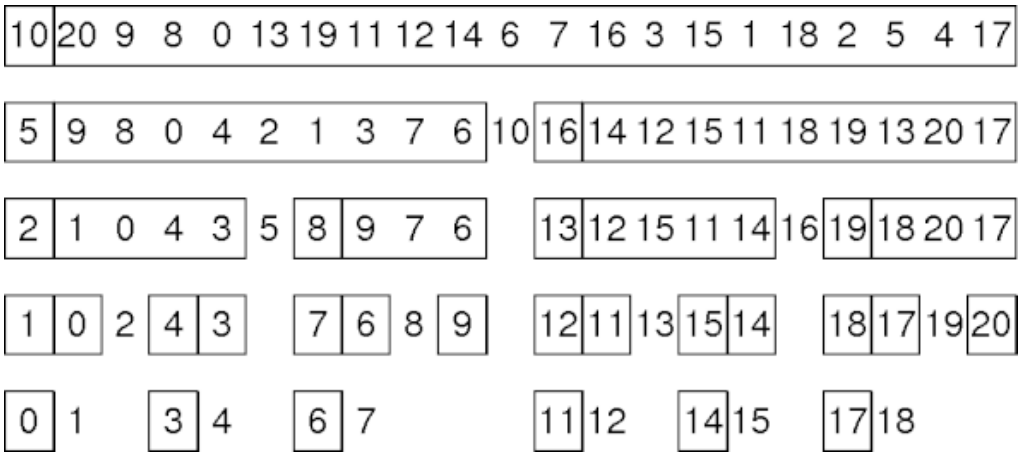
QUICK SORT

It is also known as partition exchange sort. Quick sort is a divide and conquer algorithm. Quick sort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quick sort can then recursively sort the sub-lists.

The steps are:

1. Pick an element, called a **pivot**, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

Example

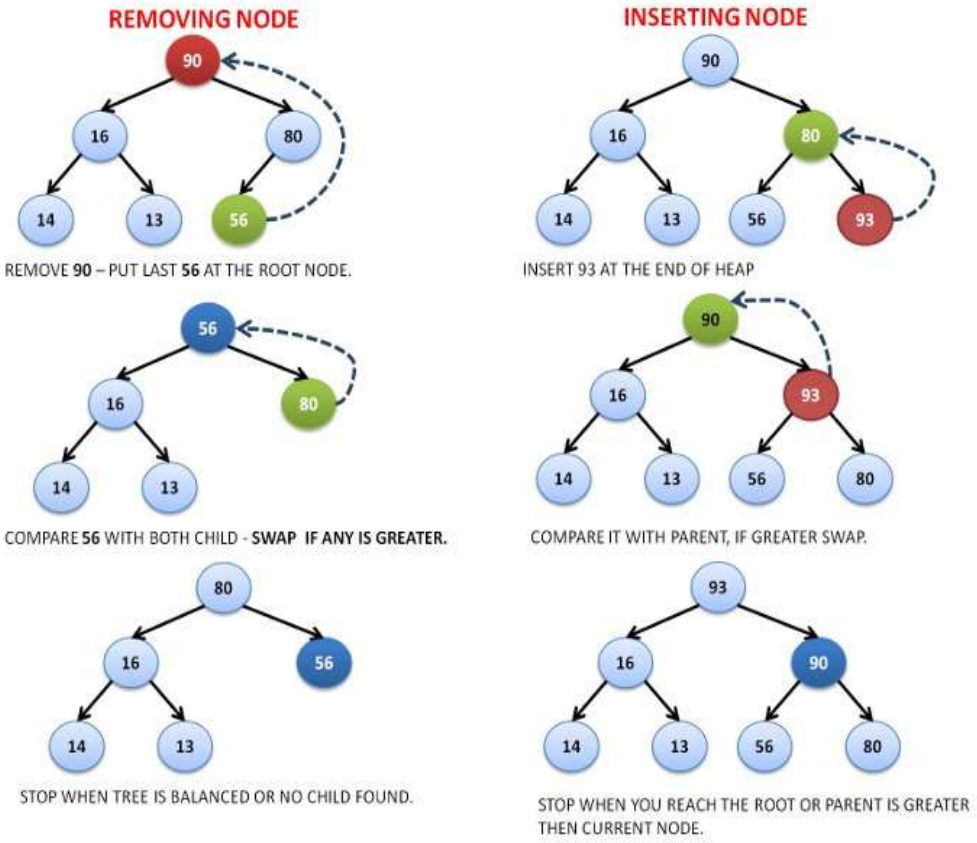


Consider the following example in which five elements 8, 9, 7, 6, 4 are to be sorted using quick sort.

HEAP SORT

In heap sort, we use the binary tree in which the nodes are arranged in specific prearranged order. The rule prescribes that each node should have bigger value than its child node. The following steps are to be followed in heap sort:

1. Arrange the nodes in the binary tree form.
2. Node should be arranged as per the specific rules.
3. If the inserted node is bigger than its parent node then replace the node.
4. If the inserted node is lesser than its parent node then do not change the position.
5. Do not allow entering nodes on right side until the child nodes of the left are fulfilled.
6. Do not allow entering nodes on left side until the child nodes of the right are fulfilled.
7. The procedure from step 3 to step 6 is repeated for each entry of the node.



RADIX SORT

The radix sort is a technique, which is based on the position of digits. The number is represented with different positions. The number has units, tens, hundreds positions onwards. Based on its position the sorting is done. For example, consider the number 456, which is selected for sorting. In this number 6 is at units position and 5 is at tens and 4 is at the hundreds position. 3 passes would be needed for the sorting of this number with this procedure. In the first pass, we place all numbers at the unit place. In the second pass all numbers are placed in the list with consents to the tens position digit value. Also, in the third pass the numbers are placed with consent to the value of the hundreds position digit.

Merge Sort

Merge sort is based on the **divide-and-conquer** paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p \,..\, r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.

To sort $A[p \,..\, r]$:

1. **Divide Step**

If a given array *A* has zero or one element, simply return; it is already sorted. Otherwise, split *A*[*p* .. *r*] into two subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*], each containing about half of the elements of *A*[*p* .. *r*]. That is, *q* is the halfway point of *A*[*p* .. *r*].

2. **Conquer Step**

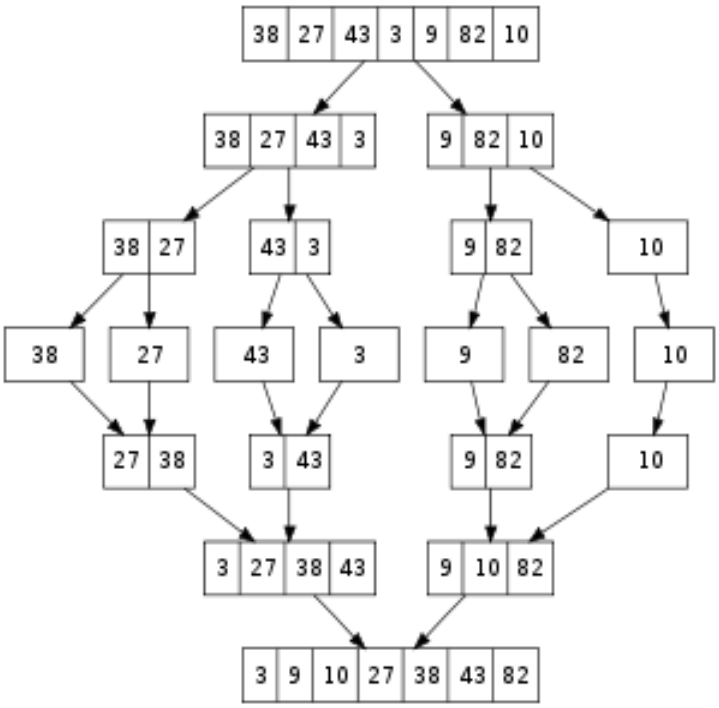
Conquer by recursively sorting the two subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*].

3. **Combine Step**

Combine the elements back in *A*[*p* .. *r*] by merging the two sorted subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (*A*, *p*, *q*, *r*).

Note that the recursion bottoms out when the subarray has just one element, so that it is trivially sorted.

Example




```
void BubbleSort(int a[])
{
   int i,j;
```

```
   for (i=MAXLENGTH; --i >=0;) {
      swapped = 0;
      for (j=0; j<i;j++) {
        if (a[j]>a[j+1]) {
          Swap[a[j],a[j+1]);
          swapped=1;
        }
      }
      if (!swapped) return;
   }
}

void Heapify(int A[],int i,int HeapSize)
{
  int left=2*i, right=2*i+1;
  int largest;

  if ((left <= HeapSize) &&
      (A[left] > A[i]))
      largest = left;
  else
      largest = i;

  if ((right <= HeapSize) &&
      (A[right] > A[largest]))
      largest = right;

  if (largest != i) {
    Swap(&A[i],&A[largest]);
    Heapify(A,largest,HeapSize);
  }
}

void HeapSort(int A[], int n)
{
     int i, HeapSize = n;

     for (i= HeapSize/2; i >= 1; i--)
         Heapify(A,i,HeapSize);

     for (i=n; i>=2; i--) {
          Swap(&A[i],&A[1]);
          HeapSize--;
          Heapify(A,1,HeapSize);
     }
```

```
}

void InsertionSort(LIST A)
{
  int f, i;
  KEYTYPE temp;

  for (f = 1; f < MAXDIM; f++) {
     if (A[f] > A[f-1]) continue;
     temp = A[f];
     i    = f-1;
     while ((i>=0)&&(A[i]>temp)) {
        A[i+1] = A[i];
        i--;
     }
     A[i+1]=temp;
  }
}


int FindPivot(int A[],int l, int r)
{
    switch (choice) {
       case 1: return l;
       case 2: return pivot2(A,l,h)
       case 3: return l+random(r-l);
    }
}

int partition(int A[], int l, int r)
{
   int i,pivot, pivotpos;

   pivotpos    = FindPivot(A,l,r);
   swap(&A[l],&A[pivotpos]);
   pivotpos = l;
   pivot = A[pivotpos];

   for (i = l+1; i <= r; i++) {
     if (A[i] < pivot) {
        pivotpos++;
        swap(&A[pivotpos],&A[i]);
     }
   }

   swap(&A[l],&A[pivotpos]);
```

```
   return pivotpos;
}

void QuickSort(int A[], int l,int r,
               int threshold)
{
  int i, pivot;

  if (r-l>threshold) {
     delay(CompareDelay);
     pivot = partition(A,l,r);

     QuickSort(A,l,pivot-1,threshold);
     QuickSort(A,pivot+1,r,threshold);
  }
}

int pivot2(int A[], int l, int r)
{
  int i = (r+l)/2;

  if ((A[l] <= A[i]) && (A[i] <= A[r]))
     return i;
  if ((A[r] <= A[i]) && (A[i] <= A[l]))
     return i;
  if ((A[r] <= A[l]) && (A[l] <= A[i]))
     return l;
  if ((A[i] <= A[l]) && (A[l] <= A[r]))
     return l;
  if ((A[l] <= A[r]) && (A[r] <= A[i]))
     return r;
  if ((A[i] <= A[r]) && (A[r] <= A[l]))
     return r;
}


void SelectSort(int A)
{
     int i, j, min, t;

     for (i =1; i<= MAXSIZE; i++) {
          min  = i;
          for (j = i+1; j<=MAXSIZE; j++)
            if (A[j] < A[min])
              min = j;
```

```
        Swap(&A[min],&A[i]);
    }
}


void ShellSort(int A[])
{
  int i, j, h=1, v;

  do
    h = 3*h+1;
  while (h <= MAXSIZE);

  do {
    h /= 3;
    for (i=h+1; i<= MAXSIZE; i++) {
      v = A[i];
      j = i;
      while ((j>h) && (A[j-h] > v)) {
        A[j] = A[j-h];
        j -= h;
      }
      A[j] = v;
    }
  } while (h > 1);
}
```

Graph

A ***graph*** is a finite set of points, called ***vertices***, together with a finite set of lines, called ***edges***, that join some or all of these points. More formally:.

A graph ***G*** consists of a finite set ***V***(***G***) of ***vertices*** and a finite set of ***E***(***G***) of ***edges***.

* Each edge is associated with a set consisting of either one or two vertices; the ***endpoints***.

***** An edge with just one endpoint is called a ***loop***.

***** Two distinct edges with the same set of endpoints are said to be ***parallel***.

***** Two vertices that are connected by an edge are called ***adjacent***.

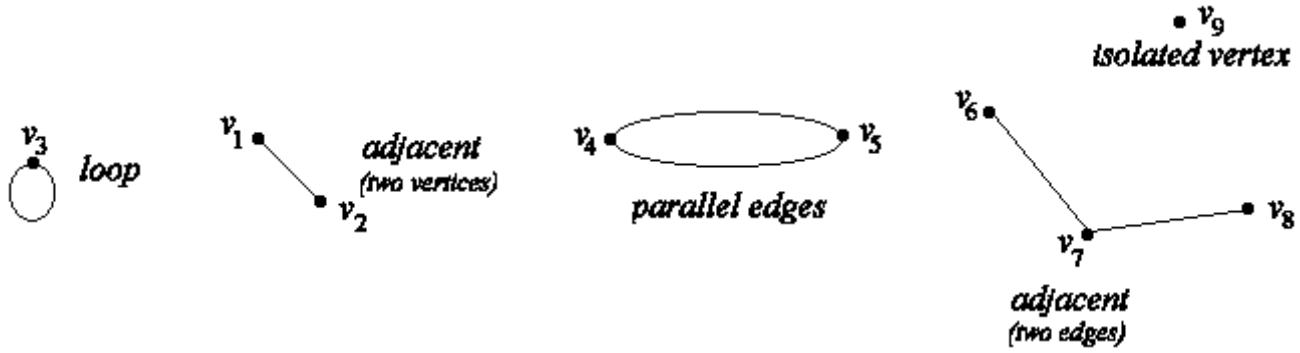
***** An edge is said to be ***incident on*** each of its endpoints.

***** Two edges incident on the same endpoint are ***adjacent***.

***** A vertex on which no edges are incident is ***isolated***.

***** A graph containing no vertices is called ***empty***; and a graph with vertices is ***nonempty***.

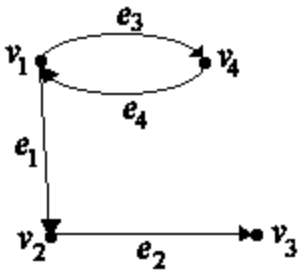
Graphs have pictorial representations in which the vertices are represented by dots and the edges by line segments, as shown by the following diagrams:



Directed Graph

A directed graph consists of a finite set ***V***(***G***) of vertices and a finite set ***D***(***G***) of directed edges, where each edge is associated with an *ordered* pair of vertices called its ***endpoints***. This is also known as a ***digraph.***

For example, consider the following diagram.



Here, the set of vertices $V(G) = \{v_1, v_2, v_3, v_4\}$ and the set of edges $D(G) = \{ e_1, e_2, e_3, e_4\}$

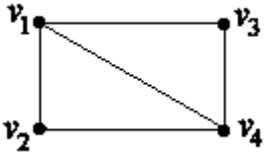
The edge-endpoint function is described by the following table:

Edges	Endpoints
e_1	$\{v_1, v_2\}$
e_2	$\{v_2, v_3\}$
e_3	$\{v_1, v_4\}$
e_4	$\{v_4, v_1\}$

Simple Graphs

A simple graph ***G*** is an ordered pair (***V***, ***E***), where ***V*** is a finite nonempty set of nodes (**vertices**) and ***E*** is another finite set of **edges** of the graph. A simple graph does not contain loops or parallel edges.

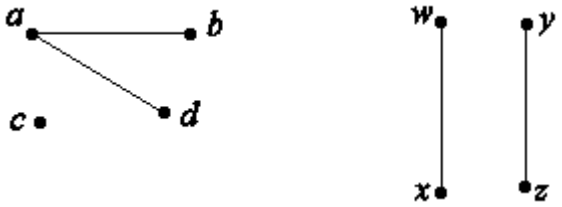
Example: Based on the graph ***G*** shown below,



$V = \{v_1, v_2, v_3, v_4\}$
$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_3, v_4\}\}$

$G = (V, E)$ is a simple graph.

Besides the above example, the diagrams shown below are also considered to be simple graphs.

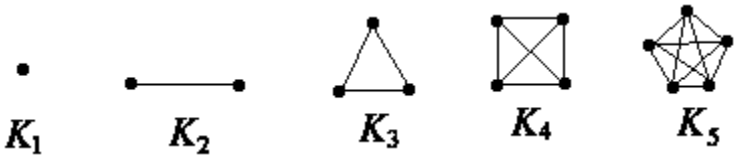


Complete Graphs

A simple graph is ***complete*** provided that any two vertices in the graph are adjacent. A complete graph with ***n*** vertices is called a complete graph *on* ***n*** *vertices*, and is denoted by K_n.

(***K*** is taken from the German word *komplett*, meaning "complete.")

The drawings below represent K_n for each ***n*** = 1, 2, 3, 4, 5.



Complete Bipartite Graphs

Bipartite graphs: A bipartite graph ***G*** is a simple graph whose vertex set can be partitioned into two mutually disjoint nonempty subsets $\mathbf{V_1}$ and $\mathbf{V_2}$ such that vertices in $\mathbf{V_1}$ may be connected to vertices in $\mathbf{V_2}$, but no vertices in $\mathbf{V_1}$ are connected to other vertices in $\mathbf{V_1}$ and no vertices in $\mathbf{V_2}$ are connected to other vertices in $\mathbf{V_2}$. Consider the following example, the graph ***G*** in **diagram 1** can be redrawn as shown in **diagram 2**. In **diagram 2**, we can see that ***G*** is bipartite with mutually disjoint vertex sets $\mathbf{V_1} = \{v_1, v_3, v_4\}$ and $\mathbf{V_2} = \{v_2, v_5, v_6\}$.

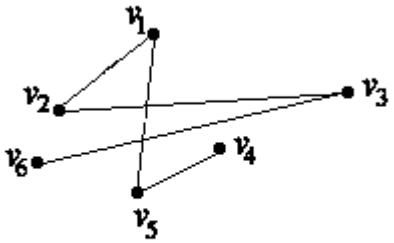


Diagram 1

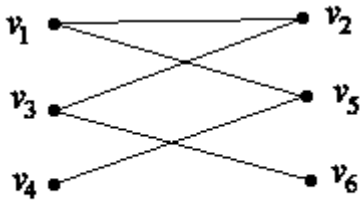


Diagram 2

A ***complete bipartite graph*** on (***m***, ***n***) vertices, denoted $K_{m,n}$, is a simple graph with vertices $v_1, v_2, ..., v_m$ and $w_1, w_2, ..., w_n$ that satisfies the following properties:

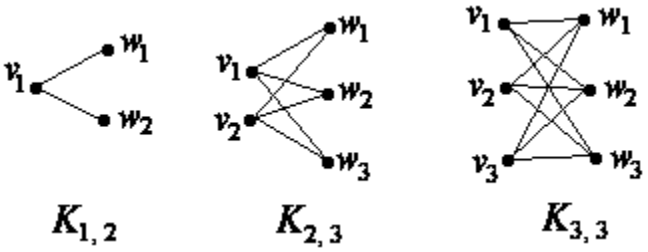
for all $i, j = 1, 2, ..., m$ and for all $k, l = 1, 2, ..., n$,

1. There is an edge from each vertex v_i to each vertex w_k;

2. There is no edge from any vertex v_i to any other vertex v_j;

3. There is no edge from any vertex w_k to any other vertex w_l.

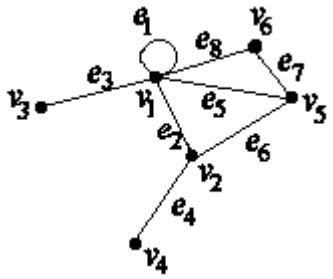
The bipartite graphs $K_{1,2}$, $K_{2,3}$ and $K_{3,3}$ are illustrated below.



Subgraphs

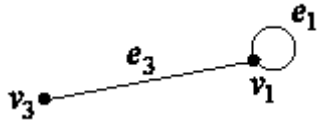
A graph ***H*** is a subgraph of ***G*** provided that the set of vertices in ***H*** is a subset of the set of vertices in ***G*** and the set of edges in ***H*** is a subset of the set of edges in ***G***. Moreover, every edge in ***H*** must have the same endpoints as in ***G***.

Example: List three nonempty subgraphs of the graph ***G*** shown below.

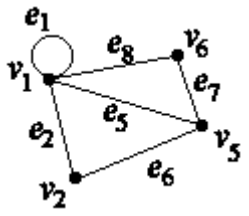


Solutions:

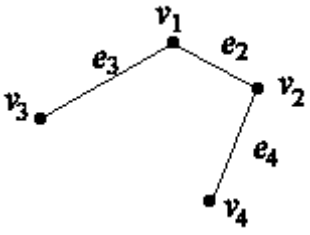
1.



2.



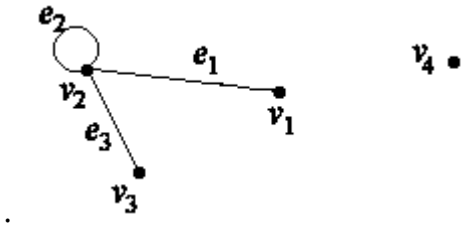
3.



The Degree of A Graph

Let ***G*** be a graph and ***v*** a vertex of ***G***. The degree of ***v***, denoted ***deg***(***v***), is the number of edges that are incident with ***v***, with an edge that is a loop counted twice. The total degree of ***G*** is the sum of the degrees of all the vertices of ***G***.

Example: Find the degree of each vertex of the graph ***G*** shown below and the total degree of ***G,*** then compare your answer to ours.



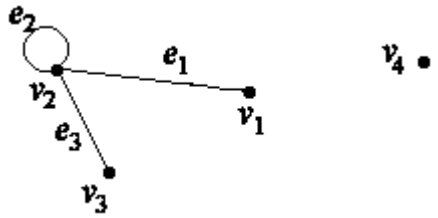
Other properties:

For any graph ***G***, the sum of the degrees of all the vertices of ***G*** equals twice the number of edges of ***G***. This means that if the vertices of ***G*** are v_1, v_2, ..., v_n, where ***n*** is a positive integer, then

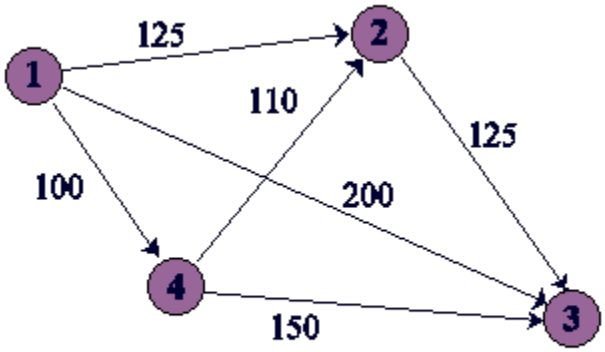
the total degree of ***G*** = $deg(v_1) + deg(v_2) + ... + deg(v_n)$

= 2 * (the number of edges of ***G***)

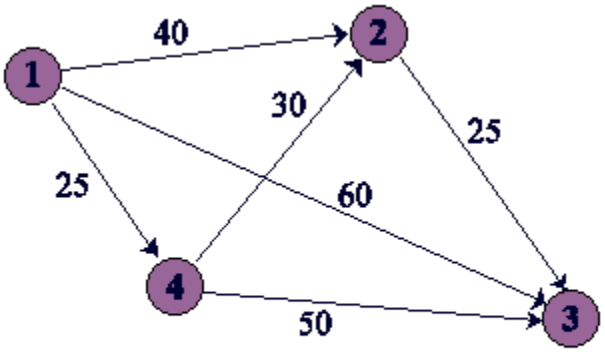
Looking at the previous example, there are three edges in the graph. Hence, the total degree of the graph = 2 * 3 = 6 which is same as the above answer.



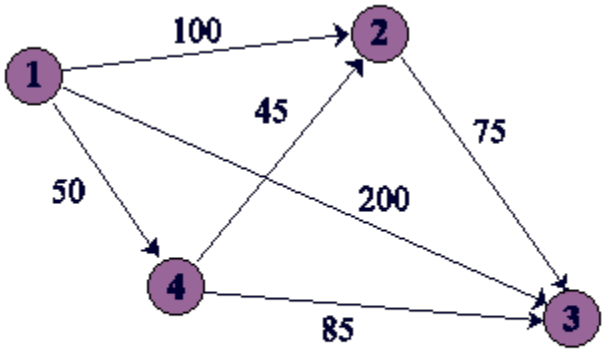
A ***weighted graph*** is a digraph which has values attached to the directed edges. These values represent the *cost* of travelling from one node to the next. The cost can be measured in many terms, depending upon the application. For instance, the 3 digraphs below all represent the same thing; a graph of airport terminals and flight lines between those terminals. The only difference is the meaning of the weights of the edges: Graph A represents the flight distance between terminals; graph B represents the average flight time in minutes; and graph C represents the dollar cost of a plane ticket between the hubs. Looking at graphs A and B, you can see that the relative *cost* of a direct flight from hub 1 to 3 is cheaper in terms of time and miles than the more roundabout route of travel from hub 1 to 2 to 3. However, in graph C, the ticket costs of a roundabout route is cheaper than the direct flight.



A) Miles



B) Travel time in Minutes



C) Ticket cost

Representations of graph

Following two are the most commonly used representations of graph.

- Adjacency Matrix
- Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

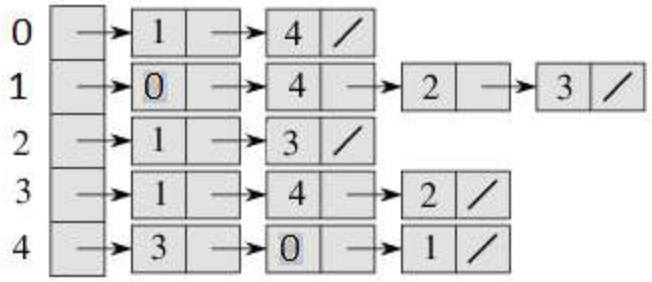
Adjacency Matrix:
Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency Matrix Representation of the above graph

Adjacency List:
An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



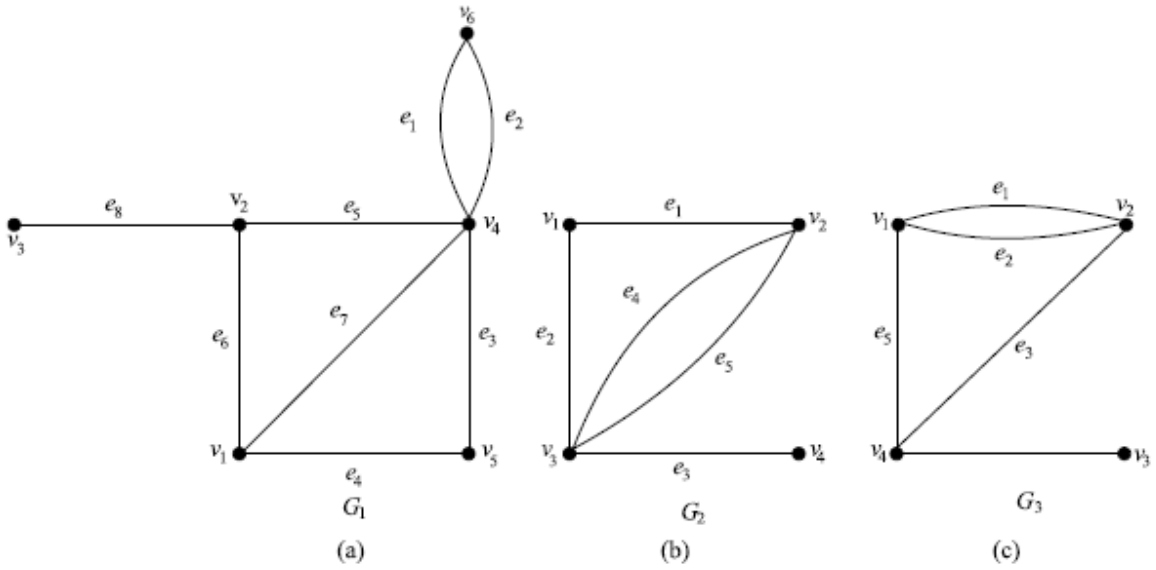
Adjacency List Representation of the above Graph

Incidence Matrix

Let G be a graph with n vertices, m edges and without self-loops. The incidence matrix A of G is an $n \times m$ matrix $A = [a_{ij}]$ whose n rows correspond to the n vertices and the m columns correspond to m edges such that

$$a_{ij} = \begin{cases} 1, & if\ jth\ edge\ m_j\ is\ incident\ on\ the\ ith\ vertex \\ 0, & otherwise. \end{cases}$$

It is also called *vertex-edge incidence matrix* and is denoted by $A(G)$.



The incidence matrix of G_1 is

$$A(G_1) = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{array}{c} \begin{array}{cccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}.$$

The incidence matrix of G_2 is

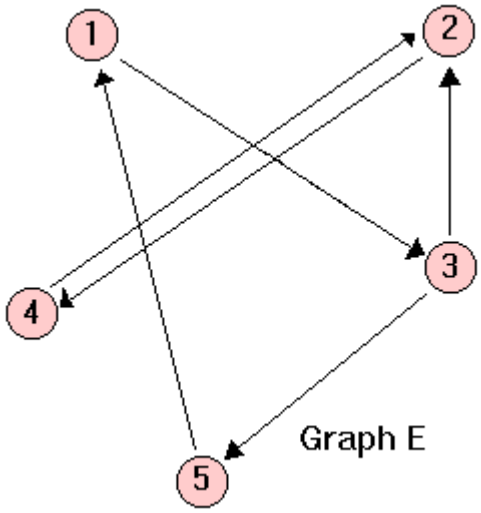
$$A(G_2) = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array}.$$

The incidence matrix of G_3 is

$$A(G_3) = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} \end{array}.$$

Warshall's Algorithm

In it's simplest form, ***Warshall's Algorithm*** is used to compute the existence of paths within a digraph using Boolean operators and matrices. Begin by creating an adjacency matrix **A** for Graph **E** as before, with one notable difference - instead of using weights (and you will have noticed by now that none are assigned to digraph **E**), we will use Boolean operators. That is to say, if there is a path, enter a 1 in matrix **A**, and enter 0 if no path exists.



Graph E

A	1	2	3	4	5
1	0	0	1	0	0
2	0	0	0	1	0
3	0	1	0	0	1
4	0	1	0	0	0
5	1	0	0	0	0

This matrix tells us whether or not there is a path ***p*** of length 1 between two adjacent nodes. Building upon matrix **A**, we will create a new matrix $\mathbf{A^1}$, for which we will choose one vertex to act as a *pivot* -- an intermediate point between 2 other vertices. Initially, we will chose vertex 1 as our pivot for $\mathbf{A^1}$. The value we are seeking is that of $p^{(1)}{}_{i\,j}$. For vertices v_i and v_j, $p^{(1)}{}_{i\,j}$ is one of the following:

> 1, if there exists an edge between vertices v_i and v_j, or if there is a path of length > 2 from v_i to v_1 and from v_1 to v_j; else
>
> 0, if there is no path.

First, realize that all paths of length 1 between vertices v_i and v_j are already established. We are searching now for any paths of length 2 which use vertex 1 as a pivot point. The only way vertex 1 can be a pivot is if a path already lies between some vertex v_i and 1, and between 1 and some vertex v_j.

A^1	1	2	3	4	5
1	0	0	1	0	0
2	0	0	0	1	0
3	0	1	0	0	1
4	0	1	0	0	0
5	1	0	1	0	0

MATRIX A^1

> Begin by scanning *column 1* of matrix **A**; the only v_i which connects to v_1 is vertex 5.
>
> Now scan *row 1*; the only path from v_1 to v_j is to vertex 3.
>
> Since we have established that a path of length 2 lies

A^2	1	2	3	4	5

between v_5 and v_3 , we update matrix $\mathbf{A}^1$ accordingly.

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

MATRIX $\mathbf{A}^2$

Next create matrix $\mathbf{A}^2$, using vertex 2 as the pivot point.

Begin by scanning *column 2* of matrix **A**; the v_i which connect to v_2 are vertices 3 and 4.

Now scan *row 2*; only 1 path from v_2 exists to v_j = vertex 4.

We have now established paths between the following vertices:

v_3 to v_4 and v_4 to v_4.

Newly added paths have been highlighted in gray. Notice that each new path created is being built upon previously existing paths.

$\mathbf{A}^3$	1	2	3	4	5
1	0	1	1	1	1
2	0	0	0	1	0
3	0	1	0	1	1
4	0	1	0	1	0
5	1	1	1	1	1

MATRIX $\mathbf{A}^3$

Matrix $\mathbf{A}^3$ will use vertex 3 as the pivot point.

As before, scan column 3 to see which vertices v_i connect to v_3 . In this case, vertices 1 and 5 have a path to 3.

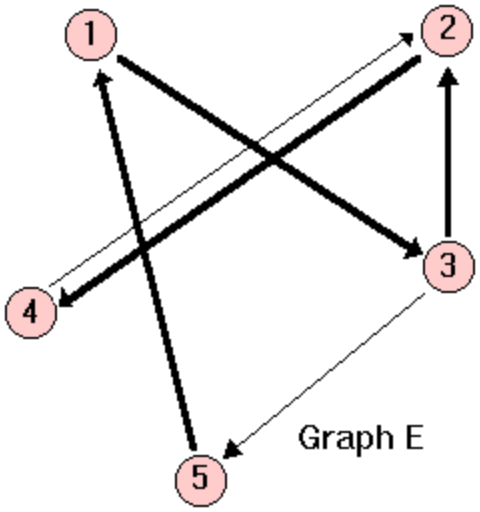
Now, scanning row 3, v_3 connects to vertices 2, 4, 5. We have now established paths between the following:

v_1 to v_2	and	v_5 to v_2
v_1 to v_4		v_5 to v_4
v_1 to v_5		v_5 to v_5

$\mathbf{A}^4$	1	2	3	4	5
1	0	1	1	1	1
2	0	1	0	1	0
3	0	1	0	1	1
4	0	1	0	1	0
5	1	1	1	1	1

Notice now that some paths have exceeded length 2. This is because the newly established paths are not using just 3 as a pivot point, but also the previous pivots points. For instance, vertex 5 has a path to 4 by travelling along the path of 5 → 1 → 3 → 2 → 4, shown below.

$\mathbf{A}^5$	1	2	3	4	5
1	1	1	1	1	1
2	0	1	0	1	0



3	1	1	1	1	1
4	0	1	0	1	0
5	1	1	1	1	1

If you haven't guessed by now, we will be creating 2 more adjacency matrices, $\mathbf{A}^4$ and $\mathbf{A}^5$. Generally speaking, if a graph has ***n*** vertices, it will require ***n*** matrices to produce $\mathbf{A}^n = \mathbf{P}^n$, where $\mathbf{P}^n$ is the path matrix. To produce $\mathbf{A}^4$ and $\mathbf{A}^5$, proceed as before.

MATRICES $\mathbf{A}^4$ and $\mathbf{A}^5$

For $\mathbf{A}^4$, first scan column 4. At this point, all vertices now have a path to vertex 4.

Scanning row 4, we see that 4 has a path only to vertex 2, indicating that all vertices have a path to 2. However, the only vertex which doesn't already have a path to vertex 2 is 2 itself, so we update the matrix accordingly.

Completing this process, scan column 5 to see that vertices 1, 3 and 5 all have paths to vertex 5. Scanning row 5 indicates that 5 has a path to all other vertices. Consequently, we add 1's to rows 1, 3 and 5 to reflect that vertices 1, 3 and 5 have paths to all other vertices.

This completes the path matrix for Graph E.

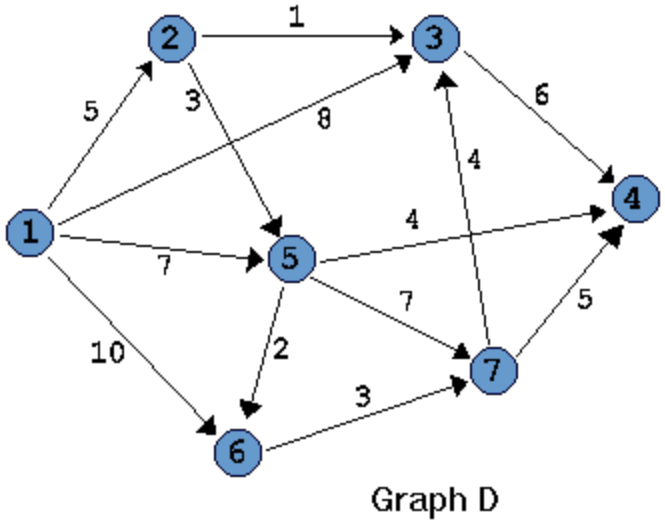
Warshall's Algorithm for computing a path matrix:

```
procedure Warshall (
A: BoolMatrix;                  /*Input, the adjacency matrix of a given graph*/
var P: BoolMatrix;              /*Output, the path matrix of the graph*/
n: integer                      /*Input, the size of the matrix (i.e., the number of
        );                      vertices)*/
```

```
int i, j, k;
begin
for i :=1 to n do                    /*Step 1: Copy adjacency matrix into path matrix*/
    for j := 1 to n do
        P[i, j] := A[i, j];
for k := 1 to n do                   /*Step 2: Allow vertex k as a pivot point*/
    for i := 1 to n do               /*Step 3: Process rows*/
        for j := 1 to n do           /*Step 4: Process columns*/
            P[i, j] := P[i, j] or (P[i, k] and    /*Step 5*/
P[k, j])
end;
```

Dijkstra's Algorithm

Suppose that we are given the following graph and are to find the shortest path between vertex 1 and all other vertices. We will begin by creating a set **V** which contains all vertices in graph **D; V** = {1, 2, 3, 4, 5, 6, 7}. Likewise, we can represent the shortest path between any 2 nodes as a set of vertices **W.** The approach we will take to solving this problem is to begin with **W** = {1}, and then we will progressively enlarge **W** one vertex at a time until **W** contains all vertices ***v*** in **V**. We will also create an array, **ShortestPath**[***s***], which will keep track of the minimum distance between vertex 1 and each vertex in **W**, and between vertex 1 and any vertex ***s*** ∈ **V-W** via a path ***p*** from the vertices in **W.** All vertices in path ***p*** will lie in **W** except for the last vertex ***s***, which is in **V-W**.



Graph D

At each step, the following will happen:

1. We will chose a vertex ***w*** to add to **W** from the set **V-W** (the difference of set **V** with respect to **W**) where there exists an edge ***e*** = (v_i, v_j), where vertex v_i ∈ **W**, and v_j ∈ **V-W,** and which has the minimum weight of all ***e*** *leaving* **W**.

2. We will update the array **ShortestPath**[***s***], so that at each step, the shortest path between 1 and all vertices will be recorded. The notation $\Delta(s)$ represents the shortest distance between the origin (vertex 1) and vertex ***s***.

```
void ShortestPath(void)
{

(Let MinDist be a variable which contains edge weights as values)
(and let Minimum(x,y) be a function which returns the lesser value of x and y.)

/*Let v1 ∈ V be the vertex at which ShortestPath begins.*/
/*Initialize W and ShortDist[x] as follows:*/

        W = {v1};
        ShortDist[v1] = 0;
        for (each x in V - {v1}) ShortDist[x] = T[v1][x];

/*Begin enlarging W 1 vertex at a time until W contains all vertices in V.*/

        while (W != V) {

                /*Find the vertex w ∈ V-W which has the shortest distance from v1.*/

                        MinDist = ∞;

                                for (each w ∈ V-W) {
                                    if(ShortDist[v] < MinDist) {
                                        MinDist = ShortDist[v];
                                        w = v;
                                    }
                                }

                        /*Add w to W*/

                                W = W ∪ {w};

                        /*Update the ShortDist array.*/

                                for (each x ∈ V-W) {
                                    ShortDist[x] =
                                        Minimum(ShortDist[x], ShortDist[w] + T[w][x]);
                                }

        }

}
```

Graph Traversal

Graph traversal is the problem of visiting all the nodes in a graph in a particular manner, updating and/or checking their values along the way.

Depth-first search (DFS)

A depth-first search (DFS) is an algorithm for traversing a finite graph. DFS visits the child nodes before visiting the sibling nodes; that is, it traverses the depth of any particular path before exploring its breadth. A stack (oftentimes the program's call stack via recursion) is generally used when implementing the algorithm.

The algorithm begins with a chosen "root" node; it then iteratively transitions from the current node to an adjacent, unvisited node, until it can no longer find an unexplored node to transition to from its current location. The algorithm then backtracks along previously visited nodes, until it finds a node connected to yet more uncharted territory. It will then proceed down the new path as it had before, backtracking as it encounters dead-ends, and ending only when the algorithm has backtracked past the original "root" node from the very first step.

Breadth-first search

A breadth-first search (BFS) is another technique for traversing a finite graph. BFS visits the sibling nodes before visiting the child nodes. Usually a queue is used in the search process. It's usually used to find the shortest path from a node to another.

Kruskal's Algorithm

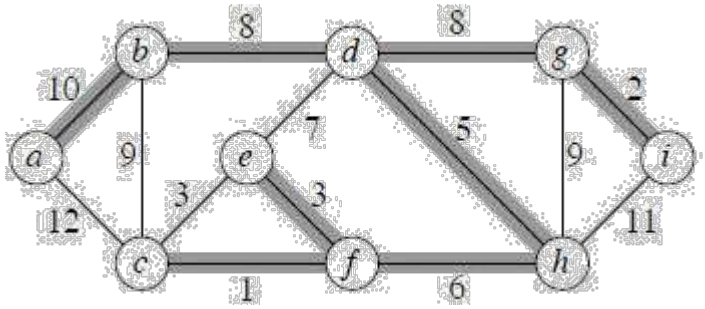
Kruskal's Algorithm, as described in CLRS, is directly based on the generic MST algorithm. It builds the MST in forest. Initially, each vertex is in its own tree in forest. Then, algorithm consider each edge in turn, order by increasing weight. If an edge (u, v) connects two different trees, then (u, v) is added to the set of edges of the MST, and two trees connected by an edge (u, v) are merged into a single tree on the other hand, if an edge (u, v) connects two vertices in the same tree, then edge (u, v) is discarded.

A little more formally, given a connected, undirected, weighted graph with a function $w : E \rightarrow R$.

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).
- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

Illustrative Examples

Lets run through the following graph quickly to see how Kruskal's algorithm works on it:



We get the shaded edges shown in the above figure.

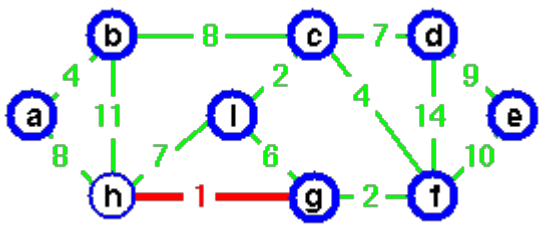
Edge (c, f) : safe
Edge (g, i) : safe
Edge (e, f) : safe
Edge (c, e) : reject
Edge (d, h) : safe
Edge (f, h) : safe
Edge (e, d) : reject
Edge (b, d) : safe
Edge (d, g) : safe
Edge (b, c) : reject
Edge (g, h) : reject
Edge (a, b) : safe

At this point, we have only one component, so all other edges will be rejected. [We could add a test to the main loop of KRUSKAL to stop once |V| − 1 edges have been added to A.]

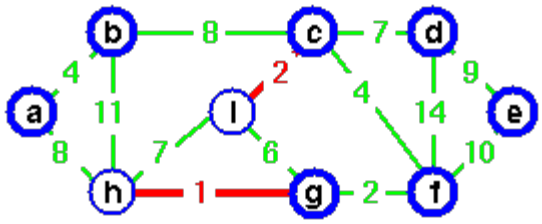
Note Carefully: Suppose we had examined (c, e) before (e, f). Then would have found (c, e) safe and would have rejected (e, f).

Example (CLRS) Step-by-Step Operation of Kurskal's Algorithm.

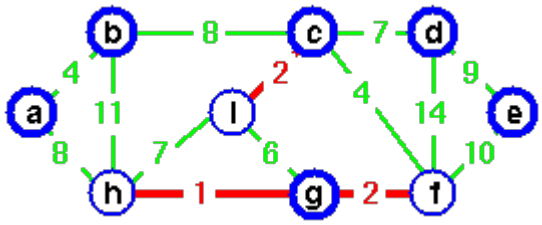
Step 1. In the graph, the Edge(*g*, *h*) is shortest. Either vertex *g* or vertex *h* could be representative. Lets choose vertex g arbitrarily.



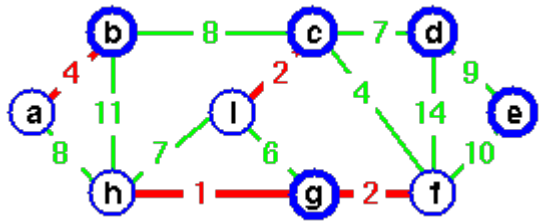
Step 2. The edge (*c*, *i*) creates the second tree. Choose vertex *c* as representative for second tree.



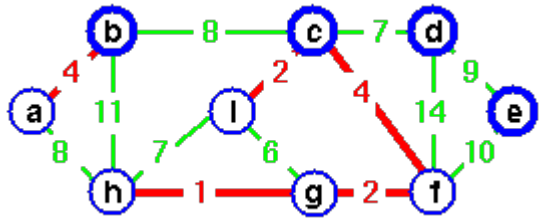
Step 3. Edge (g, g) is the next shortest edge. Add this edge and choose vertex g as representative.



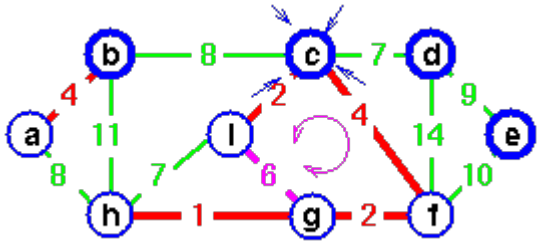
Step 4. Edge (a, b) creates a third tree.



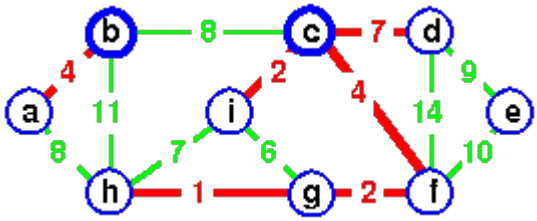
Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.



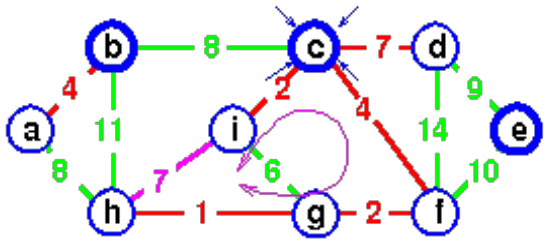
Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.



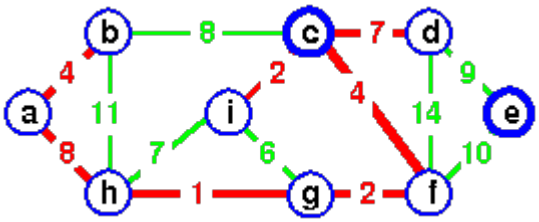
Step 7. Instead, add edge (c, d).



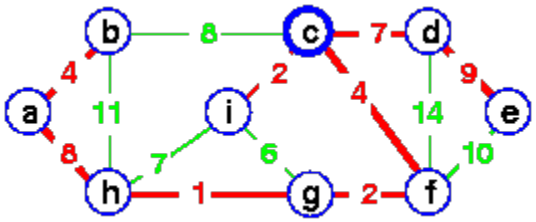
Step 8. If we add edge (h, i), edge(h, i) would make a cycle.



Step 9. Instead of adding edge (h, i) add edge (a, h).



Step 10. Again, if we add edge (b, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.



Shortest Path algorithm (Dijkstra's Algorithm)

Definition of Dijkstra's Shortest Path

1. To find the shortest path between points, the weight or length of a path is calculated as the sum of the weights of the edges in the path.

2. A path is a shortest path is there is no path from x to y with lower weight.

3. Dijkstra's algorithm finds the shortest path from x to y in order of increasing distance from x. That is, it chooses the first minimum edge, stores this value and adds the next minimum value from the next edge it selects.

4. It starts out at one vertex and branches out by selecting certain edges that lead to new vertices.

5. It is similar to the minimum spanning tree algorithm, in that it is "greedy", always choosing the closest edge in hopes of an optimal solution.

```
procedure sequential_dijkstra
begin
  d_s = 0
  d_i = ∞, for i ≠ s
  T = V
  for i = 0 to N-1
    find v_m ∈ T with minimum d_m
    for each edge (v_m, v_t) with v_t ∈ T
      if (d_t > d_m + length((v_m, v_t))) then d_t = d_m + length((v_m, v_t))
    endfor
    T = T − v_m
  endfor
end
```

Algorithm 3.2 Dijkstra's single-source shortest-path algorithm.

Chapter 9

Algorithms

Divide-and-Conquer Algorithm

Divide-and-conquer is a top-down technique for designing algorithms that consists of dividing the problem into smaller sub problems hoping that the solutions of the sub problems are easier to find and then composing the partial solutions into the solution of the original problem.

Little more formally, divide-and-conquer paradigm consists of following major phases:

- Breaking the problem into several sub-problems that are similar to the original problem but smaller in size,
- Solve the sub-problem recursively (successively and independently), and then
- Combine these solutions to sub problems to create a solution to the original problem.

Heuristic Algorithms

The term **heuristic** is used for algorithms which find solutions among all possible ones, but they do not guarantee that the best will be found, therefore they may be considered as approximately and not accurate algorithms. These algorithms, usually find a solution close to the best one and they find it fast and easily. Sometimes these algorithms can be accurate, that is they actually find the best solution, but the algorithm is still called heuristic until this best solution is proven to be the best. The method used from a heuristic algorithm is one of the known methods, such as greediness, but in order to be easy and fast the algorithm ignores or even suppresses some of the problem's demands.

Deterministic algorithm

In computer science, a **deterministic algorithm** is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states. Deterministic algorithms are by far the most studied and familiar kind of algorithm, as well as one of the most practical, since they can be run on real machines efficiently.

Formally, a deterministic algorithm computes a mathematical function; a function has a unique value for any given input, and the algorithm is a process that produces this particular value as output.

Deterministic algorithms can be defined in terms of a state machine: a *state* describes what a machine is doing at a particular instant in time. State machines pass in a discrete manner from one state to another. Just after we enter the input, the machine is in its *initial state* or *start state*. If the machine is deterministic, this means that from this point onwards, its current state determines what its next state will be; its course through the set of states is predetermined. Note that a machine can be deterministic and still never stop or finish, and therefore fail to deliver a result.

Examples of particular abstract machines which are deterministic include the deterministic Turing machine and deterministic finite automaton.

Nondeterministic algorithm

In computer science, a **nondeterministic algorithm** is an algorithm that can exhibit different behaviors on different runs, as opposed to a deterministic algorithm. There are several ways an algorithm may behave differently from run to run. A concurrent algorithm can perform differently on different runs due to a race condition. A probabilistic algorithm's behaviors depend on a random number generator. An algorithm that solves a problem in nondeterministic polynomial time can run in polynomial time or exponential time depending on the choices it makes during execution.

Serial Vs **parallel**
Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. This is a serial algorithm, as opposed to parallel algorithms, which take advantage of computer architectures to process several instructions at once. They divide the problem into sub-problems and pass them to several processors. Iterative algorithms are generally parallelizable. Sorting algorithms can be parallelized efficiently.

Big-O Notation

Big-O Notation measures the complexity of an algorithm or order of magnitude of the number of operations required to perform a function. In other words, how the number of processing steps increase as the number of items being processed increases. Processing may not increase in a constant manner. For instance if processing one item takes 2 seconds, processing 100 item does not necessarily take 200 seconds (2 * 100).

Big-O notation is a type of language or notation or terminology to discuss the the reality which is - algorithms need to process efficiently. Without knowing Big-O notation, an experienced programmer can look at an algorithm and understand that a step that can be moved to process outside of a loop will reduce the processing steps by the number of loops the algorithm has to perform less one. Instead of processing once for each item in the list, it can process that step only one time. For me, this underlying understanding of processing steps is more practical than reciting Big-O Notation, but we must succumb to industry standards when applying for jobs, and to communicate about complexity of an algorithm with other programmers. So I conform.

Common Big-O notation orders:

O(1) represents function that runs in constant time

The time to run an algorithm doesn't change based if the number of items being processed changes. Whether running 1 or 1000 items the time remains constant. This is rare. Looking up an element in

an array usually takes the same amount of time no matter how many items are in the array and would be said to run in constant time.

O(N) represents function that runs in linear time

Operations to run directly proportional to number of items processed. For instance if it take 3 minutes to process 1 item it will take 3 X 10 = 30 minutes to process 10 items.

O(N^2) represents a function that runs in quadratic time.

The equation for quadratic time is (N^2 - N) / 2. Or in other words 0+1+2+...+(N-1). In Big-O notation constants are dropped so we have N^2 - N. As the number of items increases, subtracting N from the result becomes a negligible difference so we can skip that and end up with N^2. So if you have 2 items a function that runs in O(N^2) roughly takes 4 processing steps and with 10 items takes 100 processing steps. An example would be inserting items being processed into an array in the first position and having to move every single item of the array each time to insert the new item. Functions running in quadratic time are typically not acceptable for interactive applications.

O(log N) and O(N log N) represent a functions that runs in logarithmic time.

The running time of an algorithm increases with the log of the number of items being processed. These generally mean that the algorithm deals with a data set that is partitioned into small groups of data as it is processed, like a balanced binary tree.

For instance if your asked to find the number I'm thinking of out of 100 you could ask, is it greater than or less than 50. I say greater. You say is it greater or less than 75. I say greater. You say is it greater or less than 87.5. I say greater...and continues until you get to the number I am thinking of which is 88. This is more efficient than saying, "Is it one? Is it two? Is it three?..." etc.