UNIT I

The C Language is developed for creating system applications that direct interacts to the hardware devices such as drivers, kernals etc.

C programming is considered as the base for other programming languages, that is why it is known as mother language.

It can be defined by following ways:

1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid level programming language

1) C as a mother language

C language is considered as the mother language of all the modern languages because **most of the compilers, JVMs, Kernals etc. are written in C language** and most of languages follows c syntax e.g. C++, Java etc.

It provides the core concepts like array, functions, file handling etc. that is being used in many languages like C++, java, C# etc.

2) C as a system programming language

A system programming language is used to create system softwares. C language is a system programming language because it **can be used to do low level programming (e.g. driver and kernel)**. It is generally used to create hardware devices, OS, drivers, kernels etc. For example, linux kernel is written in C.

It can?t be used in internet programming like java, .net, php etc.

3) C as a procedural language

A procedure is known as function, method, routine, subroutine etc. A procedural language **specifies a series of steps or procedures for the program to solve the problem**.

A procedural language breaks the program into functions, data structures etc.

C is a procedural language. In C, variables and function prototypes must be declared before being used.

4) C as a structured programming language

A structured programming language is a subset of procedural language. **Structure means to break a program into parts or blocks** so that it may be easy to understand.

In C language, we break the program into parts using functions. It makes the program easier to understand and modify.

5) C as a mid-level programming language

C is considered as a middle level language because it **supports the feature of both low-level and high level language**. C language program is converted into assembly code, supports pointer arithmetic (low level), but it is machine independent (feature of high level).

Low level language is specific to one machine i.e. machine dependent. It is machine dependent, fast to run. But it is not easy to understand.

High Level language is not specific to one machine i.e. machine independent. It is easy to understand.

C++

What is C++

C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming.

C++ is a middle-level language, as it encapsulates both high and low level language features.

C++ is an object-oriented programming language. It is an extension to C programming.

Object-Oriented Programming (OOPs)

C++ supports the object-oriented programming, the four major pillar of object oriented programming used in C++ are:

1. Inheritance
2. Polymorphism
3. Encapsulation
4. Abstraction

Standard Libraries

Standard C++ programming is divided into three important parts:

- The core library includes the data types, variables and literals, etc.
- The standard library includes the set of functions manipulating strings, files, etc.
- The Standard Template Library (STL) includes the set of methods manipulating a data structure.

Usage of C++

By the help of C++ programming language, we can develop different types of secured and robust applications:

- Window application
- Client-Server application
- Device drivers
- Embedded firmware etc

C vs C++

No.	C	C++
1)	C follows the **procedural style programming.**	C++ is multi-paradigm. It supports both **procedural and object oriented.**
2)	Data is less secured in C.	In C++, you can use modifiers for class members to make it inaccessible for outside users.
3)	C follows the **top-down approach.**	C++ follows the **bottom-up approach.**
4)	C does not support function overloading.	C++ supports function overloading.
5)	In C, you can't use functions in structure.	In C++, you can use functions in structure.
6)	C does not support reference variables.	C++ supports reference variables.
7)	In C, **scanf() and printf()** are mainly used for input/output.	C++ mainly uses stream **cin and cout** to perform input and output operations.

8)	Operator overloading is not possible in C.	Operator overloading is possible in C++.
9)	C programs are divided into **procedures and modules**	C++ programs are divided into **functions and classes.**
10)	C does not provide the feature of namespace.	C++ supports the feature of namespace.
11)	Exception handling is not easy in C. It has to perform using other functions.	C++ provides exception handling using Try and Catch block.

C++ history

History of C++ language is interesting to know. Here we are going to discuss brief history of C++ language.

C++ programming language was developed in 1980 by Bjarne Stroustrup at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A.

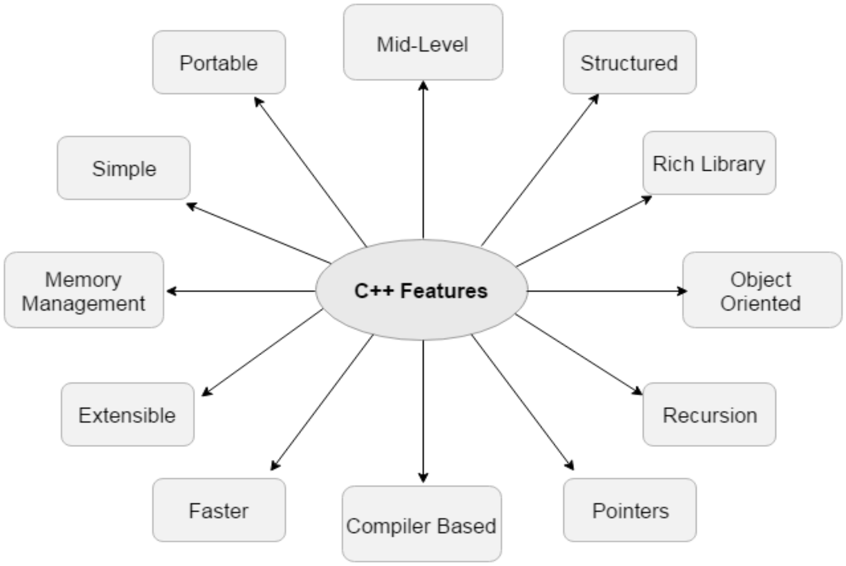
Bjarne Stroustrup is known as the **founder of C++ language.**

It was develop for adding a feature of **OOP (Object Oriented Programming)** in C without significantly changing the C component.

C++ programming is "relative" (called a superset) of C, it means any valid C program is also a valid C++ program.

C++ Features

C++ is object oriented programming language. It provides a lot of **features** that are given below.



1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. Structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible
11. Object Oriented
12. Compiler based

1) Simple

C++ is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Machine Independent or Portable

Unlike assembly language, c programs can be executed in many machines with little bit or no change. But it is not platform-independent.

3) Mid-level programming language

C++ is also used to do low level programming. It is used to develop system applications such as kernel, driver etc. It also supports the feature of high level language. That is why it is known as mid-level language.

4) Structured programming language

C++ is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

5) Rich Library

C++ provides a lot of inbuilt functions that makes the development fast.

6) Memory Management

It supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory at any time by calling the free() function.

7) Speed

The compilation and execution time of C++ language is fast.

8) Pointer

C++ provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array etc.

9) Recursion

In C++, we can call the function within the function. It provides code reusability for every function.

10) Extensible

C++ language is extensible because it can easily adopt new features.

11) Object Oriented

C++ is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

12) Compiler based

C++ is a compiler based programming language, it means without compilation no C++ program can be executed. First we need to compile our program using compiler and then we can execute our program.

Input and output in C++

C++ Basic Input / Output

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as output operation.

If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as input operation.

I/O Library Header Files

Let us see the common header files used in C++ programming are:

Header File	**Function and Description**
<iostream>	It is used to define the cout, cin and cerr objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.
<iomanip>	It is used to declare services useful for performing formatted I/O, such as setprecision and setw.
<fstream>	It is used to declare services for user-controlled file processing.

Standard output stream (cout)

The cout is a predefined object of ostream class. It is connected with the standard output device, which is usually a display screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console

Let's see the simple example of standard output stream (cout):

```
#include <iostream>

using namespace std;

int main( ) {

char ary[] = "Welcome to C++ tutorial";

cout << "Value of ary is: " << ary << endl;

}
```

Output:

```
Value of ary is: Welcome to C++ tutorial
```

Standard input stream (cin)

The **cin** is a predefined object of **istream** class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

Let's see the simple example of standard input stream (cin):

```
 #include <iostream>
 using namespace std;
 int main( ) {
 int age;
 cout << "Enter your age: ";
 cin >> age;
 cout << "Your age is: " << age << endl;
 }
```

Output:

```
Enter your age: 22
Your age is: 22
```

Standard end line (endl)

The **endl** is a predefined object of **ostream** class. It is used to insert a new line characters and flushes the stream.

Let's see the simple example of standard end line (endl):

```
    #include <iostream>
```

```
using namespace std;
int main( ) {
cout << "C++ Tutorial";
cout << " Javatpoint"<<endl;
cout << "End of line"<<endl;
}
```

Output:

```
C++ Tutorial Javatpoint
End of line
```

C++ Functions

The function in C++ language is also known as procedure or subroutine in other programming languages.

To perform any task, we can create function. A function can be called many times. It provides modularity and code reusability.

Advantage of functions in C

There are many advantages of functions.

1) Code Reusability

By creating functions in C++, you can call it many times. So we don't need to write the same code again and again.

2) Code optimization

It makes the code optimized, we don't need to write much code.

Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions

There are two types of functions in C programming:

1. Library Functions: are the functions which are declared in the C++ header files such as ceil(x), cos(x), exp(x), etc.

2. User-defined functions: are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.

Declaration of a function

The syntax of creating function in C++ language is given below:

```
return_type function_name(data_type parameter...)
{
//code to be executed
}
```

C++ Function Example

Let's see the simple example of C++ function.

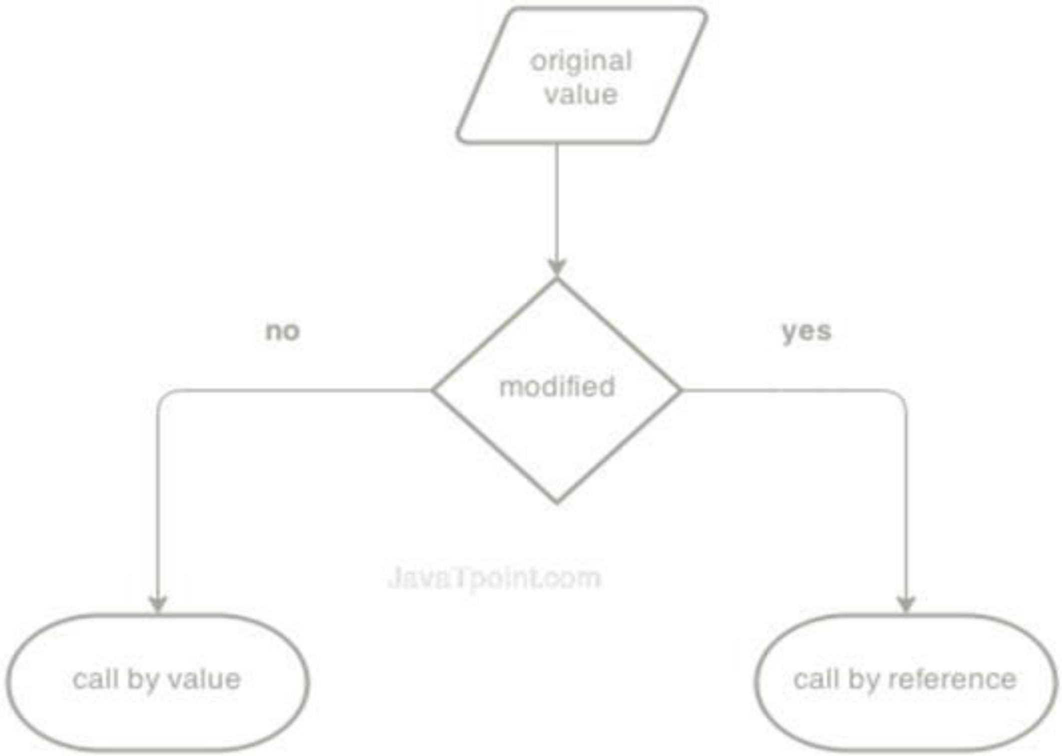
```
#include <iostream>
using namespace std;
void func() {
static int i=0; //static variable
int j=0; //local variable
i++;
j++;
cout<<"i=" << i<<" and j=" <<j<<endl;
}
int main()
{
func();
func();
func();
}
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

Call by value and call by reference in C++

There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.



Let's understand call by value and call by reference in C++ language one by one.

Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Let's try to understand the concept of call by value in C++ language by the example given below:

```
#include <iostream>
using namespace std;
void change(int data);
int main()
{
int data = 3;
change(data);
cout << "Value of the data is: " << data<< endl;
return 0;
}
void change(int data)
```

```
{
data = 5;
}
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

In call by reference, original value is modified because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Let's try to understand the concept of call by reference in C++ language by the example given below:

```
#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
int swap;
swap=*x;
*x=*y;
*y=swap;
}
int main()
{
int x=500, y=100;
swap(&x, &y);  // passing value to function
cout<<"Value of x is: "<<x<<endl;
cout<<"Value of y is: "<<y<<endl;
return 0;
}
```

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

C++ Overloading (Function and Operator)

If we create two or more members having same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

- **Function overloading**
- **Operators overloading**

C++ Function Overloading

Having two or more function with same name but different in parameters, is known as function overloading in C++.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for same action.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

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

```
class Cal {
public:
static int add(int a,int b){
return a + b;
}
static int add(int a, int b, int c)
{
return a + b + c;
}
};
int main(void) {
Cal C;
cout<<C.add(10, 20)<<endl;
cout<<C.add(12, 20, 23);
return 0;
}
```

Output:

```
30
55
```

C++ Operators Overloading

Operator overloading is used to overload or redefine most of the operators available in C++. It is used to perform operation on user define data type.

The advantage of Operators overloading is to perform different operations on the same operand.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

```
#include <iostream>
using namespace std;
class Test
{
private:
int num;
public:
Test(): num(8){}
void operator ++()
{
num = num+2;
}
void Print() {
cout<<"The Count is: "<<num;
}
};
int main()
{
Test tt;
++tt;  // calling of a function "void operator ++()"
tt.Print();
return 0;
}
```

Output:

```
The Count is: 10
```

Function templates

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>

myType GetMax (myType a, myType b) {

return (a>b?a:b);

}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

function_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;

GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:

```
// function template

#include <iostream>

using namespace std;



template <class T>

T GetMax (T a, T b) {

T result;

result = (a>b)? a : b;

return (result);

}



int main () {
```

```
6

10
```

```
int i=5, j=6, k;

long l=10, m=5, n;

k=GetMax<int>(i,j);

n=GetMax<long>(l,m);

cout << k << endl;

cout << n << endl;

return 0;

}
```

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type T is used within the GetMax() template function even to declare new objects of that type:

```
T result;
```

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:

```
int i,j;

GetMax (i,j);
```

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:

```
// function template II
#include <iostream>
using namespace std;


template <class T>
T GetMax (T a, T b) {
return (a>b?a:b);
}


int main () {
int i=5, j=6, k;
long l=10, m=5, n;
k=GetMax(i,j);
n=GetMax(l,m);
cout << k << endl;
cout << n << endl;
return 0;
}
```

```
6
10
```

Notice how in this case, we called our function template GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i;
long l;
k = GetMax (i,l);
```

This would not be correct, since our GetMax function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U>
T GetMin (T a, U b) {
return (a<b?a:b);
}
```

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int i,j;
long l;
i = GetMin<int,long> (j,l);
```

or simply:

```
i = GetMin (j,l);
```

even though j and l have different types, since the compiler can determine the appropriate instantiation anyway.

Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair {
T values [2];
public:

mypair (T first, T second)

{

values[0]=first; values[1]=second;

}

};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:

```
// class templates

#include <iostream>

using namespace std;


template <class T>

class mypair {

T a, b;

public:

mypair (T first, T second)
```

```
100
```

```
{a=first; b=second;}
T getmax ();
};


template <class T>
T mypair<T>::getmax ()
{
T retval;
retval = a>b? a : b;
return retval;
}


int main () {
mypair <int> myobject (100, 75);
cout << myobject.getmax();
return 0;
}
```

Notice the syntax of the definition of member function getmax:

```
template <class T>
T mypair<T>::getmax ()
```

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second Trefers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Template specialization

If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template.

For example, let's suppose that we have a very simple class called mycontainer that can store one element of any type and that it has just one member function called increase, which increases its value. But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function member uppercase, so we decide to declare a class template specialization for that type:

```
// template specialization
#include <iostream>
using namespace std;


// class template:
template <class T>
class mycontainer {
T element;
public:
mycontainer (T arg) {element=arg;}
T increase () {return ++element;}
};


// class template specialization:
template <>
class mycontainer <char> {
char element;
public:
mycontainer (char arg) {element=arg;}
char uppercase ()
{
```

```
8
J
```

```
if ((element>='a')&&(element<='z'))
element+='A'-'a';
return element;
}
};


int main () {
mycontainer<int> myint (7);
mycontainer<char> mychar ('j');
cout << myint.increase() << endl;
cout << mychar.uppercase() << endl;
return 0;
}
```

This is the syntax used in the class template specialization:

```
template <> class mycontainer <char> { ... };
```

First of all, notice that we precede the class template name with an empty template<> parameter list. This is to explicitly declare it as a template specialization.

But more important than this prefix, is the <char> specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (char). Notice the differences between the generic class template and the specialization:

```
template <class T> class mycontainer { ... };
template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization.

When we declare specializations for a template class, we must also define all its members, even those exactly equal to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

Non-type parameters for templates

Besides the template arguments that are preceded by the class or typename keywords , which represent types, templates can also have regular typed parameters, similar to those found in functions. As an example, have a look at this class template that is used to contain sequences of elements:

```
// sequence template
#include <iostream>
using namespace std;

template <class T, int N>
class mysequence {
T memblock [N];
public:
void setmember (int x, T value);
T getmember (int x);
};

template <class T, int N>
void mysequence<T,N>::setmember (int x, T value) {
memblock[x]=value;
}

template <class T, int N>
T mysequence<T,N>::getmember (int x) {
return memblock[x];
}
```

```
100
3.1416
```

```
int main () {
mysequence <int,5> myints;
mysequence <double,5> myfloats;
myints.setmember (0,100);
myfloats.setmember (3,3.1416);
cout << myints.getmember(0) << '\n';
cout << myfloats.getmember(3) << '\n';
return 0;
}
```

It is also possible to set default values or types for class template parameters. For example, if the previous class template definition had been:

```
template <class T=char, int N=10> class mysequence {..};
```

We could create objects using the default template parameters by declaring:

```
mysequence<> myseq;
```

Which would be equivalent to:

```
mysequence<char,10> myseq;
```

C++ Exception Handling

Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Advantage

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

C++ Exception Classes

In C++ standard exceptions are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:

All the exception classes in C++ are derived from std::exception class. Let's see the list of C++ common exception classes.

Exception	**Description**
std::exception	It is an exception and parent class of all standard C++ exceptions.
std::logic_failure	It is an exception that can be detected by reading a code.
std::runtime_error	It is an exception that cannot be detected by reading a code.
std::bad_exception	It is used to handle the unexpected exceptions in a c++ program.
std::bad_cast	This exception is generally be thrown by **dynamic_cast.**
std::bad_typeid	This exception is generally be thrown by **typeid.**
std::bad_alloc	This exception is generally be thrown by **new.**

C++ Exception Handling Keywords

In C++, we use 3 keywords to perform exception handling:

- try
- catch, and
- throw

C++ try/catch

In C++ programming, exception handling is performed using try/catch statement. The C++ **try block** is used to place the code that may occur exception. The **catch block** is used to handle the exception.

C++ example without try/catch

```
#include <iostream>
using namespace std;
float division(int x, int y) {
return (x/y);
}
int main () {
int i = 50;
int j = 0;
float k = 0;
k = division(i, j);
cout << k << endl;
return 0;
}
```

Output:

```
Floating point exception (core dumped)
```

C++ try/catch example

```
#include <iostream>
using namespace std;
float division(int x, int y) {
if( y == 0 ) {
throw "Attempted to divide by zero!";
}
return (x/y);
}
int main () {
int i = 25;
int j = 0;
float k = 0;
try {
k = division(i, j);
cout << k << endl;
}catch (const char* e) {
cerr << e << endl;
}
return 0;
}
```

Output:

```
Attempted to divide by zero!
```

C++ User-Defined Exceptions

The new exception can be defined by overriding and inheriting **exception** class functionality.

C++ user-defined exception example

Let's see the simple example of user-defined exception in which **std::exception** class is used to define the exception.

```
#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception{
public:
const char * what() const throw()
{
return "Attempted to divide by zero!\n";
}
};
int main()
{
try
{
int x, y;
cout << "Enter the two numbers : \n";
cin >> x >> y;
if (y == 0)
{
MyException z;
throw z;
}
else
{
cout << "x / y = " << x/y << endl;
}
}
catch(exception& e)
{
cout << e.what();
}
}
```

Output:

```
Enter the two numbers :
10
2
x / y = 5
```

Output:

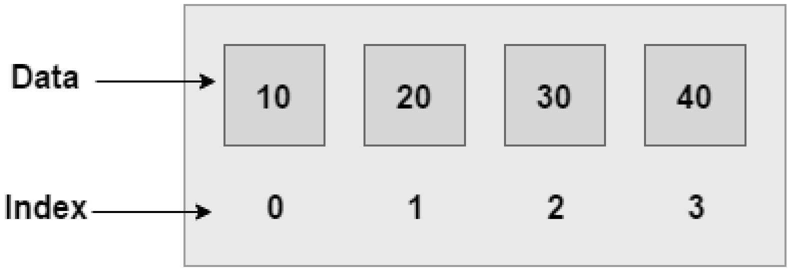
```
Enter the two numbers :
10
0
Attempted to divide by zero!
-->
```

Note: In above example what() is a public method provided by the exception class. It is used to return the cause of an exception.

C++ Arrays

Like other programming languages, array in C++ is a group of similar types of elements that have contiguous memory location.

In C++ **std::array** is a container that encapsulates fixed size arrays. In C++, array index starts from 0. We can store only fixed set of elements in C++ array.



Advantages of C++ Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

Disadvantages of C++ Array

- Fixed size

C++ Array Types

There are 2 types of arrays in C++ programming:

1. Single Dimensional Array
2. Double Dimensional Array
3. Multidimensional Array

C++ Single Dimensional Array

Let's see a simple example of C++ array, where we are going to create, initialize and traverse array.

```
#include <iostream>
using namespace std;
int main()
{
int arr[5]={10, 0, 20, 0, 30};  //creating and initializing array
//traversing array
for (int i = 0; i < 5; i++)
{
cout<<arr[i]<<"\n";
}
}
```

Output:/p>

```
10
0
20
0
30
```

C++ Array Example: Traversal using foreach loop

We can also traverse the array elements using foreach loop. It returns array element one by one.

```
#include <iostream>
using namespace std;
int main()
{
int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
//traversing array
for (int i: arr)
{
cout<<i<<"\n";
}
}
```

Output:

```
10
20
30
40
50
```

C++ Passing Array to Function

In C++, to reuse the array logic, we can create function. To pass array to function in C++, we need to provide only array name.

```
functionname(arrayname); //passing array to function
```

C++ Passing Array to Function Example: print array elements

Let's see an example of C++ function which prints the array elements.

```
#include <iostream>
using namespace std;
void printArray(int arr[5]);
int main()
{
int arr1[5] = { 10, 20, 30, 40, 50 };
int arr2[5] = { 5, 15, 25, 35, 45 };
printArray(arr1); //passing array to function
printArray(arr2);
}
void printArray(int arr[5])
{
```

```
cout << "Printing array elements:"<< endl;
for (int i = 0; i < 5; i++)
{
cout<<arr[i]<<"\n";
}
}
```

Output:

```
Printing array elements:
10
20
30
40
50
Printing array elements:
5
15
25
35
45
```

C++ Multidimensional Arrays

The multidimensional array is also known as rectangular arrays in C++. It can be two dimensional or three dimensional. The data is stored in tabular form (row ∗ column) which is also known as matrix.

C++ Multidimensional Array Example

Let's see a simple example of multidimensional array in C++ which declares, initializes and traverse two dimensional arrays.

```
#include <iostream>
using namespace std;
int main()
{
int test[3][3];  //declaration of 2D array
test[0][0]=5;  //initialization
test[0][1]=10;
test[1][1]=15;
test[1][2]=20;
test[2][0]=30;
test[2][2]=10;
//traversal
for(int i = 0; i < 3; ++i)
{
```

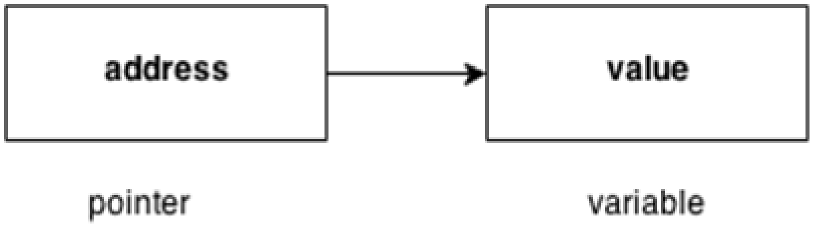
```
for(int j = 0; j < 3; ++j)
{
cout<< test[i][j]<<" ";
}
cout<<"\n"; //new line at each row
}
return 0;
}
```

Output:

```
5 10 0
0 15 20
30 0 10
```

C++ Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.



Advantage of pointer

1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.

2) We can return multiple values from function using pointer.

3) It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
∗ (asterisk sign)	Indirection operator	Access the value of an address.

Declaring a pointer

The pointer in C++ language can be declared using ∗ (asterisk symbol).

```
int ∗   a; //pointer to int
char ∗  c; //pointer to char
```

Pointer Example

Let's see the simple example of using pointers printing the address and value.

```
#include <iostream>
using namespace std;
int main()
{
int number=30;
int ∗   p;
p=&number;//stores the address of number variable
cout<<"Address of number variable is:"<<&number<<endl;
cout<<"Address of p variable is:"<<p<<endl;
cout<<"Value of p variable is:"<<*p<<endl;
return 0;
}
```

Output

```
Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30
```

Pointer Program to swap 2 numbers without using 3rd variable

```
#include <iostream>
using namespace std;
int main()
{
int a=20,b=10,∗p1=&a,∗p2=&b;
cout<<"Before swap: ∗p1="<<∗p1<<" ∗p2="<<∗p2<<endl;
∗p1=∗p1+∗p2;
∗p2=∗p1-∗p2;
∗p1=∗p1-∗p2;
cout<<"After swap: ∗p1="<<∗p1<<" ∗p2="<<∗p2<<endl;
return 0;
}
```

Output

```
Before swap: ∗p1=20 ∗p2=10
After swap: ∗p1=10 ∗p2=20
```

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts:

- **The stack:** All variables declared inside the function will take up memory from the stack.
- **The heap:** This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory previously allocated by new operator.

The new and delete operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

```
new data-type;
```

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new** operator with the following statements:

```
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
```

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below:

```
double* pvalue  = NULL;
if( !(pvalue  = new double )) {
cout << "Error: out of memory." <<endl;
exit(1);
}
```

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the delete operator as follows:

```
delete pvalue;        // Release memory pointed to by pvalue
```

Let us put above concepts and form the following example to show how new and delete work:

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

int main () {
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
*pvalue = 29494.99;     // Store value at allocated address

cout << "Value of pvalue : " << *pvalue << endl;


delete pvalue;          // free up the memory.


return 0;

}
```

If we compile and run above code, this would produce the following result:

```
Value of pvalue : 29495
```

Dynamic Memory Allocation for Arrays

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

```
char* pvalue  = NULL;   // Pointer initialized with null

pvalue  = new char[20]; // Request memory for the variable
```

To remove the array that we have just created the statement would look like this:

```
delete [] pvalue;        // Delete array pointed to by pvalue
```

Following is the syntax of new operator for a multi-dimensional array as follows:

```
int ROW = 2;

int COL = 3;

double **pvalue  = new double* [ROW]; // Allocate memory for rows


// Now allocate memory for columns

for(int i = 0; i < COL; i++) {
```

```
pvalue[i] = new double[COL];
}
```

The syntax to release the memory for multi-dimensional will be as follows:

```
for(int i = 0; i < ROW; i++) {
delete[] pvalue[i];
}
delete [] pvalue;
```

Dynamic Memory Allocation for Objects

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept:

```
#include <iostream>
using namespace std;
class Box {
public:
Box() {
cout << "Constructor called!" <<endl;
}
~Box() {
cout << "Destructor called!" <<endl;
}
};
int main( ) {
Box* myBoxArray = new Box[4];
delete [] myBoxArray; // Delete array
return 0;

}
```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result:

```
Constructor called!

Constructor called!

Constructor called!

Constructor called!

Destructor called!

Destructor called!

Destructor called!

Destructor called!
```

C++ OOPs Concepts

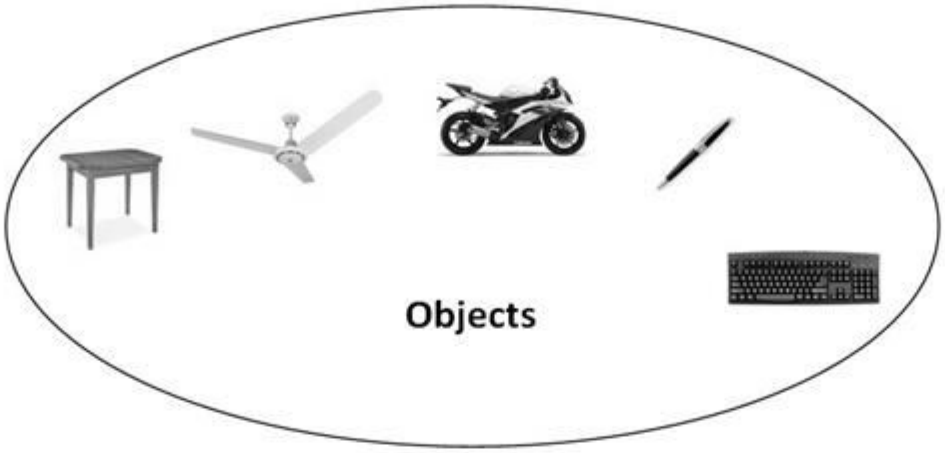
The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.

Object Oriented Programming is a paradigm that provides many concepts such as **inheritance, data binding, polymorphism etc.**

The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:



- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In C++, we use Function overloading and Function overriding to achieve polymorphism.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

C++ Object and Class

Since C++ is an object-oriented language, program is designed using objects and classes in C++.

C++ Object

In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object of student class using s1 as the reference variable.

```
Student s1;  //creating an object of Student
```

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.

C++ Class

In C++, object is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

Let's see an example of C++ class that has three fields only.

```
class Student
{
public:
int id;  //field or data member
float salary; //field or data member
String name;//field or data member
}
```

C++ Object and Class Example

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

```
#include <iostream>
using namespace std;
class Student {
public:
int id;//data member (also instance variable)
string name;//data member(also instance variable)
};
int main() {
Student s1; //creating an object of Student
s1.id = 201;
s1.name = "Sonoo Jaiswal";
cout<<s1.id<<endl;
cout<<s1.name<<endl;
return 0;
}
```

Output:

```
201
Sonoo Jaiswal
```

C++ Class Example: Initialize and Display data through method

Let's see another example of C++ class where we are initializing and displaying object through method.

```
#include <iostream>
using namespace std;
class Student {
public:
int id;//data member (also instance variable)
string name;//data member(also instance variable)
void insert(int i, string n)
{
id = i;
name = n;
}
void display()
{
cout<<id<<"  "<<name<<endl;
}
};
int main(void) {
Student s1; //creating an object of Student
Student s2; //creating an object of Student
s1.insert(201, "Sonoo");
s2.insert(202, "Nakul");
s1.display();
s2.display();
return 0;
}
```

Output:

```
201  Sonoo
202  Nakul
```

C++ Class Example: Store and Display Employee Information

Let's see another example of C++ class where we are storing and displaying employee information using method.

```
#include <iostream>
using namespace std;
class Employee {
public:
int id;//data member (also instance variable)
string name;//data member(also instance variable)
float salary;
void insert(int i, string n, float s)
{
id = i;
name = n;
salary = s;
}
void display()
{
cout<<id<<"  "<<name<<"  "<<salary<<endl;
}
};

int main(void) {

Employee e1; //creating an object of Employee

Employee e2; //creating an object of Employee

e1.insert(201, "Sonoo",990000);

e2.insert(202, "Nakul", 29000);

e1.display();

e2.display();

return 0;

}
```

Output:

```
201  Sonoo  990000
202  Nakul  29000
```

C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Let's see the simple example of C++ default Constructor.

```
#include <iostream>
using namespace std;
class Employee
{
public:
Employee()
{
cout<<"Default Constructor Invoked"<<endl;
}
};
int main(void)
{
Employee e1; //creating an object of Employee
Employee e2;
return 0;
}
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

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

```
class Employee {
public:
int id;//data member (also instance variable)
string name;//data member(also instance variable)
float salary;
Employee(int i, string n, float s)
{
id = i;
name = n;
salary = s;
}
void display()
{
cout<<id<<"  "<<name<<"  "<<salary<<endl;
}
};
int main(void) {
Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
Employee e2=Employee(102, "Nakul", 59000);
e1.display();
e2.display();
return 0;
}
```

Output:

```
101  Sonoo  890000
```

```
102  Nakul  59000
```

C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

C++ Constructor and Destructor Example

Let's see an example of constructor and destructor in C++ which is called automatically.

```
#include <iostream>
using namespace std;
class Employee
{
public:
Employee()
{
cout<<"Constructor Invoked"<<endl;
}
~Employee()
{
cout<<"Destructor Invoked"<<endl;
}
};
int main(void)
{
```

```
Employee e1; //creating an object of Employee
Employee e2; //creating an object of Employee
return 0;
}
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

C++ this Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used **to pass current object as a parameter to another method.**
- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

C++ this Pointer Example

Let's see the example of this keyword in C++ that refers to the fields of current class.

```
#include <iostream>
using namespace std;
class Employee {
public:
int id; //data member (also instance variable)
string name; //data member(also instance variable)
float salary;
Employee(int id, string name, float salary)
{
```

```
this->id = id;
this->name = name;
this->salary = salary;
}
void display()
{
cout<<id<<"  "<<name<<"  "<<salary<<endl;
}
};
int main(void) {
Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
e1.display();
e2.display();
return 0;
}
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

C++ friend function

If a function is defined as a friend function in C++ then the protected and private data of a class can be accessed using the function.

By using the keyword **friend** compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword **friend.**

Declaration of friend function in C++

```
class class_name

{

friend data_type function_name(argument/s);

};
```

C++ friend function Example

Let's see the simple example of C++ friend function used to print the length of a box.

```
#include <iostream>

using namespace std;

class Box

{

private:

int length;

public:

Box(): length(0) { }

friend int printLength(Box); //friend function

};

int printLength(Box b)

{

b.length += 10;

return b.length;

}

int main()

{

Box b;
cout<<"Length of box: "<< printLength(b)<<endl;

return 0;

}
```

Output:

```
Length of box: 10
```

C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
#include <iostream>

using namespace std;

class Account {

public:

float salary = 60000;
};
class Programmer: public Account {
public:
float bonus = 5000;
};
int main(void) {
Programmer p1;
cout<<"Salary: "<<p1.salary<<endl;
cout<<"Bonus: "<<p1.bonus<<endl;
return 0;
}
```

Output:

```
Salary: 60000
Bonus: 5000
```

In the above example, Employee is the base class and Programmer is the derived class.

C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

```
#include <iostream>
using namespace std;
class Animal {
public:
void eat() {
cout<<"Eating..."<<endl;
}
};
```

```
class Dog: public Animal
{
public:
void bark(){
cout<<"Barking...";
}
};
int main(void) {
Dog d1;
d1.eat();
d1.bark();
return 0;
}
```

Output:

```
Eating...
Barking...
```

C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

```
#include <iostream>
using namespace std;
class Animal {
public:
void eat() {
cout<<"Eating..."<<endl;
}
};
class Dog: public Animal
{
public:
void bark(){
cout<<"Barking..."<<endl;
}
};
class BabyDog: public Dog
{
public:
void weep() {
cout<<"Weeping...";
}
};
int main(void) {
BabyDog d1;
d1.eat();
d1.bark();
d1.weep();
return 0;
}
```

Output:

```
Eating...
Barking?
Weeping?
```

C++ Aggregation (HAS-A Relationship)

In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

C++ Aggregation Example

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

```
#include <iostream>
using namespace std;
class Address {
public:
string addressLine, city, state;
Address(string addressLine, string city, string state)
{
this->addressLine = addressLine;
this->city = city;
this->state = state;
}
};
class Employee
{
private:
Address* address;  //Employee HAS-A Address
public:
int id;
string name;
Employee(int id, string name, Address* address)
{
this->id = id;
this->name = name;

this->address = address;

}

void display()

{

cout<<id <<" "<<name<< " "<<

address->addressLine<< " "<< address->city<< " "<<address->state<<endl;

}

};

int main(void) {

Address a1= Address("C-146, Sec-15","Noida","UP");

Employee e1 = Employee(101,"Nakul",&a1);

e1.display();

return 0;

}
```

Output:

```
101 Nakul C-146, Sec-15 Noida UP
```

Overview of Inheritance

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class which inherits properties of other class is called **Child** or **Derived** or **Sub** class.

Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

NOTE : All members of a class except Private, are inherited

Purpose of Inheritance

1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

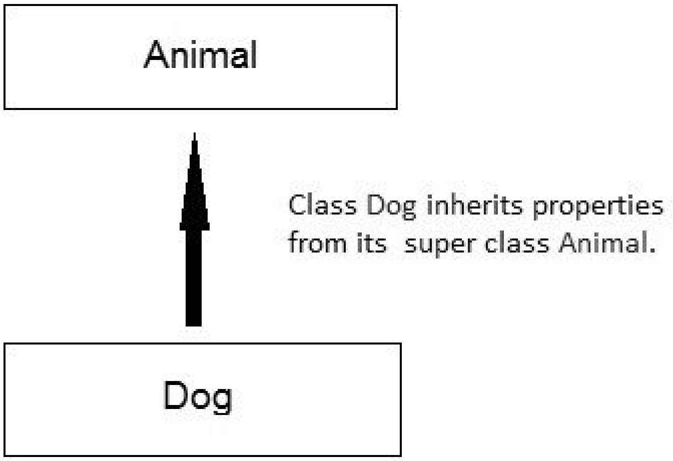
Basic Syntax of Inheritance

```
class Subclass_name : access_mode Superclass_name
```

While defining a subclass like this, the super class must be already defined or atleast declared before the subclass declaration.

Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, privtate or protected.

Example of Inheritance



Class Dog inherits properties from its super class Animal.

```
class Animal
{ public:
int legs = 4;
};
class Dog : public Animal
{ public:
int tail = 1;
};

int main()
{
Dog d;
cout << d.legs;

cout << d.tail;

}
```

Output :

```
4 1
```

Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

1) Public Inheritance

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.

class Subclass : **public** Superclass

2) Private Inheritance

In private mode, the protected and public members of super class become private members of derived class.

class Subclass : Superclass // By default its private inheritance

3) Protected Inheritance

In protected mode, the public and protected members of Super class becomes protected members of Sub class.

class subclass : **protected** Superclass

Table showing all the Visibility Modes

	Derived Class	**Derived Class**	**Derived Class**
Base class	**Public Mode**	**Private Mode**	**Protected Mode**

	Derived Class	**Derived Class**	**Derived Class**
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

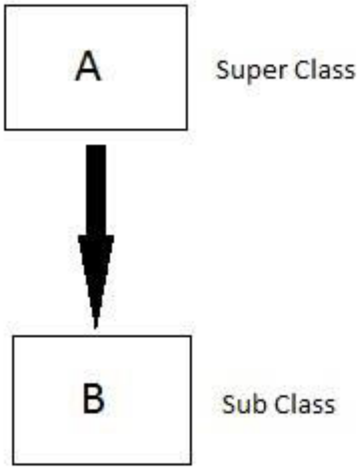
Types of Inheritance

In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

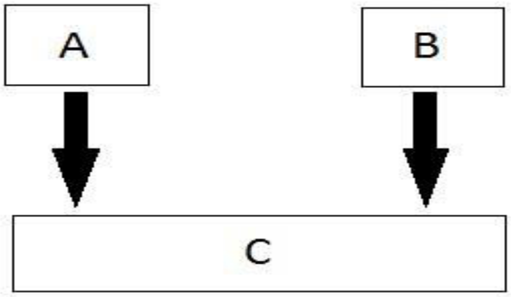
Single Inheritance

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



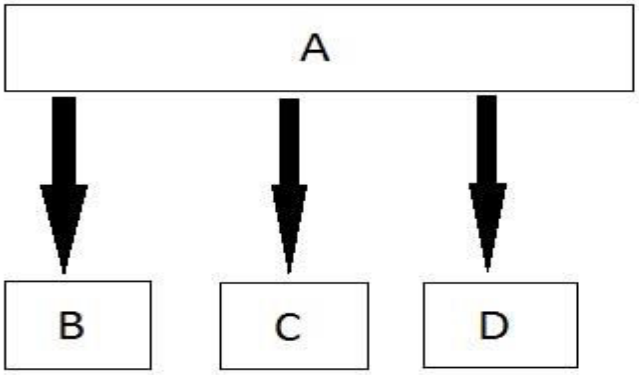
Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.



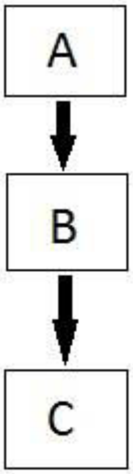
Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.



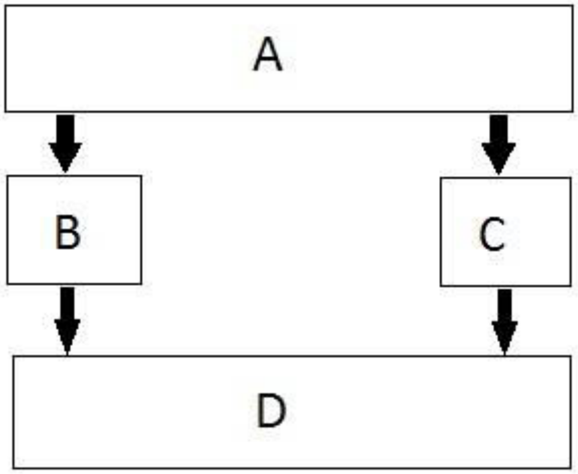
Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



C++ Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation and polymorphism.

There are two types of polymorphism in C++:

- **Compile time polymorphism:** It is achieved by function overloading and operator overloading which is also known as static binding or early binding.
- **Runtime polymorphism:** It is achieved by method overriding which is also known as dynamic binding or late binding.

C++ Runtime Polymorphism Example

Let's see a simple example of runtime polymorphism in C++.

```
#include <iostream>
using namespace std;
class Animal {
public:
void eat(){
cout<<"Eating...";
}
};
class Dog: public Animal
{
public:
void eat()
{
cout<<"Eating bread...";
}
};
int main(void) {
Dog d = Dog();
d.eat();

return 0;

}
```

Output:

```
Eating bread...
```

C++ Runtime Polymorphism Example: By using two derived class

Let's see another example of runtime polymorphism in C++ where we are having two derived classes.

```
#include <iostream>

using namespace std;

class Shape {

public:

virtual void draw(){

cout<<"drawing..."<<endl;

}

};

class Rectangle: public Shape

{

public:

void draw()

{

cout<<"drawing rectangle..."<<endl;

}

};

class Circle: public Shape
```

```
{
public:
void draw()
{
cout<<"drawing circle..."<<endl;
}
};
int main(void) {
Shape *s;
Shape sh;
Rectangle rec;
Circle cir;
s=&sh;
s->draw();
s=&rec;
s->draw();
s=○
s->draw();
}
```

Output:

```
drawing...
drawing rectangle...
drawing circle...
```

Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

```
#include <iostream>
using namespace std;
class Animal {
public:
string color = "Black";
};
class Dog: public Animal
{
public:
string color = "Grey";
};
int main(void) {
Animal d= Dog();
cout<<d.color;
}
```

Output:

```
Black
```

Algorithm Definition

An *algorithm* is a finite set of instructions that accomplishes a particular task.

Criteria

- input
- output
- definiteness: clear and unambiguous
- finiteness: terminate after a finite number of steps
- effectiveness: instruction is basic enough to be carried out

Data Type

A *data type* is a collection of *objects* and a set of *operations* that act on those objects.

Abstract Data Type

An *abstract data type(ADT)* is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations

Specificatin vs Implementation

Operation specification

- function name
- the types of arguments
- the type of the results

Implementation independent

Abstract data type Natural_Number

```
structure Natural_Number is
  objects:  an ordered subrange of the integers starting at zero and ending
            at the maximum integer (INT_MAX) on the computer
  functions:
    for all x, y ∊ Nat_Number; TRUE, FALSE ∊ Boolean
    and where +, -, <, and == are the usual integer operations.
    Nat_No Zero ( )          ::=  0
    Boolean  Is_Zero(x)      ::= if (x) return FALSE
                                  else return TRUE
    Nat_No Add(x, y)         ::= if ((x+y) <= INT_MAX) return x+y
                                  else return INT_MAX
    Boolean Equal(x,y)       ::= if (x== y) return TRUE
                                 else return FALSE
    Nat_No Successor(x)      ::= if (x == INT_MAX) return x
                                  else return x+1
    Nat_No Subtract(x,y)     ::= if (x<y) return 0
                                  else return x-y
  end Natural_Number
```

Space Complexity

$S(P)=C+S_P(I)$

Fixed Space Requirements (C)
Independent of the characteristics of the inputs and outputs

- instruction space
- space for simple variables, fixed-size structured variable, constants

Variable Space Requirements ($S_P(I)$)
depend on the instance characteristic I

- number, size, values of inputs and outputs associated with I
- recursive stack space, formal parameters, local variables, return address

Program: Simple arithmetic function

```
float abc(float a, float b, float c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;
 }
```

$S_{abc}(I) = 0$

Program: Iterative function for summing a list of numbers

```
float sum(float list[ ], int n)
{
  float tempsum = 0;
  int i;
  for (i = 0; i<n; i++)
     tempsum += list [i];
  return tempsum;
}
```

$S_{sum}(I) = 0$

Recall: pass the address of the first element of the array & pass by value

Program: Recursive function for summing a list of numbers

```
float rsum(float list[ ], int n)
{
   if (n) return rsum(list, n-1) + list[n-1];
   return 0;
 }
```

$S_{sum}(I)=S_{sum}(n)=6n$

*__Figure 1.1:__ Space needed for one recursive call of Program 1.3

Type	Name	Number of bytes
parameter: float parameter: integer return address:(used internally)	list [ ] n	2 2 2(unless a far address)
TOTAL per recursive call		6

TIME COMPLEXITY

$T(P)=C+T_P(I)$

Compile time (C)
independent of instance characteristics

$T_P(n)=c_a ADD(n)+c_s SUB(n)+c_l LDA(n)+c_{st} STA(n)$

run (execution) time T_P

Definition
A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Example

- abc = a + b + b * c + (a + b - c) / (a + b) + 4.0
- abc = a + b + c

Regard as the same unit machine independent

METHODS TO COMPUTE THE STEP COUNT

1.Introduce variable count into programs

2.Tabular method

- Determine the total number of steps contributed by each statement
  step per execution × frequency
- add up the contribution of all statements

Iterative summing of a list of numbers

```
float sum(float list[ ], int n)
{
   float tempsum = 0; count++; /* for assignment */
   int i;
   for (i = 0; i < n; i++) {
        count++;           /*for the for loop */
        tempsum += list[i]; count++;  /* for assignment */
   }
   count++;        /* last execution of for */
   return tempsum;
   count++;         /* for return */
}
```

2n + 3 steps

Program : Simplified version of Program

```
float sum(float list[ ], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
        count += 2;
   count += 3;
   return 0;
}
```

2n + 3 steps

Recursive summing of a list of numbers

PROGRAM :

```
float rsum(float list[ ], int n)
{
        count++;       /*for if conditional */
        if (n) {
                count++;  /* for return and rsum invocation */
                return rsum(list, n-1) + list[n-1];
        }
        count++;
        return list[0];
}   2n+2
```

MATRIX ADDITION

Program : Matrix addition

```
void add( int a[ ] [MAX_SIZE], int b[ ] [MAX_SIZE],
                   int c [ ] [MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++)
      for (j= 0; j < cols; j++)
        c[i][j] = a[i][j] +b[i][j];
 }
```

Program: Matrix addition with count statements

```
            void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],
                           int c[ ][MAX_SIZE], int row, int cols )
{
  int i, j;
  for (i = 0; i < rows; i++){
       count++; /* for i for loop */
       for (j = 0; j < cols; j++) {
          count++; /* for j for loop */
          c[i][j] = a[i][j] + b[i][j];
          count++; /* for assignment statement */
       }
       count++;    /* last time of j for loop */
  }
 count++;         /* last time of i for loop */
}
```

PROGRAM :

```
void add(int a[ ][MAX_SIZE], int b [ ][MAX_SIZE],
                           int c[ ][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for( i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++)
          count += 2;
          count += 2;
   }
   count++;
}
```

2rows × cols + 2rows +1

Tabular Method

Fig : Step count table for Program

Iterative function to sum a list of numbers

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1

Recursive Function to sum of a list of numbers

Fig : Step count table for recursive summing function

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total Matrix Addition			2n+2

Step count table for matrix addition

Statement	s/e	Frequency	Total steps
Void add (int a[ ][MAX_SIZE]· · ·)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i = 0; i < row; i++)	1	rows+1	rows+1
for (j=0; j< cols; j++)	1	rows· (cols+1)	rows· cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows· cols	rows· cols
}	0	0	0
Total	2rows· cols+2rows+1		

ASYMPLOTIC NOTATION

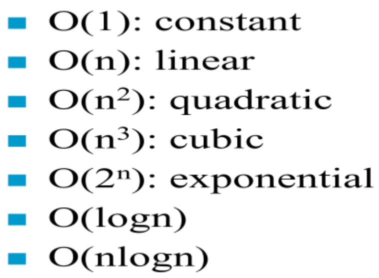
Definition
$f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$.

Examples

- $3n+2=O(n)$ /* $3n+2 \leq 4n$ for $n \geq 2$ */
- $3n+3=O(n)$ /* $3n+3 \leq 4n$ for $n \geq 3$ */
- $100n+6=O(n)$ /* $100n+6 \leq 101n$ for $n \geq 10$ */
- $10n^2+4n+2=O(n^2)$ /* $10n^2+4n+2 \leq 11n^2$ for $n \geq 5$ */
- $6*2^n+n^2=O(2^n)$ /* $6*2^n+n^2 \leq 7*2^n$ for $n \geq 4$ */

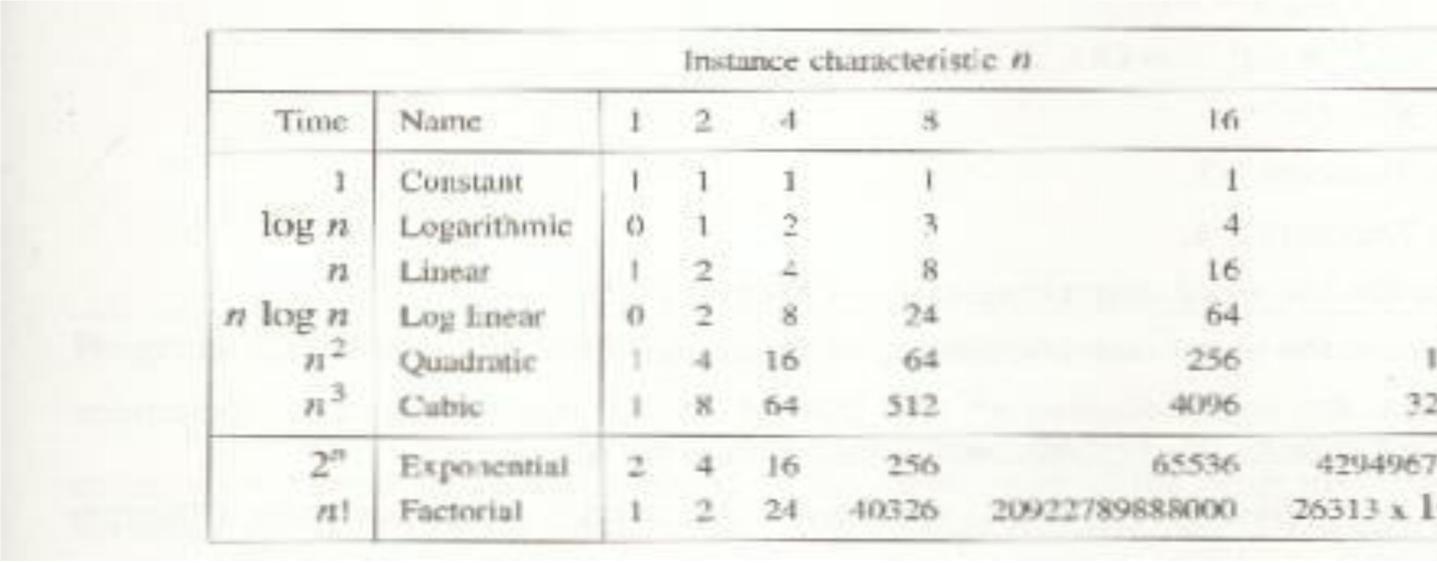
EXAMPLE

- Complexity of $c_1n^2+c_2n$ and c_3n
  - for sufficiently large of value, c_3n is faster than $c_1n^2+c_2n$
  - for small values of n, either could be faster
    - $c_1=1$, $c_2=2$, $c_3=100$ --> $c_1n^2+c_2n \leq c_3n$ for $n \leq 98$
    - $c_1=1$, $c_2=2$, $c_3=1000$ --> $c_1n^2+c_2n \leq c_3n$ for $n \leq 998$
  - break even point
    - no matter what the values of c1, c2, and c3, the n beyond which c_3n is always faster than $c_1n^2+c_2n$



- O(1): constant
- O(n): linear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential
- O(logn)
- O(nlogn)

Fig:Function values



		Instance characteristic n					
Time	Name	1	2	4	8	16	
1	Constant	1	1	1	1	1	
$\log n$	Logarithmic	0	1	2	3	4	
n	Linear	1	2	4	8	16	
$n \log n$	Log linear	0	2	8	24	64	
n^2	Quadratic	1	4	16	64	256	1
n^3	Cubic	1	8	64	512	4096	32
2^n	Exponential	2	4	16	256	65536	4294967
$n!$	Factorial	1	2	24	40326	20922789888000	26313 x 1

UNIT II

ARRAYS

Array: a set of index and value

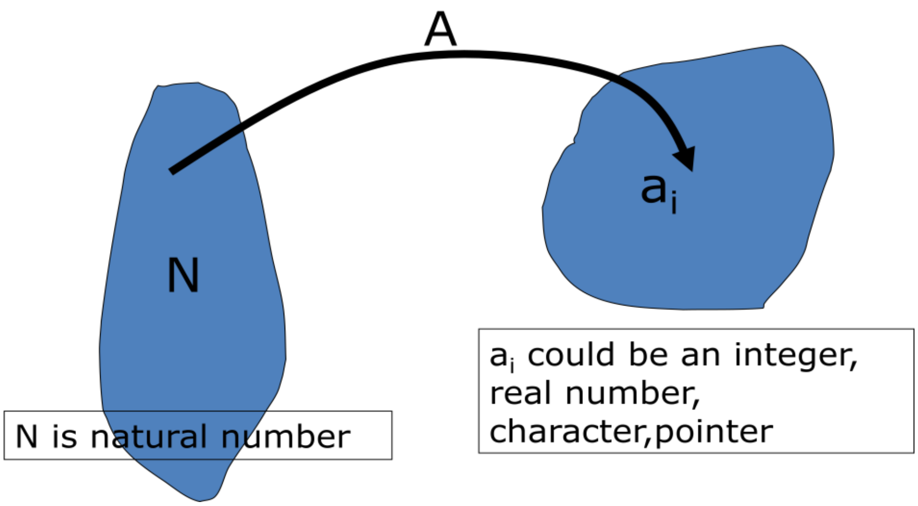
a collection of data of the same type data structure

For each index, there is a value associated with that index

representation (possible)

implemented by using consecutive memory.

Array $A(i)=a_i$ $i \in$ Integers



N is natural number

a_i could be an integer,
real number,
character,pointer

Structure *Array* is

objects: A set of pairs <*index, value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions, for example, {0, … , n-1} for one dimension, {(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)} for two dimensions, etc.

Functions:

for all A ∈ Array, *i* ∈ *index*, x ∈ *item, j, size* ∈ integer

Array Create(j, list) ::= **return** an array of *j* dimensions where list is a j-tuple whose ith element is the size of the ith dimension. *Items* are undefined.

Item Retrieve(A, *i*) ::= **if** (*i* ∈ *index*) **return** the item associated with index value *i* in array A
else return error

Array Store(A, *i, x*) ::= **if** (*i* in *index*)
return an array that is identical to array A except the new pair <*i, x*> has been inserted **else return** error

end Array

Abstract Data Type *Array*

int list[5], *plist[5];

list[5]: five integers

list[0], list[1], list[2], list[3], list[4]

*plist[5]: five pointers to integers

plist[0], plist[1], plist[2], plist[3], plist[4]

implementation of 1-D array

list[0]	base address = α
list[1]	α + 1*sizeof(int)
list[2]	α + 2*sizeof(int)
list[3]	α + 3*sizeof(int)
list[4]	α + 4*size(int)

Compare int *list1 and int list2[5] in C.

Same: list1 and list2 are pointers.

Difference: list2 reserves five locations.

Notations:

list2 - a pointer to list2[0]

(list2 + i) - a pointer to list2[i] (&list2[i])

*(list2 + i) - list2[i]

Example: 1-dimension array addressing

```
int one[] = {0, 1, 2, 3, 4};
```

Goal: print out address and value

```
void print1(int *ptr, int rows)
{
/* print out a one-dimensional array using a pointer */
        int i;
        printf("Address Contents\n");
        for (i=0; i < rows; i++)
```

```
                printf("%8u%5d\n", ptr+i, *(ptr+i));
        printf("\n");
}
```

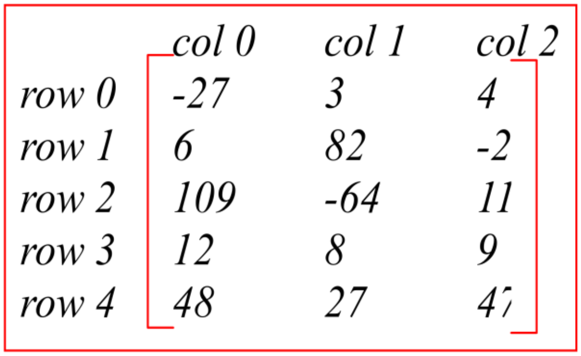
Address	Contents
1228	0
1230	1
1232	2
1234	3
1236	4

1. A[-3..2, -1..6, 2..7, 0..5] each of its elements occupies three memory spaces starting from 123

   Find the address of the element A[0,1,2,3]

   (1) In row major order (2) In column major order

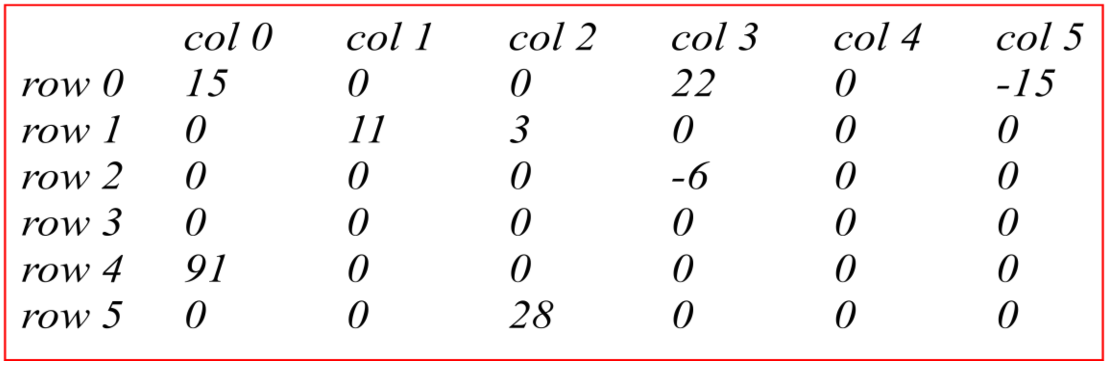
The Sparse Matrix Abstract Data Type



$$\begin{array}{c|ccc|} & col\ 0 & col\ 1 & col\ 2 \\ row\ 0 & -27 & 3 & 4 \\ row\ 1 & 6 & 82 & -2 \\ row\ 2 & 109 & -64 & 11 \\ row\ 3 & 12 & 8 & 9 \\ row\ 4 & 48 & 27 & 47 \end{array}$$

Matrix

   - Examples of matrix
- Sparse matrix
   - Many zero items
- Representation of matrix
   - A[][], standard representation
   - Sparse matrix, **store non-zero item only**



	col 0	col 1	col 2	col 3	col 4	col 5
row 0	15	0	0	22	0	-15
row 1	0	11	3	0	0	0
row 2	0	0	0	-6	0	0
row 3	0	0	0	0	0	0
row 4	91	0	0	0	0	0
row 5	0	0	28	0	0	0

Structure *Sparse_Matrix* is
objects: a set of triples, <*row, column, value*>, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item.*
functions:
for all *a, b* ∈ *Sparse_Matrix*, *x* Î *item, i, j, max_col*, *max_row* Î *index*

Sparse_Marix Create(*max_row, max_col*) ::=
return a *Sparse_matrix* that can hold up to *max_items* = *max _row* ´ *max_col* and whose maximum row size is *max_row* and whose maximum column size is *max_col.*

Sparse_Matrix Transpose(*a*) ::=
return the matrix produced by interchanging the row and column value of every triple.
Sparse_Matrix Add(*a, b*) ::=
if the dimensions of a and b are the same
return the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.
else return error
Sparse_Matrix Multiply(*a, b*) ::=
if number of columns in a equals number of rows in **b**
return the matrix *d* produced by multiplying

a by *b* according to the formula: *d* [*i*] [*j*] =
S(a[i][k]•b[k][j]) where *d (i, j)* is the *(i,j)*th
element
else return error.

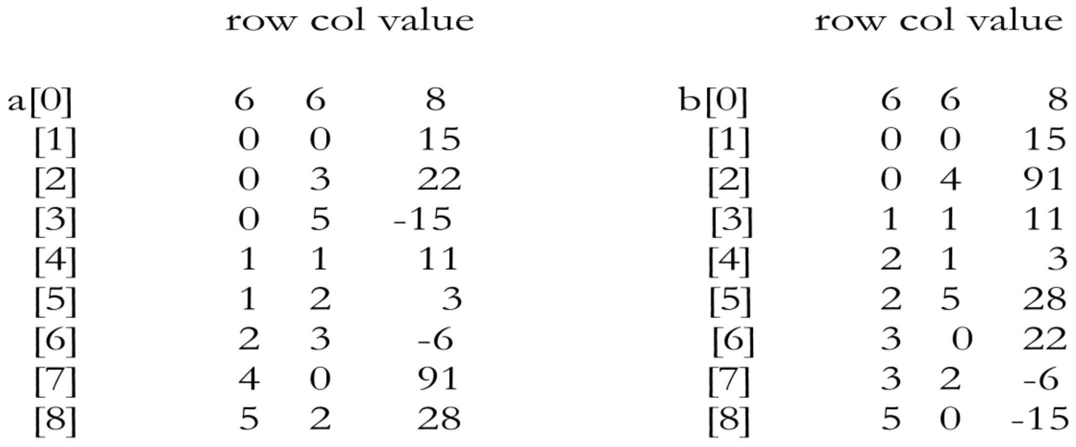
ABSTRACT DATA TYPE SPARSE-MATRIX

(1) Represented by a two-dimensional array.

Sparse matrix wastes space.

(2) Each element is characterized by <row, col, value>.

Sparse matrix and its transpose stored as triples



	row	col	value		row	col	value
a[0]	6	6	8	b[0]	6	6	8
[1]	0	0	15	[1]	0	0	15
[2]	0	3	22	[2]	0	4	91
[3]	0	5	-15	[3]	1	1	11
[4]	1	1	11	[4]	2	1	3
[5]	1	2	3	[5]	2	5	28
[6]	2	3	-6	[6]	3	0	22
[7]	4	0	91	[7]	3	2	-6
[8]	5	2	28	[8]	5	0	-15

Sparse_matrix Create(max_row, max_col) ::=

```
#define MAX_TERMS 101 /* maximum number of terms +1*/
   typedef struct {
             int col;
             int row;
             int value;
             } term;
   term a[MAX_TERMS]
```

TRANSPOSE A MATRIX

```
(1) for each row i
         take element <i, j, value> and store it
         in element <j, i, value> of the transpose.
```

difficulty: where to put <j, i, value>

```
(0, 0, 15)  ====>  (0, 0, 15)

(0, 3, 22)  ====>  (3, 0, 22)

(0, 5, -15) ====>  (5, 0, -15)

 (1, 1, 11) ====>  (1, 1, 11)
```

Move elements down very often.

```
(2) For all elements in column j,

         place element <i, j, value> in element <j, i, value>
```

```
void transpose (term a[], term b[])
/* b is set to the transpose of a */
{
   int n, i, j, currentb;
   n = a[0].value;  /* total number of elements */
   b[0].row = a[0].col;  /* rows in b = columns in a */
```

```
    b[0].col = a[0].row;  /*columns in b = rows in a */
    b[0].value = n;
    if (n > 0) {                    /*non zero matrix */
        currentb = 1;
        for (i = 0; i < a[0].col; i++)
        /* transpose by columns in a */
              for( j = 1; j <=  n; j++)
              /*  find elements from the current column */
              if (a[j].col == i) {
             /* element is in current column, add it to b */

if (n > 0) {                    /*non zero matrix */
        currentb = 1;
        for (i = 0; i < a[0].col; i++)
        /* transpose by columns in a */
              for( j = 1; j <=  n; j++)
              /*  find elements from the current column */
              if (a[j].col == i) {
             /* element is in current column, add it to b */

                 b[currentb].row = a[j].col;
                 b[currentb].col  = a[j].row;
                 b[currentb].value = a[j].value;
                 currentb++
              }
    }
}                                                              O(columns*elements)
```

Discussion: compared with 2-D array representation

O(columns*elements) vs. O(columns*rows)

elements --> columns * rows when nonsparse

O(columns*columns*rows)

Problem: Scan the array “columns” times.

Solution:

Determine the number of elements in each column of the original matrix.

==>

Determine the starting positions of each row in the transpose matrix.

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	2	0	1
starting_pos =	1	3	4	6	8	8

a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

```
void fast_transpose(term a[ ], term b[ ])
  {
  /* the transpose of a is placed in b */
    int row_terms[MAX_COL], starting_pos[MAX_COL];
    int i, j, num_cols = a[0].col, num_terms = a[0].value;
    b[0].row = num_cols; b[0].col = a[0].row;
    b[0].value = num_terms;
    if (num_terms > 0){ /*nonzero matrix*/
       for (i = 0; i < num_cols; i++)
             row_terms[i] = 0;
       for (i = 1; i  <= num_terms; i++)
             row_term [a[i].col]++
       starting_pos[0] = 1;
       for (i =1; i < num_cols; i++)
             starting_pos[i]=starting_pos[i-1] +row_terms [i-1];
  for (i=1; i <= num_terms, i++) {
                   j = starting_pos[a[i].col]++;
                   b[j].row = a[i].col;
                   b[j].col = a[i].row;
                   b[j].value = a[i].value;
```

```
        }
    }
}
```

FAST TRANSPOSE OF A SPARSE MATRIX

Compared with 2-D array representation

O(columns+elements) vs. O(columns*rows)

elements --> columns * rows

O(columns+elements) --> O(columns*rows)

Cost: Additional row_terms and starting_pos arrays are required.

Let the two arrays row_terms and starting_pos be shared.

	space	time
2D array	**O(rows * cols)**	**O(rows * cols)**
Transpose	**O(elements)**	**O(cols * elmnts)**
Fast Transpose	**O(elmnts+MAX_COL)**	**O(cols + elmnts)**

SPARSE MATRIX MULTIPLICATION

Definition: $[D]_{m*p}=[A]_{m*n}* [B]_{n*p}$

Procedure: Fix a row of A and find all elements in column j

of B for j=0, 1, …, p-1.

Alternative 1. Scan all of B to find all elements in j.

Alternative 2. Compute the transpose of B.

(Put all column elements consecutively)

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

```
void mmult (term a[ ], term b[ ], term d[ ] )
/* multiply two sparse matrices */
{
  int i, j, column, totalb = b[].value, totald = 0;
  int rows_a = a[0].row, cols_a = a[0].col,
  totala = a[0].value; int cols_b = b[0].col,
  int row_begin = 1, row = a[1].row, sum =0;
  int new_b[MAX_TERMS][3];
  if (cols_a != b[0].row){
      fprintf (stderr, "Incompatible matrices\n");
      exit (1);
  }

fast_transpose(b, new_b);
a[totala+1].row = rows_a;
new_b[totalb+1].row = cols_b;
new_b[totalb+1].col = 0;
for (i = 1; i <= totala; ) {
   column = new_b[1].row;
   for (j = 1; j <= totalb+1;) {
   /* mutiply row of a by column of b */
   if (a[i].row != row)  {
     storesum(d, &totald, row, column, &sum);
     i = row_begin;
     for (; new_b[j].row == column; j++)
       ;
     column =new_b[j].row
   }
else switch (COMPARE (a[i].col, new_b[j].col)) {
            case -1: /* go to next term in a */
                      i++; break;
            case 0: /* add terms, go to next term in a and b */
                      sum += (a[i++].value * new_b[j++].value);
                      break;
             case 1: /* advance to next term in b*/
                       j++
          }
      } /* end of for j <= totalb+1 */
      for (; a[i].row == row; i++)
          ;
       row_begin = i; row = a[i].row;
    } /* end of for i <=totala */
    d[0].row = rows_a;
    d[0].col = cols_b; d[0].value = totald;
}
```

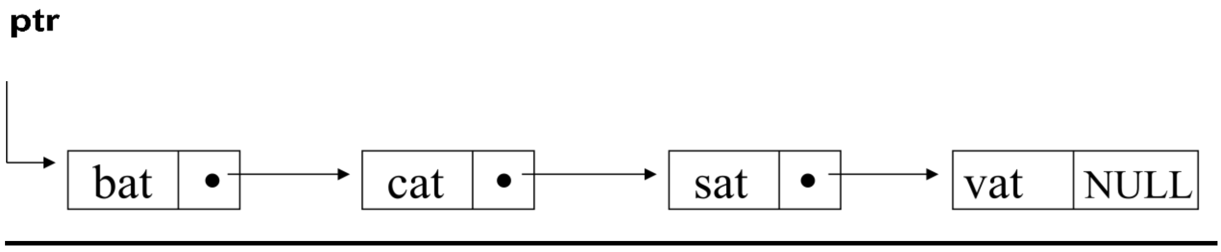
Program : Sparse matrix multiplication

Linked list

An ordered sequence of nodes with links

The nodes do not reside in sequential locations

The locations of the nodes may change on different runs



Usual way to draw a linked list

Create a linked list of words

```
typedef struct list_node *list_pointer;
typedef struct list_node {
```

```
            char data [4];
            list_pointer link;
            };
```

Creation

```
list_pointer ptr =NULL;
```

Testing

```
#define IS_EMPTY(ptr) (!(ptr))
```

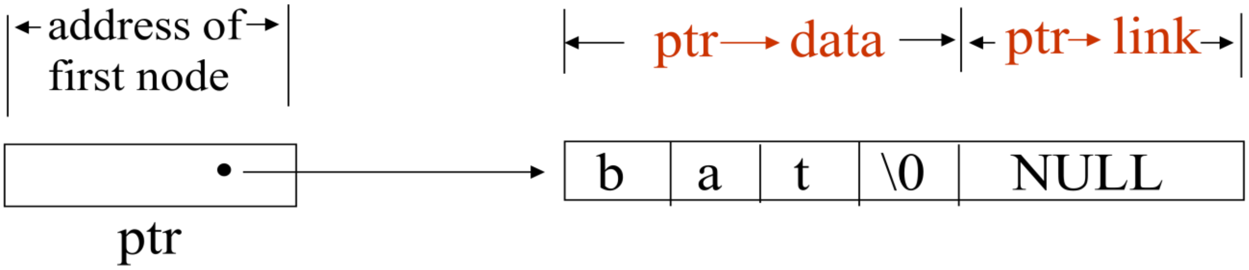
Allocation

```
ptr=(list_pointer) malloc (sizeof(list_node));
```

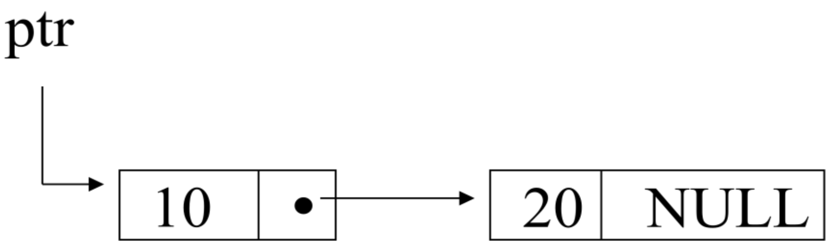
```
e -> name -> (*e).name
```

```
strcpy(ptr -> data, “bat”);
```

```
ptr -> link = NULL;
```

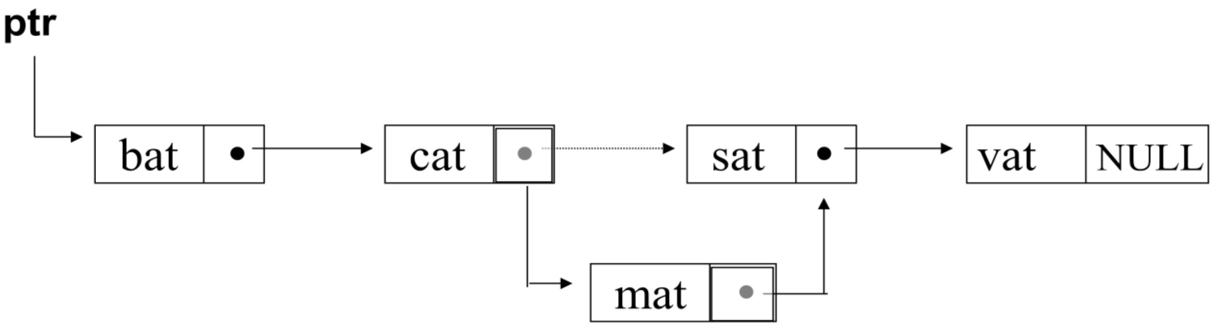


Referencing the fields of a node



```
typedef struct list_node *list_pointer;
typedef struct list_node {
            int data;
            list_pointer link;
            };
list_pointer ptr =NULL
list_pointer create2( )
{
/* create a linked list with two nodes */
   list_pointer first, second;
   first = (list_pointer) malloc(sizeof(list_node));
   second = ( list_pointer) malloc(sizeof(list_node));
   second -> link = NULL;
   second -> data = 20;
   first -> data = 10;
   first ->link = second;
   return first;



}
```

Insert mat after cat

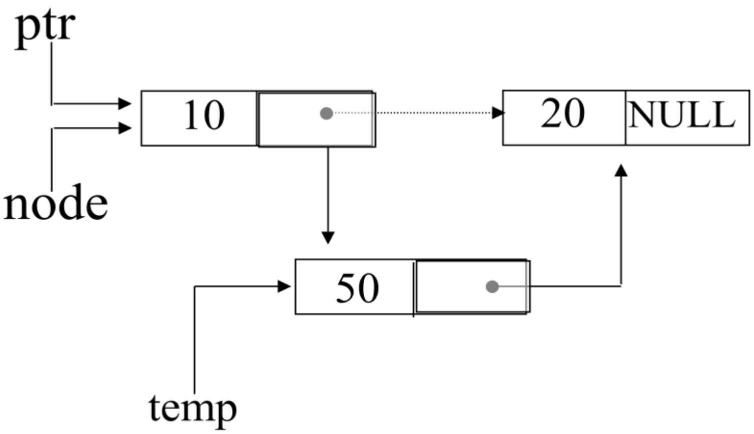


1. Get a node that is currently unused ; let its address be paddr.
2. Set the data field of this node to mat.
3. Set paddr's link field to point to the address found in the link field of the node containing cat.
4. Set the link field of the node containing cat to point to paddr.

Insert a node after a specific node

```
#define IS_FULL(p) (!(p))

void insert(list_pointer *ptr, list_pointer node)
{
/* insert a new node with data = 50 into the list ptr after node */
```

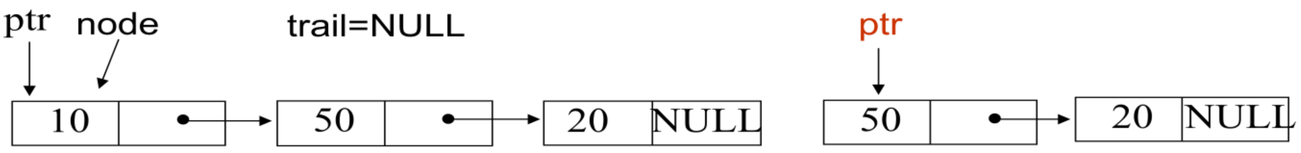
```
    list_pointer temp;
    temp = (list_pointer) malloc(sizeof(list_node));
    if (IS_FULL(temp))            //if, temp==NULL
    {
      fprintf(stderr, "The memory is full\n");
      exit (1);
    }

    temp->data = 50;
    if (*ptr) {  // nonempty list
       temp->link =node ->link;
       node->link = temp;
   }
   else {    // empty list
      temp->link = NULL;
      *ptr =temp;
    }
}
```

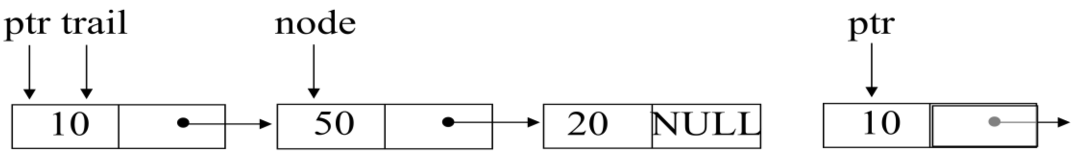


List Deletion

a) Before **b) After deletion**

Delete the first node.





Delete other than first node

```
void delete(list_pointer *ptr, list_pointer trail,
                                                list_pointer node)
{
/* delete node from the list, trail is the preceding node
   ptr is the head of the list */
    if (trail)
       trail->link = node->link;
    else
       *ptr = (*ptr) ->link;
     free(node);
}
```

Print out a list (traverse a list)

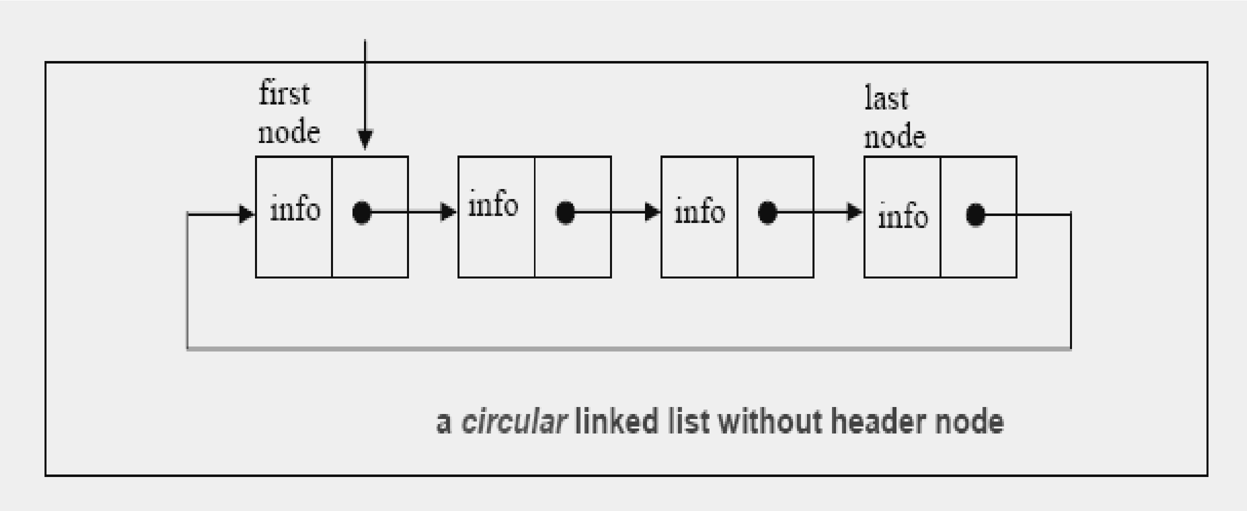
```
void print_list(list_pointer ptr)
{
   printf("The list ocntains: ");
   for ( ; ptr; ptr = ptr->link)
      printf("%4d", ptr->data);
   printf("\n");
 }
```

Circular Linked Lists

A Circular Linked List is a special type of Linked List

It supports traversing from the end of the list to the beginning by making the last node point back to the head of the list

A Rear pointer is often used instead of a Head pointer



a *circular* linked list without header node

MOTIVATION

Circular linked lists are usually sorted

Circular linked lists are useful for playing video and sound files in “looping” mode

They are also a stepping stone to implementing graphs, an important topic in comp171.

Definition

```
#include <stdio.h>

using namespace std;

struct Node{

    int data;

    Node* next;

};

typedef Node* NodePtr;
```

Circular Linked List Operations

```
insertNode(NodePtr& Rear, int item)
    //add new node to ordered circular linked list
deleteNode(NodePtr& Rear, int item)
    //remove a node from circular linked list
print(NodePtr Rear)
    //print the Circular Linked List once
```

Traverse The List

```
void print(NodePtr Rear){
        NodePtr Cur;
        if(Rear != NULL){
                Cur = Rear->next;
                do{
                        cout << Cur->data << " ";
                        Cur = Cur->next;
                }while(Cur != Rear->next);
                cout << endl;
        }
}
```

Insert Node

- **Insert into an empty list**

```
NotePtr New = new Node;
New->data = 10;
```

```
Rear = New;

Rear->next = Rear;
```

⌧ **Insert to head of a Circular Linked List**

```
New->next = Cur;  // same as: New->next = Rear->next;

Prev->next = New; // same as: Rear->next = New;
```

⌧ **Insert to middle of a Circular Linked List between Pre and Cur**

```
New->next = Cur;

Prev->next = New;
```

Insert to end of a Circular Linked List

```
New->next = Cur;   // same as: New->next = Rear->next;

Prev->next = New;           // same as: Rear->next = New;

Rear = New;

    void insertNode(NodePtr& Rear, int item){

            NodePtr  New, Cur, Prev;

            New = new Node;

            New->data = item;

            if(Rear == NULL){     // insert into empty list

                    Rear = New;

                    Rear->next = Rear;

                    return;

            }

            Prev = Rear;

            Cur = Rear->next;

            do{                                     // find Prev and Cur

                    if(item <= Cur->data)
                    break;
                Prev = Cur;
                Cur = Cur->next;
            }while(Cur != Rear->next);
            New->next = Cur;       // revise pointers
            Prev->next = New;
            if(item > Rear->data)  //revise Rear pointer if adding to end
                Rear = New;
    }
```

⌧ Delete a node from a single-node Circular Linked List

```
Rear = NULL;
delete Cur;
```

⌧ Delete the head node from a Circular Linked List

```
Prev->next = Cur->next;  // same as: Rear->next = Cur->next
```

⌧ Delete a middle node Cur from a Circular Linked List

```
Prev->next = Cur->next;
delete Cur;
DelPrev->next = Cur->next;   // same as: Rear->next;
delete Cur;
Rear = Prev;
void deleteNode(NodePtr& Rear, int item){
    NodePtr Cur, Prev;
    if(Rear == NULL){
            cout << "Trying to delete empty list" << endl;
            return;
        }
        Prev = Rear;
        Cur = Rear->next;
        do{                                 // find Prev and Cur
            if(item <= Cur->data)  break;
            Prev = Cur;
            Cur = Cur->next;
        }while(Cur != Rear->next);
        if(Cur->data != item){// data does not exist
            cout << "Data Not Found" << endl;
            return;
        }
        if(Cur == Prev){                    // delete single-node list
            Rear = NULL;
            delete Cur;
            return;
        }
        if(Cur == Rear)                     // revise Rear pointer if deleting end
            Rear = Prev;
        Prev->next = Cur->next;             // revise pointers
        delete Cur;
    }
```

```
void main(){
     NodePtr Rear = NULL;


     insertNode(Rear, 3);
     insertNode(Rear, 1);
     insertNode(Rear, 7);
     insertNode(Rear, 5);
     insertNode(Rear, 8);
     print(Rear);
     deleteNode(Rear, 1);
     deleteNode(Rear, 3);
     deleteNode(Rear, 8);
     print(Rear);
     insertNode(Rear, 1);
     insertNode(Rear, 8);
     print(Rear);
 }
```

Result is:

```
1 3 5 7 8
5 7
1 5 7 8
```

Doubly Linked List

Move in forward and backward direction.

Singly linked list (in one direction only)

How to get the preceding node during deletion or insertion?

Using 2 pointers

Node in doubly linked list

left link field (llink)

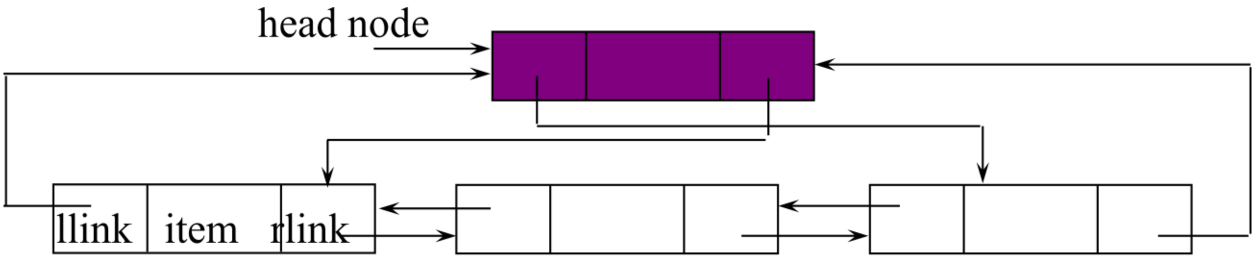
data field (item)

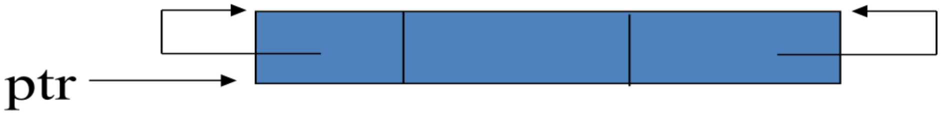
right link field (rlink)

```
typedef struct node *node_pointer;

typedef struct node {

node_pointer llink;

element item;

node_pointer rlink;
```

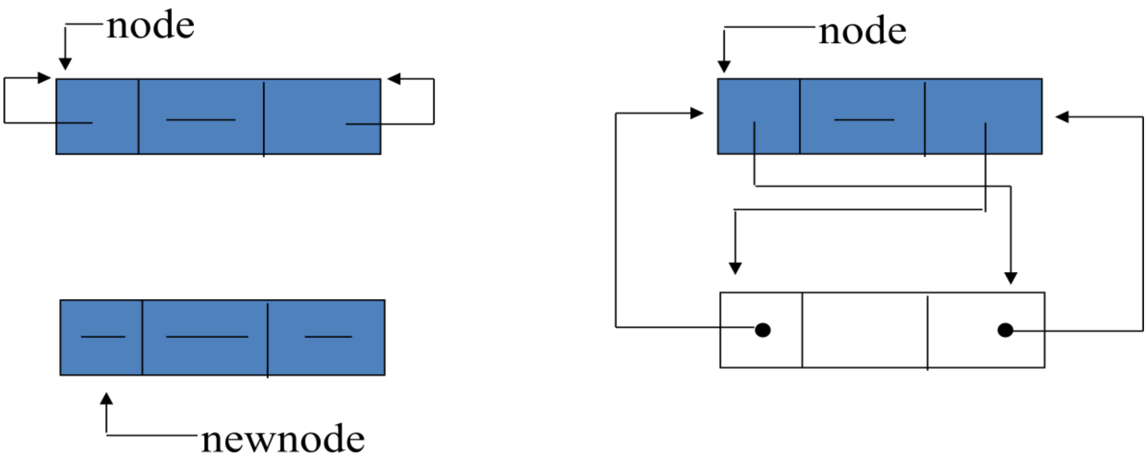
}

ptr
= ptr->rlink->llink
= ptr->llink->rlink





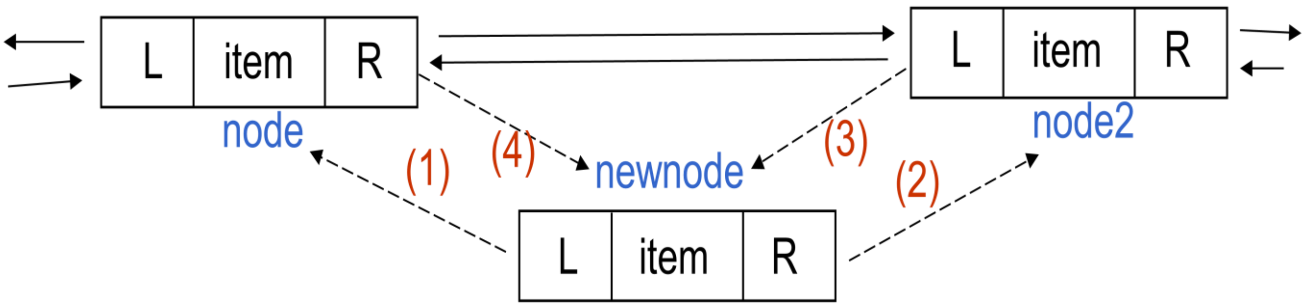
<u>Empty doubly linked circular list with head node</u>



Insertion into an empty doubly linked circular list

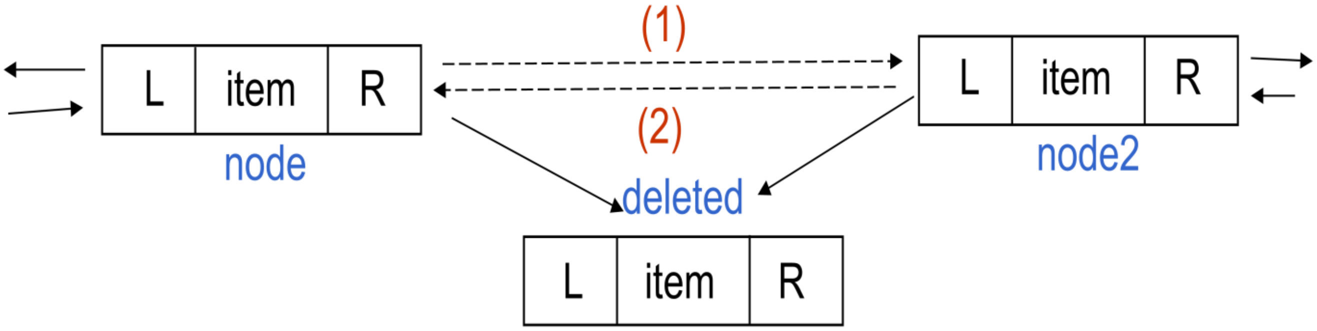
INSERT

```
void dinsert(node_pointer node, node_pointer newnode)

{

    (1) newnode->llink = node;

    (2) newnode->rlink = node->rlink;

    (3) node->rlink->llink = newnode;

    (4) node->rlink = newnode;

}
```



DELETE

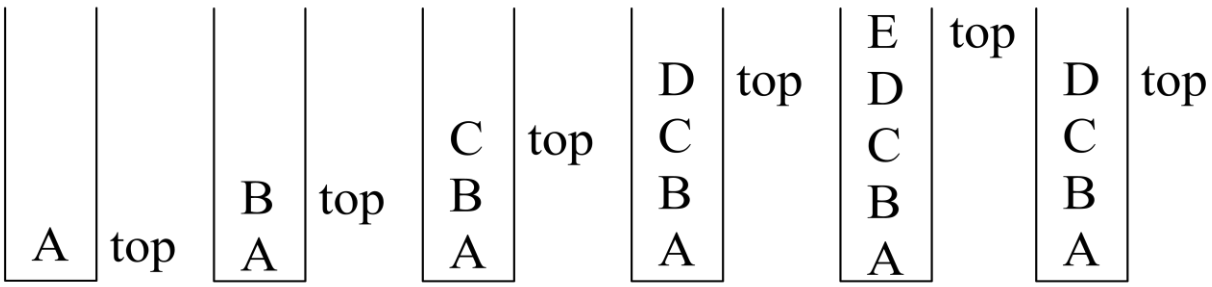
```
void ddelete(node_pointer node, node_pointer deleted)
{
   if (node==deleted)
               printf("Deletion of head node not permitted.\n");
   else {
      (1) deleted->llink->rlink= deleted->rlink;
      (2) deleted->rlink->llink= deleted->llink;
         free(deleted);
   }
}
```



STACKS AND QUEUES

STACK (STACK: A LAST-IN-FIRST-OUT (LIFO) LIST)

- Stack
  - An ordered list
  - Insertions and deletions are made at one end, called top
- Illustration



Inserting and ***deleting*** elements in a stack

APPLICATIONS OF STACK

- Implementing recursive call
- Expression evaluation
  - Infix to postfix
  - Postfix evaluation
- Maze problem
- Breadth First Search

ABSTRACT DATA TYPE FOR STACK

```
        structure Stack is
 objects: a finite ordered list with zero or more elements.
 functions:
   for all stack ∊ Stack, item ∊ element, max_stack_size
   ∊ positive integer
  Stack CreateS(max_stack_size) ::=
          create an empty stack whose maximum size is
          max_stack_size
  Boolean IsFull(stack, max_stack_size) ::=
          if (number of elements in stack == max_stack_size)
          return TRUE
          else return FALSE
  Stack Add(stack, item) ::=
          if (IsFull(stack)) stack_full
          else insert item into top of stack and return
```

```
    Boolean IsEmpty(stack) ::=
            if(stack == CreateS(max_stack_size))
               return TRUE

               else return FALSE

Element Delete(stack) ::=
              if(IsEmpty(stack)) return

                  else remove and return the item on the top of the stack
```

Structure 2.1: *Abstract data type Stack*

Implementation: using array

***Stack* CreateS(max_stack_size)** ::=

```
#define MAX_STACK_SIZE 100 /* maximum stack size */
typedef struct {
        int key;
        /* other fields */
        } element;
element stack[MAX_STACK_SIZE];
int top = -1;
```

***Boolean* IsEmpty(Stack)** ::= top< 0;

***Boolean* IsFull(Stack)** ::= top >= MAX_STACK_SIZE-1;

ADD DATA INTO STACK

```
void add(int *top, element item)
{
   if (*top >= MAX_STACK_SIZE-1)  {
       stack_full( );
       return;
   }
   stack[++*top] = item;
}
```

program : *Add to a stack*

DELETE FROM A STACK

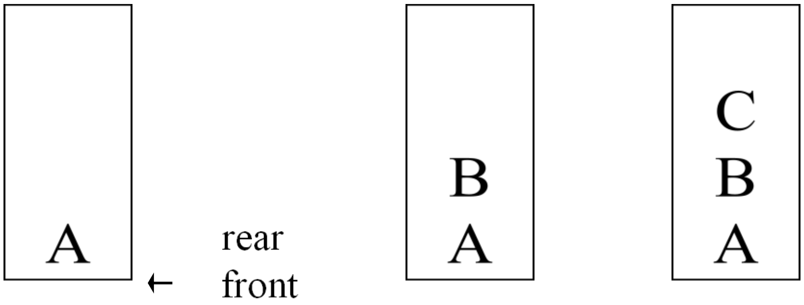
```
        element delete(int *top)
{
   if (*top == -1)
```

```
        return stack_empty( );  /* returns and error key */

     return stack[(*top)--];
 }
```

Program: *Delete from a stack*

QUEUE

- Queue
  - An ordered list
  - All insertions take place at one end, ***rear***
  - All deletions take place at the opposite end, ***front***
- Illustration



APPLICATIONS OF QUEUE

- Job scheduling
- Event list in simulator
- Server and Customs

Application: Job scheduling

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added

Figure 3.2: Insertion and deletion from a sequential queue

QUEUE ADT

```
structure Queue is
  objects: a finite ordered list with zero or more elements.
  functions:
     for all queue ∈ Queue, item ∈ element,
            max_ queue_ size ∈ positive integer
     Queue CreateQ(max_queue_size) ::=
            create an empty queue whose maximum size is
            max_queue_size
     Boolean IsFullQ(queue, max_queue_size) ::=
            if(number of elements in queue == max_queue_size)
            return TRUE
            else return FALSE
     Queue AddQ(queue, item) ::=
            if (IsFullQ(queue)) queue_full
           else insert item at rear of queue and return queue

Boolean IsEmptyQ(queue) ::=
            if (queue ==CreateQ(max_queue_size))
                   return TRUE

            else return FALSE

     Element DeleteQ(queue) ::=
            if (IsEmptyQ(queue)) return

                   else remove and return the item at front of                    queue.
```

***Structure :** Abstract data type Queue*

Implementation 1: using array

```
Queue CreateQ(max_queue_size) ::=
# define MAX_QUEUE_SIZE 100/* Maximum queue size */
typedef struct {
                int key;
                /* other fields */
                } element;
element queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = -1;
Boolean IsEmpty(queue) ::= front == rear
Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1
```

ADD TO A QUEUE

```
void addq(int *rear, element item)
{
   if (*rear == MAX_QUEUE_SIZE_1) {
      queue_full( );
      return;
   }
  queue [++*rear] = item;
}
```

Program : *Add to a queue*

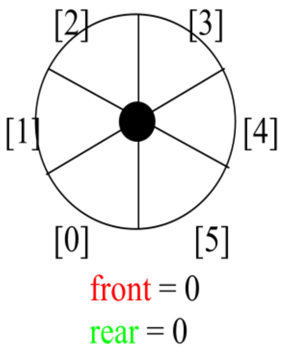
DELETE FROM A QUEUE

```
element deleteq(int *front, int rear)
{
   if ( *front == rear)
       return queue_empty( );    /* return an error key */

   return queue [++ *front];
}
```

Program : *Delete from a queue*

Implementation 2: regard an array as a circular queue

front: one position counterclockwise from the first element



Empty circular Queue

rear: current end

ADD TO A CIRCULAR QUEUE

```
void addq(int front, int *rear, element item)
{
   *rear = (*rear +1) % MAX_QUEUE_SIZE;
    if (front == *rear) /* reset rear and print error */
    return;
  }
    queue[*rear] = item;
}
```

**Program 3.5: Add to a circular queue*

DELETE FROM A CIRCULAR QUEUE

```
element deleteq(int* front, int rear)
{
  element item;
  if (*front == rear)
       return queue_empty( );
              /* queue_empty returns an error key */
    *front = (*front+1) % MAX_QUEUE_SIZE;
    return queue[*front];
}
```

Program : Delete from a circular queue

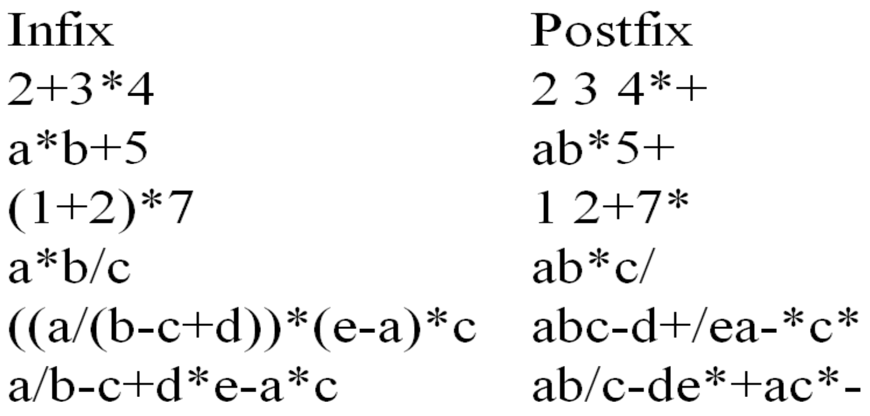
EVALUATION OF EXPRESSIONS

- Evaluating a complex expression in computer

    - ((rear+1==front)||((rear==MaxQueueSize-1)&&!front))
    - x= a/b- c+ d*e- a*c
- Figuring out the order of operation within any expression
    - A ***precedence*** hierarchy within any programming language
    - See Figure 3.12

      Evaluation of Expressions (Cont.)

- Ways to write expressions
    - Infix (standard)
    - Prefix
    - Postfix
        - compiler, a parenthesis-free notation



Infix	Postfix
2+3*4	2 3 4*+
a*b+5	ab*5+
(1+2)*7	1 2+7*
a*b/c	ab*c/
((a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*+ac*-

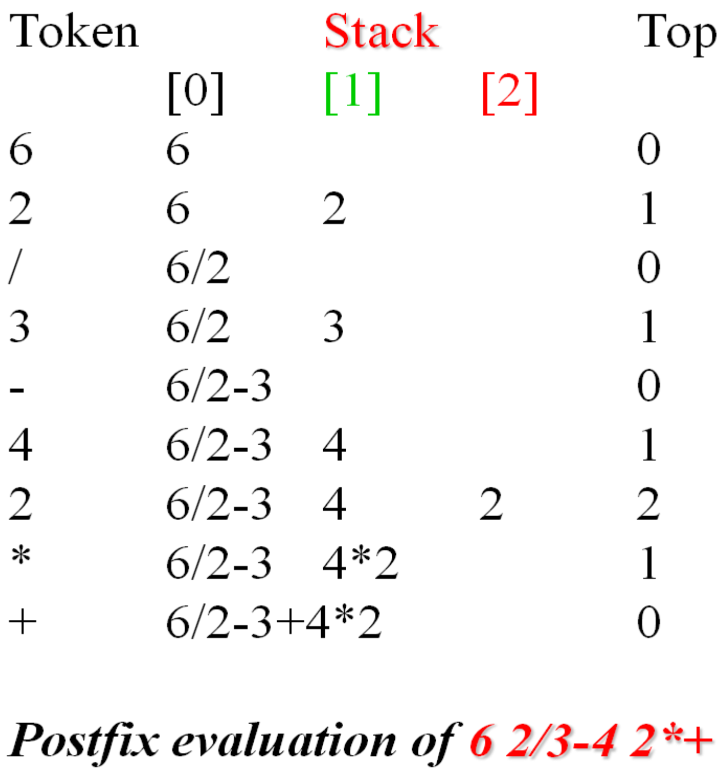
EVALUATION OF POSTFIX EXPRESSION

- Left-to-right scan Postfix expression,
    1) Stack operands until find an operator,
    2) Meet operator, remove correct operands for this operator,
    3) Perform the operation,

4) Stack the result

- Remove the answer from the top of stack

Evaluation of postfix expression



Token		Stack		Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

Postfix evaluation of ***6 2/3-4 2*+***

Assumptions:

operators: +, -, *, /, %

operands: single digit integer

```
#define MAX_STACK_SIZE 100
#define MAX_EXPR_SIZE 100 /* max size of expression */
typedef enum{1paran, rparen, plus, minus, times, divide,
                        mod, eos, operand} precedence;
int stack[MAX_STACK_SIZE];  /* global stack */
char expr[MAX_EXPR_SIZE];  /* input string

int eval(void)
{
  precedence token;
  char symbol;
  int op1, op2;
  int n = 0;  /* counter for the expression string */
  int top = -1;
  token = get_token(&symbol, &n);
  while (token != eos)  {
      if (token == operand)
           add(&top, symbol-’0’);   /* stack add */

else {
            /* remove two operands, perform operation, and
                return result to the stack */
       op2 = delete(&top);  /* stack delete */
       op1 = delete(&top);
       switch(token) {
            case plus: add(&top, op1+op2); break;
            case minus: add(&top, op1-op2); break;
            case times: add(&top, op1*op2); break;
            case divide: add(&top, op1/op2); break;
            case mod: add(&top, op1%op2);
       }
    }
    token = get_token (&symbol, &n);
 }
 return delete(&top); /* return result */
}
```

Program: *Fuprecedence get_token(char *symbol, int *n)*

```
{
  *symbol =expr[(*n)++];
  switch (*symbol)  {
     case ‘(‘ : return lparen;
     case ’)’ : return rparen;
     case ‘+’: return plus;
     case ‘-’ : return minus;
     case ‘/’ :  return divide;
      case ‘*’ : return times;
     case ‘%’ : return mod;
     case ‘\0‘ : return eos;
     default  : return operand;
     }
}
```

Program : Function to get a token from the input string (p.123)nction to evaluate a postfix expression

INFIX TO POSTFIX

1) Method I

    1) Fully parenthesize the expression

    2) Move all binary operators so that they replace their corresponding right parentheses

    3) Delete all parentheses

- Examples:***a/b-c+d*e-a*c***

    - ((((a/b)-c)+(d*e))-(a*c)), fully parentheses

    - ab/c-de*+ac*-, replace right parentheses and delete all parentheses

- Disadvantage

    - inefficient, two passes

- Method II

    - scan the infix expression left-to-right

    - output operand encountered

    - output operators depending on their precedence, i.e., higher precedence operators first

- Example: *a+b*c*, simple expression

Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*

- Example: ***a*(b+c)*d*** , parenthesized expression

Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc+
*	*			0	abc+*
d	*			0	abc+*d
eos	*			0	abc+*d*

Infix to Postfix

- Last two examples suggests a precedence-based scheme for stacking and unstacking operators
  - **isp** (in-stack precedence)
  - **icp** (iprecedence stack[MaxStackSize];
  - /* isp and icp arrays - index is value of precedence
  - lparen, rparen, plus, minus, time divide, mod, eos */
  - *static int* **isp**[]= { 0, 19, 12, 12, 13, 13, 13, 0};
  - *static int* **icp**[]= {20, 19, 12, 12, 13, 13, 13, 0}n-coming precedence)

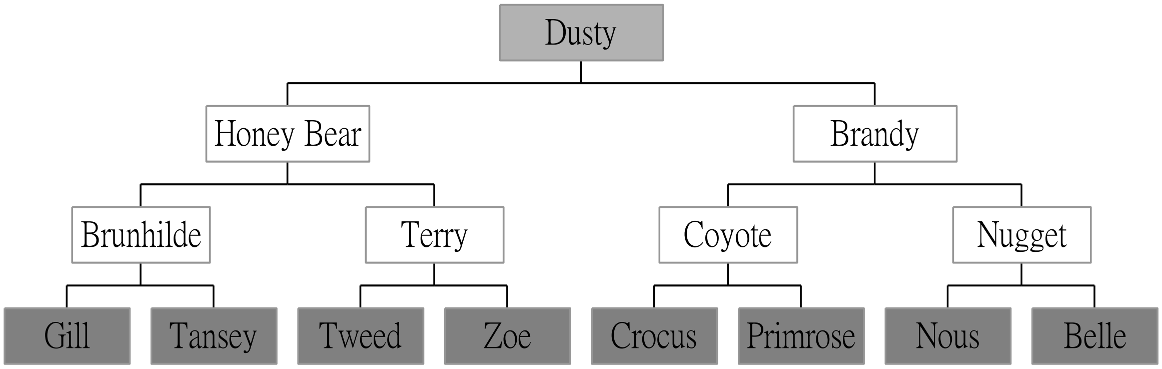
```
void postfix(void)
{
/* output the postfix of the expression. The expression
   string, the stack, and top are global */
  char symbol;
  precedence token;
  int n = 0;
  int top = 0; /* place eos on stack */
  stack[0] = eos;
```

```
   for (token = get _token(&symbol, &n); token != eos;
                       token = get_token(&symbol, &n)) {
     if (token == operand)
        printf ("%c", symbol);
     else if (token == rparen ){

      /*unstack tokens until left parenthesis */
      while (stack[top] != lparen)
          print_token(delete(&top));
      delete(&top); /*discard the left parenthesis */
     }
     else{
      /* remove and print symbols whose isp is greater
          than or equal to the current token's icp */
      while(isp[stack[top]] >= icp[token] )
          print_token(delete(&top));
      add(&top, token);
     }
  }
  while ((token = delete(&top)) != eos)
       print_token(token);
  print("\n");
}
```

***Program** : Function to convert from infix to postfix*

UNIT III

Trees



DEFINITION OF A TREE

- A tree is a finite set of one or more nodes such that:
- There is a specially designated node called the root.
- The remaining nodes are partitioned into n>=0 disjoint sets T1, ..., Tn, where each of these sets is a tree.
- We call T1, ..., Tn the subtrees of the root.

TERMINIOLOGY

Node, Degree of a node, Leaf (terminal), Nonterminal, Parent, Children, Sibling, Degree of a tree, Ancestor,Level of a node

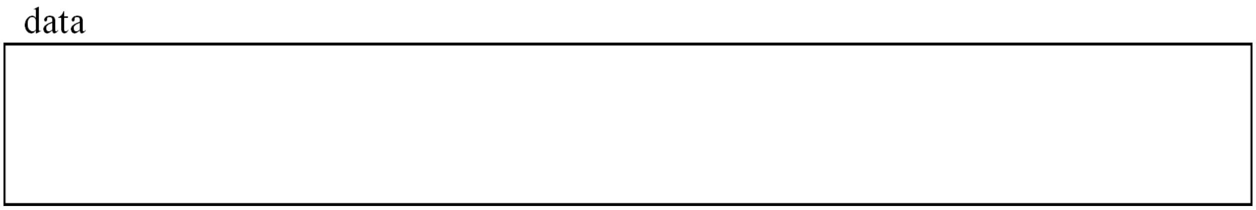
- Height of a tree The degree of a node is the number of subtrees of the node
    - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*.

- The ancestors of a node are all the nodes along the path from the root to the node.

REPRESENTATION OF TREES

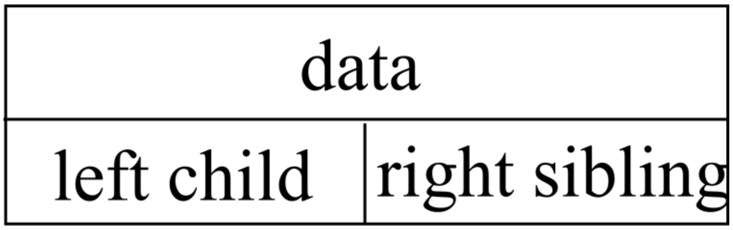
List Representation

- (A (B (E (K, L), F), C (G), D (H (M), I, J)))
- The root comes first, followed by a list of sub-trees



How many link fields are needed in such a representation?

LEFT CHILD-RIGHT CHILD SIBLING



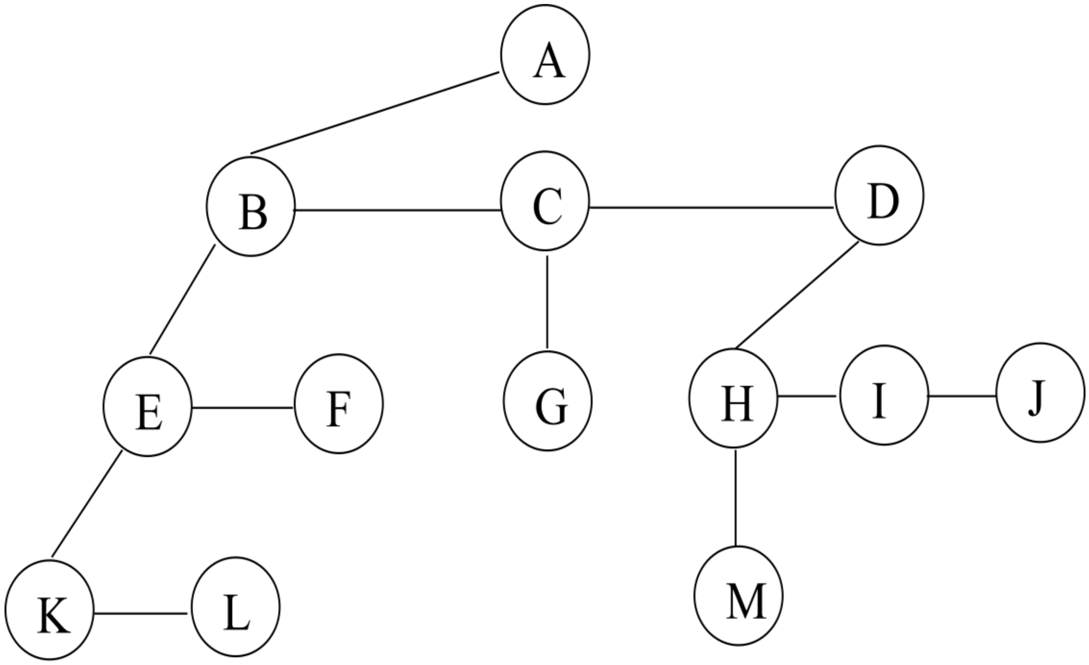
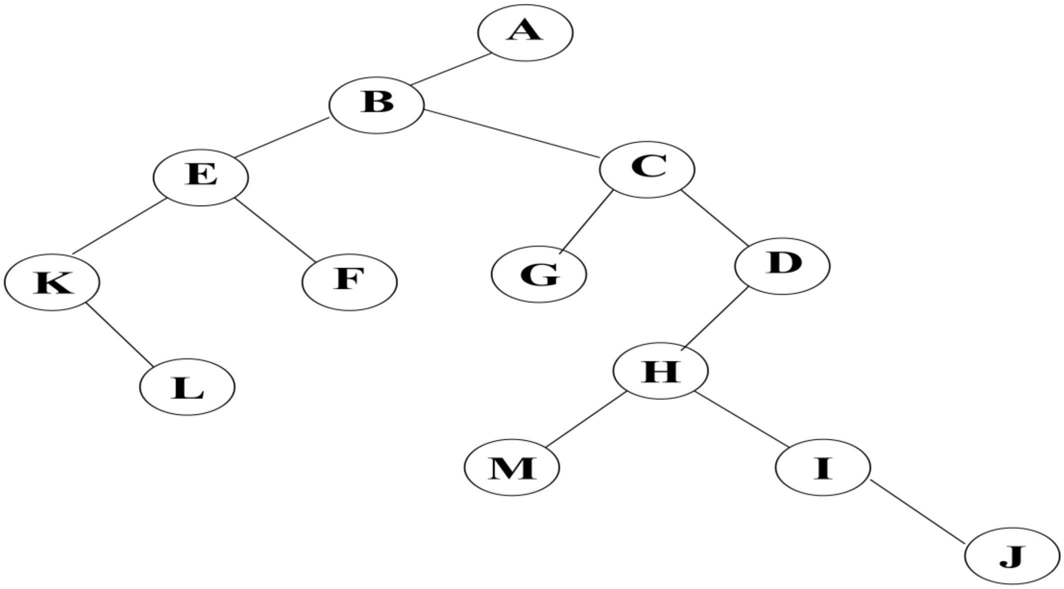


FIG: Left child and right child representation of a tree

BINARY TREES

- A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation
- The left subtree and the right subtree are distinguished.



BINARY TREE ADT

structure Binary_Tree(abbreviated BinTree) is objects: a finite set of nodes either empty or consisting of a root node, left Binary_Tree, and right Binary_Tree.

functions:

for all bt, bt1, bt2 ∈ BinTree, item ∈ element

Bintree Create() ::= creates an empty binary tree

Boolean IsEmpty(bt) ::= if (bt==empty binary tree) return TRUE else return FALSE

BinTree MakeBT(bt1, item, bt2)::= return a binary tree

whose left subtree is bt1, whose right subtree is bt2,

and whose root node contains the data item

Bintree Lchild(bt)::= if (IsEmpty(bt)) return error
else return the left subtree of bt

element Data(bt)::= if (IsEmpty(bt)) return error
else return the data in the root node of bt

Bintree Rchild(bt)::= if (IsEmpty(bt)) return error
else return the right subtree of bt

Samples of Trees

Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1.
- The maximum nubmer of nodes in a binary tree
  of depth k is 2^k-1, k>=1.

Relations between Number of Leaf Nodes and Nodes of Degree 2

For any nonempty binary tree, T, if n0 is the number of leaf nodes and n2 the number of nodes of degree 2, then n0=n2+1

proof:

Let n and B denote the total number of nodes & branches in T.

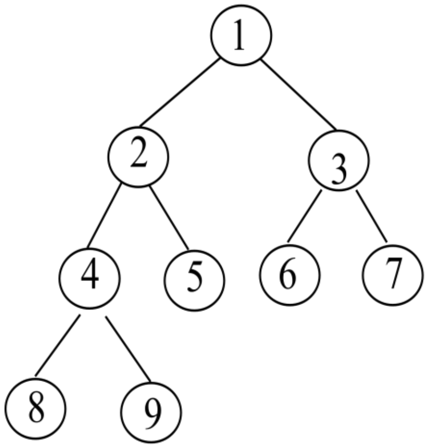
Let n0, n1, n2 represent the nodes with no children, single child, and two children respectively.

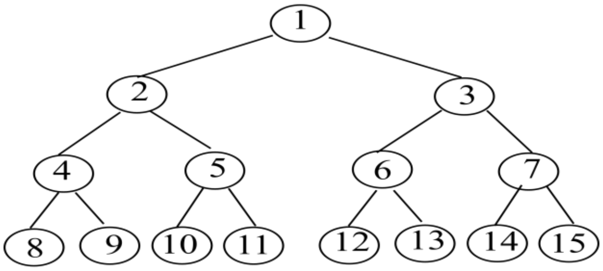
n= n0+n1+n2, B+1=n, B=n1+2n2 ==> n1+2n2+1= n,

n1+2n2+1= n0+n1+n2 ==> n0=n2+1

Full BT VS Complete BT

- A full binary tree of depth k is a binary tree of depth k having 2 -1 nodes, k>=0.
- A binary tree with n nodes and depth k is complete iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.





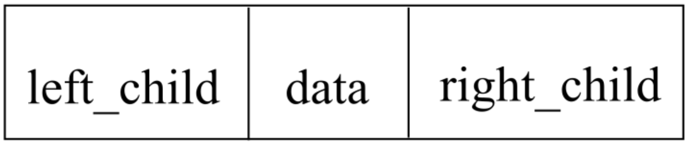
Full binary tree of depth 4

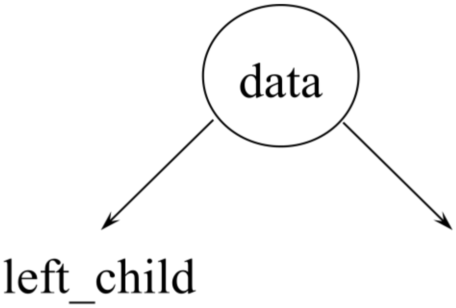
BINARY TREE REPRESENTATIONS

- If a complete binary tree with *n* nodes (depth =log *n* + 1) is represented sequentially, then for any node with index *i*, 1<=*i*<=*n*, we have:
    - *parent*(*i*) is at *i*/2 if *i*!=1. If *i*=1, *i* is at the root and has no parent.
    - *left_child*(*i*) ia at 2*i* if 2*i*<=*n*. If 2*i*>n, then *i* has noleft child.
    - *right_child*(*i*) ia at 2*i*+1 if 2*i* +1 <=*n*. If 2*i* +1 >n, then *i* has no right child.
- Sequential Representation

Linked Representation

```
typedef struct node *tree_pointer;

typedef struct node {

                              int data;

                            tree_pointer left_child, right_child;

};
```





BINARY TREE TRAVERSALS

Let L, V, and R stand for moving left, visiting the node, and moving right.

There are six possible combinations of traversal

LVR, LRV, VLR, VRL, RVL, RLV

Adopt convention that we traverse left before right, only 3 traversals remain

LVR, LRV, VLR

inorder, postorder, preorder

Inorder Traversal (recursive version)

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
   if (ptr) {
      inorder(ptr->left_child);
      printf(“%d”, ptr->data);
      inorder(ptr->right_child);
   }
}
```

PREORDER TRAVERSAL (RECURSIVE VERSION)

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf("%d", ptr->data);
        preorder(ptr->left_child);
        preorder(ptr->right_child);
    }
}
```

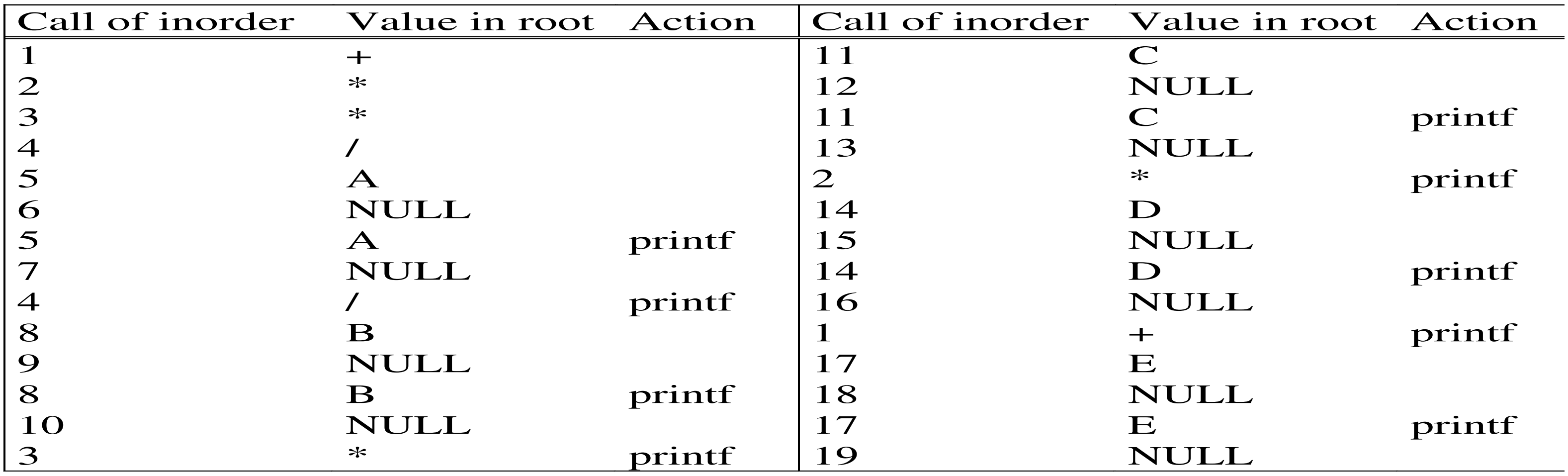
Postorder Traversal (recursive version)

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->left_child);
        postorder(ptr->right_child);
        printf("%d", ptr->data);
    }
}
```

Iterative Inorder Traversal (using stack)

```
void iter_inorder(tree_pointer node)
{
  int top= -1; /* initialize stack */
  tree_pointer stack[MAX_STACK_SIZE];
  for (;;) {
   for (; node; node=node->left_child)
     add(&top, node);/* add to stack */
   node= delete(&top);
                /* delete from stack */
   if (!node) break; /* empty stack */
   printf(“%d”, node->data);

   node = node->right_child;

  }

 }
```

Trace Operations of Inorder Traversal



Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

LEVEL ORDER TRAVERSAL (USING QUEUE)

```
void level_order(tree_pointer ptr)

/* level order tree traversal */

{

  int front = rear = 0;

  tree_pointer queue[MAX_QUEUE_SIZE];

  if (!ptr) return; /* empty queue */

  addq(front, &rear, ptr);

  for (;;) {

    ptr = deleteq(&front, rear);

if (ptr) {

      printf(“%d”, ptr->data);
```

```
     if (ptr->left_child)
       addq(front, &rear,
                    ptr->left_child);
     if (ptr->right_child)
       addq(front, &rear,
                    ptr->right_child);
   }
   else break;
 }
                                                      }
```

NON RECURSIVE BINARY TREE TRAVERSALS

NONRECURSIVE INORDER TRAVERSAL: GENERAL ALGORITHM

```
  1. current = root;  //start traversing the binary tree at
                 // the root node
  2. while(current is not NULL or stack is nonempty)
   if(current is not NULL)
   {
     push current onto stack;
     current = current->llink;
   }
   else
   {
     pop stack into current;
     visit current;    //visit the node
     current = current->rlink;     //move to the
                              //right child
  }
```

NONRECURSIVE PREORDER TRAVERSAL

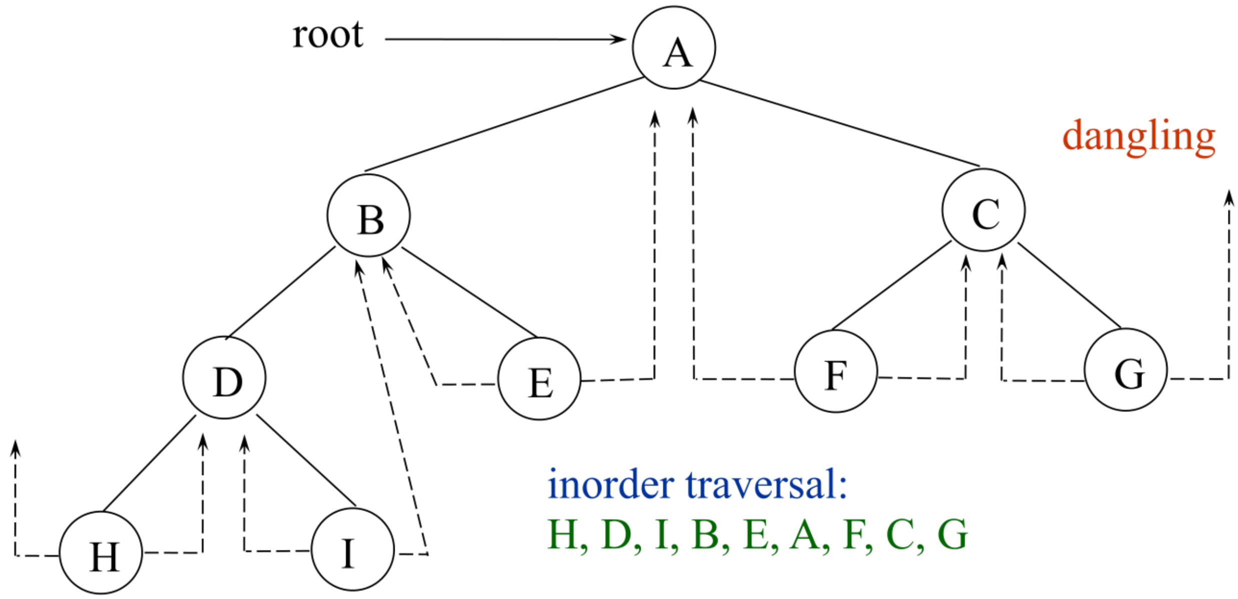
```
1. current = root;  //start the traversal at the root node
2. while(current is not NULL or stack is nonempty)
  if(current is not NULL)
  {
    visit current;
    push current onto stack;
    current = current->llink;
  }
  else
  {
    pop stack into current;
    current = current->rlink;    //prepare to visit
                                 //the right subtree
  }
```

NONRECURSIVE POSTORDER TRAVERSAL

```
1. current = root;  //start traversal at root node
2. v = 0;
3. if(current is NULL)
          the binary tree is empty
4. if(current is not NULL)
      a. push current into stack;
      b. push 1 onto stack;
      c.  current = current->llink;
      d.  while(stack is not empty)
    if(current is not NULL and v is 0)
    {
    push current and 1 onto stack;
    current = current->llink;
          }
else
  {
            pop stack into current and v;
            if(v == 1)
            {
                push current and 2 onto stack;
            current = current->rlink;
            v = 0;
    }
    else
            visit current;
   }
```

THREADED BINARY TREES

- Two many null pointers in current representation of binary trees n: number of nodes number of non-null links: n-1 total links: 2n null links: 2n-(n-1)=n+1
- Replace these null pointers with some useful “threads”.
- If ptr->left_child is null, replace it with a pointer to the node that would be visited *before* ptr in an *inorder traversal*
- If ptr->right_child is null, replace it with a pointer to the node that would be visited *after* ptr in an *inorder traversal*



HEAP

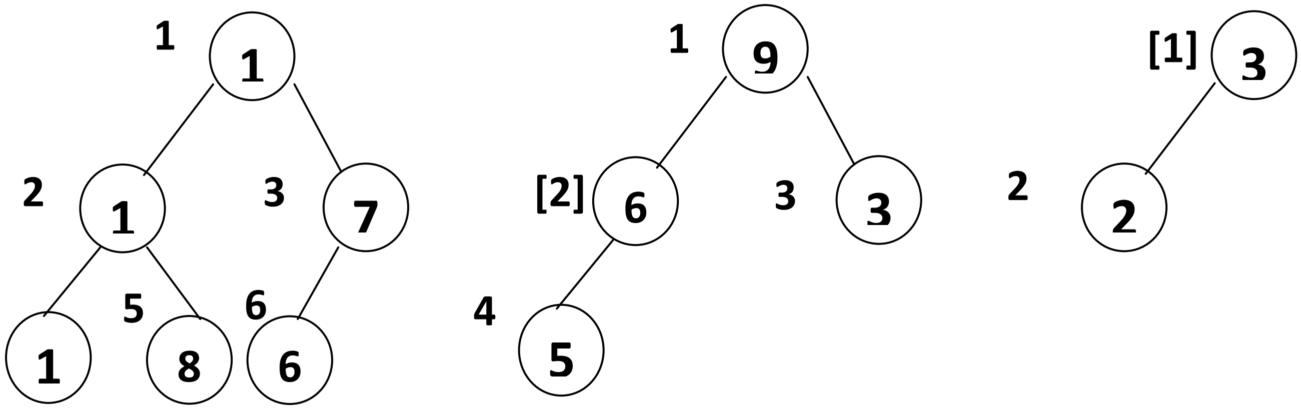
A max tree is a tree in which the key value in each node is no smaller than the key values in its children. A max heap is a complete binary tree that is also a max tree.A min tree is a tree in which the key value in each node is no larger than the key values in its children. A min heap is a complete binary tree that is also a min tree.

Operations on heaps

- creation of an empty heap
- insertion of a new element into the heap;
- deletion of the largest element from the heap

\

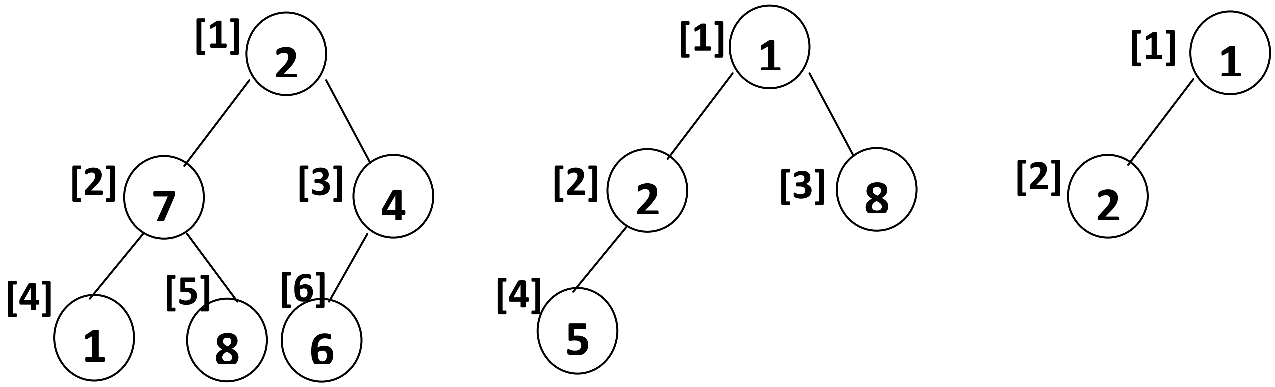
Fig Sample max heaps



Property:

The root of max heap (min heap) contains the largest (smallest).

Fig: Sample min heaps



ADT for Max Heap

structure MaxHeap

objects: a complete binary tree of n > 0 elements organized so that the value in each node is at least as large as those in its children

functions:

for all heap belong to MaxHeap, item belong to Element, n, max_size belong to integer

MaxHeap Create(max_size)::= create an empty heap that can
hold a maximum of max_size elements

Boolean HeapFull(heap, n)::= if (n==max_size) return TRUE
else return FALSE

MaxHeap Insert(heap, item, n)::= if (!HeapFull(heap,n)) insert
item into heap and return the
resulting heap else return error

Boolean HeapEmpty(heap, n)::= if (n>0) return FALSE

else return TRUE

Element Delete(heap,n)::= if (!HeapEmpty(heap,n)) return one
instance of the largest element in the heap
and remove it from the heap

else return error

Application: priority queue

- machine service
  - amount of time (min heap)
  - amount of payment (max heap)
- factory
  - time tag

Data Structures

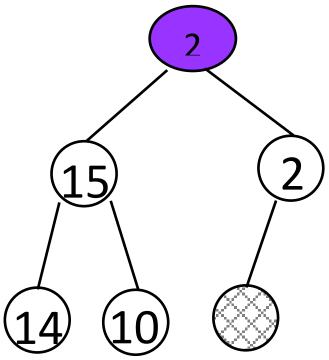
- unordered linked list
- unordered array
- sorted linked list
- sorted array
- heap

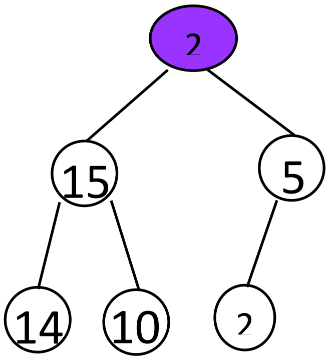
Fig: Priority queue representations

Representation	Insertion	Deletion
Unordered array	$\Theta(1)$	$\Theta(n)$
Unordered linked list	$\Theta(1)$	$\Theta(n)$
Sorted array	$O(n)$	$\Theta(1)$
Sorted linked list	$O(n)$	$\Theta(1)$
Max heap	$O(\log_2 n)$	$O(\log_2 n)$

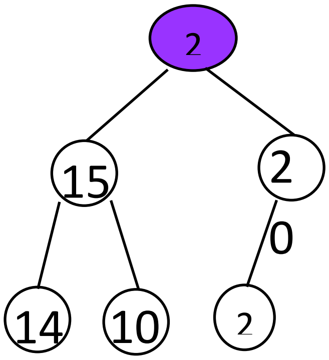
Example of Insertion to Max Heap

initial location of new node





insert 5 into heap

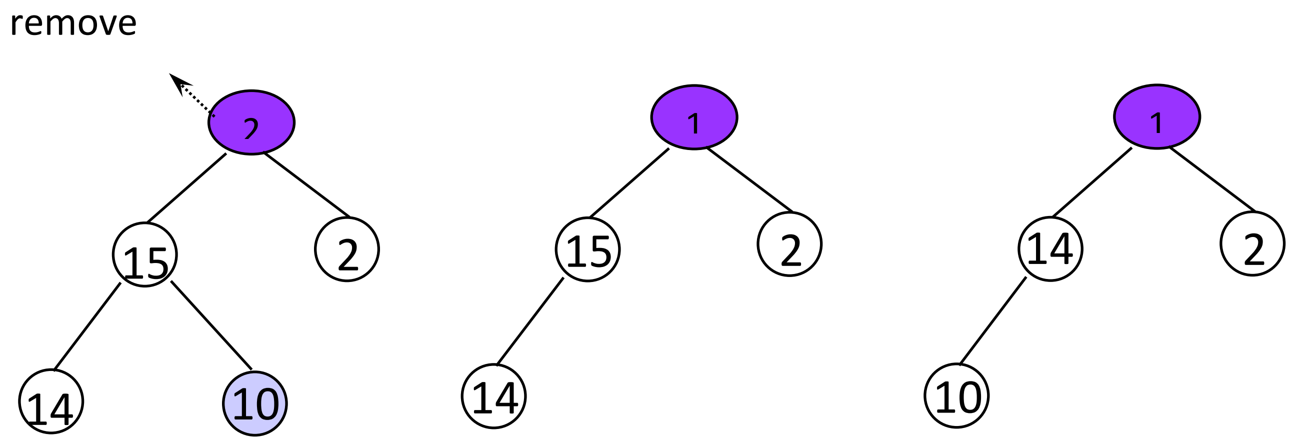


insert 21 into heap

INSERTION INTO A MAX HEAP

```
void insert_max_heap(element item, int *n)
{
  int i;
  if (HEAP_FULL(*n)) {
    fprintf(stderr, “the heap is full.\n”);
    exit(1);
  }
  i = ++(*n);
  while ((i!=1)&&(item.key>heap[i/2].key)) {
    heap[i] = heap[i/2];
    i /= 2;
  }
  heap[i]= item;
}
```

Example of Deletion from Max Heap



(a) Heap structure (b) 10 inserted at the root (c) Finial heap

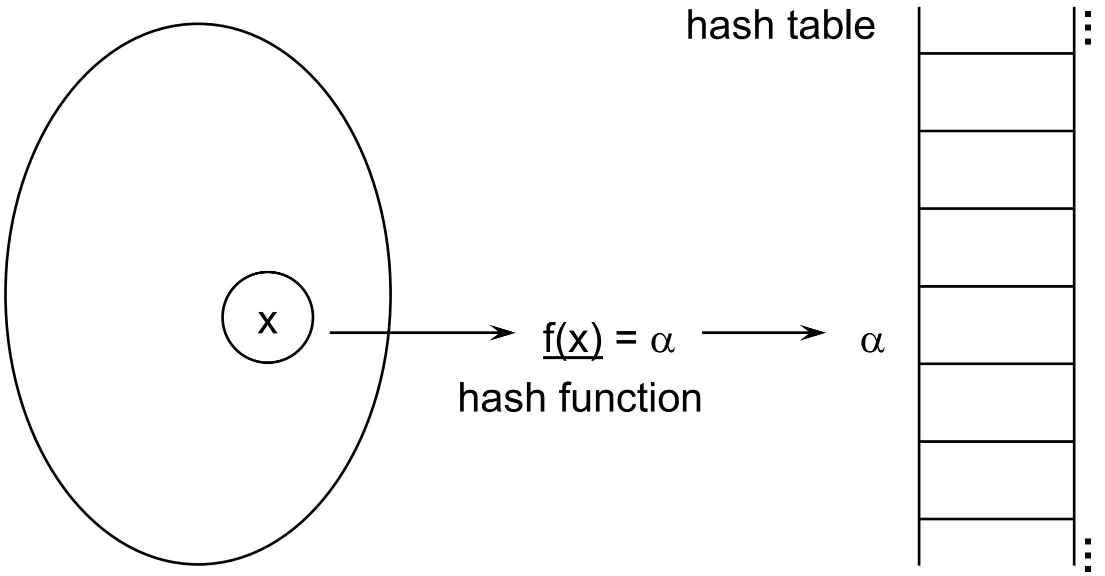
Deletion from a Max Heap

```
element delete_max_heap(int *n)
{
  int parent, child;
  element item, temp;
  if (HEAP_EMPTY(*n)) {
    fprintf(stderr, “The heap is empty\n”);
    exit(1);
  }
  /* save value of the element with the
   highest key */
  item = heap[1];
  /* use last element in heap to adjust heap */
  temp = heap[(*n)--];
  parent = 1;
  child = 2;
while (child <= *n) {
    /* find the larger child of the current
       parent */
    if ((child < *n)&&
        (heap[child].key<heap[child+1].key))
      child++;
    if (temp.key >= heap[child].key) break;
    /* move to the next lower level */
    heap[parent] = heap[child];
    child *= 2;
  }
  heap[parent] = temp;
  return item;
}
```

use hashing

- very good expected performance: O(1)

- **Static Hashing**

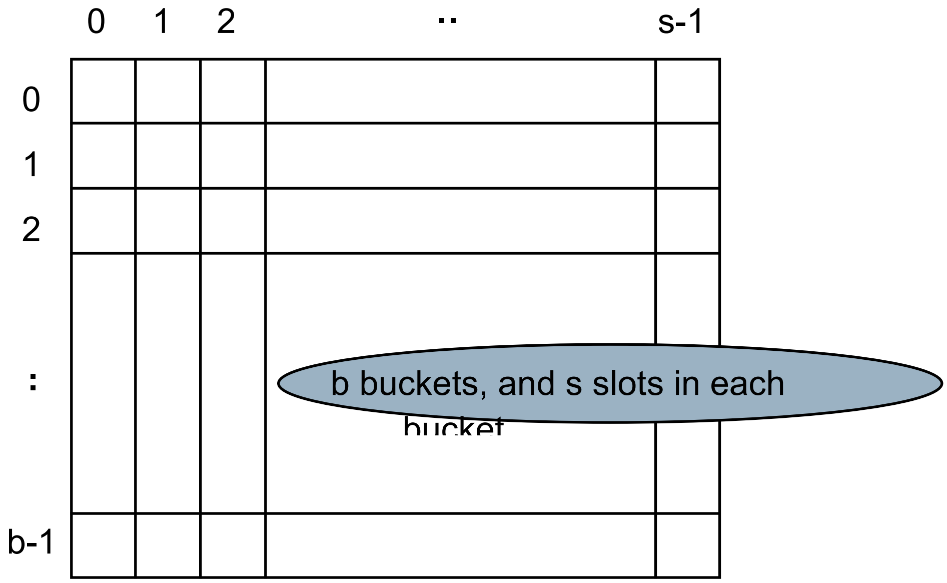


- **Hash Table**

hash tables

- store the identifiers in a fixed

size table called a hash table



- **Hash Table**

Def)

- identifier density of a hash table:

n/T where

n: number of identifiers in table

T: total number of possible

identifiers

- loading density or loading factor

of a hash table:

$a = n/(s \cdot b)$ where

s: number of slots in each bucket

b: number of bucket

- **Hash Table**

- two identifiers i1 and i2 are *synonyms* with respect to f, if f(i1) = f(i2) where i1 ¹ i2

- an *overflow* occurs when we hash a new identifier, i, into a full bucket

- a *collision* occurs when we hash two nonidentical identifiers into the same bucket

  - collisions and overflows occur simultaneously iff bucket size is 1

- **Hash Table**

Example) hash table *ht* with *b*=26, *s*=2, n=10

<u>hash function f</u>

- 1st character of identifier

	slot 0	slot 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
•••		
25		

Identifiers

acos

define

float

exp

char

atan

ceil

floor

clock

ctime

hash table with 26 bucket and two slots per bucket

Hash Function

requirements for a hash function

- easy to compute

- minimizes the number of collision (but, we can not avoid collisions)

uniform hash function

- for randomly chosen x from the identifier space, P[f(x)=i] = 1/b, for all buckets i

- a random x has an equal chance of hashing into any of the b buckets

mid-square

- middle of square hash function- frequently used in symbol table applications

hash function fm

1)squaring the identifier

2)obtain the bucket address by using an appropriate number of bits from the middle of the square

3)if we use r bits, 2r buckets are necessary

division(modular)

- use the modulus(%) operator

fD(x) = x % M

where M: table size

- range of bucket address: 0 ~ M-1

- the choice of M is critical

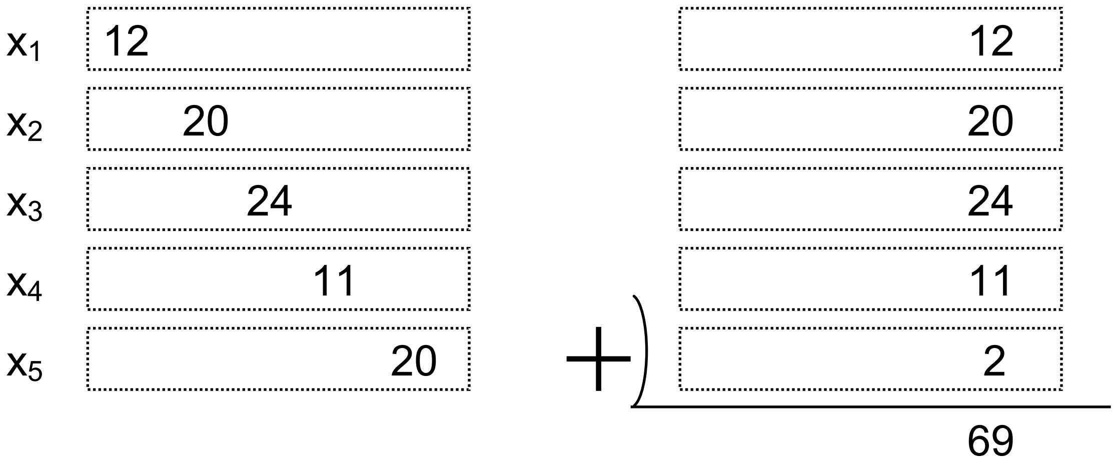
- choose M as a prime number such that M does not divide rk±a for small k and a

- choose M such that it has no prime divisors less than 20

folding

1)shift folding

ex) identifier x = 12320324111220



2)folding at the boundaries

→ 123 + 302 + 241 + 211 + 20 = 897

digit analysis

- used in case all the identifiers are known in advance

- examine the digits of each identifier

- delete those digits that have skewed distributions

- select the digit positions to be used to calculate the hash address

- **Overflow Handling**

<u>linear open addressing</u>

1) linear probing

- when overflow occurs, linear search for the empty slot in the hash table using circular rotation

linear probing

- represent hash table as a

one-dimensional array

```
#define MAX_CHAR 10
/* max number of characters in an identifier */
#define TABLE_SIZE 13
/* max table size = prime number*/
typedef struct {
   char key[MAX_CHAR];
   /* other filed */
} element;
element hash_table[TABLE_SIZE];
```

initialize the table

- allow overflows and collisions to be detected
- all slots to empty(null) string

```
void init_table(element ht[]) {
int i;
for (i = 0; i < TABLE_SIZE; i++) {
 ht[i].key[0] = NULL;
}
```

initialization of a hash table

to insert an element, transform a key into a number and calculate hash address

```
int transform(char *key) {
/* simple additive approach to create a natural
   number that is within the integer range */
   int number = 0;
   while (*key)
      number += *key++;
   return number;
}

int hash(char *key) {
/* calculate hash address */
   return(transform(key) % TABLE_SIZE);
}
```

creation of a hash function

insert element into the hash table

- find another bucket if the new
  element is hashed into a full
  bucket: linear probing

Example) b = 13, s = 1

[0]	function
[1]	
[2]	for
[3]	do
[4]	while
[5]	
[6]	
[7]	
[8]	
[9]	else
[10]	
[11]	
[12]	if

hash table with linear probing

(13 buckets, 1 slot/bucket)

4 cases in insertion process

examine the hash table buckets- ht[(f(x)+j) % TABLE_SIZE], where

0 £ j £ TABLE_SIZE1)the bucket contains x

- simply report a duplicate identifier

- update information in the other fields of the element

2)the bucket contains the empty string

- bucket is empty, and insert the new element into it

3)the bucket contains a nonempty string other than x

- proceed to examine the next bucket

4)return to the home bucket ht[f(x)](j = TABLE_SIZE)

- the home bucket is being examined for the second time and all remaining buckets have been examined

- report an error condition and exit

```
void linear_insert(element item, element ht[]) {
```

```
/* insert the key into the table using the linear
   probing technique, exit the function if the table
   is full */
   int i, hash_value;
   hash_value = hash(item.key);
   i = hash_value;
   while (strlen(ht[i].key)) {
      if (!strcmp(ht[i].key, item.key)) {
         fprintf(stderr, ”duplicate entry\n”);
         exit(1);
      }
      i = (i + 1) % TABLE_SIZE;
      if (i == hash_value) {
         fprintf(stderr,”the table is full\n”);
         exit(1);
      }
   }
   ht[i] = item;
}
```

linear insert into a hash table

characteristics of linear probing to
resolve overflow

- identifiers tend to cluster together
- increases the search time

Ex) enter the C built-in functions into
a 26-bucket hash table in the order
"acos, atoi, char, define, exp,
ceil, cos, float, atol, floor,
ctime"

- b = 26, s = 1

bucket	x	# of comparisons
0	acos	1
1	atoi	2
2	char	1
3	define	1
4	exp	1
5	ceil	4
6	cos	5
7	float	3
8	atol	9
9	floor	5
10	ctime	9
···		
25		

hash table with linear probing(26 buckets, 1 slot/bucket) cluster of identifiers in linear probing

- tend to merge as more identifiers is entered into the table
- bigger cluster

solutions

- quadratic probing
- random probing
- rehashing

2) quadratic probing

- examine the hash table buckets

ht[f(x)],

ht[(f(x) + i2) % b],

ht[(f(x) - i2) % b],

for 0 £ i £ (b-1)/2,

where

b: number of buckets in the table

- reduce the average number of probes

3) rehashing

- use a series of hashing functions

f1, f2, ··· , fb

- bucket fi(x) is examined for

i = 1, 2, ⋯ , b

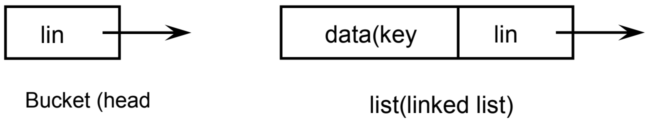
chaning

defect of linear probing

- comparison of identifiers with different hash values

maintain list of identifiers

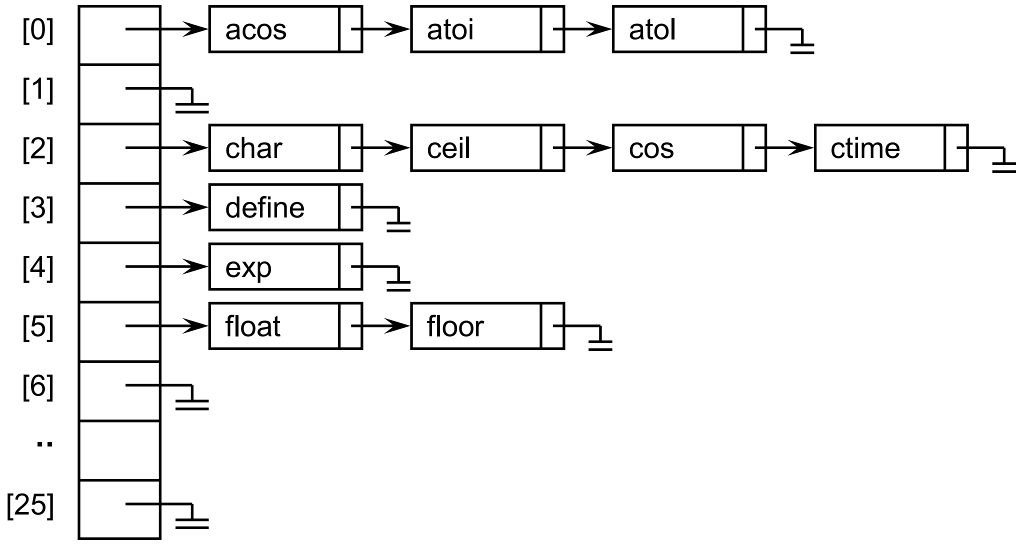
- one list per one bucket
- each list has all the synonyms
- requires a head node for each chain



```
#define MAX_CHAR 10
#define TABLE_SIZE 13
#define IS_FULL(ptr) (!(ptr))
typedef struct {
  char key[MAX_CHAR];
  /* other fields */
} element;

typedef struct list *list_ptr;
typedef struct list {
  element item;
  list_ptr link;
}
list_ptr hash_table[TABLE_SIZE];
void chain_insert(element item, list_ptr ht[]) {
  int hash_value = hash(item.key);
  list_ptr ptr, trail = NULL;
  list_ptr lead = ht[hash_value];
  for (; lead; trail=lead, lead = lead->link)
     if (!strcmp(lead->item.key, item.key)) {
       fprintf(stderr,”the key is in the table\n”);
       exit(1);
     }
  }
  ptr = (list_ptr)malloc(sizeof(list));
  if (IS_FULL(ptr)) {
    fprintf(stderr,“the memory is full\n”);
    exit(1);
  }
  ptr->item = item;
  ptr->link = NULL;
  if (trail) trail->link = ptr;
  else ht[hash_value] = ptr;
}
```

chain insert into a hash table

- hash chains



List Verification

- Compare lists to verify that they are identical or identify the discrepancies.
- example
  - international revenue service (e.g., employee vs. employer)
- complexities
  - random order: O(mn)
  - ordered list:
    O(tsort(n)+tsort(m)+m+n)

***Program 4.3:** verifying using a sequential search

```
void verify1(element list1[], element list2[ ], int n, int m)
/* compare two unordered lists list1 and list2 */
{
int i, j;
int marked[MAX_SIZE];

for(i = 0; i<m; i++)
   marked[i] = FALSE;
for (i=0; i<n; i++)
  if ((j = seqsearch(list2, m, list1[i].key)) < 0)
```

```
     printf("%d is not in list 2\n ", list1[i].key);
 else
 /* check each of the other fields from list1[i] and list2[j], and print out any discrepancies */

marked[j] = TRUE;
for ( i=0; i<m; i++)
      if (!marked[i])
          printf("%d is not in list1\n", list2[i]key);
}
```

Program : Fast verification of two lists

```
void verify2(element list1[ ], element list2 [ ], int n, int m)
/* Same task as verify1, but list1 and list2 are sorted */
{
   int i, j;
   sort(list1, n);
   sort(list2, m);
   i = j = 0;
   while (i < n && j < m)
        if  (list1[i].key < list2[j].key) {
             printf ("%d is not in list 2 \n", list1[i].key);
             i++;
        }
        else if (list1[i].key == list2[j].key) {
        /* compare list1[i] and list2[j] on each of the other field
           and report any discrepancies */
        i++; j++;
        }

else {
     printf("%d is not in list 1\n", list2[j].key);
     j++;
 }
for(; i < n; i++)
     printf ("%d is not in list 2\n", list1[i].key);
for(; j < m; j++)
     printf("%d is not in list 1\n", list2[j].key);
}
```

UNIT IV

SEARCHING AND SORTING

SEQUENTIAL SEARCH (LINEAR SEARCH)

Example
44, 55, 12, 42, 94, 18, 06, 67

unsuccessful search

- n+1

successful search

$$\sum_{i=0}^{n-1} (i+1)/n = \frac{n+1}{2}$$

```
# define MAX-SIZE 1000/* maximum size of  list  plus one */
typedef struct {
            int key;
            /* other fields */
            } element;
element list[MAX_SIZE];
```

__*Program 4.1:__ <u>Sequential search</u>

```
int seqsearch( int list[ ], int searchnum, int n )
{
/*search an array, list, that has n numbers. Return i, if list[i]=searchnum. Return -1, if searchnum
is not in the list */
    int i;
    list[n]=searchnum;    sentinel
    for (i=0; list[i] != searchnum; i++)
       ;
    return (( i<n) ? i : -1);
}
```

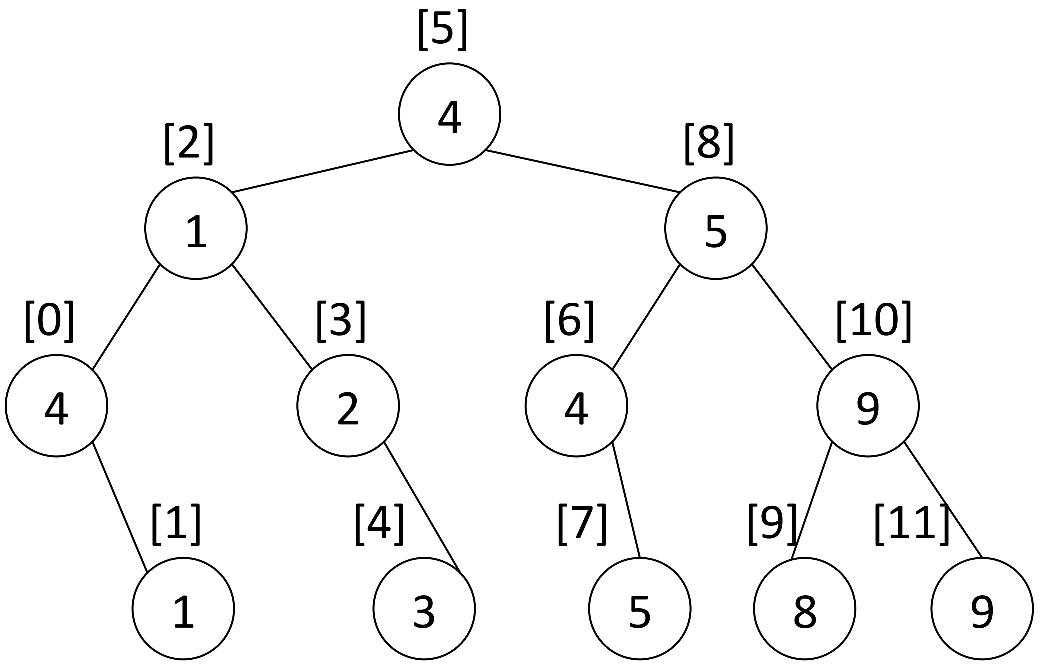
BINARY SEARCH

Program : Binary search

```
int binsearch(element list[ ], int searchnum, int n)
{
/* search list [0], ..., list[n-1]*/
   int left = 0, right = n-1, middle;
   while (left <= right) {
       middle = (left+ right)/2;
   switch (COMPARE(list[middle].key, searchnum)) {
       case -1: left = middle +1;
                   break;
       case  0: return middle;
       case  1:right = middle - 1;
        }
      }
     return -1;
}
```

$O(\log_2 n)$

Figure : Decision tree for binary search



4, 15, 17, 26, 30, 46, 48, 56, 58, 82, 90, 95

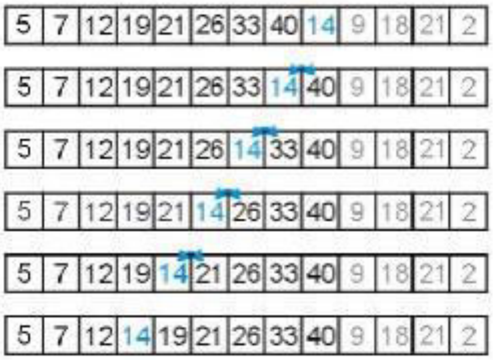
INSERTION SORT

Suppose we have a list of size $k - 1$ that is already sorted. We can easily insert a kth new object into that list by starting at the back and moving items over until we find a location for it. For example, the list in Figure 1 has the first eight objects already sorted.

5	**7**	**12**	**19**	**21**	**26**	**33**	**40**	**14**	**9**	**18**	**21**	**2**

Figure 1. A sorted list of eight objects.

If we were to insert 14 into this sorted list of eight objects, we could would proceed backward: swap 14 and 40, then with 33, 26, 21, and 19. At this point, 12 < 14, so we are finished. This is quickly shown in Figure 2.



5 7 12 19 21 26 33 40 14 9 18 21 2

5 7 12 19 21 26 33 14 40 9 18 21 2

5 7 12 19 21 26 14 33 40 9 18 21 2

5 7 12 19 21 14 26 33 40 9 18 21 2

5 7 12 19 14 21 26 33 40 9 18 21 2

5 7 12 14 19 21 26 33 40 9 18 21 2

Figure 2. Inserting 14 into the sorted list of eight objects.

This creates a list of size 9. We can then proceed by inserting the next object, 9, into this sorted list.

THE INSERTION SORT ALGORITHM

The algorithm form insertion sort is:

1.Given a list of *n* items, treat the first item to be a sorted list of size 1.

2.Then, for *k* from 1 to *n* – 1:

a.Insert the (*k* + 1)st object in the array into its appropriate location.

b.This produces a list of *k* + 1 sorted objects.

After *n* – 1 steps, this produces a list of *n* sorted objects. This is clearly using the *insertion strategy* for sorting.

IMPLEMENTATION

Assuming we are trying to place the (*k* + 1)st object into the previously list of sorted entries. As soon as we find that the entry is in the correct location, we're finished—we can break out of the loop.

```
                    for ( int j = k; j> 0;  --j ) {
                      if ( array[j -1] >  array[j] ) {
                        std::swap(array[j - 1], array[j] );
}else {
//As soon as we don't need to swap, the (k + 1)st
//is in the correct location
break;
}
}
```

This would be part of a larger function that would call this loop once for each value from *k* = 1 to *n* – 1:

```
    template <typename Type>
void insertion_sort( Type *const array, int const n ) { for ( int k = 1; k < n; ++k ) {
    for ( int j = k; j > 0; --j ) {
if ( array[j - 1] > array[j] ) { std::swap( array[j - 1], array[j] );
    }else {
    //As soon as we don't need to swap, the (k + 1)st
    //is in the correct location
    break;
    }
    }
    }
    }
```

RUN-TIME ANALYSIS

To do a run-time analysis, we will begin with the outer loop: *k* takes on the values 1, 2, 3 and so on until *k* = *n* in which case, the condition fails. Therefore, the body of the outer for-loop will run *n* – 1 times with *k* taking the values 1 through *n* – 1.

The body of the inner loop contains an if-statement all components of which run in Θ(1) time. Thus, the body of the inner for-loop will run in Θ(1) time. As inner loop goes from *k* to 1, inclusive, the inner loop will run in O(*k*). I use O instead of Θ because there is a break statement in the inner for-loop—the loop may terminate early.

In the worst case, however, the inner loop will run *k* times and therefore the worst-case run time will be

$\sum n-1\ k = n(n-1) = O(n2)$.

HEAP SORT (*HEAPSORT*)

We will now look at our first Θ(*n* ln(*n*)) algorithm: heap sort. It will use a data structure that we have already seen: a binary heap.

Strategy and Run-time Analysis

Given a list of *n* objects, insert them into a min-heap and take them out in order. Now, in the worst case, both a push into and a pop from a binary min-heap with *k* entries is Θ(ln(*k*)). Therefore, the total run time will be

$$\sum_{k=1}^{n} \ln(k) = \ln\left(\prod_{k=1}^{n} k\right) = \ln(n!)$$

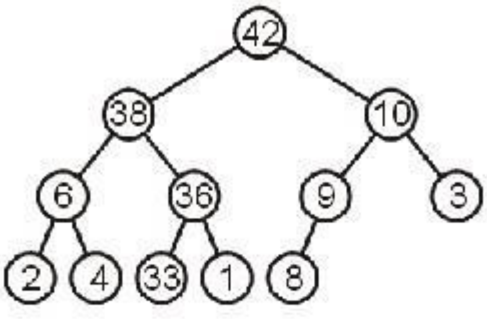
because the sum of logarithms is the logarithm of the products: ln(*a*) + ln(*b*) = ln(*ab*). The question is, what is the asymptotic growth of ln(*n*!)? Such a question is best left to Maple:

```
> asympt( ln( n! ), n );
```

$$(\ln(n)-1)n+\frac{1}{2}\ln(n)+\frac{1}{2}\ln(2\pi)+\frac{1}{2}\ln(n)+\frac{1}{12n}-\frac{1}{360n^3}+O\left(\frac{1}{n^5}\right)$$

The dominant term in this asymptotic series is (ln(n) – 1)n and thus the run time is O(n ln(n)). If you plot both ln(n!) and n ln(n), you will note that they appear to be grown at approximately the same rate.

Figure. A plot of ln(n!) in red and n ln(n) in blue.



Note that it is not possible to calculate ln(200!) directly as 200! ≈ 7.89 × 10375 which is greater than the largest double-precision floating-point number.

In-place Implementation

Issue: this implementation requires an additional array for the heap... Is it possible to do a heap sort in- place? That is, can we do heap sort with only Θ(1) additional memory?

Suppose we have a max-heap as is shown in Figure 2.

Figure 2. A max-heap.

The array representation of this max-heap is

1	2	3	4	5	6	7	8	9	10	11	12
42	38	10	6	36	9	3	2	4	33	1	8

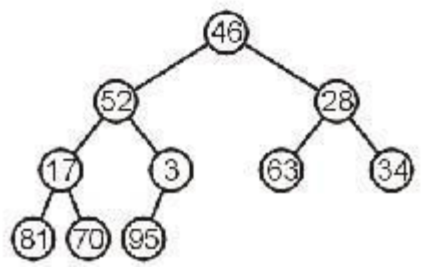
If we pop the top entry of the heap, this creates a vacancy in the last position:

1	2	3	4	5	6	7	8	9	10	11	12
38	36	10	6	33	9	3	2	4	8	1	**?**

We could place 42 into this last position but ignore it the next time we pop 38:

1	2	3	4	5	6	7	8	9	10	11	12
36	33	10	6	8	9	3	2	4	1	**?**	**42**

After *n* pops, we would have a sorted list.



The problem, however, is that we do not start with a max heap. Consider, for example, the unsorted array

0	1	2	3	4	5	6	7	8	9
46	52	28	17	3	63	34	81	70	95

If we interpret this as a binary tree, we have the tree shown in Figure 3.

Figure. The tree-representation of the given unsorted array.

Unfortunately, this is neither a min-heap, a max-heap, nor a binary search tree. It's just a binary tree. We need to convert this tree into a max-heap and we must do it in place.

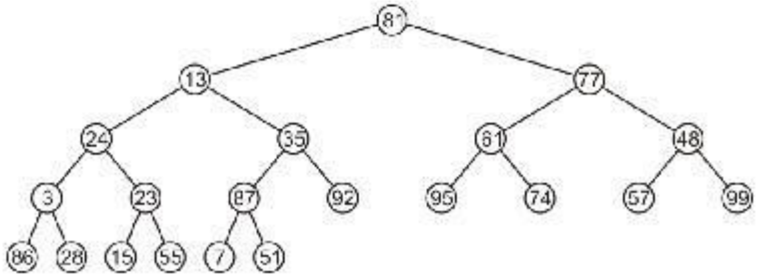
Another issue is indexing: recall that with binary heaps, we left the location 0 empty so that we could use the simple formulas that for the entry k, its children are locations 2*k and 2*k + 1 and its parent is in location k/2. Because all arrays start at 0, we can use the formulas 2*k + 1, 2*k +

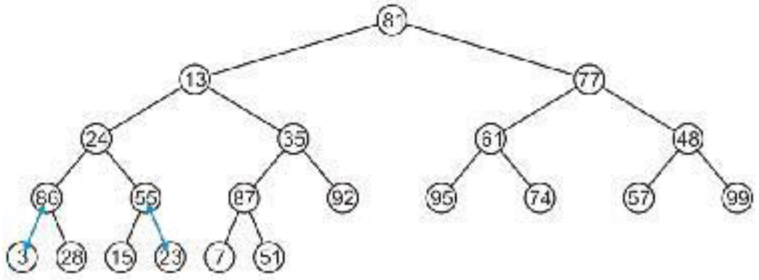
2for the children and (k + 1)/2 – 1 for the parent. Fortunately, we will have to find the parent of an entry at most *n* times.

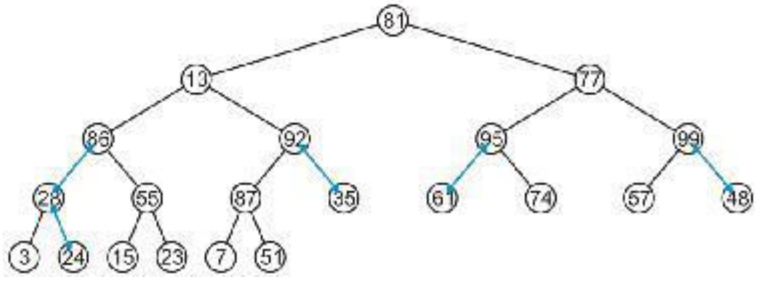
In-place Heapification

In order to convert an arbitrary complete tree into a binary max-heap, there are two strategies we could use:

1. Similar to insertion sort, assume the root is a binary max-heap and keep inserting the next entry in the array into the existing max-heap, or
2. Start from the bottom and note that all the leaf nodes are already valid max-heaps. Now, carry on working from right to left, and for each previous element, make whatever changes are necessary to convert the tree rooted at that node into a max-heap working all the way back to the first entry.







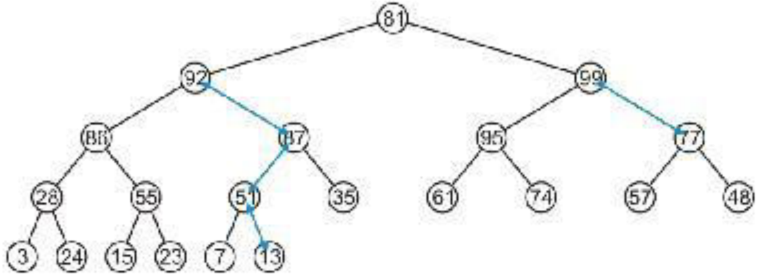
Let us work from the leaves up. In order to do this, let's consider the larger tree shown in Figure 4.

Figure . The binary tree representation of an unsorted list with 21 entries.

Starting from the last entry, 51, and working our way in a reverse-breadth-first traversal, we note that all the leaf nodes are trivial max-heaps of size 1. Now, look at node 87: it, too, is a max-heap;however, the two trees rooted at 23 and 3 are not max heaps. We can, never-the-less, convert these two trees into max heaps by percolating the root elements down, as is shown in Figure

Proceeding one row higher, we can swap 48 and 99, 61 and 95, 35 can be swapped with 87 and again with 51, and finally 24 can be swapped with 86 and then 28. Now all the trees rooted at depth 2 are max- heaps, as is shown in Figure.

Continuing back, 77 must be swapped with 99 and 13 must be percolated down to being a leaf node swapping with 92, 87, and 51. This is shown in Figure 7.



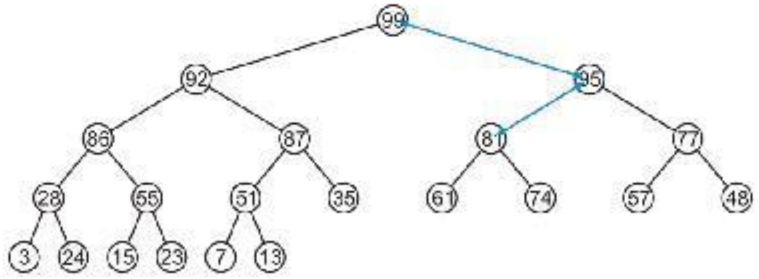


Figure . The trees rooted at 99 and 92 are now max-heaps.

Finally, 81 is swapped with 99 and then again with 95. This produces the max-heap shown in Figure .

Run-time Analysis

If we consider a perfect tree of height *h*, there are 2*k* nodes at a depth *k* and a node at depth *k* will be compared and possibly swapped with at most *h* – *k* nodes. Thus, the maximum number of comparisons will be

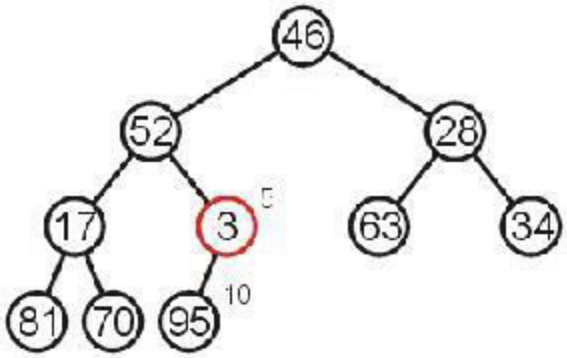
$$\sum_{k=0}^{h}(2k\,(h - \square k\,)) = \square(2h{+}1 - 1) - (h - 1).$$

Now, $n = 2h + 1 - 1$ and h = lg(n + 1) – 1 thus, the run time is n – lg(n) = O(n).

Note that if we chose to create the max-heap using the same strategy as insertion sort, the run time would be calculated by

h
∑(2*k k*)=□2*h*+1 (*h* −1)+□2
k =0
=(2*h*+1 +1)(*h* −1)−□(*h* −1)+□2
=*n*(lg (*n* +1)−□2)−□lg (*n* +1)+□4 Θ□=(*n* ln (*n*))

which is significantly worse. Note that this does not affect the overall run time, as the subsequent operation will be Θ(n ln(n)).



Example

As an example of a heap sort, consider the following unsorted list:

46	52	28	17	3	63	34	81	70	95

First, we must transform the array into a max-heap

We begin with the entry containing 3 and it has one child so we swap 3 and 95:

46	52	28	17	**95**	63	34	81	70	**3**

Next, the children of 17 are 81 and 70, so we swap 17 and 81:

46	52	28	**81**	95	63	34	**17**	**70**	3

The children of 28 are 63 and 34, so was swap 28 and 63:

46	52	**63**	81	95	**28**	**34**	17	70	3

The children of 52 are 81 and 95, so we swap 52 and 95. In the new location, 52 has one child, 3, and thus no swap is necessary:

46	**95**	63	**81**	**52**	28	34	17	70	**3**

Finally, the children of 46 are 95 and 28, so we swap 46 and 95. In its new position, the children of 46 are now 81 and 52, so we continue by swapping 46 and 81. In that position, the children of 46 are 17 and 70, so we swap 46 and 70.

95	**46**	**63**	81	52	28	34	17	70	3
95	**81**	63	**46**	**52**	28	34	17	70	3
95	81	63	**70**	52	28	34	**17**	**46**	3

Second, we must convert the max-heap into a sorted list:

95	81	63	70	52	28	34	17	46	3

Swap 95 and 3 and percolate 3 into a max-heap of size 9:

81	**70**	63	**46**	52	28	34	17	**3**	95

Swap 81 and 3 and percolate 3 into a max-heap of size 8:

70	**52**	63	46	**3**	28	34	17	81	95

Swap 70 and 17 and percolate 17 into a max-heap of size 7:

63	52	**34**	46	3	28	**17**	70	81	95

Swap 63 and 17 and percolate 17 into a max-heap of size 6:

52	**46**	34	**17**	3	28	63	70	81	95

Swap 52 and 28 and percolate 28 into a max-heap of size 5:

46	**28**	34	**17**	3	52	63	70	81	95

Swap 46 and 3 and percolate 3 into a max-heap of size 4:

34	28	**3**	17	46	52	63	70	81	95

Swap 34 and 17 and percolate 17 into a max-heap of size 3:

28	**17**	3	34	46	52	63	70	81	95

Swap 28 and 3 and percolate 3 into a max-heap of size 2:

17	**3**	28	34	46	52	63	70	81	95

Finally, swap 17 and 3 to produce a sorted list.

3	17	28	34	46	52	63	70	81	95

Merge Sort

We will now look at a second $\Theta(n \ln(n))$ algorithm: merge sort. This is the first divide-and-conqueralgorithm: we solve the problem by dividing the problem into smaller sub-problems, we recursively call the algorithm on the sub-problems, and we then recombine the solutions to the sub-problems to create a solution to the overall problem.

We will recursively define merge sort as follows:

1.If the list is of size 1,

2.Otherwise,

a.Divide the unsorted list into two sub-lists,

b.Recursively call merge sort on each sub-list, and

c.Merge the two sorted sub-lists together into a single sorted list.

We will first assume we have two sorted lists: what is the algorithm for merging them?

Merging Sorted Lists

Suppose we have two sorted lists, how can we merge them into a single sorted list? Consider the two lists of size $n1$ and $n2$:

3	5	18	21	24	27	31

and

2	7	12	16	33	37	42

We must begin with a new list of size *n*1 + *n*2 and we start with three indices all set to 0:

3	5	18	21	24	27	31

2	7	12	16	33	37	42

🟦													

We copy the smaller of the two objects into the new array and increment that index:

3	5	18	21	24	27	31

2	**7**	12	16	33	37	42

2

Repeating this process, we compare 3 and 7 and now copy 3 into the new list and increment that index:

3	**5**	18	21	24	27	31

2	**7**	12	16	33	37	42

2 **3**

			🟦										

											🟦	🟦	🟦

At this point, we would copy from the first array again, incrementing that index:

3	5	**18**	21	24	27	31

2	**7**	12	16	33	37	42

2 3 **5**

We could repeat this, however, at some point the index of one of the two arrays will be beyond the end of the array. Having copied 31 into the new array, the index of the first array is beyond the end

3	5	18	21	24	27	31	🟥

2	7	12	16	**33**	37	42

2	3	5	7	12	16	18	21	24	27	**31**	🟦		

At this point, however, all we must do is copy the remaining entries of the second array down:

3	5	18	21	24	27	31

2	7	12	16	**33**	**37**	**42**

2 3 5 7 12 16 18 21 24 27 31 **33** **37** **42**

and we are done.

Implementing Merging

Let's assume that the two arrays are array1 and array2 with sizes n1 and n2 while the output array arrayout is of size n1 + n2. We begin by defining three indices:

```
int i1 = 0, i2 = 0, k = 0;
```

Next, we iterate through the lists:

```
                    while ( i1 < n1 && i2 < n2 ) {
                    if ( array1[i1] < array2[i2] ) {
                    arrayout[k] = array1[i1];
                    ++i1;
                    } else {
                    assert( array1[i1] >= array2[i2] );          // Requires #include <cassert>

arrayout[k] = array2[i2]; ++i2;
}

++k;
}
```

Finally, all the entries in one of the two arrays has not yet been copied over, so we finish by copying those over:

```
for ( ; i1 < n1; ++i1, ++k ) { arrayout[k] = array1[in1];
    }

for ( ; i2 < n2; ++i2, ++k ) { arrayout[k] = array2[i2];
    }
```

Analysis of merging

You will note that one of these two loops will never run: the first array ran until one of i1 < n1or i2 < n2 evaluated to false (0).

The run-time of merging can be quickly determined by realizing that the statement ++k will only be executed $n1 + n2$ times, and therefore the run time is $\Theta(n1 + n2)$. If the sizes of the arrays are comparable, that is, $n = n1$ and $n1 \approx n2$, we can say that the run time is $\Theta(n)$.

We do, however, have one significant problem: the merging of two lists—even if they are adjacent— requires the allocation of another array at least equal to the smaller of the two arrays being merged. Thus, if we are merging two lists of size n, the memory requirements will also be $\Theta(n)$.

The Algorithm

Thus, of the five sorting techniques (insertion, exchange, selection, merging, and distribution), ours falls into the fourth case, merging. Now that we know we can merge two lists in $\Theta(n)$, we will simply apply the algorithm:

1.If the array is of size 1, it is sorted and we are finished;
2.Otherwise,
a.Split the list into two approximately equal sub-lists,
b.Recursively call merge sort on those sub-lists, and
c.Merge the resulting sorted sub-lists together into one sorted list.

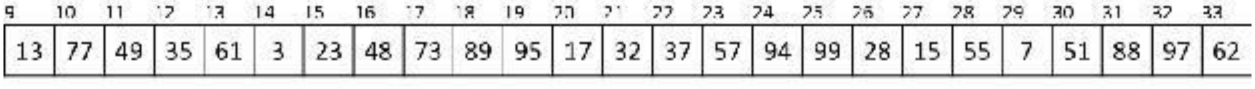
Question: does it make sense to recursively call merge sort on a list of size 2? Consider the overhead: two function calls, allocating a new array, and merging the two lists together.

In fact, should we even call merge sort recursively on a list of size under 8 or under 16? Certainly the overhead of making two function calls can be expensive.

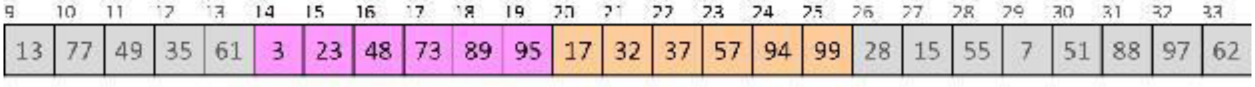
Consequently, it is reasonable to consider an alternative algorithm:

1.If the size of the array is less than some constant N, use **insertion sort** to sort it,
2.Otherwise,
a.Split the list into two approximately equal sub-lists,

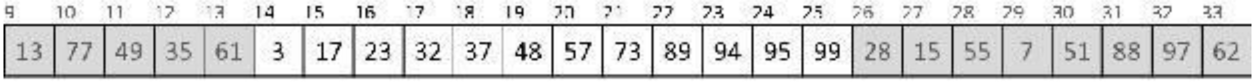
b.Recursively call merge sort on those sub-lists, and
c.Merge the resulting sorted sub-lists together into one sorted list.



9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
13	77	49	35	61	3	23	48	73	89	95	17	32	37	57	94	99	28	15	55	7	51	88	97	62



9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
13	77	49	35	61	3	23	48	73	89	95	17	32	37	57	94	99	28	15	55	7	51	88	97	62



9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
13	77	49	35	61	3	17	23	32	37	48	57	73	89	94	95	99	28	15	55	7	51	88	97	62

Thus, if the list is sufficiently small, insertion sort will be quicker than merge sort. In reality, this constant *N* can be very large: in one experiment, the author found *N* = 64 as being appropriate.

Implementation

Assume we have a merging function

template <typename Type>
void merge(Type *array, int a, int b, int c);

that assumes that the entries in positions a through b - 1 are sorted and the entries in positions b through c - 1 are sorted and returns with the entries from a through c - 1 merged together into a sorted list. Such a function will have to allocation additional memory internally.

For example, consider this array where the entries from 14 to 19 are sorted, and the entries from 20 to 25 are sorted.

Suppose we wish to merge these two together.

To do this, we call merge(array, 14, 20, 26), which results in:

Note that surrounding entries are not affected.

We will now implement a function

```
void merge_sort( Type *array, int first, first last );
```

which will sort the entries of the argument array from indices first <= i to i < last, inclusive. We will start by checking if there are fewer than *N* elements, in which case, we will call insertion_sort on those entries. Otherwise, we will find the mid-point, recursively call merge sort on both halves, and then merge the results:

```
    template <typename Type>
void merge_sort( Type *array, int first, int last ) { if ( last - first <= N ) {
        insertion_sort( array, first, last ); } else {
    int midpoint = (first + last)/2;

    merge_sort( array, first, midpoint ); merge_sort( array, midpoint, last ); merge( array, first,
    midpoint, last );
    }
    }
```

The implementation of insertion sort would also be restricted to soring entries on a sub-range of the array:

```
    template <typename Type>
void insertion_sort( Type *array, int first, int last ) { for ( int k = first + 1; k < last; ++k ) {
    Type tmp = array[k];
for ( int j = k; k > first; --j ) { if ( array[j - 1] > tmp ) {
        array[j] = array[j - 1]; } else {
    array[j] = tmp; goto finished;
    }
    }
    array[first] = tmp; finished: ;
    }
    }
```

Example

Consider sorting the following array of size 25:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	2
13	**77**	**49**	**35**	**61**	**48**	**73**	**23**	**95**	**3**	**89**	**37**	**57**	**99**	**17**	**32**	**94**	**28**	**15**	**55**	**7**	**51**	**88**	**97**	**62**

We will call insertion sort whenever the sub-list has a size less than *N* = 6.

Thus, we start with a call to merge_sort(array, 0, 25);

We begin by noting that the first and last entries have indices first = 0 and last = 25, and because 25 – 0 > 6, we will calculate the midpoint and recursively call merge sort:

```
// Code fragment 1
int midpoint = (0 + 25)/2; // == 12 merge_sort( array, 0, 12 ); merge_sort( array, 12, 25 );
merge( array, 0, 12, 25 );
```

However, we begin by executing the first of these, so now we call merge sort on the first half:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	**77**	**49**	**35**	**61**	**48**	**73**	**23**	**95**	**3**	**89**	**37**	57	99	17	32	94	28	15	55	7	51	88	97	62

Because 12 – 0 > 6, we will recursively call merge sort:

```
// Code fragment 2
int midpoint = (0 + 12)/2; // == 6 merge_sort( array, 0, 6 ); merge_sort( array, 6, 12 );
merge( array, 0, 6, 12 );
```

but again, we start by calling the first call to merge sort:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	**77**	**49**	**35**	**61**	**48**	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

At this point, 6 – 0 ≤ 6, so we call insertion sort which is performed in-place:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	**35**	**48**	**49**	**61**	**77**	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

Insertion sort finishes and returns, and consequently, the call to merge_sort(array, 0, 6)finishes and returns, too. We now go to continue executing the second function call in Code fragment 2. We are now calling merge sort on the second half of the first half the array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	**73**	**23**	**95**	**3**	**89**	**37**	57	99	17	32	94	28	15	55	7	51	88	97	62

At this point, 12 – 6 ≤ 6, so we call insertion sort which is performed in-place:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	**3**	**23**	**37**	**73**	**89**	**95**	57	99	17	32	94	28	15	55	7	51	88	97	62

Insertion sort finishes and returns, and consequently, the call to merge_sort(array, 6, 12)finishes and returns, too.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	**35**	**48**	**49**	**61**	**77**	**3**	**23**	**37**	**73**	**89**	**95**	57	99	17	32	94	28	15	55	7	51	88	97	62

We now go to continue executing the third function call in Code fragment 2. We will now call merge(array, 0, 6, 12) to merge the two sub-arrays. This results in:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	**13**	**23**	**35**	**37**	**48**	**49**	**61**	**73**	**77**	**89**	**95**	57	99	17	32	94	28	15	55	7	51	88	97	62

At this point, merge_sort(array, 0, 12) is finished. It returns back to the function that called it: merge_sort(array, 0, 25). Thus, we continue to execute the second function in Code Fragment 1: calling merge sort on the entries 12 through 24.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	2
3	13	23	35	37	48	49	61	73	77	89	95	**57**	**99**	**17**	**32**	**94**	**28**	**15**	**55**	**7**	**51**	**88**	**97**	**62**

Because 25 – 12 > 6, we will recursively call merge sort:

```
// Code fragment 3
int midpoint = (12 + 25)/2; // == 18 merge_sort( array, 12, 18 ); merge_sort( array, 18, 25 );
merge( array, 12, 18, 25 );
```

but again, we start by calling the first call to merge sort:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	**57**	**99**	**17**	**32**	**94**	**28**	15	55	7	51	88	97	62

Here, 18 – 12 ≤ 6, so we call insertion sort which is performed in-place:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	**17**	**28**	**32**	**57**	**94**	**99**	15	55	7	51	88	97	62

Insertion sort finishes and returns, and consequently, the call to merge_sort(array, 12, 18)finishes and returns, too. We now go to continue executing the second function call in Code fragment 3. We are now calling merge sort on the second half of the second half the array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	2
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	**15**	**55**	**7**	**51**	**88**	**97**	**62**

Because 25 – 18 > 6, we will recursively call merge sort:

```
// Code fragment 4
int midpoint = (18 + 25)/2; // == 21 merge_sort( array, 18, 21 ); merge_sort( array, 21, 25 );
merge( array, 18, 21, 25 );
```

but again, we start by calling the first call to merge sort:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	**15**	**55**	**7**	51	88	97	62

At this point, 21 – 18 ≤ 6, so we call insertion sort which is performed in-place:

01		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	202122			23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	**7**	**15**	**55**	51	88	97	62

Insertion sort finishes and returns, and consequently, the call to merge_sort(array, 18, 21) finishes

and returns, too. It returns back to the function that called it: merge_sort(array, 18, 25). Thus, we continue to execute the second function in Code Fragment 4: calling merge sort on the entries 21 through 24.

01		2	34		5	6	7	8	9	10	11	12	13	14	15	16	17	1819		20	21	2223		24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	**51**	**88**	**97**	**62**

Here, also, 25 – 21 ≤ 6, so we call insertion sort which is performed in-place:

01		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	1819		20	21	22	23	2
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	**51**	**62**	**88**	**97**

Insertion sort finishes and returns, and consequently, the call to merge_sort(array, 18, 21)finishes and returns, too. We now call the final function in Code Fragment 4: merging the two sub-arrays. We now more the arrays from 18 to 20 and 21 to 24 to get:

01		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	2
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	**7**	**15**	**51**	**55**	**62**	**88**	**97**

The call to merge_sort(array, 18, 25) returns with the entries from 18 to 24 sorted. We now return to execution of the function call merge_sort(array, 12, 25) and execute the last function in Code Fragment 3: merging the sub-arrays from 12 to 17 and 18 to 24. This is performed and results in

01		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	2
3	13	23	35	37	48	49	61	73	77	89	95	**7**	**15**	**17**	**28**	**32**	**51**	**55**	**57**	**62**	**88**	**94**	**97**	**99**

Having completed this merging process, the call to merge_sort(array, 12, 25) returns, and we are back executing the last function call in our original call to merge_sort(array, 0, 25), namely, merging the two sub-arrays from 0 to 11 and 12 to 24. This yields

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	2
3	**7**	**13**	**15**	**17**	**23**	**28**	**32**	**35**	**37**	**48**	**49**	**51**	**55**	**57**	**61**	**62**	**73**	**77**	**88**	**89**	**94**	**95**	**97**	**99**

which is a sorted list.

Run-time Analysis

The time required to perform a merge sort (ignoring our optimization by calling insertion sort) on an array of size *n* is:

1.The time required to sort the left half containing approximately *n*/2 entries,
2.The time required to sort the right half, again with *n*/2 entries, and
3.The time required to merge the results.
This gives us

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

Solving this in Maple gives us:

> rsolve({T(n) = 2*T(n/2) + n, T(1) = 1}, T(n)); $\frac{n(\ln(2) + \ln(n))}{\ln(2)}$

which can be simplified to *n* + *n* lg(*n*); thus, the run-time of merge sort is Θ(*n* lg(*n*)).

Later on, we will see the *master theorem* which will give us the run-time of most variations of *divide-and- conquer* algorithms.

There are no best-case and no worst-case scenarios for merge sort. They will all have the same *n* lg(*n*) run time.

In practice, merge sort is faster than heap sort; however, unlike heap sort, merge sort requires the allocation of an addition array for the merging process: this requires Θ(*n*) additional memory. Next we will see quick sort which is, on average, faster than both heap sort and merge sort and usually requires only Θ(ln(*n*)) additional memory.

Quicksort

Merge sort divided an unsorted list into two approximately equal sub-lists based on location. We will look at an alternate strategy for dividing a list into two sub-lists: select one entry in the list (call it a *pivot*) and separate all other entries as to whether they are smaller than or larger than this pivot.

Using this idea, quicksort is, on average, faster than merge sort and has the following properties, but there are some issues, as are shown in Table 1.

Table 1. The run times of quicksort.

	Run Time	Memory
Average Case	$\Theta(n \ln(n))$	$\Theta(\ln(n))$
Worst Case	$\Theta(n2)$	$\Theta(n)$

We will look at this algorithm but we will also look at strategies for avoiding the worst-case scenario.

Strategy and Run-time Analysis

Suppose, we split a list into two sub-lists by picking one entry (the *pivot*) and dividing all other entries into those less than the pivot and those greater than the pivot. For example, if we select 44 from this list

80	21	95	84	66	10	79	**44**	26	87	96	12	43	71	3

This would produce the list

21	10	26	12	43	3	**44**	80	95	84	66	79	87	96	71

Each of these two lists contains approximately $n/2$ entries. Notice that 44 is now in the correct location if the array is entirely sorted. We could continue by using a similar algorithm on the first six entries and the last eight entries.

Thus, if we were to reapply this algorithm and always get two sub-lists of approximately half the size at each step, the run time would be similar to that of merge sort: $\Theta(n \ln(n))$. We can also apply our simplification of using insertion sort if the size of the list ever drops below some fixed N.

enario

Unfortunately, we might get unlucky:

80	21	95	84	66	10	79	**2**	26	87	96	12	43	71	3

Using 2 as a pivot results in the partition

2	80	21	95	84	66	10	79	26	87	96	12	43	71	3

At this point, we must sort the remaining list of size $n - 1$. Thus, the run time may be described by

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T(n-1) + \Theta(n) & n > 1 \end{cases}$$

This is no different from the run-time of selection sort: Θ(*n*2).

Median-of-Three Strategy

The median of *n* entries is that entry such that half of all values are less than the median (and, therefore, the other half are greater than that entry).

The politician was astonished to learn that half of all
Canadians were below the median intelligence.

The ideal case is to choose the median; however, we cannot find the median entry quickly. Instead, an alternate strategy is to choose three entries, say, the first, middle, and last entries and choose the median of these three entries. Going back to our initial example, the median of the entries {80, 44, 3} is 44.

80	21	95	84	66	10	79	**44**	26	87	96	12	43	71	**3**

We can now partition the entries based on the pivot 44:

21	10	26	12	43	3	**44**	80	95	84	66	79	87	96	71

Applying the median-of-three on the first six,

21	10	26	12	43	3	**44**	80	95	84	66	79	87	96	71

would be the median of {21, 26, 3} which is 21 while the median of the last eight,

21	10	26	12	43	3	**44**	80	95	84	66	79	87	96	71

would be the median of {80, 66, 71} or 71.

UNIT –V

GRAPHS

Definition

A graph, *G=(V, E)*, consists of two sets:

- a finite set of ***vertices(V)***, and
- a finite, possibly empty set of edges***(E)***
- *V(G)* and *E(G)* represent the sets of vertices and edges of *G*, respectively

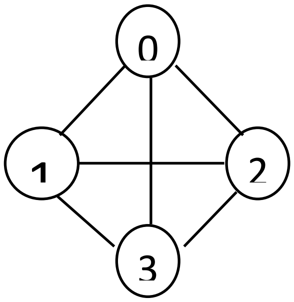
Undirected graph

- The pairs of vertices representing any edges is ***unordered***
- e.g., *(v0, v1)* and *(v1, v0)* represent the same edge

Directed graph

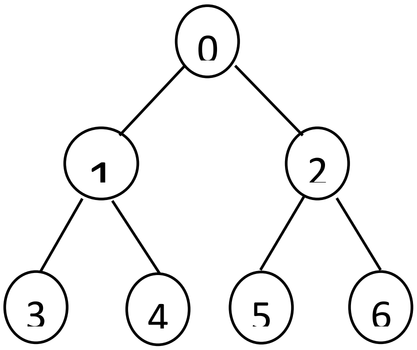
- Each edge as a directed pair of vertices
- e.g. *<v0, v1>* represents an edge, *v0* is the tail and *v1* is the head.

Examples for Graph



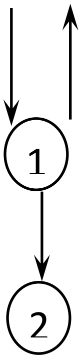
G_1

complete graph



G_2

incomplete graph



G_3

V(G1)={0,1,2,3}	E(G1)={(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)}
V(G2)={0,1,2,3,4,5,6}	E(G2)={(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)}
V(G3)={0,1,2}	E(G3)={<0,1>,<1,0>,<1,2>}

complete undirected graph: n(n-1)/2 edges

complete directed graph: n(n-1) edges

A complete graph is a graph that has the maximum number of edges

- for undirected graph with n vertices, the maximum number of edges is n(n-1)/2
- for directed graph with n vertices, the maximum number of edges is n(n-1)
- example: G1 is a complete graph

Adjacent and Incident

If (v0, v1) is an edge in an undirected graph,

  - v0 and v1 are adjacent
  - The edge (v0, v1) is incident on vertices v0 and v1

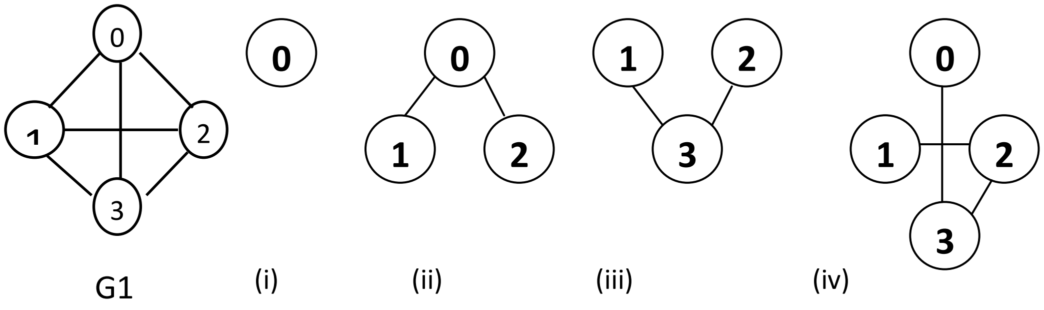
If <v0, v1> is an edge in a directed graph

  - v0 is adjacent to v1, and v1 is adjacent from v0
  - The edge <v0, v1> is incident on v0 and v1

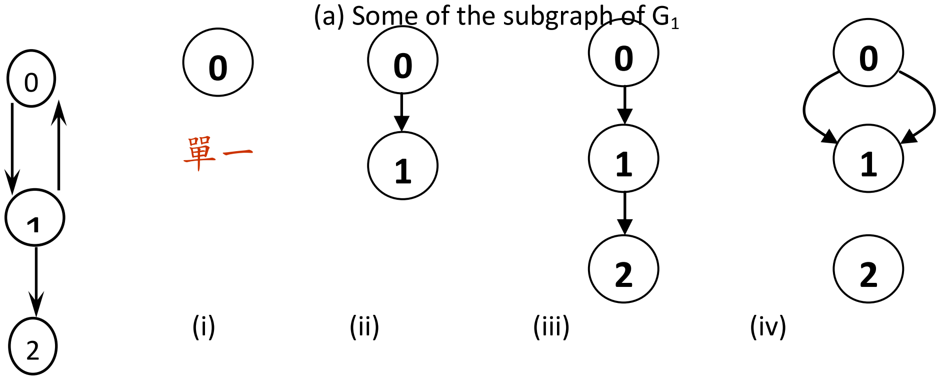
Subgraph and Path

- A subgraph of G is a graph G' such that V(G') is a subset of V(G) and E(G') is a subset of E(G)
- A path from vertex vp to vertex vq in a graph G, is a sequence of vertices, vp, vi1, vi2, ..., vin, vq, such that (vp, vi1), (vi1, vi2), ..., (vin, vq) are edges in an undirected graph
- The length of a path is the number of edges on it

Fig: subgraphs of G1 and G3



(a) Some of the subgraph of G_1



(b) Some of the subgraph of G_3

G3

Simple Path and Style

- A simple path is a path in which all vertices, except possibly the first and the last, are distinct
- A cycle is a simple path in which the first and the last vertices are the same
- In an undirected graph G, two vertices, v0 and v1, are connected if there is a path in G from v0 to v1
- An undirected graph is connected if, for every pair of distinct vertices vi, vj, there is a path
  from vi to vj

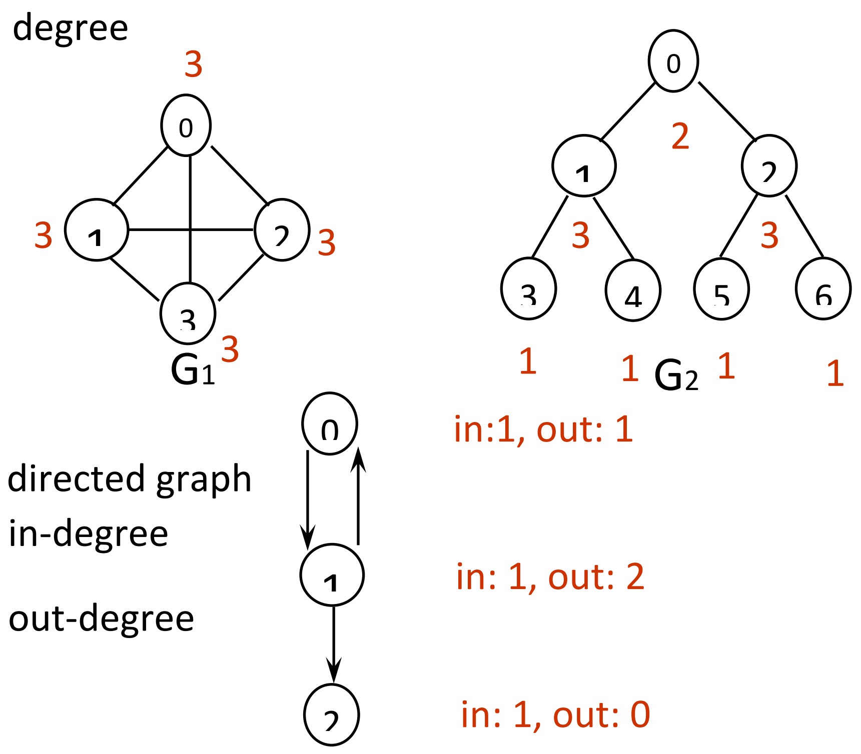
Connected Component

- A connected component of an undirected graph is a maximal connected subgraph.
- A tree is a graph that is connected and acyclic.
- A directed graph is strongly connected if there is a directed path from vi to vj and also from vj to vi.
- A strongly connected component is a maximal subgraph that is strongly connected

DEGREE

- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail
  - if *di* is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

$$e = (\sum_{0}^{n-1} d_i)/2$$



<u>ADT for Graph</u>

structure Graph is

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* ∈ *Graph*, *v*, *v*1 and *v*2 ∈ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no incident edge.

Graph InsertEdge(*graph*, *v1*,*v2*)::= return a graph with new edge between *v1* and *v2*

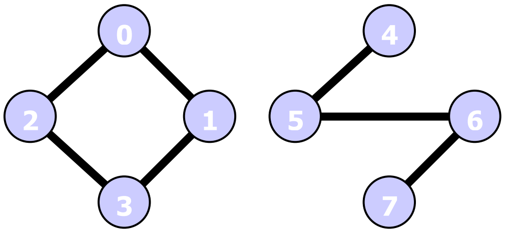
Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*) is removed

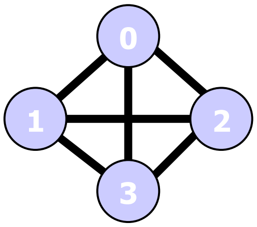
Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE

else return FALSE

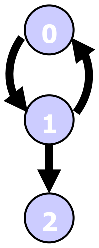
List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*



G4



G1



G3

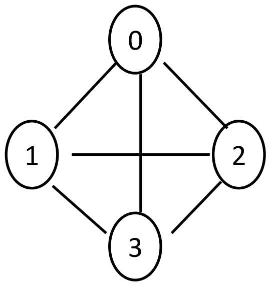
Graph Representations

Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.

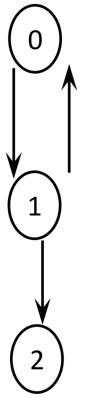
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (vi, vj) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric
-

Examples for Adjacency Matrix



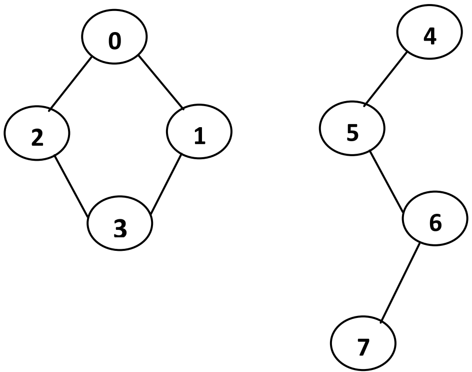
$$\begin{bmatrix} O & 1 & 1 & 1 \\ 1 & O & 1 & 1 \\ 1 & 1 & O & 1 \\ 1 & 1 & 1 & O \end{bmatrix}$$

G_1



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

G_2



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

symmetric

undirected: $n^2/2$

directed: n^2

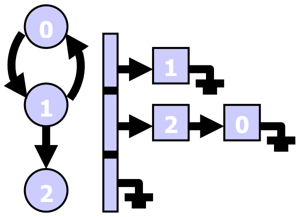
Merits of Adjacency Matrix

- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is
- For a digraph, the row sum is the out_degree, while the column sum is the in_degree

$$ind(vi) = \sum_{j=O}^{n-1} A[j,i]$$

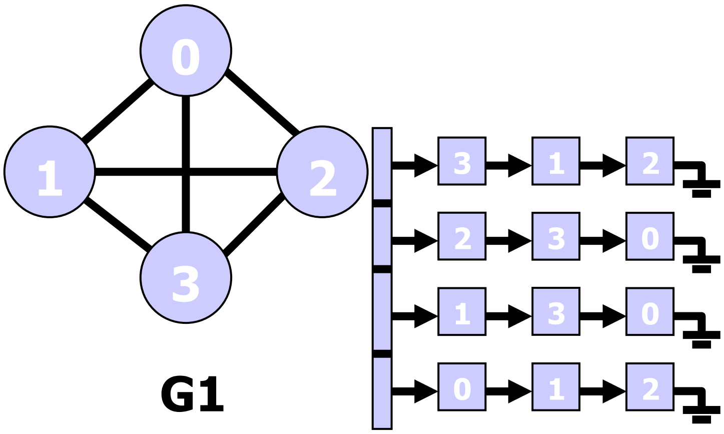
Adjacency lists

- linked list



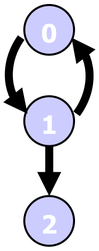
G3

```
#define MAX_VERTICES 50
typedef struct node *node_ptr;
typedef struct node {
                int vertex;
                node_ptr link;
} node;
node_ptr graph[MAX_VERTICES];
int n = 0; /* number of nodes
```



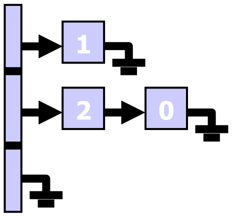
G1

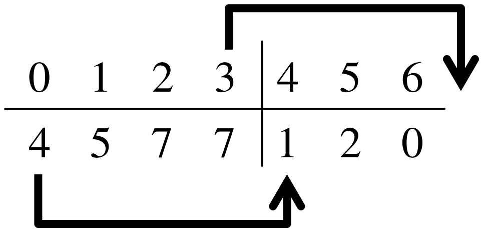
Adjacency lists, by array



G3

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

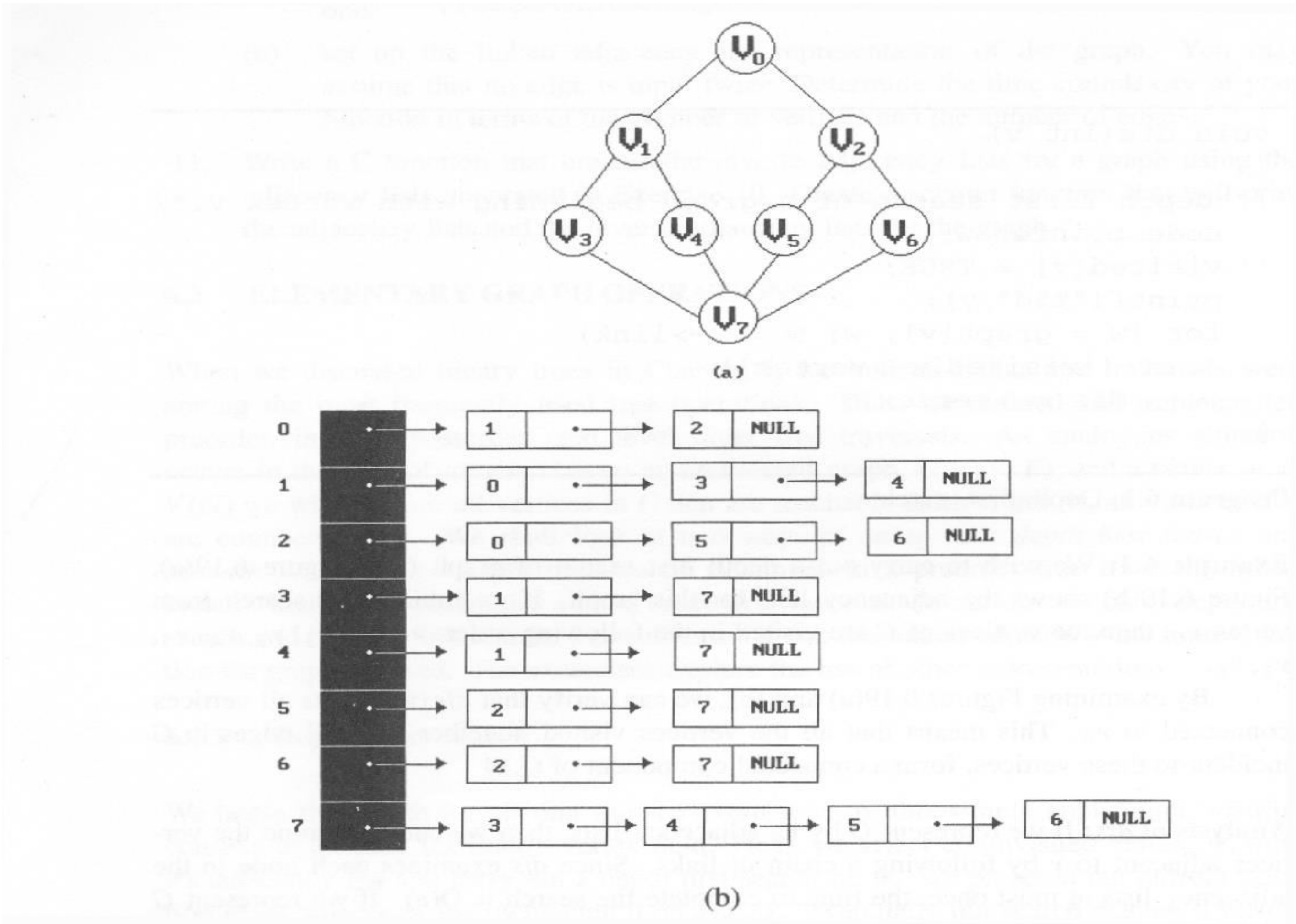




0	1	2	3	4	5	6
4	5	7	7	1	2	0

Some Graph Operations

- Traversal
  Given G=(V,E) and vertex v, find all w∈V, such that w connects v.
  - Depth First Search (DFS) preorder tree traversal
  - Breadth First Search (BFS) level order tree traversal
- Connected Components
- Spanning Trees



Depth First Search

```
#define FALSE 0

#define TRUE 1

short int visited[MAX_VERTICES];

void dfs(int v)

{

  node_pointer w;

  visited[v]= TRUE;

  printf("%5d", v);

  for (w=graph[v]; w; w=w->link)

    if (!visited[w->vertex])
    dfs(w->vertex);
}
```

Data structure: a) adjacency list: O(e) b) adjacency matrix: O(n2)

Breadth-First Search

```
typedef struct queue *queue_ptr;
typedef struct queue {
        int vertex;
        queue_ptr link;
};
void addq(queue_ptr *, queue_ptr *, int);
Int deleteq(queue_ptr);
```

```
void bfs(int v) {
        node_ptr w;
        queue_ptr front, rear;
        front=rear=NULL;
        printf("%5d",v);
        visited[v]=TRUE;
        addq(&front, &rear, v);
        while(front) {
                v = deleteq(&front);
                for(w=graph[v]; w; w=w->link)
                        if(!visited[w->vertex]) {
                                printf("%5d", w->vertex);
                                addq(&front, &rear, w->vertex);
                                visited[w->vertex] = TRUE;
                        }
```

B-TREES: BALANCED TREE DATA STRUCTURES

INTRODUCTION

Tree structures support various basic dynamic set operations including *Search*, *Predecessor*, *Successor*, *Minimum*, *Maximum*, *Insert*, and *Delete* in time proportional to the height of the tree. Ideally, a tree will be balanced and the height will be *log n* where *n* is the number of nodes in the tree. To ensure that the height of the tree is as small as possible and therefore provide the best running time, a balanced tree structure like a red-black tree, AVL tree, or b-tree must be used.

When working with large sets of data, it is often not possible or desirable to maintain the entire structure in primary storage (RAM). Instead, a relatively small portion of the data structure is maintained in primary storage, and additional data is read from secondary storage as needed. Unfortunately, a magnetic disk, the most common form of secondary storage, is significantly slower than random access memory (RAM). In fact, the system often spends more time retrieving data than actually processing data.

B-trees are balanced trees that are optimized for situations when part or all of the tree must be maintained in secondary storage such as a magnetic disk. Since disk accesses are expensive (time consuming) operations, a b-tree tries to minimize the number of disk accesses. For example, a b-tree with a height of 2 and a branching factor of 1001 can store over one billion keys but requires at most two disk accesses to search for any node (Cormen 384).

THE STRUCTURE OF B-TREES

Unlike a binary-tree, each node of a b-tree may have a variable number of keys and children. The keys are stored in non-decreasing order. Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key but greater than the preceeding key. A node also has an additional rightmost child that is the root for a subtree containing all keys greater than any keys in the node.

A b-tree has a minumum number of allowable children for each node known as the *minimization factor*. If *t* is this *minimization factor*, every node must have at least *t - 1* keys. Under certain circumstances, the root node is allowed to violate this property by having fewer than *t - 1* keys. Every node may have at most *2t - 1* keys or, equivalently, *2t* children.

Since each node tends to have a large branching factor (a large number of children), it is typically neccessary to traverse relatively few nodes before locating the desired key. If access to each node requires a disk access, then a b-tree will minimize the number of disk accesses required. The minimzation factor is usually chosen so that the total size of each node corresponds to a multiple of the block size of the underlying storage device. This choice simplifies and optimizes disk access. Consequently, a b-tree is an ideal data structure for situations where all data cannot reside in primary storage and accesses to secondary storage are comparatively expensive (or time consuming).

HEIGHT OF B-TREES

For *n* greater than or equal to one, the height of an *n*-key b-tree T of height *h* with a minimum degree *t* greater than or equal to 2,

$$h \leq \log_t \frac{n+1}{2}$$

The worst case height is *O(log n)*. Since the "branchiness" of a b-tree can be large compared to many other balanced tree structures, the base of the logarithm tends to be large; therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures. Although this does not affect the asymptotic worst case height, b-trees tend to have smaller heights than other trees with the same asymptotic height.

OPERATIONS ON B-TREES

The algorithms for the *search*, *create*, and *insert* operations are shown below. Note that these algorithms are single pass; in other words, they do not traverse back up the tree. Since b-trees strive to minimize disk accesses and the nodes are usually stored on disk, this single-pass approach will reduce the number of node visits and thus the number of disk accesses. Simpler double-pass approaches that move back up the tree to fix violations are possible.

Since all nodes are assumed to be stored in secondary storage (disk) rather than primary storage (memory), all references to a given node be be preceeded by a read operation denoted by *Disk-Read*. Similarly, once a node is modified and it is no longer needed, it must be written out to secondary storage with a write operation denoted by *Disk-Write*. The algorithms below assume that all nodes referenced in parameters have already had a corresponding *Disk-Read* operation. New nodes are created and assigned storage with the *Allocate-Node* call. The implementation details of the *Disk-Read*, *Disk-Write*, and *Allocate-Node* functions are operating system and implementation dependent.

B-TREE-SEARCH(X, K)

```
i <- 1
while i <= n[x] and k > $key_i[x]$
    do i <- i + 1
if i <= n[x] and k = $key_i[x]$
    then return (x, i)
if leaf[x]
    then return NIL
    else Disk-Read($c_i[x]$)
        return B-Tree-Search($c_i[x]$, k)
```

The search operation on a b-tree is analogous to a search on a binary tree. Instead of choosing between a left and a right child as in a binary tree, a b-tree search must make an n-way choice. The correct child is chosen by performing a linear search of the values in the node. After finding the value greater than or equal to the desired value, the child pointer to the immediate left of that value is followed. If all values are less than the desired value, the rightmost child pointer is followed. Of course, the search can be terminated as soon as the desired node is found. Since the running time of the search operation depends upon the height of the tree, *B-Tree-Search* is $O(log_t\, n)$.

B-Tree-Create(T)

```
x <- Allocate-Node()
leaf[x] <- TRUE
n[x] <- 0
Disk-Write(x)
root[T] <- x
```

The *B-Tree-Create* operation creates an empty b-tree by allocating a new root node that has no keys and is a leaf node. Only the root node is permitted to have these properties; all other nodes must meet the criteria outlined previously. The *B-Tree-Create* operation runs in time *O(1)*.

B-Tree-Split-Child(x, i, y)

```
z <- Allocate-Node()
leaf[z] <- leaf[y]
n[z] <- t - 1
for j <- 1 to t - 1
    do $key_j[z]$ <- $key_{j+t}[y]$
if not leaf[y]
    then for j <- 1 to t
        do $c_j[z]$ <- $c_{j+t}[y]$
n[y] <- t - 1
for j <- n[x] + 1 downto i + 1
    do $c_{j+1}[x]$ <- $c_j[x]$
$c_{i+1}$ <- z
for j <- n[x] downto i
    do $key_{j+1}[x]$ <- $key_j[x]$
$key_i[x]$ <- $key_t[y]$
n[x] <- n[x] + 1
Disk-Write(y)
Disk-Write(z)
Disk-Write(x)
```

If is node becomes "too full," it is necessary to perform a split operation. The split operation moves the median key of node *x* into its parent *y* where *x* is the i^{th} child of *y*. A new node, *z*, is allocated, and all keys in *x* right of the median key are moved to *z*. The keys left of the median key remain in the original node *x*. The new node, *z*, becomes the child immediately to the right of the median key that was moved to the parent *y*, and the original node, *x*, becomes the child immediately to the left of the median key that was moved into the parent *y*.

The split operation transforms a full node with *2t - 1* keys into two nodes with *t - 1* keys each. Note that one key is moved into the parent node. The *B-Tree-Split-Child* algorithm will run in time *O(t)* where *t* is constant.

B-Tree-Insert(T, k)

```
r <- root[T]
if n[r] = 2t - 1
    then s <- Allocate-Node()
        root[T] <- s
                     leaf[s] <- FALSE
                     n[s] <- 0
                     $c_1$ <- r
                     B-Tree-Split-Child(s, 1, r)
                     B-Tree-Insert-Nonfull(s, k)
    else B-Tree-Insert-Nonfull(r, k)
```

B-Tree-Insert-Nonfull(x, k)

```
i <- n[x]
if leaf[x]
    then while i >= 1 and k < $key_i[x]$
          do $key_{i+1}[x]$ <- $key_i[x]$
                         i <- i - 1
        $key_{i+1}[x]$ <- k
                     n[x] <- n[x] + 1
                     Disk-Write(x)
    else while i >= and k < $key_i[x]$
          do i <- i - 1
                     i <- i + 1
                     Disk-Read($c_i[x]$)
                     if n[$c_i[x]$] = 2t - 1
                         then B-Tree-Split-Child(x, i, $c_i[x]$)
                             if k > $key_i[x]$
                                       then i <- i + 1
        B-Tree-Insert-Nonfull($c_i[x]$, k)
```

To perform an insertion on a b-tree, the appropriate node for the key must be located using an algorithm similiar to *B-Tree-Search*. Next, the key must be inserted into the node. If the node is not full prior to the insertion, no special action is required; however, if the node is full, the node must be split to make room for the new key. Since splitting the node results in moving one key to the parent node, the parent node must not be full or another split operation is required. This process may repeat all the way up to the root and may require splitting the root node. This approach requires two passes. The first pass locates the node where the key should be inserted; the second pass performs any required splits on the ancestor nodes.

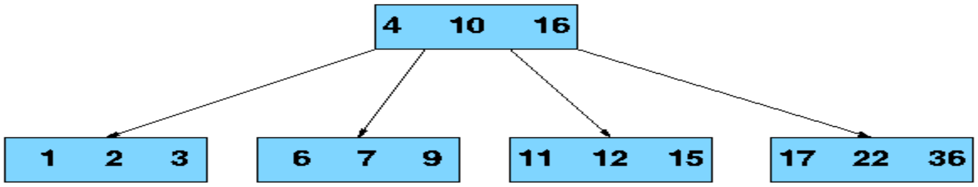
Since each access to a node may correspond to a costly disk access, it is desirable to avoid the second pass by ensuring that the parent node is never full. To accomplish this, the presented algorithm splits any full nodes encountered while descending the tree. Although this approach may result in unecessary split operations, it guarantees that the parent never needs to be split and eliminates the need for a second pass up the tree. Since a split runs in linear time, it has little effect on the $O(t \log_t n)$ running time of *B-Tree-Insert*.

Splitting the root node is handled as a special case since a new root must be created to contain the median key of the old root. Observe that a b-tree will grow from the top.

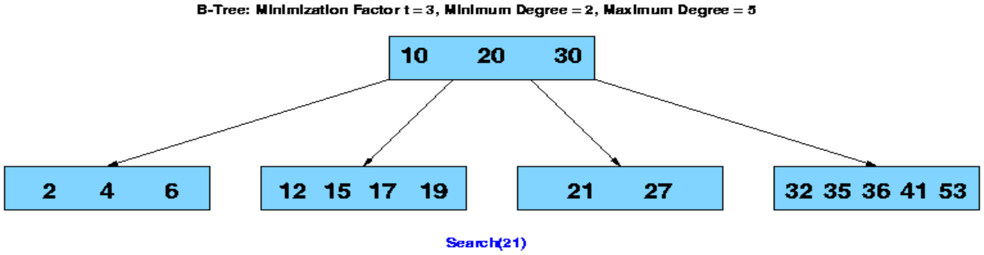
B-Tree-Delete

Deletion of a key from a b-tree is possible; however, special care must be taken to ensure that the properties of a b-tree are maintained. Several cases must be considered. If the deletion reduces the number of keys in a node below the minimum degree of the tree, this violation must be corrected by combining several nodes and possibly reducing the height of the tree. If the key has children, the children must be rearranged.

Examples Sample B-Tree



Searching a B-Tree for Key 21


B-Tree: Minimization Factor t=3, Minimum Degree=2, Maximum Degree=5
10 20 30
2 4 6
12 15 17 19
21 27
32 35 36 41 53
Search(21)

AVL TREES

Binary search tree

time complexity

- average case: O(log2n)

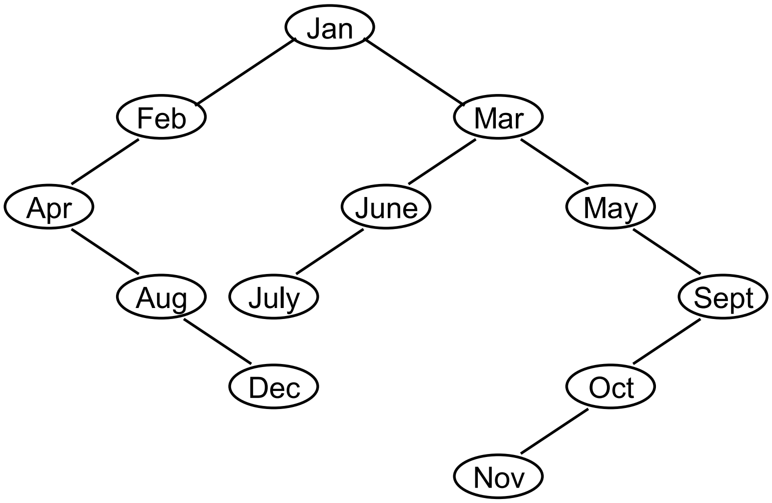
- worst case: O(n)

maintain the binary search tree as a complete binary tree

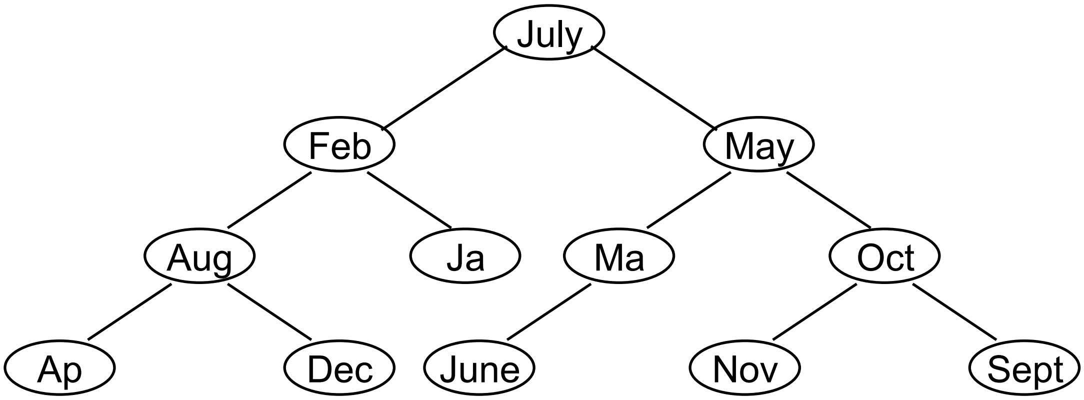
- minimize the average and maximum search time

- average and worst case: O(log2n)

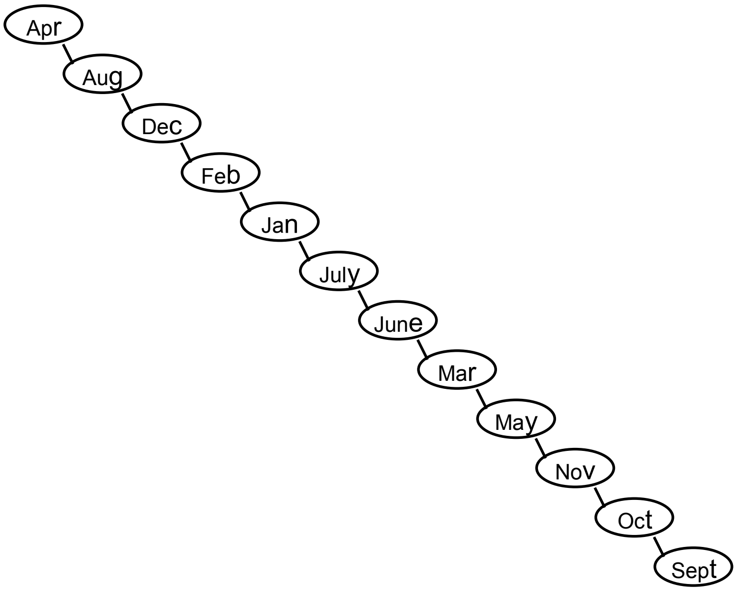
- a significant increase in the time required to add new element



binary search tree obtained for the months of the year



a balanced tree for the months of   the year



degenerate binary search tree

AVL Trees

- balanced binary trees

- average and worst case: O(log2n)

Def) *height balanced binary tree*

- an empty binary tree is height balanced

- if T is a nonempty binary tree with TL and TR as its left and right subtrees

- T is height balanced iff

  1) TL and TR are height balanced, and

  2) |hL - hR| £ 1 where hL and hR are height of TL and TR, respectively

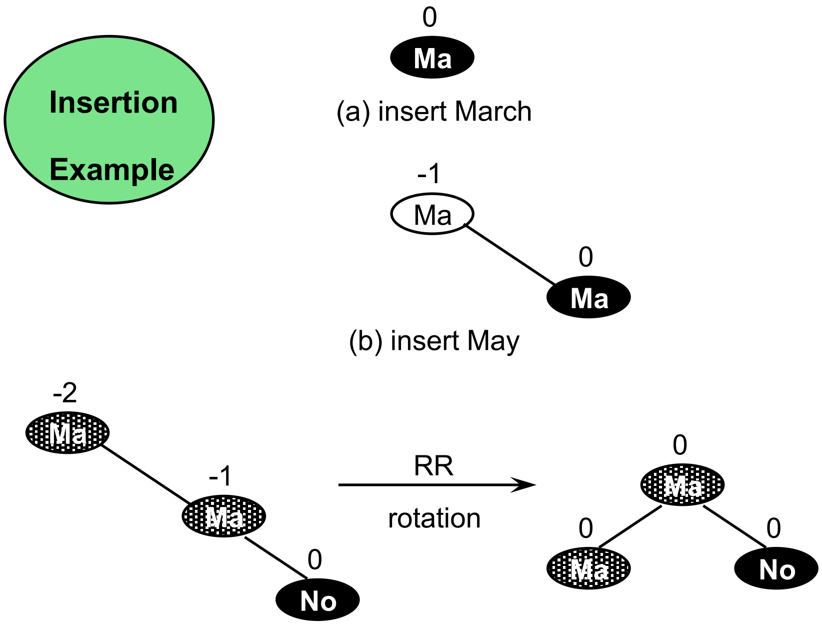
Def) *balance factor*, BF(T), of node T

in a binary tree

- hL - hR where hL and hR are heights of left and right subtree of T

- for any node T in an AVL tree, BF(T) = -1, 0, or 1

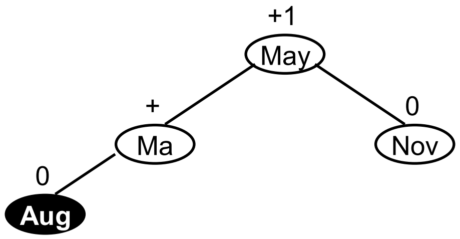
Insertion Example



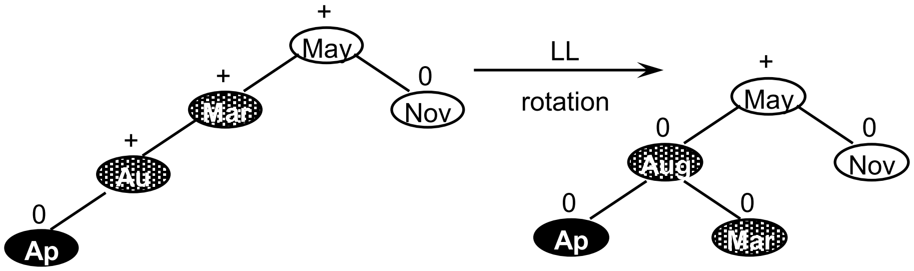
(a) insert March

(b) insert May

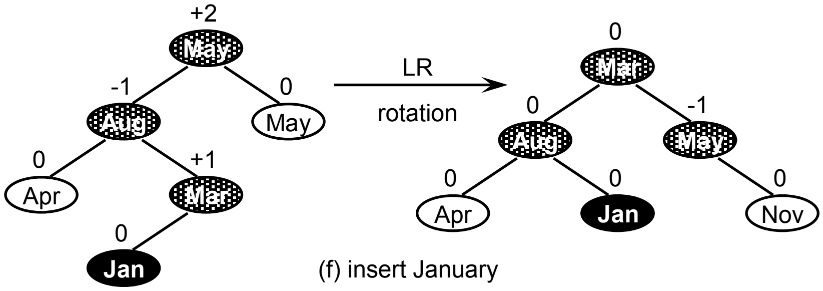
(c) insert November



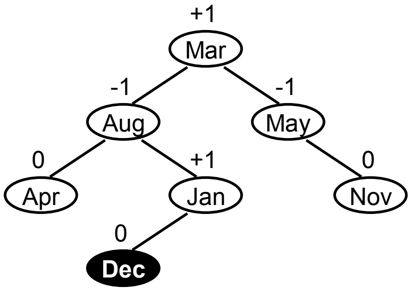
(d) insert August

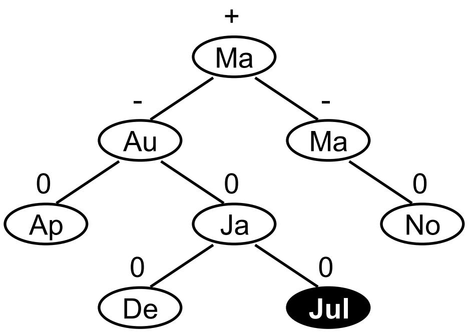


(e) insert April

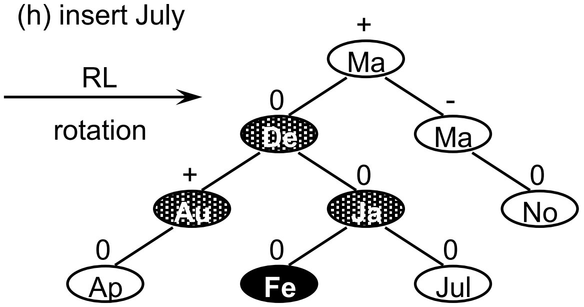


(f) insert January





(h) insert July



(i) insert February

four kinds of rotations to rebalance

- LL, LR, RR, RL

- LL and RR are symmetric

- LR and RL are symmetric

Let Y: new inserted node, and

A: the nearest ancestor of Y, whose balance factor becomes ±2

LL: Y is inserted in the left subtree of the left subtree of A

LR: Y is inserted in the right subtree of the left subtree of A

RR: Y is inserted in the right subtree

of the right subtree of A**RL**: Y is inserted in the left subtree of the right subtree of A

- height of the subtrees which are not involved in the rotation remain unchanged

RED-BLACK TREES

- Balanced" binary search trees guarantee an O(lgn) running time
- Red-black-tree
  - Binary search tree with an additional attribute for its nodes: color which can be **red** or **black**
  - Constrains the way nodes can be colored on any path from the root to a leaf:

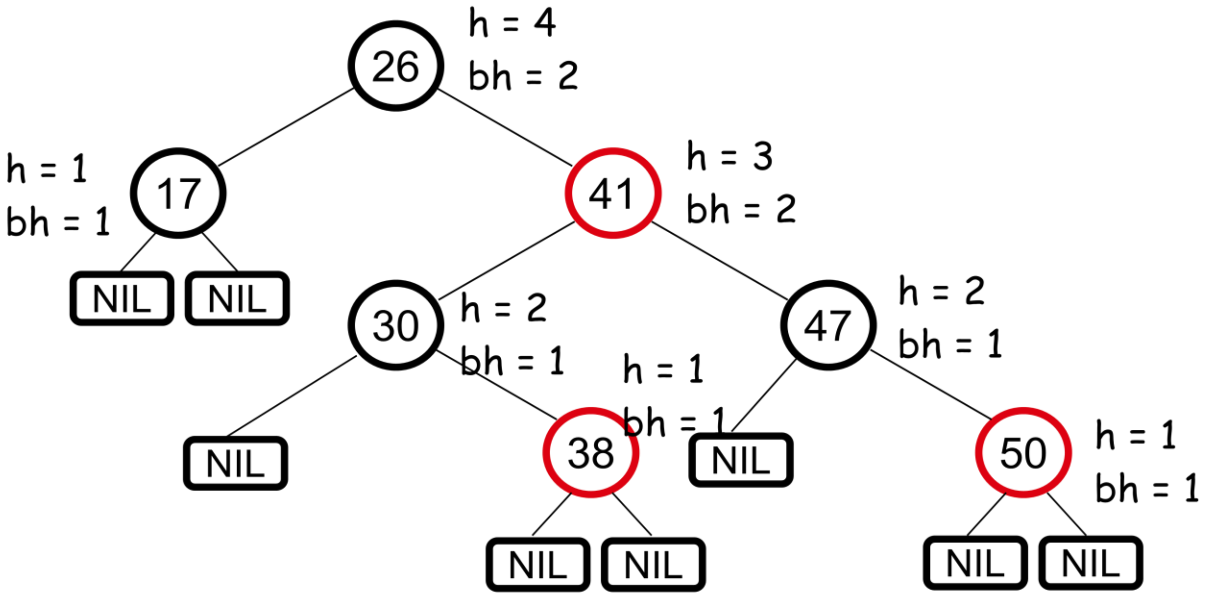
Ensures that no path is more than twice as long as any other path ⇒ the tree is balanced

- For convenience we use a sentinel NIL[T] to represent all the NIL nodes at the leafs
  - NIL[T] has the same fields as an ordinary node
  - Color[NIL[T]] = BLACK
  - The other fields may be set to arbitrary values

Red-Black-Trees Properties

1. Every node is either **red** or **black**
2. The root is **black**
3. Every leaf (NIL) is **black**
4. If a node is **red**, then both its children are **black**
   - No two consecutive red nodes on a simple path from the root to a leaf
1. For each node, all paths from that node to descendant leaves contain the same number of **black** nodes

Black-Height of a Node



- **Height of a node:** the number of edges in the **longest** path to a leaf
- **Black-height** of a node x: bh(x) is the number of black nodes (including NIL) on the path from x to a leaf,

not counting x

Overview: Most important property of Red-Black-Trees

A red-black tree with n internal nodes

has height at most 2lg(n + 1)

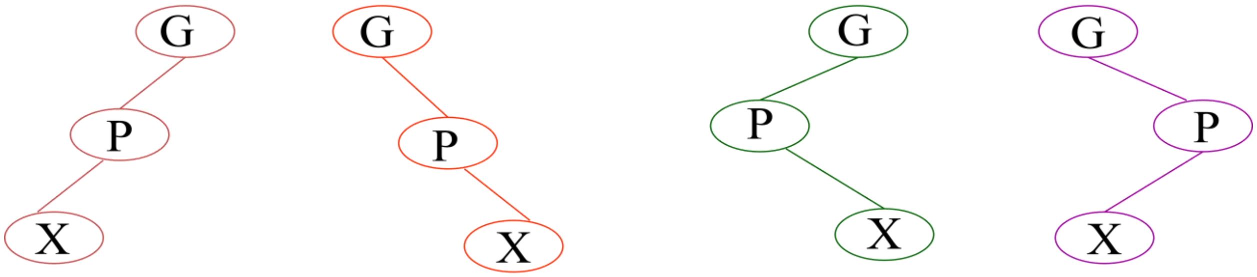
Need to prove two claims first …Any node x with height h(x) has bh(x) ≥ h(x)/2
Proof
By property 4, at most h/2 red nodes on the path from the node to a leaf
Hence at least h/2 are black

SPLAY TREES

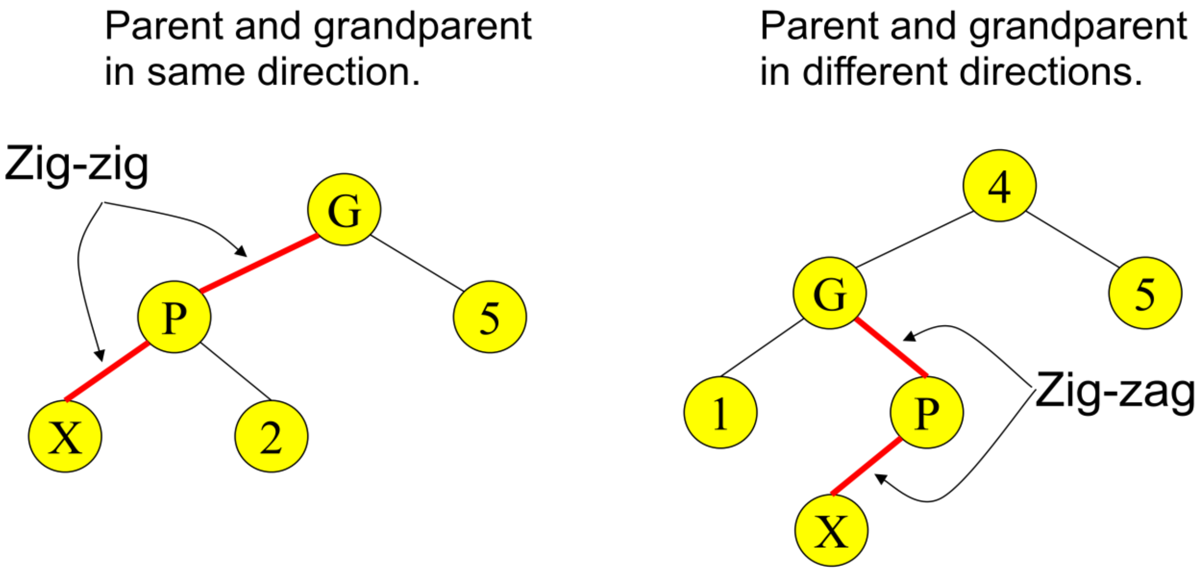
- Splay trees are tree structures that:
  - › Are not perfectly balanced all the time
  - › Data most recently accessed is near the root. (principle of locality; 80-20 “rule”)
- The procedure:

  - › After node X is accessed, perform “splaying” operations to bring X to the root of the tree.
  - › Do this in a way that leaves the tree more balanced as a whole

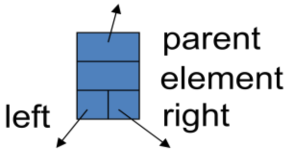
- Let X be a non-root node with ≥ 2 ancestors.
  - P is its parent node.
  - G is its grandparent node.



Zig-Zig and Zig-Zag



Splay Tree Operations



- Single Rotations (X has a P (the root) but no G)
  ZigFromLeft, ZigFromRight
- Double Rotations (X has both a P and a G)
  ZigZigFromLeft, ZigZigFromRight
  ZigZagFromLeft, ZigZagFromRight

In this unit pattern matching, is the act of checking some sequence of tokens for the presence of the constituents of some pattern .Uses of pattern matching include outputting the locations of a pattern within a token sequence, to output some component of the matched pattern, and to substitute the matching pattern with some other token sequence (I.e., search and replace).

Contents:

1. Pattern matching algorithms
2. Standard Tries, Compressed Tries, Suffix tries.

Brute Force algorithm

Main features

- no preprocessing phase;
- constant extra space needed;
- always shifts the window by exactly 1 position to the right;
- comparisons can be done in any order;
- searching phase in $\boldsymbol{O}(mn)$ time complexity;
- $2n$ expected text characters comparisons.

Description

The brute force algorithm consists in checking, at all positions in the text between 0 and n-m, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right.

The brute force algorithm requires no preprocessing phase, and a constant extra space in addition to the pattern and the text. During the searching phase the text character comparisons can be done in any order. The time complexity of this searching phase is $\boldsymbol{O}(mn)$ (when searching for $a^{m-1}b$ in a^n for instance). The expected number of text character comparisons is $2n$.

Boyer-Moore algorithm

Main features

- performs the comparisons from right to left;
- preprocessing phase in $\boldsymbol{O}(m+\sigma)$ time and space complexity;
- searching phase in $\boldsymbol{O}(mn)$ time complexity;
- $3n$ text character comparisons in the worst case when searching for a non periodic pattern;
- $\boldsymbol{O}(n / m)$ best performance.

Description

The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of it or the entire algorithm is often implemented in text editors for the «search» and «substitute» commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two shift functions are called the ***good-suffix shift*** (also called matching shift and the ***bad-character shift*** (also called the occurrence shift).

Assume that a mismatch occurs between the character *x*[*i*]=*a* of the pattern and the character *y*[*i*+*j*]=*b* of the text during an attempt at position *j*. Then, *x*[*i*+1 .. *m*-1]=*y*[*i*+*j*+1 .. *j*+*m*-1]=u and *x*[*i*] ≠ y[*i*+*j*]. The good-suffix shift consists in aligning the segment *y*[*i*+*j*+1 .. *j*+*m*-1]=*x*[*i*+1 .. *m*-1] with its rightmost occurrence in *x* that is preceded by a character different from *x*[*i*] (*see figure*).

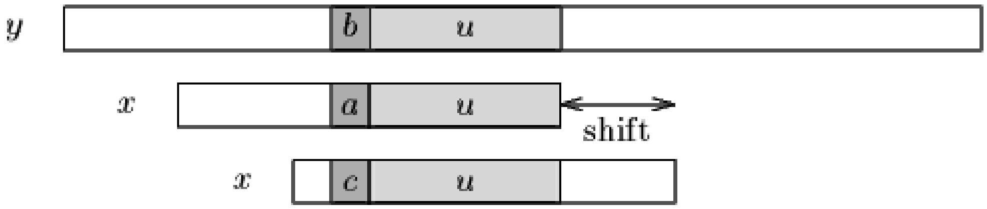


Figure . The good-suffix shift, *u* re-occurs preceded by a character *c* different from *a*.

If there exists no such segment, the shift consists in aligning the longest suffix *v* of *y*[*i*+*j*+1 .. *j*+*m*-1] with a matching prefix of *x* (*see figure*).

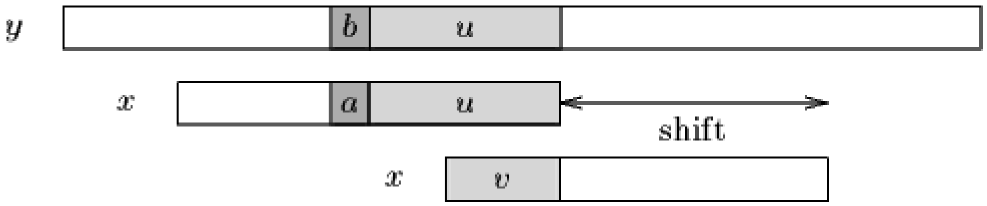


Figure . The good-suffix shift, only a suffix of *u* re-occurs in *x*.

The bad-character shift consists in aligning the text character *y*[*i*+*j*] with its rightmost occurrence in *x*[0 .. *m*-2]. (*see figure*)

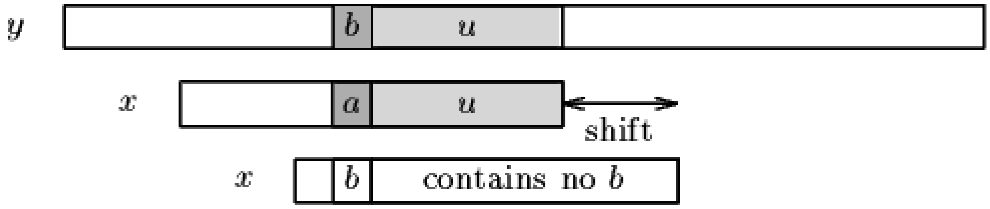


Figure . The bad-character shift, *a* occurs in *x*.

If $y[i+j]$ does not occur in the pattern x, no occurrence of x in y can include $y[i+j]$, and the left end of the window is aligned with the character immediately after $y[i+j]$, namely $y[i+j+1]$ (*see figure*).

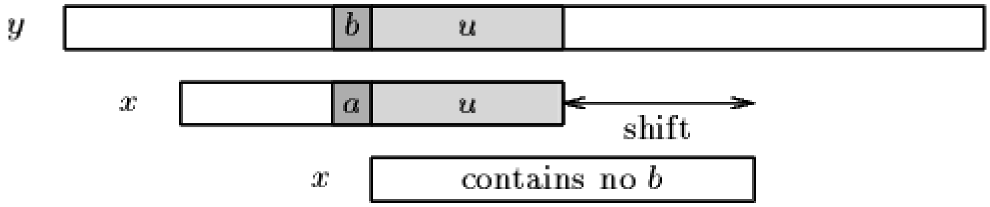


Figure . The bad-character shift, *b* does not occur in *x*.

Note that the bad-character shift can be negative, thus for shifting the window, the Boyer-Moore algorithm applies the maximum between the the good-suffix shift and bad-character shift. More formally the two shift functions are defined as follows.

The good-suffix shift function is stored in a table *bmGs* of size $m+1$.

Let us define two conditions:

i, s): for each k such that $i < k < m$, $s \geqslant k$ or $x[k-s]=x[k]$ and

i, s): if $s < i$ then $x[i-s] \neq x[i]$

Then, for $0 \leqslant i < m$: $bmGs[i+1]=\min\{s>0 : Cs(i, s) \text{ and } Co(i, s) \text{ hold}\}$
and we define $bmGs[0]$ as the length of the period of x. The computation of the table *bmGs* use a table *suff* defined as follows: for $1 \leqslant i < m$, $suff[i]=\max\{k : x[i-k+1 .. i]=x[m-k .. m-1]\}$

The bad-character shift function is stored in a table *bmBc* of size σ. For c in Σ: $bmBc[c] = \min\{i : 1 \leqslant i < m-1 \text{ and } x[m-1-i]=c\}$ if c occurs in x, m otherwise.

Tables *bmBc* and *bmGs* can be precomputed in time $\boldsymbol{O}(m+\sigma)$ before the searching phase and require an extra-space in $\boldsymbol{O}(m+\sigma)$. The searching phase time complexity is quadratic but at most $3n$ text character comparisons are performed when searching for a non periodic pattern. On large alphabets (relatively to the length of the pattern) the algorithm is extremely fast. When searching for $a^{m-1}b$ in b^n

the algorithm makes only ***O***(*n* / *m*) comparisons, which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed.

Knuth-Morris-Pratt string matching

The problem: given a (short) pattern and a (long) text, both strings, determine whether the pattern appears somewhere in the text. Last time we saw how to do this with finite automata. This time we'll go through the Knuth-Morris-Pratt (KMP) algorithm, which can be thought of as an efficient way to build these automata. I also have some working C++ source code which might help you understand the algorithm better.

First let's look at a naive solution.
suppose the text is in an array: char T[n]
and the pattern is in another array: char P[m].

One simple method is just to try each possible position the pattern could appear in the text.

Naive string matching:

```
for (i=0; T[i] != '\0'; i++)
{
for (j=0; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++) ;
if (P[j] == '\0') found a match
}
```

There are two nested loops; the inner one takes O(m) iterations and the outer one takes O(n) iterations so the total time is the product, O(mn). This is slow; we'd like to speed it up.

In practice this works pretty well -- not usually as bad as this O(mn) worst case analysis. This is because the inner loop usually finds a mismatch quickly and move on to the next position without going through all m steps. But this method still can take O(mn) for some inputs. In one bad example, all characters in T[] are "a"s, and P[] is all "a"'s except for one "b" at the end. Then it takes m comparisons each time to discover that you don't have a match, so mn overall.

Here's a more typical example. Each row represents an iteration of the outer loop, with each character in the row representing the result of a comparison (X if the comparison was unequal). Suppose we're looking for pattern "nano" in text "banananobano".

```
 0  1  2  3  4  5  6  7  8  9 10 11
  T: b  a  n  a  n  a  n  o  b  a  n  o

i=0: X
```

```
i=1:  X
i=2:    n a n X
i=3:      X
i=4:        n a n o
i=5:          X
i=6:            n X
i=7:              X
i=8:                X
i=9:                  n X
i=10:                   X
```

Some of these comparisons are wasted work! For instance, after iteration i=2, we know from the comparisons we've done that T[3]="a", so there is no point comparing it to "n" in iteration i=3. And we also know that T[4]="n", so there is no point making the same comparison in iteration i=4.

Skipping outer iterations

The Knuth-Morris-Pratt idea is, in this sort of situation, after you've invested a lot of work making comparisons in the inner loop of the code, you know a lot about what's in the text. Specifically, if you've found a partial match of j characters starting at position i, you know what's in positions T[i]...T[i+j-1].

You can use this knowledge to save work in two ways. First, you can skip some iterations for which no match is possible. Try overlapping the partial match you've found with the new match you want to find:

```
i=2: n a n
i=3:   n a n o
```

Here the two placements of the pattern conflict with each other -- we know from the i=2 iteration that T[3] and T[4] are "a" and "n", so they can't be the "n" and "a" that the i=3 iteration is looking for. We can keep skipping positions until we find one that doesn't conflict:

```
i=2: n a n
i=4:     n a n o
```

Here the two "n"'s coincide. Define the *overlap* of two strings x and y to be the longest word that's a suffix of x and a prefix of y. Here the overlap of "nan" and "nano" is just "n". (We don't allow the overlap to be all of x or y, so it's not "nan"). In general the value of i we want to skip to is the one corresponding to the largest overlap with the current partial match:

String matching with skipped iterations:

```
i=0;
while (i<n)
{
for (j=0; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++) ;
if (P[j] == '\0') found a match;
i = i + max(1, j-overlap(P[0..j-1],P[0..m]));
}
```

Skipping inner iterations

The other optimization that can be done is to skip some iterations in the inner loop. Let's look at the same example, in which we skipped from i=2 to i=4:

```
i=2: n  a  n
i=4:       n  a  n  o
```

In this example, the "n" that overlaps has already been tested by the i=2 iteration. There's no need to test it again in the i=4 iteration. In general, if we have a nontrivial overlap with the last partial match, we can avoid testing a number of characters equal to the length of the overlap.

Standard Tries

•The standard trie for a set of strings S is an ordered tree such that:

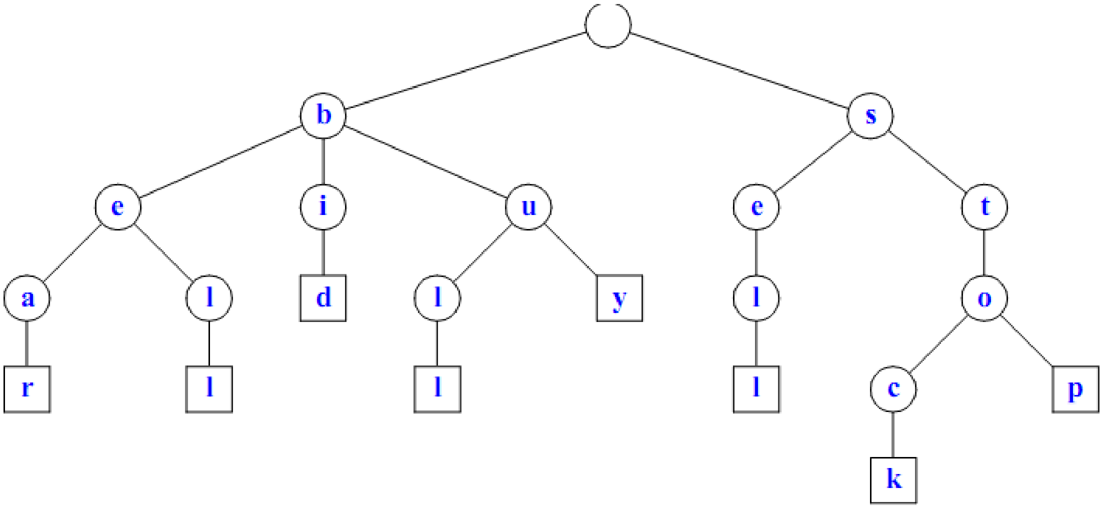
-each node but the root is labeled with a character

-the children of a node are alphabetically ordered

-the paths from the external nodes to the root yield the strings of S

Example: standard trie for the set of strings

S = { bear, bell, bid, bull, buy, sell, stock, stop }



A standard trie uses O(n) space. Operations (find, insert, remove) take time O(dm) each, where:

-n = total size of the strings in S,

-m =size of the string parameter of the operation

-d =alphabet size,

Applications of Tries

•A standardtrie supports the following operations on a preprocessed text in time O(m), where m = |X|

- wordmatching:find the first occurrence of word X in the text

- prefix matching: find the first occurrence of the longest prefix of word X in the text

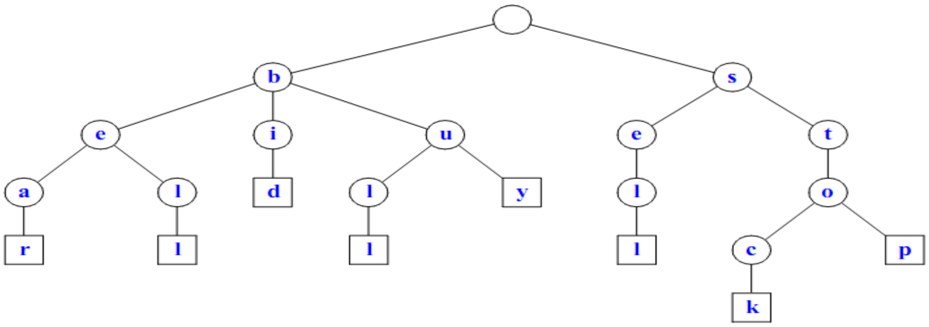
•Each operation is performed by tracing a path in the trie starting at the root

Compressed Tries

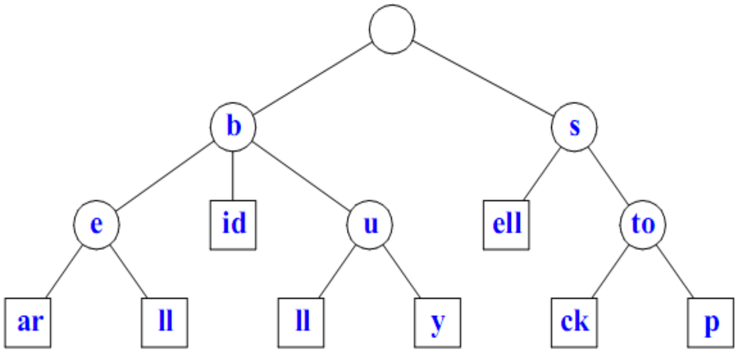
•Trie with nodes of degree at least 2

•Obtained from standard trie by compressing chains of redundant nodes

- Standard Trie:



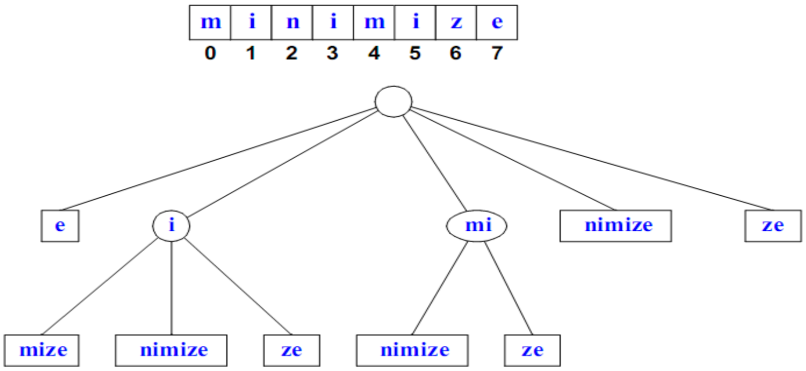
- Compressed Trie:



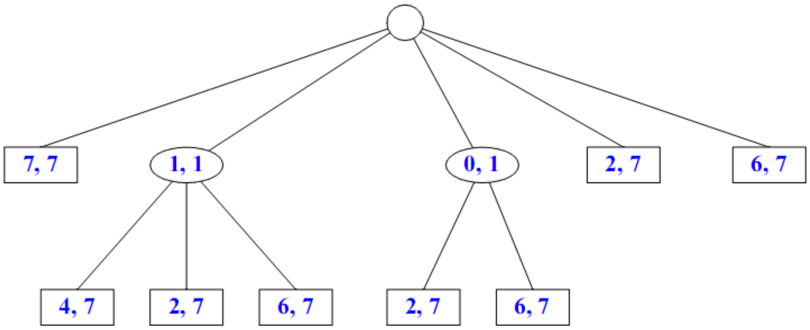
Suffix Tries

•A suffix trie is a compressed trie for all the suffixes of a text

Example



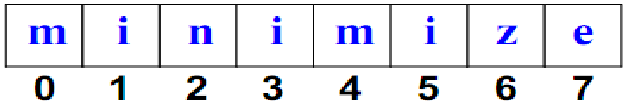
- Compact representation:



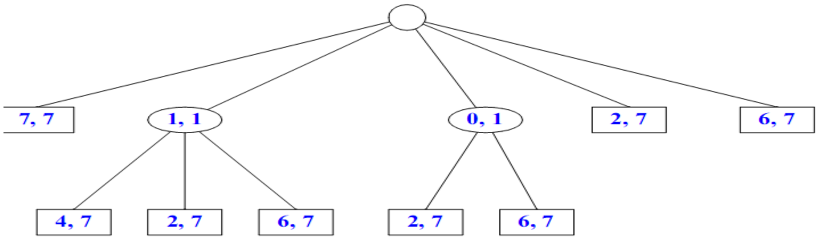
Properties of Suffix Tries

•The suffixtrie foratextXofsize n from an alphabet of size d -stores all the n(n−1)/2 suffixes of X in O(n) space

-supports arbitrary patternmatching and prefix matching queries in O(dm) time, where m is the length of the pattern -can be constructed in O(dn) time



m	i	n	i	m	i	z	e
0	1	2	3	4	5	6	7


7, 7
1, 1
0, 1
2, 7
6, 7
4, 7
2, 7
6, 7
2, 7
6, 7