



CATHOLIC UNIVERSITY OF RWANDA

P.O. Box 49 BUTARE / HUYE - RWANDA
TEL: (00250) 0252 530 893 FAX: (00250) 0252 530 627
E-mail: administration@cur.ac.rw
Web: www.cur.ac.rw

FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE
LEVEL II WEEKEND PROGRAM

**MODULE: DATA STRUCTURES AND
ALGORITHMS ANALYSIS (DSAA2513)
SYLLABUS**

LECTURER: Mr. Elias NTAWUZUMUNSI

ACADEMIC YEAR: 2017-2018

Table of Contents

| | |
|---|-----|
| CHAPTER 1: INTRODUCTION TO C++ PROGRAMMING LANGUAGE..... | 3 |
| Basics of C++..... | 4 |
| Data Types in C++..... | 5 |
| Operators in C++ | 10 |
| Decision making in C++ | 14 |
| Looping in C++..... | 18 |
| Functions in C++ | 22 |
| CHAPTER 2: OBJECT ORIENTED PROGRAMMING | 26 |
| Defining Class and Declaring Objects | 31 |
| Overview of Inheritance | 34 |
| Inheritance Visibility Mode | 35 |
| Types of Inheritance | 36 |
| Polymorphism..... | 38 |
| CHAPTER III: FUNDAMENTAL DATA STRUCTURES | 39 |
| 1.STACK | 39 |
| 2. QUEUES..... | 47 |
| 3. LINKED LISTS | 51 |
| 4. HASHING..... | 55 |
| 5. TREE DATA STRUCTURES | 62 |
| 6. GRAPHS..... | 74 |
| CHAPTER 4: ANALYSIS OF ALGORITHM | 80 |
| 4.1 Algorithmic Complexity | 80 |
| 4.2 How to Design Algorithms | 84 |
| 4.3 How to Express Algorithms | 85 |
| 4.4 Fundamental Concepts of Algorithm | 85 |
| 4.5 FUNDAMENTAL ALGORITHMS STRATEGIES | 86 |
| 4.6 GRAPH AND TREE ALGORITHMS | 104 |
| 4.6.1 Depth-first search (DFS) for undirected graphs..... | 104 |
| 4.6.2 Breadth-First Search Traversal Algorithm..... | 109 |
| 4.6.3 Dijkstra's shortest path algorithm..... | 115 |
| 4.6.4 Kruskal's Algorithm..... | 121 |

CHAPTER 1: INTRODUCTION TO C++ PROGRAMMING LANGUAGE

What we will cover in Basics of C++

- Overview and OOPS concept
- Basic Syntax and Structure of programming
- Data Types and Modifiers
- Variables and Operators in C++
- Decision Making
- Types of Loops
- Types of Storage Classes
- Introduction to Functions

C++, as we all know is an extension to C language and was developed by **Bjarne Stroustrup** at bell labs. C++ is an intermediate level language, as it comprises a confirmation of both high level and low level language features. C++ is a statically typed, free form, multiparadigm, compiled general-purpose language.

C++ is an **Object Oriented Programming language** but is not purely Object Oriented. Its features like `friend` and `virtual`, violate some of the very important OOPS features, rendering this language unworthy of being called completely Object Oriented. Its a middle level language.

Benefits of C++ over C Language

The major difference being OOPS concept, C++ is an object oriented language whereas C language is a procedural language. Apart from this there are many other features of C++ which gives this language an upper hand on C language.

Following features of C++ makes it a stronger language than C,

1. There is Stronger Type Checking in C++.
2. All the OOPS features in C++ like Abstraction, Encapsulation, Inheritance etc makes it more worthy and useful for programmers.
3. C++ supports and allows user defined operators (i.e Operator Overloading) and function overloading is also supported in it.
4. Exception Handling is there in C++.
5. The Concept of Virtual functions and also Constructors and Destructors for Objects.
6. Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.
7. Variables can be declared anywhere in the program in C++, but must be declared before they are used.

Basics of C++

In this section we will cover the basics of C++, it will include the syntax, variable, operators, loop types, pointers, references and information about other requirements of a C++ program. You will come across lot of terms that you have already studied in C language.

Syntax and Structure of C++ program

Here we will discuss one simple and basic C++ program to print "Hello this is C++" and its structure in parts with details and uses.

First C++ program

```
include <iostream>
using namespace std;
int main()
{
cout << "Hello this is C++";
}
```

Header files are included at the beginning just like in C program. Here `iostream` is a header file which provides us with input & output streams. Header files contained predeclared function libraries, which can be used by users for their ease.

Using namespace std, tells the compiler to use standard namespace. Namespace collects identifiers used for class, object and variables. Namespace can be used by two ways in a program, either by the use of `using` statement at the beginning, like we did in above mentioned program or by using name of namespace as prefix before the identifier with scope resolution (`::`) operator.

Example `:std::cout << "A";`

main(), is the function which holds the executing part of program its return type is `int`.

cout <<, is used to print anything on screen, same as `printf` in C language. **cin** and **cout** are same as `scanf` and `printf`, only difference is that you do not need to mention format specifiers like, `%d` for `int` etc, in `cout` & `cin`.

Comments

For single line comments, use `//` before mentioning comment, like

```
cout<<"single line"; // This is single line comment
For multiple line comment, enclose the comment between /* and */
```

```
/*this is
a multiple line
```

```
comment */
```

Using Classes

Classes name must start with capital letter, and they contain data variables and member functions. This is a mere introduction to classes, we will discuss classes in detail throughout the C++ tutorial.

```
class Abc
{
    int i;           //data variable
    void display()  //Member Function
    {
        cout<<"Inside Member Function";
    }
}; // Class ends here

int main()
{
    Abc obj; // Creatig Abc class's object
    obj.display(); //Calling member function using class object
}
```

This is how class is defined, its object is created and the member functions are used.

Variables can be declared anywhere in the entire program, but must be declared, before they are used. Hence, we don't need to declare variable at the start of the program.

Data Types in C++

They are used to define type of variables and contents used. Data types define the way you use storage in the programs you write. Data types can be built in or abstract.

Built in Data Types

These are the data types which are predefined and are wired directly into the compiler. eg: int, char etc.

User defined or Abstract data types

These are the type, that user creates as a class. In C++ these are classes where as in C it was implemented by structures.

Basic Built in types

| | |
|------|----------------------------------|
| char | for character storage (1 byte) |
| int | for integral number (2 bytes) |

| | |
|--------|---|
| float | single precision floating point (4 bytes) |
| double | double precision floating point numbers (8 bytes) |

Example :

```
char a = 'A';           // character type
int a = 1;             // integer type
float a = 3.14159;     // floating point type
double a = 6e-4;      // double type (e is for exponential)
```

Other Built in types

| | |
|---------|---------------------------|
| bool | Boolean (True or False) |
| void | Without any Value |
| wchar_t | Wide Character |

Enum as Data type

Enumerated type declares a new type-name and a sequence of value containing identifiers which has values starting from 0 and incrementing by 1 every time.

For Example :

```
enum day(mon, tues, wed, thurs, fri) d;
```

Here an enumeration of days is defined with variable *d*. *mon* will hold value 0, *tue* will have 1 and so on. We can also explicitly assign values, like, `enum day(mon, tue=7, wed);`. Here, *mon* will be 0, *tue* is assigned 7, so *wed* will have value 8.

Modifiers

Specifiers modify the meanings of the predefined built-in data types and expand them to a much larger set. There are four data type modifiers in C++, they are :

1. long
2. short
3. signed
4. unsigned

Below mentioned are some important points you must know about the modifiers,

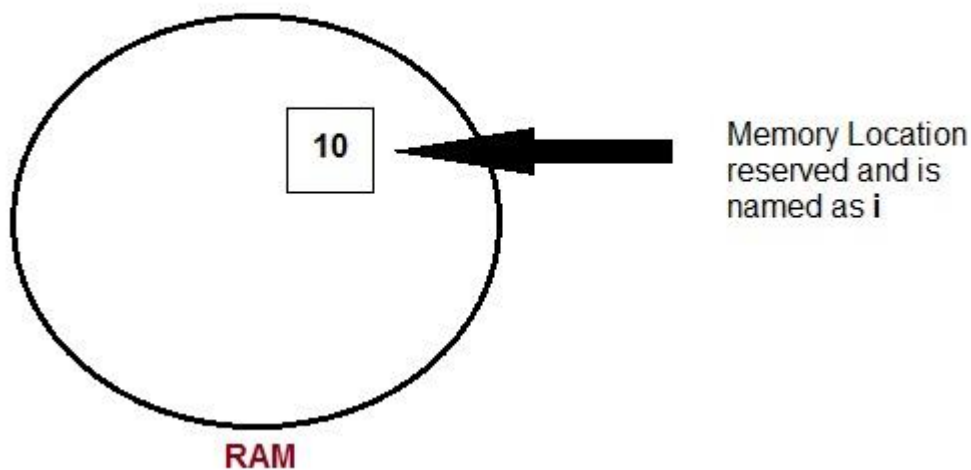
- **long** and **short** modify the maximum and minimum values that a data type will hold.
- A plain int must have a minimum size of **short**.

- Size hierarchy : `short int < int < long int`
- Size hierarchy for floating point numbers is : `float < double < long double`
- **long float** is not a legal type and there are no **short floating point** numbers.
- **Signed** types includes both positive and negative numbers and is the default type.
- **Unsigned**, numbers are always without any sign, that is always positive.

What are Variables

Variable are used in C++, where we need storage for any value, which will change in program. Variable can be declared in multiple ways each with different memory requirements and functioning. Variable is the name of memory location allocated by the compiler depending upon the datatype of the variable.

Example : `int i=10; // declared and initialised`



Basic types of Variables

Each variable while declaration must be given a datatype, on which the memory assigned to the variable depends. Following are the basic types of variables,

| | |
|--|---|
| <code>bool</code> | For variable to store boolean values(True or False) |
| <code>char</code> | For variables to store character types. |
| <code>int</code> | for variable with integral values |
| <code>float</code> and <code>double</code> are also types for variables with large and floating point values | |

Declaration and Initialization

Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

Example :

```
int i;          // declared but not initialised
char c;
int i, j, k;   // Multiple declaration
```

Initialization means assigning value to an already declared variable,

```
int i;  // declaration
i = 10; // initialization
```

Initialization and declaration can be done in one single step also,

```
int i=10;          //initialization and declaration in same step
int i=10, j=11;
```

If a variable is declared and not initialized by default it will hold a garbage value. Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

```
int i,j;
i=10;
j=20;
int j=i+j;    //compile time error, cannot redeclare a variable in same
scope
```

Scope of Variables

All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable. For most of the cases its between the curly braces,in which variable is declared that a variable exists, not outside it. We will study the storage classes later, but as of now, we can broadly divide variables into two main types,

- Global Variables
- Local variables

Global variables

Global variables are those, which ar once declared and can be used throughout the lifetime of the program by any class or any function. They must be declared outside the `main()` function. If only declared, they can be assigned different values at different time in program lifetime. But even if they are declared and initialized at the same time outside the `main()` function, then also they can be assigned any value at any point in the program.

Example : Only declared, not initialized

```
include <iostream>
using namespace std;
int x;                // Global variable declared
int main()
{
    x=10;              // Initialized once
    cout <<"first value of x = "<< x;
    x=20;              // Initialized again
    cout <<"Initialized again with value = "<< x;
}
```

Local Variables

Local variables are the variables which exist only between the curly braces, in which its declared. Outside that they are unavailable and leads to compile time error.

Example :

```
include <iostream>
using namespace std;
int main()
{
    int i=10;
    if(i<20)          // if condition scope starts
    {
        int n=100;    // Local variable declared and initialized
    }                 // if condition scope ends
    cout << n;        // Compile time error, n not available here
}
```

Some special types of variable

There are also some special keywords, to impart unique characteristics to the variables in the program. Following two are mostly used, we will discuss them in details later.

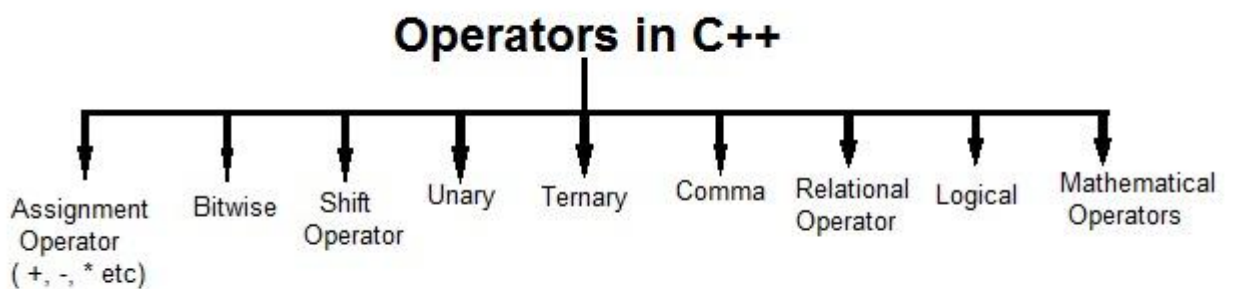
1. **Final** - Once initialized, its value can't be changed.
2. **Static** - These variables holds their value between function calls.

Example :

```
include <iostream>
using namespace std;
int main()
{
    final int i=10;
    static int y=20;
}
```

Operators in C++

Operators are special type of functions that takes one or more arguments and produces a new value. For example: addition (+), subtraction (-), multiplication (*) etc, are all operators. Operators are used to perform various operations on variables and constants.



Types of operators

1. Assignment Operator
2. Mathematical Operators
3. Relational Operators
4. Logical Operators
5. Bitwise Operators
6. Shift Operators
7. Unary Operators
8. Ternary Operator
9. Comma Operator

Assignment Operator (=)

Operates '=' is used for assignment, it takes the right-hand side (called rvalue) and copy it into the left-hand side (called lvalue). Assignment operator is the only operator which can be overloaded but cannot be inherited.

Mathematical Operators

There are operators used to perform basic mathematical operations. Addition (+) , subtraction (-) , division (/) multiplication (*) and modulus (%) are the basic mathematical operators. Modulus operator cannot be used with floating-point numbers.

C++ and C also use a shorthand notation to perform an operation and assignment at same type.

Example,

```
int x=10;
x += 4 // will add 4 to 10, and hence assign 14 to X.
x -= 5 // will subtract 5 from 10 and assign 5 to x.
```

Relational Operators

These operators establish a relationship between operands. The relational operators are : less than (<), greater than (>), less than or equal to (<=), greater than equal to (>=), equivalent (==) and not equivalent (!=).

You must notice that assignment operator is (=) and there is a relational operator, for equivalent (==). These two are different from each other, the assignment operator assigns the value to any variable, whereas equivalent operator is used to compare values, like in if-else conditions, *Example*

```
int x = 10; //assignment operator
x=5;       // again assignment operator
if(x == 5) // here we have used equivalent relational operator, for
comparison
{
    cout <<"Successfully compared";
}
```

Logical Operators

The logical operators are AND (&&) and OR (||). They are used to combine two different expressions together.

If two statements are connected using AND operator, the validity of both statements will be considered, but if they are connected using OR operator, then either one of them must be valid. These operators are mostly used in loops (especially `while` loop) and in Decision making.

Bitwise Operators

They are used to change individual bits into a number. They work with only integral data types like `char`, `int` and `long` and not with floating point values.

- Bitwise AND operators &

- Bitwise OR operator |
- And bitwise XOR operator ^
- And, bitwise NOT operator ~

They can be used as shorthand notation too, & = , |= , ^= , ~= etc.

Shift Operators

Shift Operators are used to shift Bits of any variable. It is of three types,

1. Left Shift Operator <<
2. Right Shift Operator >>
3. Unsigned Right Shift Operator >>>

Unary Operators

These are the operators which work on only one operand. There are many unary operators, but increment ++ and decrement -- operators are most used.

Other Unary Operators : address of &, dereference *, **new** and **delete**, bitwise not ~, logical not !, unary minus - and unary plus +.

Ternary Operator

The ternary if-else ? : is an operator which has three operands.

```
int a = 10;
a > 5 ? cout << "true" : cout << "false"
```

Comma Operator

This is used to separate variable names and to separate expressions. In case of expressions, the value of last expression is produced and used.

Example :

```
int a,b,c; // variables declaration using comma operator
a=b++, c++; // a = c++ will be done.
```

sizeof operator in C++

sizeof is also an operator not a function, it is used to get information about the amount of memory allocated for data types & Objects. It can be used to get size of user defined data types too.

sizeof operator can be used with and without parentheses. If you apply it to a variable you can use it without parentheses.

```
cout << sizeof(double); //Will print size of double
int x = 2;
int i = sizeof x;
```

typedef Operator

typedef is a keyword used in C language to assign alternative names to existing types. Its mostly used with user defined data types, when names of data types get slightly complicated. Following is the general syntax for using typedef,

```
typedef existing_name alias_name
```

Lets take an example and see how typedef actually works.

```
typedef unsigned long ulong;
```

The above statement define a term **ulong** for an unsigned long type. Now this **ulong** identifier can be used to define unsigned long type variables.

```
ulong i, j ;
```

typedef and Pointers

typedef can be used to give an alias name to pointers also. Here we have a case in which use of typedef is beneficial during pointer declaration.

In Pointers * binds to the right and not the left.

```
int* x, y ;
```

By this declaration statement, we are actually declaring **x** as a pointer of type int, whereas **y** will be declared as a plain integer.

```
typedef int* IntPtr ;
IntPtr x, y, z;
```

But if we use **typedef** like in above example, we can declare any number of pointers in a single statement.

Decision making in C++

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C++ handles decision-making by supporting the following statements,

- *if* statement
- *switch* statement
- conditional operator statement
- *goto* statement

Decision making with *if* statement

The *if* statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

1. Simple *if* statement
2. *If...else* statement
3. Nested *if...else* statement
4. *else if* statement

Simple *if* statement

The general form of a simple *if* statement is,

```
if( expression )
{
    statement-inside;
}
statement-outside;
```

If the *expression* is true, then 'statement-inside' it will be executed, otherwise 'statement-inside' is skipped and only 'statement-outside' is executed.

Example :

```
#include< iostream.h>
int main( )
{
    int x,y;
    x=15;
    y=13;
    if (x > y )
    {
        cout << "x is greater than y";
    }
}
```

Output : x is greater than y

***if...else* statement**

The general form of a simple *if...else* statement is,

```
if( expression )
{
    statement-block1;
}
else
{
    statement-block2;
}
```

If the 'expression' is true, the 'statement-block1' is executed, else 'statement-block1' is skipped and 'statement-block2' is executed.

Example :

```
void main( )
{
    int x,y;
    x=15;
    y=18;
    if (x > y )
    {
        cout << "x is greater than y";
    }
    else
    {
        cout << "y is greater than x";
    }
}
```

Output : y is greater than x

Nested *if...else* statement

The general form of a nested *if...else* statement is,

```
if( expression )
{
    if( expression1 )
    {
        statement-block1;
    }
    else
    {
        statement-block 2;
    }
}
```

```

    }
}
else
{
    statement-block 3;
}

```

if 'expression' is false the 'statement-block3' will be executed, otherwise it continues to perform the test for 'expression 1' . If the 'expression 1' is true the 'statement-block1' is executed otherwise 'statement-block2' is executed.

Example :

```

void main( )
{
    int a,b,c;
    clrscr();
    cout << "enter 3 number";
    cin >> a >> b >> c;
    if(a > b)
    {
        if( a > c)
        {
            cout << "a is greatest";
        }
        else
        {
            cout << "c is greatest";
        }
    }
    else
    {
        if( b> c)
        {
            cout << "b is greatest";
        }
        else
        {
            printf("c is greatest");
        }
    }
    getch();
}

```

else-if ladder

The general form of else-if ladder is,

```

if(expression 1)
{
    statement-block1;
}

```



```

}
else if(expression 2)
{
    statement-block2;
}
else if(expression 3 )
{
    statement-block3;
}
else
    default-statement;

```

The expression is tested from the top(of the ladder) downwards. As soon as the true condition is found, the statement associated with it is executed.

Example :

```

void main( )
{
    int a;
    cout << "enter a number";
    cin >> a;
    if( a%5==0 && a%8==0)
    {
        cout << "divisible by both 5 and 8";
    }
    else if( a%8==0 )
    {
        cout << "divisible by 8";
    }
    else if(a%5==0)
    {
        cout << "divisible by 5";
    }
    else
    {
        cout << "divisible by none";
    }
    getch();
}

```

Points to Remember

1. In *if* statement, a single statement can be included without enclosing it into curly braces { }
2. int a = 5;
3. if(a > 4)
4. cout << "success";

No curly braces are required in the above case, but if we have more than one statement inside *if* condition, then we must enclose them inside curly braces.

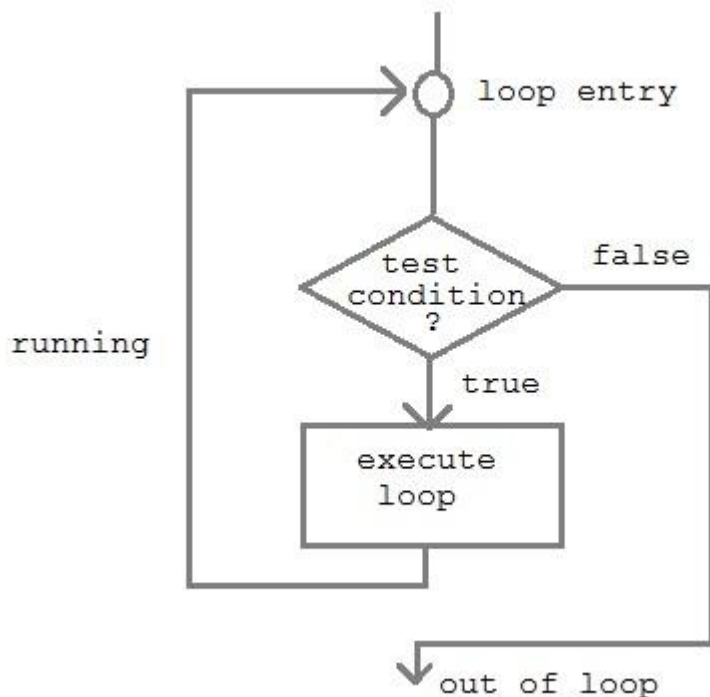
5. == must be used for comparison in the expression of *if* condition, if you use = the expression will always return true, because it performs assignment not comparison.
6. Other than **0(zero)**, all other values are considered as true.
7. `if(27)`
8. `cout << "hello";`

In above example, hello will be printed.

Looping in C++

In any programming language, loops are used to execute a set of statements repeatedly until a particular condition is satisfied.

How it works



A sequence of statement is executed until a specified condition is true. This sequence of statement to be executed is kept inside the curly braces { } known as loop body. After every execution of loop body, condition is checked, and if it is found to be **true** the loop body is executed again. When condition check comes out to be **false**, the loop body will not be executed.

There are 3 type of loops in C++ language

1. *while* loop
2. *for* loop
3. *do-while* loop

while loop

while loop can be address as an **entry control** loop. It is completed in 3 steps.

- Variable initialization.(e.g int x=0;)
- condition(e.g while(x<=10))
- Variable increment or decrement (x++ or x-- or x=x+2)

Syntax :

```
variable initialization ;
while (condition)
{
    statements ;
    variable increment or decrement ;
}
```

for loop

for loop is used to execute a set of statement repeatedly until a particular condition is satisfied. we can say it an **open ended loop**. General format is,

```
for(initialization; condition ; increment/decrement)
{
    statement-block;
}
```

In **for** loop we have exactly two semicolons, one after initialization and second after condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. **for** loop can have only one **condition**.

Nested for loop

We can also have nested **for** loop, i.e one **for** loop inside another **for** loop. Basic syntax is,

```
for(initialization; condition; increment/decrement)
{
    for(initialization; condition; increment/decrement)
```

```
    {  
        statement ;  
    }  
}
```

do while loop

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of **do-while** loop. **do** statement evaluates the body of the loop first and at the end, the condition is checked using **while** statement. General format of **do-while** loop is,

```
do  
{  
    ....  
    .....  
}  
while (condition)
```

Jumping out of loop

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becomes true, that is jump out of loop. C language allows jumping from one statement to another within a loop as well as jumping out of the loop.

1) break statement

When **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

2) continue statement

It causes the control to go directly to the test-condition and then continue the loop process. On encountering continue, cursor leave the current cycle of loop, and starts with the next cycle.

Storage Classes in C++

Storage classes are used to specify the lifetime and scope of variables. How storage is allocated for variables and How variable is treated by compiler depends on these storage classes.

These are basically divided into 5 different types :

1. Global variables
2. Local variables
3. Register variables
4. Static variables
5. Extern variables

Global Variables

These are defined at the starting , before all function bodies and are available throughout the program.

```
using namespace std;
int globe;          // Global variable
void func();
int main()
{
    .....
}
```

Local variables

They are defined and are available within a particular scope. They are also called **Automatic variable** because they come into being when scope is entered and automatically go away when the scope ends.

The keyword **auto** is used, but by default all local variables are auto, so we don't have to explicitly add keyword auto before variable declaration. Default value of such variable is **garbage**.

Register variables

This is also a type of local variable. This keyword is used to tell the compiler to make access to this variable as fast as possible. Variables are stored in registers to increase the access speed.

But you can never use or compute **address of register variable** and also , a register variable can be declared only within a **block**, that means, you cannot have *global* or *static register variables*.

Static Variables

Static variables are the variables which are initialized & allocated storage only once at the beginning of program execution, no matter how many times they are used and called in the program. A static variable retains its value until the end of program.

```

void fun()
{
    static int i = 10;
    i++;
    cout << i;
}
int main()
{
    fun();      // Output = 11
    fun();      // Output = 12
    fun();      // Output = 13
}

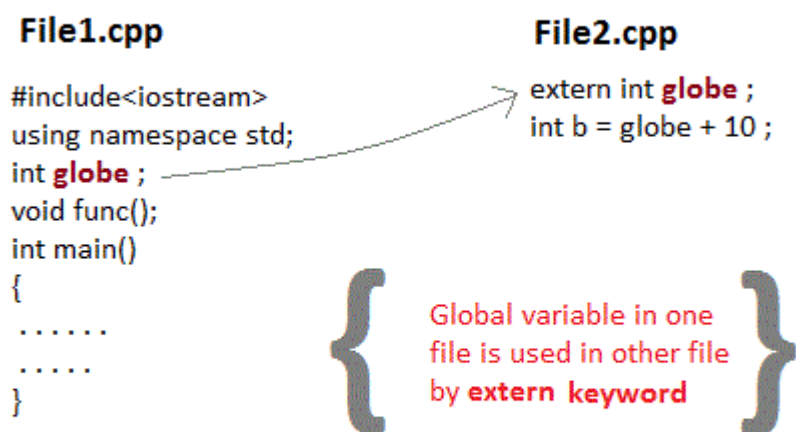
```

As, *i* is static, hence it will retain its value through function calls, and is initialized only once at the beginning.

Static specifiers are also used in classes, but that we will learn later.

Extern Variables

This keyword is used to access variable in a file which is declared & defined in some other file, that is the existence of a global variable in one file is declared using extern keyword in another file.



Functions in C++

Functions are used to provide modularity to a program. Creating an application using function makes it easier to understand, edit, check errors etc.

Syntax of Function

```

return-typefunction-name (parameters)
{
    // function-body
}

```

- **return-type** : suggests what the function will return. It can be int, char, some pointer or even a class object. There can be functions which does not return anything, they are mentioned with **void**.
- **Function Name** : is the name of the function, using the function name it is called.
- **Parameters** : are variables to hold values of arguments passed while function is called. A function may or may not contain parameter list.

```

void sum(int x, int y)
{
    int z;
    z = x + y;
    cout << z;
}

```

```

int main()
{
    int a = 10;
    int b = 20;
    sum (a, b);
}

```

Here, **a** and **b** are sent as arguments, and **x** and **y** are parameters which will hold values of a and b to perform required operation inside function.

- **Function body** : is he part where the code statements are written.

Declaring, Defining and Calling Function

Function declaration, is done to tell the compiler about the existence of the function. Function's return type, its name & parameter list is mentioned. Function body is written in its definition. Lets understand this with help of an example.

```

#include < iostream>
using namespace std;
int sum (int x, int y);    //declaring function
int main()
{
    int a = 10;
    int b = 20;
    int c = sum (a, b);    //calling function
    cout << c;
}
int sum (int x, int y)    //defining function
{

```

```
    return (X + y);  
}
```

Here, initially the function is **declared**, without body. Then inside main() function it is **called**, as the function returns summation of two values, hence z is their to store the value of sum. Then, at last, function is **defined**, where the body of function is mentioned. We can also, declare & define the function together, but then it should be done before it is called.

Calling a Function

Functions are called by their names. If the function is without argument, it can be called directly using its name. But for functions with arguments, we have two ways to call them,

1. Call by Value
2. Call by Reference

Call by Value

In this calling technique we pass the values of arguments which are stored or copied into the formal parameters of functions. Hence, the original values are unchanged only the parameters inside function changes.

```
void calc(int x);  
int main()  
{  
    int x = 10;  
    calc(x);  
    printf("%d", x);  
}
```

```
void calc(int x)  
{  
    x = x + 10 ;  
}
```

Output : 10

In this case the actual variable `x` is not changed, because we pass argument by value, hence a copy of `x` is passed, which is changed, and that copied value is destroyed as the function ends(goes out of scope). So the variable `x` inside main() still has a value 10.

But we can change this program to modify the original `x`, by making the function `calc()` return a value, and storing that value in `x`.

```
int calc(int x);  
int main()  
{  
    int x = 10;  
    x = calc(x);  
    printf("%d", x);  
}
```



```
}  
  
int calc(int x)  
{  
    x = x + 10 ;  
    return x;  
}  
Output : 20
```

Call by Reference

In this we pass the address of the variable as arguments. In this case the formal parameter can be taken as a reference or a pointer, in both the case they will change the values of the original variable.

```
void calc(int *p);  
int main()  
{  
    int x = 10;  
    calc(&x);    // passing address of x as argument  
    printf("%d", x);  
}  
  
void calc(int *p)  
{  
    *p = *p + 10;  
}  
  
Output : 20
```

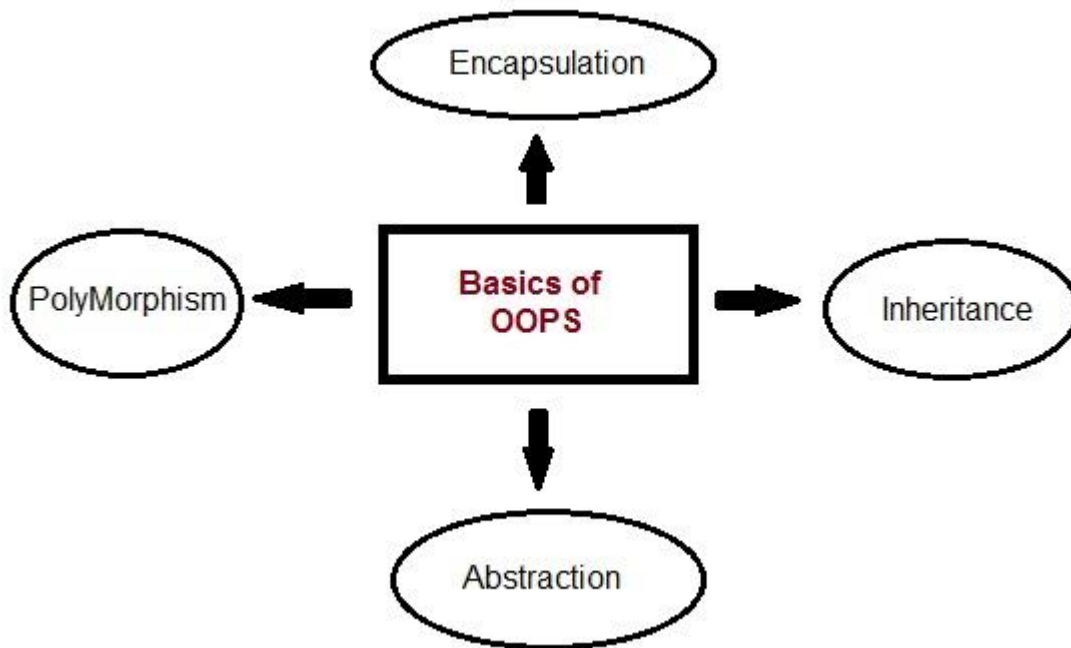
NOTE : If you do not have a prior knowledge of pointers, do study Pointers first.

CHAPTER 2: OBJECT ORIENTED PROGRAMMING

Object Oriented programming is a programming style that is associated with the concept of OBJECTS, having datafields and related member functions.

Objects are instances of classes and are used to interact amongst each other to create applications. Instance means, the object of class on which we are currently working. C++ can be said to be as C language with classes. In C++ everything revolves around object of class, which have their methods & data members.

C++ can be said to be as C language with classes. In C++ everything revolves around object of class, which have their methods & data members.



For Example : We consider human body as a class, we do have multiple objects of this class, with variable as color, hair etc. and methods as walking, speaking etc.

Now, let us discuss some of the main features of object oriented programming which you will be using in C++.

1. Objects
2. Classes
3. Abstraction
4. Encapsulation
5. Inheritance
6. Overloading
7. Exception Handling

Objects

Objects are the basic unit of OOP. They are instances of class, which have data members and uses various member functions to perform tasks.

Class

It is similar to structures in C language. Class can also be defined as user defined data type but it also contains functions in it. So, class is basically a blueprint for object. It declare & defines what data variables the object will have and what operations can be performed on the class's object.

Abstraction

Abstraction refers to showing only the essential features of the application and hiding the details. In C++, classes provide methods to the outside world to access & use the data variables, but the variables are hidden from direct access.

Encapsulation

It can also be said data binding. Encapsulation is all about binding the data variables and functions together in class.

Inheritance

Inheritance is a way to reuse once written code again and again. The class which is inherited is called base class & the class which inherits is called derived class. So when, a derived class inherits a base class, the derived class can use all the functions which are defined in base class, hence making code reusable.

Polymorphism

Polymorphism makes the code more readable. It is a feature, which lets us create functions with same name but different arguments, which will perform differently. That is function with same name, functioning in different

Overloading

Overloading is a part of polymorphism. Where a function or operator is made & defined many times, to perform different functions they are said to be overloaded.

Exception Handling

Exception handling is a feature of OOP, to handle unresolved exceptions or errors produced at runtime.

The classes are the most important feature of C++ that leads to Object Oriented programming. Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating instance of that class.

The variables inside class definition are called as data members and the functions are called member functions.

For example : Class of birds, all birds can fly and they all have wings and beaks. So here flying is a behavior and wings and beaks are part of their characteristics. And there are many different birds in this class with different names but they all possess this behavior and characteristics.

Similarly, class is just a blue print, which declares and defines characteristics and behavior, namely data members and member functions respectively. And all objects of this class will share these characteristics and behavior.

More about Classes

1. Class name must start with an uppercase letter. If class name is made of more than one word, then first letter of each word must be in uppercase. *Example,*

```
class Study, class StudyTonight etc
```

2. Classes contain, data members and member functions, and the access of these data members and variable depends on the access specifiers (discussed in next section).
3. Class's member functions can be defined inside the class definition or outside the class definition.
4. Class in C++ are similar to structures in C, the only difference being, class defaults to private access control, whereas structure defaults to public.
5. All the features of OOPS, revolve around classes in C++. Inheritance, Encapsulation, Abstraction etc.
6. Objects of class hold separate copies of data members. We can create as many objects of a class as we need.
7. Classes do possess more characteristics, like we can create abstract classes, immutable classes, all this we will study later.

Objects

Class is merely a blueprint or a template. No storage is assigned when we define a class. Objects are instances of class, which hold the data variables declared in class and the member functions work on these class objects.

Each object has different data variables. Objects are initialised using special class functions called **Constructors**. We will study about constructors later.

And whenever the object is out of its scope, another special class member function called **Destructor** is called, to release the memory reserved by the object. C++ doesn't have Automatic Garbage Collector like in JAVA, in C++ Destructor performs this task.

```
class Abc
{
    int x;
    void display(){} //empty function
};
```

```
in main()
{
  Abc obj;    // Object of class Abc created
}
```

Access Control in Classes

Now before studying how to define class and its objects, let's first quickly learn what are access specifiers.

Access specifiers in C++ class defines the access control rules. C++ has 3 new keywords introduced, namely,

1. public
2. private
3. protected

These access specifiers are used to set boundaries for availability of members of class be it data members or member functions

Access specifiers in the program, are followed by a colon. You can use either one, two or all 3 specifiers in the same class to set different boundaries for different class members. They change the boundary for all the declarations that follow them.

Public

Public, means all the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. Hence there are chances that they might change them. So the key members must not be declared public.

```
class PublicAccess
{
  public:    // public access specifier
  int x;    // Data Member Declaration
  void display(); // Member Function declaration
}
```

Private

Private keyword, means that no one can access the class members declared private outside that class. If someone tries to access the private member, they will get a compile time error. By default class variables and member functions are private.

```
class PrivateAccess
```

```

{
private:    // private access specifier
int x;      // Data Member Declaration
void display(); // Member Function deARATION
}

```

Protected

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class. (If class A is inherited by class B, then class B is subclass of class A. We will learn this later.)

```

class ProtectedAccess
{
protected:    // protected access specifier
int x;      // Data Member Declaration
void display(); // Member Function deARATION
}

```

Defining Class and Declaring Objects

When we define any class, we are not defining any data, we just define a structure or a blueprint, as to what the object of that class type will contain and what operations can be performed on that object.

Below is the syntax of class definition,

```

class ClassName
{
Access specifier:
Data members;
Member Functions(){}
};

```

Here is an example, we have made a simple class named Student with appropriate members,

```

class Student
{
public:
int rollno;
string name;
};

```

So its clear from the syntax and example, class definition starts with the keyword "class" followed by the class name. Then inside the curly braces comes the class body, that is data

members and member functions, whose access is bounded by access specifier. A class definition ends with a semicolon, or with a list of object declarations.

Example :

```
class Student
{
public:
int rollno;
string name;
}A,B;
```

Here A and B are the objects of class Student, declared with the class definition. We can also declare objects separately, like we declare variable of primitive data types. In this case the data type is the class name, and variable is the object.

```
int main()
{
    Student A;
    Student B;
}
```

Both A and B will have their own copies of data members.

Accessing Data Members of Class

Accessing a data member depends solely on the access control of that data member. If its public, then the data member can be easily accessed using the direct member access (.) operator with the object of that class.

If, the data member is defined as private or protected, then we cannot access the data variables directly. Then we will have to create special public member functions to access, use or initialize the private and protected data members. These member functions are also called **Accessors** and **Mutator** methods or **getter** and **setter** functions.

Accessing Public Data Members

Following is an example to show you how to initialize and use the public data members using the dot (.) operator and the respective object of class.

```
class Student
{
public:
int rollno;
string name;
};

int main()
```



```

{
  Student A;
  Student B;
  A.rollno=1;
  A.name="Adam";

  B.rollno=2;
  B.name="Bella";

  cout <<"Name and Roll no of A is :"<< A.name << A.rollno;
  cout <<"Name and Roll no of B is :"<< B.name << B.rollno;
}

```

Accessing Private Data Members

To access, use and initialize the private data member you need to create getter and setter functions, to get and set the value of the data member.

The setter function will set the value passed as argument to the private data member, and the getter function will return the value of the private data member to be used. Both getter and setter function must be defined public.

Example :

```

class Student
{
  private:    // private data member
  int rollno;

  public:    // public accessor and mutator functions
  int getRollno()
  {
    return rollno;
  }

  void setRollno(int i)
  {
    rollno=i;
  }

};

int main()
{
  Student A;
  A.rollno=1; //Compile time error
  cout<< A.rollno; //Compile time error

  A.setRollno(1); //Rollno initialized to 1
  cout<< A.getRollno(); //Output will be 1
}

```

So this is how we access and use the private data members of any class using the getter and setter methods. We will discuss this in more details later.

Accessing Protected Data Members

Protected data members, can be accessed directly using dot (.) operator inside the subclass of the current class, for non-subclass we will have to follow the steps same as to access private data member.

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

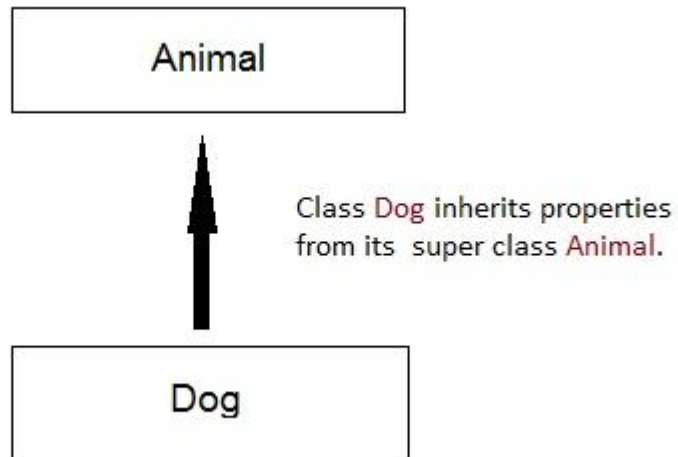
Basic Syntax of Inheritance

```
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, private or protected.

Example of Inheritance



```

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

| Base class | Derived Class | | |
|------------|---------------|---------------|----------------|
| | Public Mode | Private Mode | Protected Mode |
| Private | Not Inherited | Not Inherited | Not Inherited |
| Protected | Protected | Private | Protected |
| Public | Public | Private | Protected |

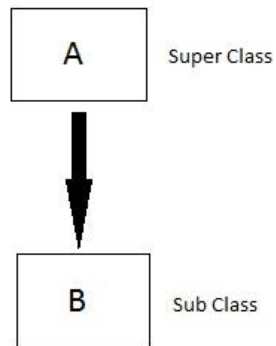
Types of Inheritance

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)

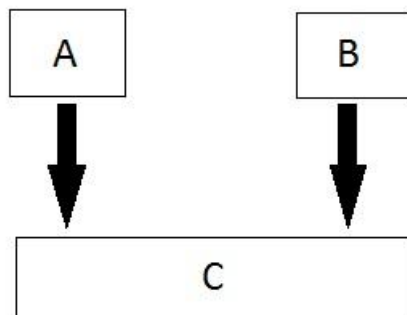
Single Inheritance

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



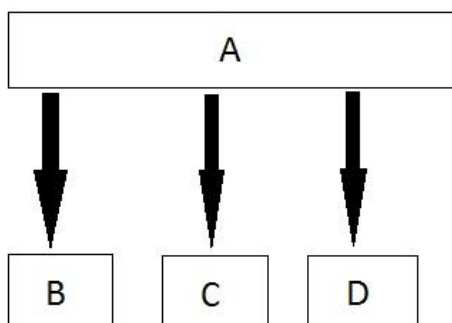
Multiple Inheritance

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



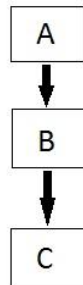
Hierarchical Inheritance

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



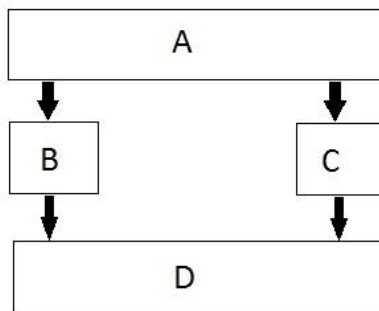
Multilevel Inheritance

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 Multilevel Inheritance.



Polymorphism

Polymorphism means having multiple forms of one thing. In inheritance, polymorphism is done, by method overriding, when both super and sub class have member function with same declaration but different definition.

CHAPTER III: FUNDAMENTAL DATA STRUCTURES

1. STACK

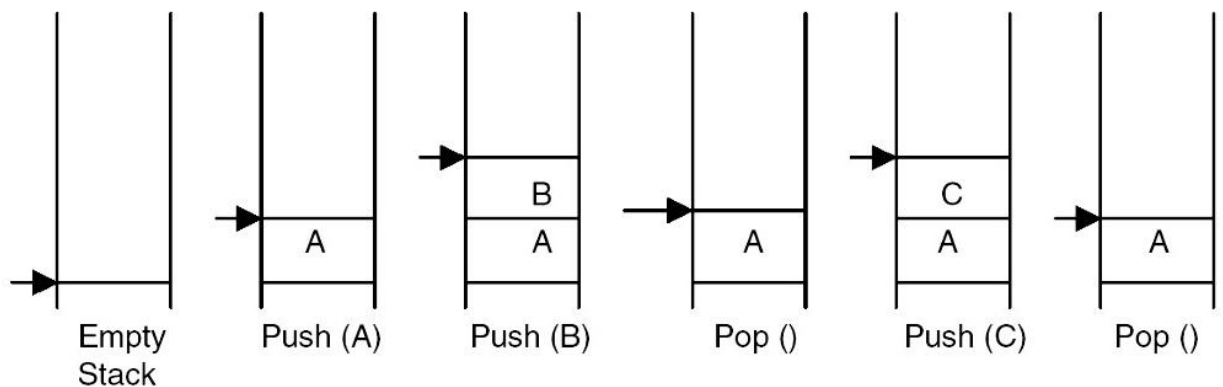
A *stack* is a version of a list that is particularly useful in applications involving reversing, such as the problem will be given later on. In a stack data structure, all insertions and deletions of entries are made at one end, called the *top* of the stack.

A helpful analogy is to think of a stack of trays or of plates sitting on the counter in a busy cafeteria. Throughout the lunch hour, customers take trays off the top of the stack, and employees place returned trays back on top of the stack. The tray most recently put on the stack is the first one taken off. The bottom tray is the first one put on, and the last one to be used.

When we add an item to a stack, we say that we *push* it onto the stack, and when we remove an item, we say that we *pop* it from the stack.

Note that the last item pushed onto a stack is always the first that will be popped from the stack. This property is called *last in, first out*, or *LIFO* for short.

Shown in the following figure are the effects of push and pop operations on the stack.



→ Indicates the stack top position

As we said before, stacks are used internally by the compiler for implementing recursion; generally, they are useful for computations that proceed “inside out”, as we will see below.

Let’s assume that our stack stores elements of some type *T*. The functionality that a stack offers us more formally is the following.

- Adding an element: **void push (const T & data)**
- Look at the top (most recently-added) element: **T top ()** (also sometimes, for instance in the textbook, called **T peek ()**).
- Remove the top (most recently-added) element: **void pop()**.

Notice that the only element that we can look at or remove is the one on top of the stack.

Also, in practice, to make the stack easier to use, you would probably add a function **bool**

`isEmpty()` which returns whether there are any elements on the stack. But the above three are really what makes a stack.

As with other data structures we have seen so far, we'd like to define formally what is and isn't a stack.

As before, a recursive definition is perhaps clearest and simplest: A stack is either

1. The empty stack, or
2. Of the form `S.push(data)`, where `S` is a stack, and `data` is a data item.

This tells us what is and isn't a stack, but it doesn't tell us the semantics of the stack functions, i.e., what exactly they do. The nice thing about stacks is that they are simple enough that we can completely specify the meaning of their functions using formulas. Some people would call these formulas the stack axioms:

1. For all stacks `S`: `s.push(data).top() = data`.
2. For all stacks `S`: `s.push(data).pop() = S`.

This says that if you read the top element after pushing something on a stack, you get that element back.

And if you remove the top element after pushing something on the stack, you get the original stack back.

The definition does not say what happens when you call `pop` or `top` on an empty stack. (In other words, we don't define `pop` or `top` for the base case of our recursive stack definition.) This is intentional: there is no real "correct" meaning that we could assign those operations. When you implement a stack, of course you will have to choose what happens, and you should document your choice. But behavior for cases that really shouldn't happen isn't really something that needs to go in a high-level definition of an abstract data type, and someone who plans to use a stack to solve a problem to solve some bigger algorithmic problems shouldn't rely on specific behavior for faulty inputs.

A second thing to reiterate is that it is really quite amazing that the stack functionality can be completely specified with two lines of math, and two simple lines at that. When we get to queues later, this will not be possible. At an intuitive level, for a stack, adding and removing come in pairs that are right next to each other, while for queues, there can be arbitrarily many elements between adding an element and seeing it again. This means that to specify the semantics of queue operations, one has to use much more advanced mathematical specifications.

At an even more philosophical level, this difference is why many mathematically inclined people prefer functional programming. Recursion by nature resembles stacks: the behavior

of a function can be summarized, and then substituted where it occurs. Analyzing for loops, on the other hand, requires something called fixed point operators, which is much less intuitive. As a result, carefully thought through recursive solutions are often much more robust.

1.1 Example

As an example of something that is really easy to do with stacks, and wouldn't at all be obvious without stacks or recursion, let us look at the following task: You are given a string of characters: lowercase letters and opening and closing parentheses, brackets, and braces “([{ }])”. The goal is to test if all of the parentheses/square brackets/braces match. To illustrate the task, here are some examples:

1. “[ab{c}]de()” is correct: all opening/closing braces match.
2. “[ab]” is incorrect: the closing square bracket does not match the opening parenthesis.
3. “ab}” is incorrect: there is no opening brace matching the closing one.

We now outline an algorithm (using a stack) to recognize whether the string has matching parentheses and brackets and braces.

1. The stack starts out empty.
2. The string is scanned character by character, left to right.
 - Each time an opening brace/bracket/parenthesis is encountered, push it onto the stack.
 - When a letter is encountered, ignore it.
 - When there is a closing brace/bracket/parenthesis, check if it matches what is on top of the stack.
 - If it does, then pop the matching brace/bracket/parenthesis off the stack.
 - If it doesn't, then signal that this is bad input (mismatch).
 - If the stack was empty, then signal a bad input (missing opening parenthesis).
3. If the stack is not empty at the end, there was an unmatched opening parenthesis/bracket/brace, so signal an error.

In a sense, we can regard the above algorithm as a very rudimentary parser. It's easy to imagine putting numbers to add or multiply in there, or putting C++ code inside the braces. Indeed, most implementations of parsers for languages (programming languages, logic formulas, arithmetic expressions, . . .) use stacks either explicitly or at least implicitly (through recursion).

When you use the above techniques to write a parser for formulas (e.g., a simple calculator, or a parser for a programming language, or for Boolean formulas), you would go one step

further: whenever you encounter a closing parenthesis, you know that you reached the end of an expression that you are now ready to evaluate.

Typically, you would now pop stuff off the stack until you have the entire internal expression, and instead push its value back on the stack. That way, you evaluate the formula inside out.

Looking ahead a little bit to proving correctness of algorithms formally, we would like to argue that the algorithm does the right thing. Of course, it is our intuition that it does — otherwise, we wouldn't have come up with this algorithm. But sometimes, we make mistakes, not just in our implementation, but also in our logic.

There are techniques for mathematically proving that a program/algorithm is correct. These are based on phrasing axioms about what exactly certain programming constructs do, such as our axioms about the stack functions above. Whenever a program contains recursion or loops, such proofs are invariably based on using induction. For loops, a key element of a correctness proof by induction is what's called a loop invariant: a property that will be true after each iteration of the loop, such that it's trivial that is true at the beginning (base case), and if we can prove that it holds at the end, our program has done the correct thing. Then, proving that the invariant holds from one loop to the next is exactly the induction step of a proof by induction.

While we won't do a full correctness proof by induction for this algorithm, for the curious student, the following should be pointed out: the algorithm runs a loop over all characters of the input string. The important part of the loop invariant is that at any time, the top of the stack is the most recent unmatched opening parenthesis/bracket/brace. If one were to really attempt the proof, this isn't enough to make the induction step work. The actual hypothesis needed is the following: at any stage of the processing, the stack contains all currently unmatched parentheses, in right-to-left order (top-to-bottom in the stack).

1.2. Implementation of a general-purpose stack

A stack can be naturally (and quite easily) implemented using either a linked list or array (so long as we dynamically resize it, the way we suggested for implementing a List).

For an implementation based on linked lists, we could insert and delete either at the head or tail. Both are roughly equally easy to implement, though inserting and deleting at the head may be even easier, because even a single-linked list will do:

- To push an item create a new element, link it to the old head, and make it the new head.

- To read the top, return the head's data item.
- To pop off the stack, remove the head from the list, and set its next element as the new head.

Notice that for none of these operations do we ever need to access any element's previous element. We could also perform similar operations on the tail of a linked list, but in order to pop the last element, we need to know its predecessor, which will be the new tail; so we would need a doubly linked list, or to scan through the entire list to find the new tail. The latter would of course be inefficient.

An implementation using arrays is not much more difficult: we have an array `a` and store the size of the stack in some variable, say, called `size`.

- To push an item, we write it into `a[size]` and increment `size`. If we now exceed the allocated array size, we expand the array by allocating a new larger array and copying the data over, much like we did for the `List` datatype.
- To read the top, we simply return `a[size-1]`.
- To pop an element off the stack, we decrement the `size`.

For both implementations, the running time of the `top` and `pop` operations is clearly $O(1)$, as we only return a directly accessible element, and update only a constant number of pointers or integer variables. For `push`, the linked list implementation is also clearly $O(1)$. For the array implementation, it is $O(1)$ except when the array size has to increase (in which case it is $\Theta(n)$). However, if we double the array size every time the array is too small (rather than, say, incrementing it only by 1), then for every operation that takes $\Theta(n)$, the next n operations will require no doubling and thus take $O(1)$. In total, those $n+1$ operations thus take $O(n)$, which means that on average, they take $O(1)$. This is another example of "amortized analysis", which was discussed in the context of the Array List implementation.

Stack Example

| Operation | output | stack |
|-----------|---------|-----------|
| • push(8) | - | (8) |
| • push(3) | - | (3, 8) |
| • pop() | 3 | (8) |
| • push(2) | - | (2, 8) |
| • push(5) | - | (5, 2, 8) |
| • top() | 5 | (5, 2, 8) |
| • pop() | 5 | (2, 8) |
| • pop() | 2 | (8) |
| • pop() | 8 | () |
| • pop() | "error" | () |
| • push(9) | - | (9) |
| • push(1) | - | (1, 9) |

Problem

Write a complete C++ program that implements a stack to solve the following problem:

Read an integer n , then read a list of n numbers, and print the list in reverse order.

Solution

```
/* Program stackNumbersSTL.cpp
```

```
Pre: The user supplies an integer n and n decimal numbers.
```

```
Post: The numbers are printed in reverse order.
```

```
Uses: The STL class stack and its methods
```

```
*/
```

```
#include <iostream>
```

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

```
using namespace std;
```

```
#include <stack>
```

```
int main(){
```

```
int n;
```

```
double item;
```

```
stack <double> numbers; // declares and initializes a stack of numbers
```

```
cout << " Type in an integer n followed by n decimal numbers." << endl
```

```
<< " The numbers will be printed in reverse order." << endl;
```

```
cin >> n;
```

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

```
cin >> item;
```

```
numbers.push(item);
```

```
}
```

```
cout << endl << endl;
```

```
while (!numbers.empty()) {
```

```
cout << numbers.top() << " ";
```

```
numbers.pop();
```

```
}
```

```
getchar();
```

```
getchar();
```

```
}
```

```
47
```

Sample Output:

Type in an integer n followed by n integers.

The entered integers will be printed in reverse order.

5 21 43 65 78 49

49

78

65

43

21

Questions

1. What is a stack?
2. What is the purpose of the push() member method?
3. What is the purpose of the pop() member method?
4. What is the purpose of the isFull() member method?
5. What is the purpose of the isEmpty() member method?
6. What kind of value is assigned to the top attribute?
7. Why is the top attribute initialized to -1?
8. What is the purpose of the keyword private?
9. What is the purpose of the keyword public?
10. What is the difference between a constructor and a destructor?

Answers

1. A stack is the way you group things together by placing one thing on top of another and then removing them one at a time from the top of the stack.
2. The `push()` member method places a value onto the top of a stack.
3. The `pop()` member method removes the value from the top of a stack, which is then returned by the `pop()` member method to the statement that calls the `pop()` member method.
4. The `isFull()` member method determines if there is room for one more value on the stack.
5. The `isEmpty()` member method determines if a value is at the top of the stack and is called before an attempt is made to remove the value.
6. The value at the `top` attribute is an index.
7. The `top` attribute is initialized to `-1` because when the attribute is incremented by the `push()` member method, the new value of the `top` attribute is zero, which is the index of the first element of the array used to create the stack.
8. The keyword `private` means that the attribute or member method is accessible only by a member method. The instance of the class cannot directly access a private member of the class.
9. The keyword `public` means that the attribute or member method is accessible to member methods and from the instance of the class.
10. A constructor is a member method of a class that is called when an instance of the class is declared. A destructor is a member method of a class that is called when the instance of the class falls out of scope.

2. QUEUES

Recall that queues provide “First In First Out” (FIFO) access to data. When elements are added into a queue, they can be read/removed only in exactly the order in which they were added. The functionality of queues (storing elements of some type T) is the following:

- Adding an element: **void enqueue(const T & data).**
- Look at the oldest element in the queue: **T peekfront().**
- Remove the oldest element from the queue: **void dequeue().**

As with stacks, you’d likely want to add a function `bool isEmpty()` in practice. Notice again the contrast to stacks: with a stack, you can only access (read/remove) the most recently added element, while a queue only allows you to access the oldest element.

Much like with stacks, we can define formally what is and isn’t a queue. A queue is either:

1. the empty queue, or 2. of the form `Q.enqueue(data)`, where Q is a queue, and data is a data item. As we discussed above, the precise semantics of the operations on a queue are much harder to specify, and far beyond what we can do in this class. (And unfortunately, at USC, we do not have a class on semantics of programming languages.)

2.1 Implementation of a queue

Just like with a stack, a queue can be implemented using a linked list or an array.

For an implementation using linked lists, we have two choices: (1) insert at the head and read/remove at the tail, or (2) insert at the tail and read/remove at the head. There’s a slight advantage to the second version, because a singly linked list suffices: when we delete, we only need to set `head=head->next` (and deallocate the old head). In order to delete at the tail, we would need to know the tail’s predecessor, which would require a doubly linked list or a linear-time search.

For an implementation using arrays, compared to stacks, we now need two integer indices: one (say, `newest`) for the position of the newest element in the array, and one (say, `oldest`) for the position of the oldest element. When enqueueing a new element, we can write it into `a[newest]` and increment `newest`. To implement `peekfront`, we can return `a[oldest]`, and to dequeue the oldest element, we can just increment `oldest` by one. As before, when enqueueing a new element and exceeding the current array size, we need to expand the array.

One downside of this implementation is that it may be quite wasteful of space. In particular if the queue is used for a long time, after a while, both `newest` and `oldest` may be large, which means that most of the array is unused. Instead, we could have the array “wrap around”, by doing calculations of the index `newest` modulo the array’s size. This makes

better use of space, but we now have to be careful not to overwrite the older elements with newer elements. None of this is a huge obstacle, but one has to be a bit more careful in the implementation, which perhaps makes an implementation based on linked lists slightly more attractive.

The running time analysis is pretty much identical to the one for a stack. All operations are $O(1)$

because they just involve updates of a small number of variables. As before the only exception is the possible expansion of the array, which may take $O(n)$, and which can again be amortized if the array size doubles (or gets multiplied with some number $b > 1$, rather than having something added to it) whenever the array is too small.

2.2 Why use restrictive data structures?

As we saw, a stack only allows access to the most recently added element, while a queue only allows access to the oldest element in the data structure. Couldn't we have a data structure that allows access to either, as we need it?

Indeed, we can. There is a data structure called Deque, which does exactly that: it allows adding and accessing (reading/removing) elements at both ends. If you understood how to implement a stack and a queue, implementing a Deque will be easy for you. So why wouldn't we use a Deque everywhere instead of stacks or queues?

Or, perhaps more strongly: why don't we just use the C++ vector class for all data structures? It

combines the functionality of a stack, queue, and List in the sense in which we defined it in previous lectures. So it is much more powerful.

One answer to this question is pedagogical: in this class, we want to learn how these data structures work internally, and using a powerful tool without understanding it does not contribute to learning. A second answer arises when we have to implement the data structures ourselves: if a data structure provides more functions, then it is more work to implement, and offers more places to make implementation mistakes. So if we implement them ourselves, there's something to be said for keeping our data structures simple. But of course, that argument does not apply when we use data structures provided in a programming language.

A third, and often appropriate, answer is that in order to implement more functions, the implementation of other functions may be less efficient. For instance, in order to allow access to each element in constant time, vector is implemented internally as an array. That

may slow down the implementation of some other functions, which could be implemented faster if we didn't also need access by index.

The fourth, and real, answer lies in our goal of writing code that we and our collaborators can understand easily in the future. If you write code in which you explicitly use a variable of type `Stack`, you make it clear that all your code needs is the `Stack` functionality, which will help others (and yourself, if you return to your code weeks or months later) parse the logic of your code. If instead, you had used `Deque` or `Vector` types, others may wonder whether somewhere buried in line 1729 of your code, you actually access specific elements, or need to access both the head and tail of your `Deque`. That makes it harder to understand the logic. So in general, it is a good idea to think through your code's logic ahead of time, and only declare those data structures that you really need.

We could of course try to make this clear with our variable names, say, by writing something like `vector<int> stack`. If we do this, it might convey our intention. But of course, another interpretation would be that we had initially written `Stack<int> stack`, and then later realized that we actually needed extra functionality. But because we didn't want to rename the variable everywhere, we just changed its type to `vector<int>`, so even though the variable is called `stack`, it's not actually used as one.

So the upshot is that you should ideally declare your variables to be of the type that you actually need, not giving yourself extra functionality you don't intend to use. In particular, you may not find too many immediate uses for a `Deque`, since most of the time, you'll want either a `Stack` or `Queue`.

Queue operations

Main queue operations:

- _ **enqueue(o)**: inserts element `o` at the end of the queue
- _ **dequeue()**: removes and returns the element at the front of the queue

Auxiliary queue operations:

- _ **front()**: returns the element at the front without removing it
- _ **size()**: returns the number of elements stored
- _ **isEmpty()**: returns a Boolean value indicating whether no elements are stored
- _ Attempting the execution of `dequeue` or `front` on an empty queue throws an `EmptyQueueException`

Example of Queue

| Operation | output | queue |
|------------------|---------------|--------------|
| • enqueue(5) | - | (5) |
| • enqueue(3) | - | (5, 3) |
| • dequeue() | 5 | (3) |
| • enqueue(7) | - | (3, 7) |
| • dequeue() | 3 | (7) |
| • front() | 7 | (7) |
| • dequeue() | 7 | () |
| • dequeue() | "error" | () |
| • isEmpty() | true | () |
| • enqueue(9) | - | (9) |
| • size() | 1 | (9) |

3. LINKED LISTS

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.



Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages of Linked Lists

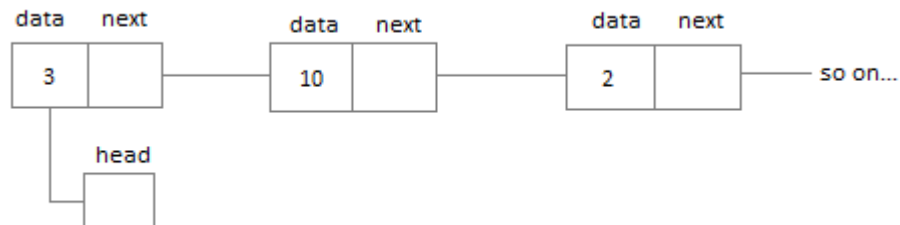
- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

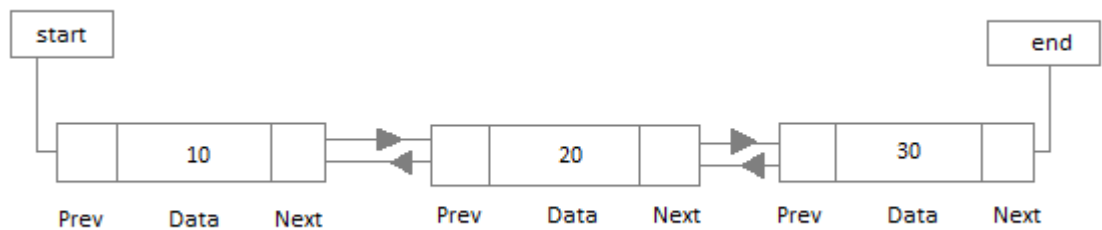
- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Types of Linked Lists

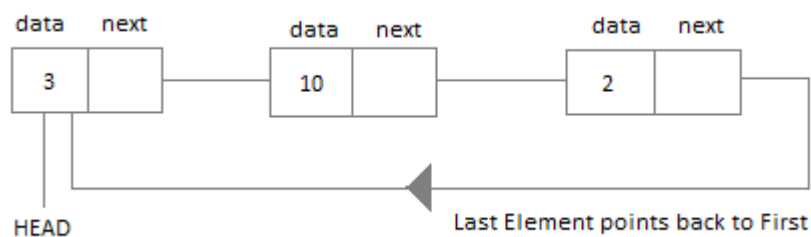
- **Singly Linked List :** Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.



- **Doubly Linked List :** In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.



- **Circular Linked List :** In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.



Linear Linked List

The element can be inserted in linked list in 2 ways :

- Insertion at beginning of the list.
- Insertion at the end of the list.

We will also be adding some more useful methods like :

- Checking whether Linked List is empty or not.
- Searching any element in the Linked List
- Deleting a particular Node from the List

Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the first Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

Inserting at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, hence making the new node the last Node.

Searching for an Element in the List

In searching we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.

Checking whether the List is empty or not

We just need to check whether the **Head** of the List is NULL or not.

4. HASHING

In previous sections we were able to make improvements in our search algorithms by taking advantage of information about where items are stored in the collection with respect to one another. For example, by knowing that a list was ordered, we could search in logarithmic time using a binary search. In this section we will attempt to go one step further by building a data structure that can be searched in $O(1)$ time. This concept is referred to as **hashing**.

A **hash table** is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a **slot**, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items so every slot is empty. We can implement a hash table by using a list with each element initialized to the special Python value `None`. Figure 1 shows a hash table of size $m=11$. In other words, there are m slots in the table, named 0 through 10.

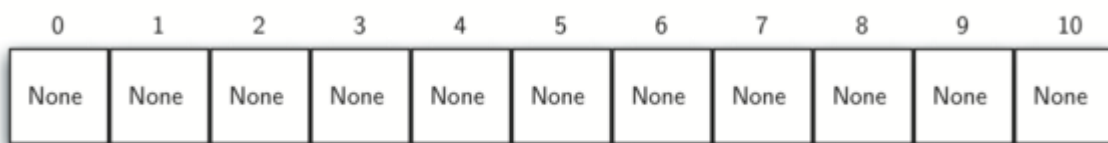


Figure 1: Hash Table with 11 Empty Slots

The mapping between an item and the slot where that item belongs in the hash table is called the **hash function**. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and $m-1$. Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31. Our first hash function, sometimes referred to as the “remainder method,” simply takes an item and divides it by the table size, returning the remainder as its hash value ($h(item)=item\%11$). Table 1 gives all of the hash values for our example items. Note that this remainder method (modulo arithmetic) will typically be present in some form in all hash functions, since the result must be in the range of slot names.

| Item | Hash Value |
|-------------|-------------------|
| 54 | 10 |
| 26 | 4 |
| 93 | 5 |
| 17 | 6 |
| 77 | 0 |
| 31 | 9 |

Once the hash values have been computed, we can insert each item into the hash table at the designated position as shown in Figure 2. Note that 6 of the 11 slots are now occupied. This is referred to as the **load factor**, and is commonly denoted by $\lambda = \text{number of items} / \text{table size}$. For this example, $\lambda = 6/11$.

| | | | | | | | | | | |
|----|------|------|------|----|----|----|------|------|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 77 | None | None | None | 26 | 93 | 17 | None | None | 31 | 54 |

Figure 2: Hash Table with Six Items

Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present. This searching operation is $O(1)$, since a constant amount of time is required to compute the hash value and then index the hash table at that location. If everything is where it should be, we have found a constant time search algorithm.

You can probably already see that this technique is going to work only if each item maps to a unique location in the hash table. For example, if the item 44 had been the next item in our collection, it would have a hash value of 0 ($44\%11=0$). Since 77 also had a hash value of 0, we would have a problem. According to the hash function, two or more items would need to be in the same slot. This is referred to as a **collision** (it may also be called a “clash”). Clearly, collisions create a problem for the hashing technique. We will discuss them in detail later.

Hash Functions

Given a collection of items, a hash function that maps each item into a unique slot is referred to as a **perfect hash function**. If we know the items and the collection will never change, then it is possible to construct a perfect hash function (refer to the exercises for more about perfect hash functions). Unfortunately, given an arbitrary collection of items, there is no systematic way to construct a perfect hash function. Luckily, we do not need the hash function to be perfect to still gain performance efficiency.

One way to always have a perfect hash function is to increase the size of the hash table so that each possible value in the item range can be accommodated. This guarantees that each item will have a unique slot. Although this is practical for small numbers of items, it is not feasible when the number of possible items is large. For example, if the items were nine-digit Social Security numbers, this method would require almost one billion slots. If we only want to store data for a class of 25 students, we will be wasting an enormous amount of memory.

Our goal is to create a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the items in the hash table. There are a number of common ways to extend the simple remainder method. We will consider a few of them here.

The **folding method** for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value. For example, if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43,65,55,46,01). After the addition, $43+65+55+46+01$, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case $210 \% 11$ is 1, so the phone number 436-555-4601 hashes to slot 1. Some folding methods go one step further and reverse every other piece before the addition. For the above example, we get $34+56+55+64+10=219$ which gives $219 \% 11=10$.

Another numerical technique for constructing a hash function is called the **mid-square method**. We first square the item, and then extract some portion of the resulting digits. For example, if the item were 44, we would first compute $44^2=1,936$. By extracting the middle two digits, 93, and performing the remainder step, we get 5 ($93 \% 11$). Table 2 shows items

under both the remainder method and the mid-square method. You should verify that you understand how these values were computed.

| Item | Remainder | Mid-Square |
|-------------|------------------|-------------------|
| 54 | 10 | 3 |
| 26 | 4 | 7 |
| 93 | 5 | 9 |
| 17 | 6 | 8 |
| 77 | 0 | 4 |
| 31 | 9 | 6 |

Collision Resolution

We now return to the problem of collisions. When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called **collision resolution**. As we stated earlier, if the hash function is perfect, collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing.

One method for resolving collisions looks into the hash table and tries to find another open slot to hold the item that caused the collision. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty. Note that we may need to go back to the first slot (circularly) to cover the entire hash table. This collision resolution process is referred to as **open addressing** in that it tries to find the next open slot or address in the hash table. By systematically visiting each slot one at a time, we are performing an open addressing technique called **linear probing**.

Figure 3 shows an extended set of integer items under the simple remainder method hash function (54,26,93,17,77,31,44,55,20). Table 1 above shows the hash values for the original items. Figure 2 shows the original contents. When we attempt to place 44 into slot 0, a collision occurs. Under linear probing, we look sequentially, slot by slot, until we find an open position. In this case, we find slot 1.

Again, 55 should go in slot 0 but must be placed in slot 2 since it is the next open position. The final value of 20 hashes to slot 9. Since slot 9 is full, we begin to do linear probing. We visit slots 10, 0, 1, and 2, and finally find an empty slot at position 3.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|----|----|----|----|----|----|------|------|----|----|
| 77 | 44 | 55 | 20 | 26 | 93 | 17 | None | None | 31 | 54 |

Figure 3: Collision Resolution with Linear Probing

Once we have built a hash table using open addressing and linear probing, it is essential that we utilize the same methods to search for items. Assume we want to look up the item 93. When we compute the hash value, we get 5. Looking in slot 5 reveals 93, and we can return True. What if we are looking for 20? Now the hash value is 9, and slot 9 is currently holding 31. We cannot simply return False since we know that there could have been collisions. We are now forced to do a sequential search, starting at position 10, looking until either we find the item 20 or we find an empty slot.

A disadvantage to linear probing is the tendency for **clustering**; items become clustered in the table. This means that if many collisions occur at the same hash value, a number of surrounding slots will be filled by the linear probing resolution. This will have an impact on other items that are being inserted, as we saw when we tried to add the item 20 above. A cluster of values hashing to 0 had to be skipped to finally find an open position. This cluster is shown in Figure 4.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|----|----|----|----|----|----|------|------|----|----|
| 77 | 44 | 55 | 20 | 26 | 93 | 17 | None | None | 31 | 54 |

Figure 4: A Cluster of Items for Slot 0

One way to deal with clustering is to extend the linear probing technique so that instead of looking sequentially for the next open slot, we skip slots, thereby more evenly distributing the items that have caused collisions. This will potentially reduce the clustering that occurs.

Figure 5 shows the items when collision resolution is done with a “plus 3” probe. This means that once a collision occurs, we will look at every third slot until we find one that is empty.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|----|------|----|----|----|----|----|------|----|----|
| 77 | 55 | None | 44 | 26 | 93 | 17 | 20 | None | 31 | 54 |

Figure 5: Collision Resolution Using “Plus 3”

The general name for this process of looking for another slot after a collision is **rehashing**. With simple linear probing, the rehash function is $newhashvalue = rehash(oldhashvalue)$ where $rehash(pos) = (pos+1) \% sizeof\ table$. The “plus 3” rehash can be defined as $rehash(pos) = (pos+3) \% sizeof\ table$. In general, $rehash(pos) = (pos+skip) \% sizeof\ table$. It is important to note that the size of the “skip” must be such that all the slots in the table will eventually be visited. Otherwise, part of the table will be unused. To ensure this, it is often suggested that the table size be a prime number. This is the reason we have been using 11 in our examples.

A variation of the linear probing idea is called **quadratic probing**. Instead of using a constant “skip” value, we use a rehash function that increments the hash value by 1, 3, 5, 7, 9, and so on. This means that if the first hash value is h , the successive values are $h+1$, $h+4$, $h+9$, $h+16$, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares. Figure 6 shows our example values after they are placed using this technique.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|----|----|----|----|----|----|------|------|----|----|
| 77 | 44 | 20 | 55 | 26 | 93 | 17 | None | None | 31 | 54 |

Figure 6: Collision Resolution with Quadratic Probing

An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items. **Chaining** allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper

slot of the hash table. As more and more items hash to the same location, the difficulty of searching for the item in the collection increases. Figure 7 shows the items as they are added to a hash table that uses chaining to resolve collisions.

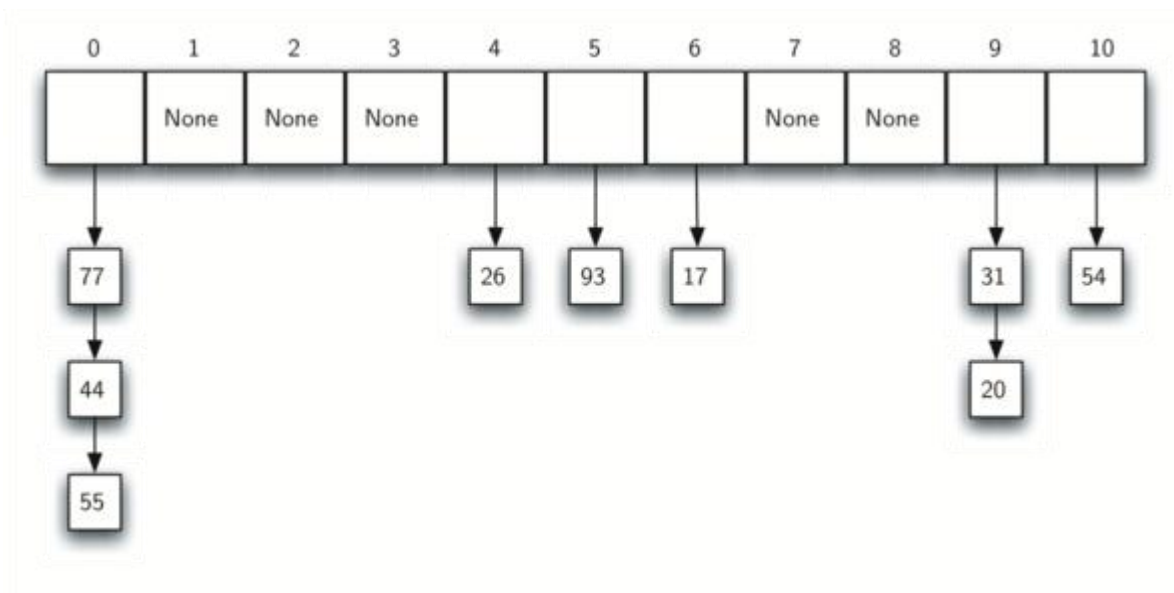


Figure 7: Collision Resolution with Chaining

When we want to search for an item, we use the hash function to generate the slot where it should reside. Since each slot holds a collection, we use a searching technique to decide whether the item is present. The advantage is that on the average there are likely to be many fewer items in each slot, so the search is perhaps more efficient. We will look at the analysis for hashing at the end of this section.

Self Check

Q-1: In a hash table of size 13 which index positions would the following two keys map to?
27, 130

Q-2: Suppose you are given the following set of keys to insert into a hash table that holds exactly 11 values: 113 , 117 , 97 , 100 , 114 , 108 , 116 , 105 , 99 Which of the following best demonstrates the contents of the has table after all the keys have been inserted using linear probing?

5. TREE DATA STRUCTURES

Very often we have to describe a group of real life objects, which have such relation to one another that we cannot use linear data structures for their description. In this chapter, we will give examples of such branched structures. We will explain their properties and the real life problems, which inspired their creation and further development.

A tree-like data structure or branched data structure consists of set of elements (nodes) which could be linked to other elements, sometimes hierarchically, sometimes not. Trees represent hierarchies, while graphs represent more general relations such as the map of city.

Example – Hierarchy of the Participants in a Project

We have a team, responsible for the development of certain software project.

The participants in it have manager-subordinates relations. Our team consists of 9 teammates:

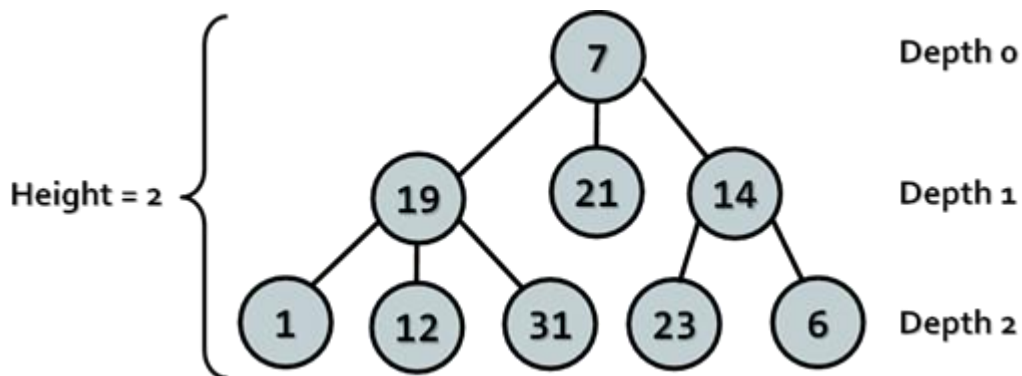


What is the information we can get from this hierarchy? The direct boss of the developers is the "Team Leader", but indirectly they are subordinate to the "Project Manager". The "Team Leader" is subordinate only to the "Project Manager". On the other hand "Developer 1" has no subordinates. The "Project Manager" is the highest in the hierarchy and has no manager.

The same way we can describe every participant in the project. We see that such a little figure gives us so much information.

Trees Terminology

For a better understanding of this part of the chapter we recommend to the reader at every step to draw an analogy between the abstract meaning and its practical usage in everyday life.



We will simplify the figure describing our hierarchy. We assume that it consists of circles and lines connecting them. For convenience we name the circles with unique numbers, so that we can easily specify about which one we are talking about.

We will call every circle a node and each line an edge. Nodes "19", "21", "14" are below node "7" and are directly connected to it. These nodes are called **direct descendants** (child nodes) of node "7", and node "7" their parent. The same way "1", "12" and "31" are children of "19" and "19" is their parent. Intuitively we can say that "21" is sibling of "19", because they are both children of "7" (the reverse is also true – "19" is sibling of "21"). For "1", "12", "31", "23" and "6" node "7" precedes them in the hierarchy, so he is their indirect parent – ancestor, and they are called his descendants.

Root is called the node without parent. In our example this is node "7"

Leaf is a node without child nodes. In our example – "1", "12", "31", "21", "23" and "6".

Internal nodes are the nodes, which are not leaf or root (all nodes, which have parent and at least one child). Such nodes are "19" and "14".

Path is called a sequence of nodes connected with edges, in which there is no repetition of nodes. Example of path is the sequence "1", "19", "7" and "21". The sequence "1", "19" and "23" is not a path, because "19" and "23" are not connected.

Path length is the number of edges, connecting the sequence of nodes in the path. Actually it is equal to the number of nodes in the path minus 1. The length of our example for path ("1", "19", "7" and "21") is three.

Depth of a node we will call the length of the path from the root to certain node. In our example "7" as root has depth zero, "19" has depth one and "23" – depth two.

Tree definition

Tree: a recursive data structure, which consists of nodes, connected with edges. The following statements are true for trees:

- Each node can have 0 or more direct descendants (children).
- Each node has at most one parent. There is only one special node without parent – the root (if the tree is not empty).
- All nodes are reachable from the root – there is a path from the root to each node in the tree.

We can give more simple definition of tree: a node is a tree and this node can have zero or more children, which are also trees.

Height of tree – is the maximum depth of all its nodes. In our example the tree height is 2.

Degree of node we call the number of direct children of the given node. The degree of "19" and "7" is three, but the degree of "14" is two. The leaves have degree zero.

Branching factor is the maximum of the degrees of all nodes in the tree. In our example the maximum degree of the nodes is 3, so the branching factor is 3.

Depth-First-Search (DFS) Traversal

In the class `Tree<T>` is implemented the method `TraverseDFS()`, that calls the private method `PrintDFS(TreeNode<T> root, string spaces)`, which traverses the tree in depth and prints on the standard output its elements in tree layout using right displacement (adding spaces).

The Depth-First-Search algorithm aims to visit each of the tree nodes exactly one. Such a visit of all nodes is called tree traversal. There are multiple algorithms to traverse a tree but in this course we will discuss only two of them: **DFS (depth-first search)** and **BFS (breadth-first search)**.

The DFS algorithm starts from a given node and goes as deep in the tree hierarchy as it can. When it reaches a node, which has no children to visit or all have been visited, it returns to the previous node. We can describe the depth-first search algorithm by the following simple steps:

1. Traverse the current node (e.g. print it on the console or process it in some way).
2. Sequentially traverse recursively each of the current nodes' child nodes (traverse the sub-trees of the current node). This can be done by a recursive call to the same method for each child node.

Breath-First-Search (BFS)

Let's have a look at another way of traversing trees. Breath-First-Search (BFS) is an algorithm for traversing branched data structures (like trees and graphs). The BFS algorithm first traverses the start node, then all its direct children, then their direct children and so on. This approach is also known as the wavefront traversal, because it looks like the waves caused by a stone thrown into a lake.

The Breath-First-Search (BFS) algorithm consists of the following steps:

1. Enqueue the start node in queue Q.
2. While Q is not empty repeat the following two steps:
 - Dequeue the next node v from Q and print it.
 - Add all children of v in the queue.

The BFS algorithm is very simple and always traverses first the nodes that are closest to the start node, and then the more distant and so on until it reaches the furthest. The BFS algorithm is very widely used in problem solving, e.g. for finding the shortest path in a labyrinth.

Binary Trees

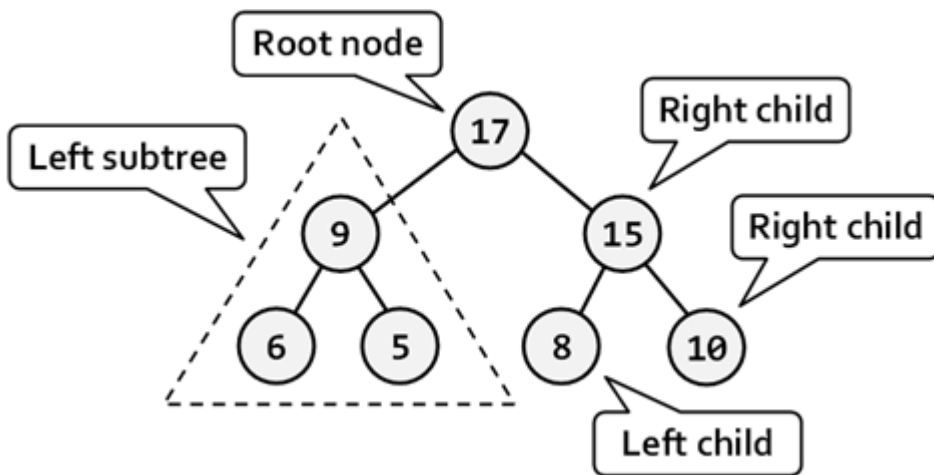
In the previous section we discussed the basic structure of a tree. In this section we will have a look at a specific type of tree – binary tree. This type of tree turns out to be very useful in programming. The terminology for trees is also valid about binary trees. Despite that below we will give some specific explanations about this structure.

Binary Tree: a tree, which nodes have a degree equal or less than 2 or we can say that it is a tree with branching degree of 2. Because every node's children are at most 2, we call them left child and right child. They are the roots of the left sub-tree and the right sub-tree of their parent node. Some nodes may have only left or only right child, not both. Some nodes may have no children and are called leaves.

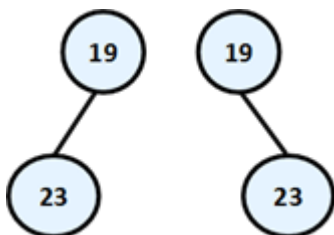
Binary tree can be recursively defined as follows: a single node is a binary tree and can have left and right children which are also binary trees.

Binary Tree – Example

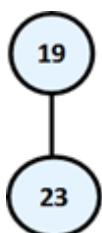
Here we have an example of binary tree. The nodes are again named with some numbers. In the figure we can see the root of the tree – "17", the left sub-tree (with root 9) and the right sub-tree (with root 15) and a right and left child – "3" and "21".



We have to note that there is one very big difference in the definition of binary tree from the definition of the classical tree – the order of the children of each node. The next example will illustrate that difference:



On this figure above two totally different binary trees are illustrated – the first one has root "19" and its left child "23" and the second root "19" and right child "23". If that was an ordinary tree they would have been the same. That's why such tree we would illustrate the following way:



Binary Tree Traversal

The traversal of binary tree is a classic problem which has classical solutions. Generally there are few ways to traverse a binary tree recursively:

- **In-order (Left-Root-Right)** – the traversal algorithm first traverses the left sub-tree, then the root and last the right sub-tree. In our example the sequence of such traversal is: "23", "19", "10", "6", "21", "14", "3", "15".
- **Pre-order (Root-Left-Right)** – in this case the algorithm first traverses the root, then the left sub-tree and last the right sub-tree. The result of such traversal in our example is: "14", "19", "23", "6", "10", "21", "15", "3".
- **Post-order (Left-Right-Root)** – here we first traverse the left sub-tree, then the right one and last the root. The result after the traversal is: "23", "10", "21", "6", "19", "3", "15", "14".

Ordered Binary Search Trees

Till this moment we have seen how we can build traditional and binary trees. These structures are very summarized in themselves and it will be difficult for us to use them for a bigger project. Practically, in computer science special and programming variants of binary and ordinary trees are used that have certain special characteristics, like order, minimal depth and others. Let's review the most important trees used in programming.

As examples for a useful properties we can give the ability to quickly search of an element by given value (Red-Black tree); order of the elements in the tree (ordered search trees); balanced depth (balanced trees); possibility to store an ordered tree in a persistent storage so that searching of an element to be fast with as little as possible read operations (B-tree), etc.

Comparability between Objects

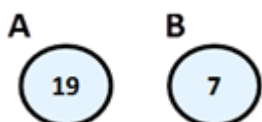
Before continuing, we will introduce the following definition, which we will need for the further exposure.

Comparability – we call two objects A and B comparable, if exactly one of following three dependencies exists:

- "A is less than B"
- "A is bigger than B"
- "A is equal to B"

Similarly we will call two keys A and B comparable, if exactly one of the following three possibilities is true: $A < B$, $A > B$ or $A = B$.

The nodes of a tree can contain different fields but we can think about only their unique keys, which we want to be comparable. Let's give an example. We have two specific nodes A and B:



In this case, the keys of A and B hold the integer numbers 19 and 7. From Mathematics we know that the integer numbers (unlike the complex numbers) are comparable, which according the above reasoning give us the right to use them as keys. That's why we can say that "A is bigger than B", because "19 is bigger than 17".

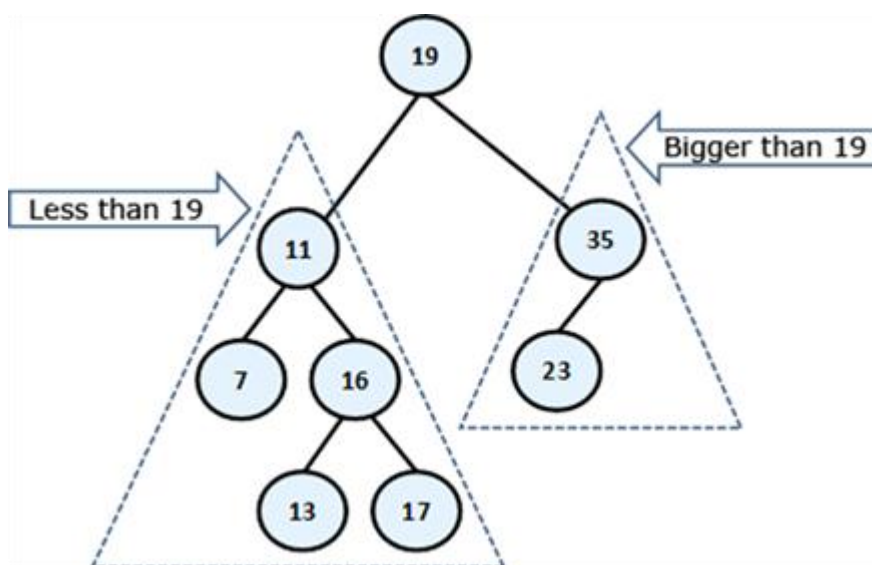
And we arrive to the definition of the ordered binary search tree:

Ordered Binary Tree (binary search tree) is a binary tree, in which every node has a unique key, every two of the keys are comparable and the tree is organized in a way that for every node the following is satisfied:

- All keys in the left sub-tree are smaller than its key.
- All keys in the right sub-tree are bigger than its key.

Properties of the Ordered Binary Search Trees

On the figure below we have given an example of an ordered binary search tree. We will use this example, to give some important properties of the binary tree's order:



By definition we know that the left sub-tree of every node consists only of elements, which are smaller than itself, while in the right sub-tree there are only bigger elements. This means that if we want to find a given element, starting from the root, either we have found it or should search it respectively in its left or its right sub-tree, which will save unnecessary comparisons. For example, if we search 23 in our tree, we are not going to search for it in the left sub-tree of 19, because 23 is not there for sure (23 is bigger than 19, so eventually it is in the right sub-tree). This saves us 5 unnecessary comparisons with each of the left sub-tree elements, but if we were using a linked list, we would have to make these 5 comparisons.

From the elements' order follows that the smallest element in the tree is the leftmost successor of the root, if there is such or the root itself, if it does not have a left successor. In our example this is the minimal element 7 and the maximal – 35. Next useful property from this is, that every single element from the left sub-tree of given node is smaller than every single element from the right sub-tree of the same node.

Ordered Binary Search Trees – Example

The next example shows a simple implementation of a binary search tree. Our point is to suggest methods for adding, searching and removing an element in the tree. For every single operation from the above, we will give an explanation in details. Note that our binary search tree is not balanced and may have poor performance in certain circumstances.

Inserting an Element

Inserting (or adding) an element in a binary search tree means to put a new element somewhere in the tree so that the tree must stay ordered. Here is the algorithm: if the tree is empty, we add the new element as a root. Otherwise:

- If the element is smaller than the root, we call recursively the same method to add the element in the left sub-tree.
- If the element is bigger than the root, we call recursively to the same method to add the element in the right sub-tree.
- If the element is equal to the root, we don't do anything and exit from the recursion.

We can clearly see how the algorithm for inserting a node, conforms to the rule “elements in the left sub-tree are less than the root and the elements in the right sub-tree are bigger than the root”.

Searching for an Element

Searching in a binary search tree is an operation which is more intuitive. In the sample code we have shown how the search of an element can be done without recursion and with iteration instead. The algorithm starts with element node, pointing to the root. After that we do the following:

- If the element is equal to node, we have found the searched element and return it.
- If the element is smaller than node, we assign to node its left successor, i.e. we continue the searching in the left sub-tree.
- If the element is bigger than node, we assign to node its right successor, i.e. we continue the searching in the right sub-tree.

At the end, the algorithm returns the found node or null if there is no such node in the tree. Additionally we define a Boolean method that checks if certain value belongs to the tree.

Removing an Element

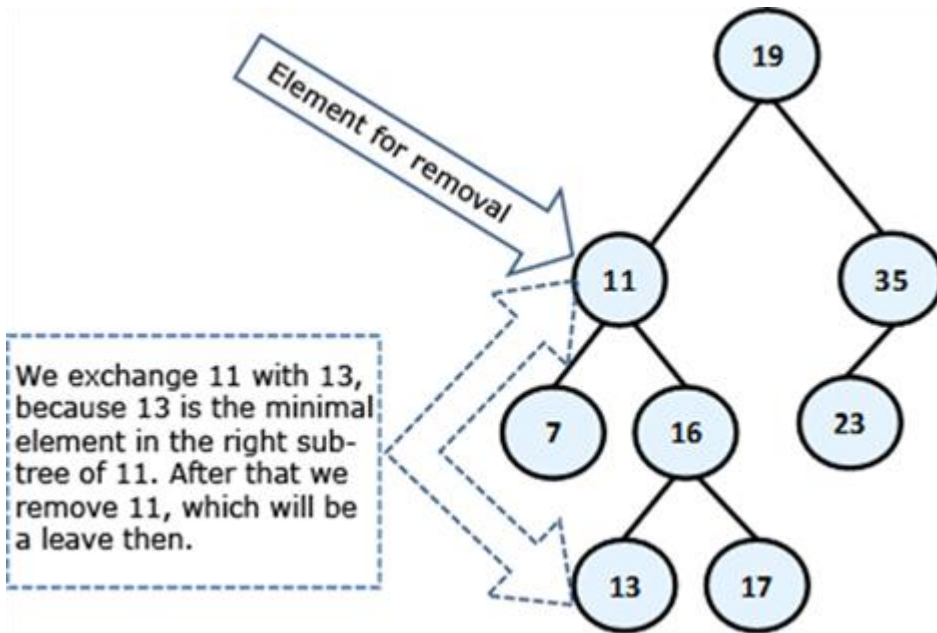
Removing is the most complicated operation from the basic binary search tree operations. After it the tree must keep its order.

The first step before we remove an element from the tree is to find it. We already know how it happens. After that, we have 3 cases:

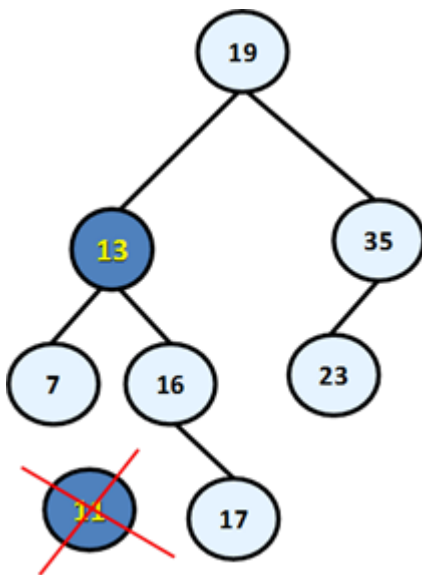
- If the node is a leaf – we point its parent's reference to null. If the element has no parent, it means that it is a root and we just remove it.
- If the node has only one sub-tree – left or right, it is replacing with the root of this sub-tree.
- The node has two sub-trees. Then we have to find the smallest node in the right sub-tree and swap with it. After this exchange the node will have one sub-tree at most and then we remove it grounded on some of the above two rules. Here we have to say that it can be done analogical swap, just that we get the left sub-tree and it is the biggest element.

We leave to the reader to check the correctness of these three steps, as a little exercise.

Now, let's see a sample removal in action. Again we will use our ordered tree, which we have displayed at the beginning of this point. For example, let's remove the element with key 11.



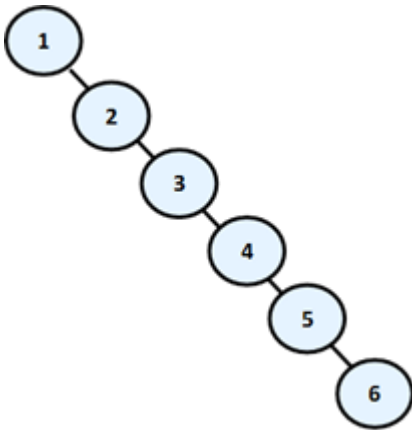
The node 11 has two sub-trees and according to our algorithm, it must be exchanged with the smallest element from the right sub-tree, i.e. with 13. After the exchange, we can remove 11 (it is a leaf). Here is the final result:



Balanced Trees

As we have seen above, the ordered binary trees are a very comfortable structure to search within. Defined in this way, the operations for creating and deleting the tree have a hidden flaw: they don't balance the tree and its depth could become very big.

Think a bit what will happen if we sequentially include the elements: 1, 2, 3, 4, 5, 6? The ordered binary tree will look like this:



In this case, the binary tree degenerates into a linked list. Because of this the searching in this tree is going to be much slower (with N steps, not with $\log(N)$), as to check whether an item is inside, in the worst case we will have to go through all elements.

We will briefly mention the existence of data structures, which save the logarithmic behavior of the operations adding, searching and removing an element in the common case. We will introduce to you the following definitions before we go on to explain how they are achieved:

Balanced binary tree – a binary tree in which no leaf is at “much greater” depth than any other leaf. The definition of “much greater” is rough depends on the specific balancing scheme.

Perfectly balanced binary tree – binary tree in which the difference in the left and right tree nodes’ count of any node is at most one.

Without going in details we will mention that when given binary search tree is balanced, even not perfectly balanced, then the operations of adding, searching and removing an element in it will run in approximately a logarithmic number of steps even in the worst case. To avoid imbalance in the tree to search, apply operations that rearrange some elements of the tree when adding or removing an item from it. These operations are called rotations in most of the cases. The type of rotation should be further specified and depends on the implementation of the specific data structure. As examples for structures like these we can give Red-Black tree, AVL-tree, AA-tree, Splay-tree and others.

Balanced search trees allow quickly (in general case for approximately $\log(n)$ number of steps) to perform the operations like searching, adding and deleting of elements. This is due to two main reasons:

- Balanced search trees keep their elements ordered internally.

- Balanced search trees keep themselves balanced, i.e. their depth is always in order of $\log(n)$.

Due to their importance in computer science we will talk about balanced search trees and their standard implementations in .NET Framework many times when we discuss data structures and their performance in this chapter and in the next few chapters.

Balanced search trees can be binary or non-binary.

Balanced binary search trees have multiple implementations like **Red-Black Trees**, **AA Trees** and **AVL Trees**. All of them are ordered, balanced and binary, so they perform insert / search / delete operations very fast.

Non-binary balanced search trees also have multiple implementations with different special properties. Examples are **B-Trees**, **B+ Trees** and **Interval Trees**. All of them are ordered, balanced, but not binary. Their nodes can typically hold more than one key and can have more than two child nodes. These trees also perform operations like insert / search / delete very fast.

For a more detailed examination of these and other structures we recommend the reader to look closely at literature about algorithms and data structures.

6. GRAPHS

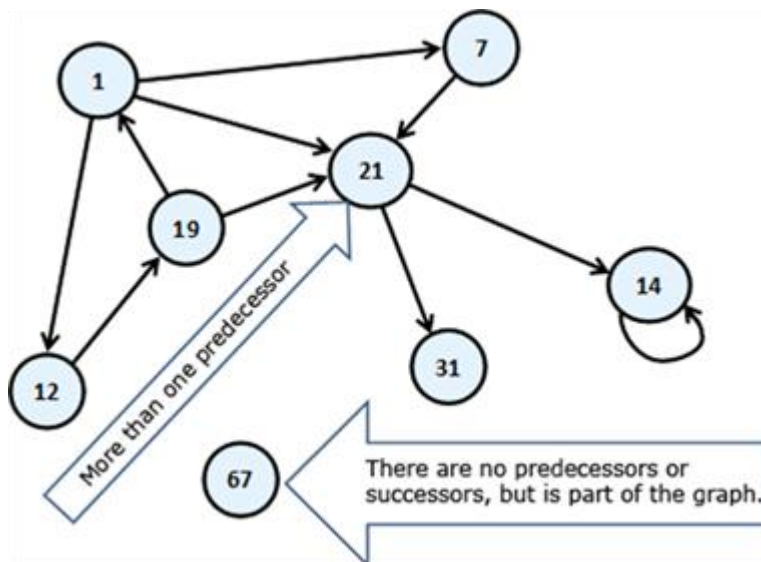
The graphs are very useful and fairly common data structures. They are used to describe a wide variety of relationships between objects and in practice can be related to almost everything. As we will see later, trees are a subset of the graphs and also lists are special cases of trees and thus of graphs, i.e. the graphs represent a generalized structure that allows modeling of very large set of real-world situations.

Frequent use of graphs in practice has led to extensive research in "graph theory", in which there is a large number of known problems for graphs and for most of them there are well-known solutions.

Graphs – Basic Concepts

In this section we will introduce some of the important concepts and definitions. Some of them are similar to those introduced about the tree data structure, but as we shall see, there are very serious differences, because trees are just special cases of graphs.

Let's consider the following sample graph (which we would later call a finite and oriented). Again, like with trees, we have numbered the graph, as it is easier to talk about any of them specifically:



The circles of this scheme we will call vertices (nodes) and the arrows connecting them we will call directed edges. The vertex of which the arrow comes out we will call predecessor of that the arrow points. For example "19" is a predecessor of "1". In this case, "1" is a successor of "19". Unlike the structure tree, here each vertex can have more than one

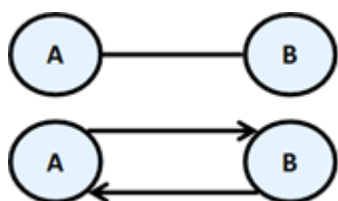
predecessor. Like “21”, it has three – “19”, “1” and “7”. If two of the vertices are connected with edge, then we say these two vertices are adjacent through this edge.

Finite directed graph.

Finite directed graph is called the couple (V, E) , in which V is a finite set of vertices and E is a finite set of directed edges. Each edge e that belongs to E is an ordered couple of vertices u and v or $e = (u, v)$, which are defining it in a unique way.

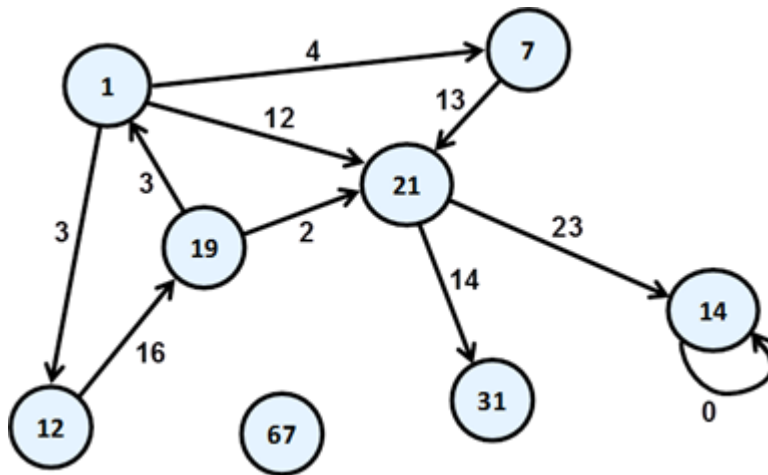
For better understanding of this definition we are strongly recommending to the reader to think of the vertices as they are cities, and the directed edges as one-way roads. That way, if one of the vertices is Sofia and the other is Paris, the one-way path (edge) will be called Sofia – Paris. In fact this is one of the classic examples for the use of the graphs – in tasks with paths.

If instead of arrows, the vertices are connected with segments, then the segments will be called undirected edges, and the graph – undirected. Practically we can imagine that an undirected edge from vertex A to vertex B is two-way edge and equivalent to two opposite directed edges between the same two vertices:



Two vertices connected with an edge are called **neighbors (adjacent)**.

For the edges a weight function can be assigned, that associates each edge to a real number. These numbers we will call weights (costs). For examples of the weights we can mention some distance between neighboring cities, or the length of the directed connections between two neighboring cities, or the crossing function of a pipe, etc. A graph that has weights on the edges is called weighted. Here is how it is illustrated a weighted graph.



Path in a graph is a sequence of vertices v_1, v_2, \dots, v_n , such as there is an edge from v_i to v_{i+1} for every i from 1 to $n-1$. In our example path is the sequence "1", "12", "19", "21". "7", "21" and "1" is not a path because there is no edge starting from "21" and ending in "1".

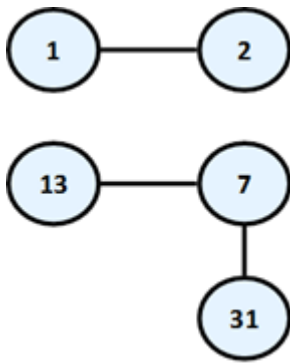
Length of path is the number of edges connecting vertices in the sequence of the vertices in the path. This number is equal to the number of vertices in the path minus one. The length of our example for path "1", "12", "19", "21" is three.

Cost of path in a weighted graph, we call the sum of the weights (costs) of the edges involved in the path. In real life the road from Sofia to Madrid, for example, is equal to the length of the road from Sofia to Paris plus the length of the road from Madrid to Paris. In our example, the length of the path "1", "12", "19" and "21" is equal to $3 + 16 + 2 = 21$.

Loop is a path in which the initial and the final vertex of the path match. Example of vertices forming loop are "1", "12" and "19". In the same time "1", "7" and "21" do not form a loop.

Looping edge we will call an edge, which starts and ends in the same vertex. In our example the vertex "14" is looped.

A connected undirected graph we call an undirected graph in which there is a path from each node to each other. For example, the following graph is not connected because there is no path from "1" to "7".



So we already have enough knowledge to define the concept tree in other way, as a special kind of graph:

Tree – undirected connected graph without loops.

As a small exercise we let the reader show why all definitions of tree we gave in this chapter are equivalent.

Graphs – Presentations

There are a lot of different ways to present a graph in the computer programming. Different representations have different properties and what exactly should be selected depends on the particular algorithm that we want to apply. In other words – we present the graph in a way, so that the operations that our algorithm does on it to be as fast as possible. Without falling into greater details we will set out some of the most common representations of graphs.

- **List of successors** – in this representation for each vertex v a list of successor vertices is kept (like the tree's child nodes). Here again, if the graph is weighted, then to each element of the list of successors an additional field is added indicating the weight of the edge to it.
- **Adjacency matrix** – the graph is represented as a square matrix $g[N][N]$, where if there is an edge from v_i to v_j , then the position $g[i][j]$ is contains the value 1. If such an edge does not exist, the field $g[i][j]$ is contains the value 0. If the graph is weighted, in the position $g[i][j]$ we record weight of the edge, and matrix is called a matrix of weights. If between two nodes in this matrix there is no edge, then it is recorded a special value meaning infinity. If the graph is undirected, the adjacency matrix will be symmetrical.
- **List of the edges** – it is represented through the list of ordered pairs (v_i, v_j) , where there is an edge from v_i to v_j . If the graph is weighted, instead ordered pair we have ordered triple, and its third element shows what the weight of the edge is.

- **Matrix of incidence between vertices and edges** – in this case, again we are using a matrix but with dimensions $g[M][N]$, where N is the number of vertices, and M is the number of edges. Each column represents one edge, and each row a vertex. Then the column corresponding to the edge (v_i, v_j) will contain 1 only at position i and position j , and other items in this column will contain 0. If the edge is a loop, i.e. is (v_i, v_i) , then on position i we record 2. If the graph we want to represent is oriented and we want to introduce edge from v_i to v_j , then to position i we write 1 and to the position j we write -1.

The most commonly used representation of graphs is the list of successors.

Graphs – Basic Operations

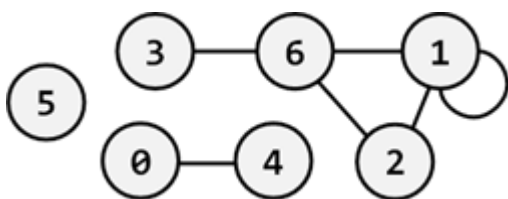
The basic operations in a graph are:

- Creating a graph
- Adding / removing a vertex / edge
- Check whether an edge exists between two vertices
- Finding the successors of given vertex

We will offer a sample implementation of the graph representation with a list of successors and we will show how to perform most of the operations. This kind of implementation is good when the most often operation we need is to get the list of all successors (child nodes) for a certain vertex. This graph representation needs a memory of order $N + M$ where N is the number of vertices and M is the number of edges in the graph.

In essence the vertices are numbered from 0 to $N-1$ and our Graph class holds for each vertex a list of the numbers of all its child vertices. It does not work with the nodes, but with their numbers in the range $[0..N-1]$.

By definition in undirected graph if a path exists between two nodes, they belong to the same connected component and if no path exists between two nodes, they belong to different connected components. For example consider the following undirected graph:



It has 3 connected components: $\{0, 4\}$, $\{1, 2, 6, 3\}$ and $\{5\}$.

Common Graph Applications

Graphs are used to model many situations of reality, and tasks on graphs model multiple real problems that often need to be resolved. We will give just a few examples:

- Map of a city can be modeled by a weighted oriented graph. On each street, edge is compared with a length, corresponding to the length of the street, and direction – the direction of movement. If the street is a two-way, it can be compared to two edges in both directions. At each intersection there is a node. In such a model there are natural tasks such as searching for the shortest path between two intersections, checking whether there is a road between two intersections, checking for a loop (if we can turn and go back to the starting position) searching for a path with a minimum number of turns, etc.
- Computer network can be modeled by an undirected graph, whose vertices correspond to the computers in the network, and the edges correspond to the communication channels between the computers. To the edges different numbers can be compared, such as channel capacity or speed of the exchange, etc. Typical tasks for such models of a network are checking for connectivity between two computers, checking for double-connectivity between two points (existence of double-secured channel, which remains active after the failure of any computer), finding a minimal spanning tree (MST), etc. In particular, the Internet can be modeled as a graph, in which are solved problems for routing packets, which are modeled as classical graph problems.
- The river system in a given region can be modeled by a weighted directed graph, where each river is composed of one or more edges, and each node represents the place where two or more rivers flow into another one. On the edges can be set values, related to the amount of water that goes through them. Naturally with this model there are tasks such as calculating the volume of water, passing through each vertex and anticipate of possible flood in increasing quantities.

You can see that the graphs can be used to solve many real-world problems. Hundreds of books and research papers are written about graphs, graph theory and graph algorithms. There are dozens of classic tasks for graphs, for which there are known solutions or it is known that there is no efficient solution.

CHAPTER 4: ANALYSIS OF ALGORITHM

What is an algorithm?

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

An algorithm is said to be *correct* if, for every input instance, it halts with the correct output. We say that a correct algorithm *solves* the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with other than the desired answer. Contrary to what one might expect, incorrect algorithms can sometimes be useful, if their error rate can be controlled. Concerned only with correct algorithms. An algorithm can be specified in English, as a computer program, or even as hardware

4.1 Algorithmic Complexity

Algorithmic complexity is concerned about how fast or slow particular algorithm performs. We define complexity as a numerical function $T(n)$ - time versus the input size n . We want to define time taken by an algorithm without depending on the implementation details. But you agree that $T(n)$ does depend on the implementation! A given algorithm will take different amounts of time on the same inputs depending on such factors as: processor speed; instruction set, disk speed, brand of compiler and etc. The way around is to estimate efficiency of each algorithm *asymptotically*. We will measure time $T(n)$ as the number of elementary "steps" (defined in any way), provided each such step takes constant time.

Let us consider two classical examples: addition of two integers. We will add two integers digit by digit (or bit by bit), and this will define a "step" in our computational model. Therefore, we say that addition of two n -bit integers takes n steps. Consequently, the total computational time is $T(n) = c * n$, where c is time taken by addition of two bits. On different computers, addition of two bits might take different time, say c_1 and c_2 , thus the

addition of two n-bit integers takes $T(n) = c_1 * n$ and $T(n) = c_2 * n$ respectively. This shows that different machines result in different slopes, but time $T(n)$ grows linearly as input size increases.

The process of abstracting away details and determining the rate of resource usage in terms of the input size is one of the fundamental ideas in computer science.

Asymptotic Notations

The goal of computational complexity is to classify algorithms according to their performances. We will represent the time function $T(n)$ using the "big-O" notation to express an algorithm runtime complexity. For example, the following statement

$$T(n) = O(n^2)$$

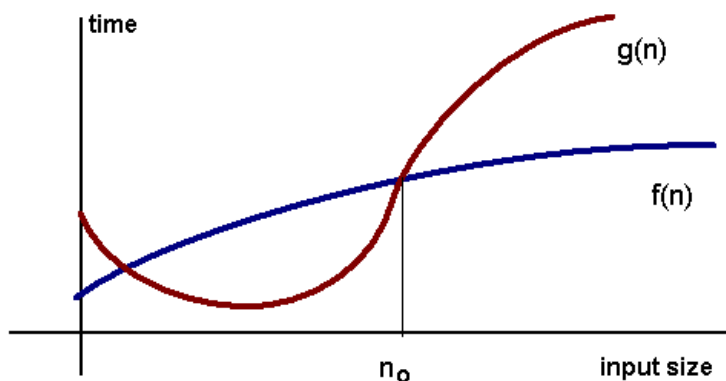
says that an algorithm has a quadratic time complexity.

Definition of "big Oh"

For any monotonic functions $f(n)$ and $g(n)$ from the positive integers to the positive integers, we say that $f(n) = O(g(n))$ when there exist constants $c > 0$ and $n_0 > 0$ such that $f(n) \leq c * g(n)$, for all $n \geq n_0$

Intuitively, this means that function $f(n)$ does not grow faster than $g(n)$, or that function $g(n)$ is an **upper bound** for $f(n)$, for all sufficiently large $n \rightarrow \infty$

Here is a graphic representation of $f(n) = O(g(n))$ relation:



Examples:

- $1 = O(n)$
- $n = O(n^2)$
- $\log(n) = O(n)$
- $2n + 1 = O(n)$

The "big-O" notation is not symmetric: $n = O(n^2)$ but $n^2 \neq O(n)$.

Exercise. Let us prove $n^2 + 2n + 1 = O(n^2)$. We must find such c and n_0 that $n^2 + 2n + 1 \leq c \cdot n^2$. Let $n_0=1$, then for $n \geq 1$

$$1 + 2n + n^2 \leq n + 2n + n^2 \leq n^2 + 2n^2 + n^2 = 4n^2$$

Therefore, $c = 4$.

Constant Time: $O(1)$

An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size. Examples:

- array: accessing any element
- fixed-size stack: push and pop methods
- fixed-size queue: enqueue and dequeue methods

Linear Time: $O(n)$

An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases. Examples:

- array: linear search, traversing, find minimum
- ArrayList: contains method
- queue: contains method

Logarithmic Time: $O(\log n)$

An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size. Example:

- binary search

Recall the "twenty questions" game - the task is to guess the value of a hidden number in an interval. Each time you make a guess, you are told whether your guess is too high or too low. Twenty questions game imploies a strategy that uses your guess number to halve the interval size. This is an example of the general problem-solving method known as **binary search**:

locate the element a in a sorted (in ascending order) array by first comparing a with the middle element and then (if they are not equal) dividing the array into two subarrays; if a is less than the middle element you repeat the whole procedure in the left subarray, otherwise - in the right subarray. The procedure repeats until a is found or subarray is a zero dimension.

Note, $\log(n) < n$, when $n \rightarrow \infty$. Algorithms that run in $O(\log n)$ does not use the whole input.

Quadratic Time: $O(n^2)$

An algorithm is said to run in logarithmic time if its time execution is proportional to the square of the input size. Examples:

- bubble sort, selection sort, insertion sort

Definition of "big Omega"

We need the notation for the **lower bound**. A capital omega Ω notation is used in this case. We say that $f(n) = \Omega(g(n))$ when there exist constant c that $f(n) \geq c \cdot g(n)$ for for all sufficiently large n . Examples

- $n = \Omega(1)$
- $n^2 = \Omega(n)$
- $n^2 = \Omega(n \log(n))$
- $2n + 1 = O(n)$

Definition of "big Theta"

To measure the complexity of a particular algorithm, means to find the upper and lower bounds. A new notation is used in this case. We say that $f(n) = \Theta(g(n))$ if and only $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Examples

- $2n = \Theta(n)$

- $n^2 + 2n + 1 = \Theta(n^2)$

Analysis of Algorithms

The term analysis of algorithms is used to describe approaches to the study of the performance of algorithms. In this course we will perform the following types of analysis:

- The *worst-case runtime complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size a .
- The *best-case runtime complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size a .
- The *average case runtime complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size a .
- The *amortized runtime complexity* of the algorithm is the function defined by a sequence of operations applied to the input of size a and averaged over time.

Example. Let us consider an algorithm of sequential searching in an array of size n .

Its *worst-case runtime complexity* is $O(n)$

Its *best-case runtime complexity* is $O(1)$

Its *average case runtime complexity* is $O(n/2)=O(n)$

4.2 How to Design Algorithms

Creating an algorithm design is an art, which may never be fully automated. Various algorithm design techniques that have proven to be useful in that they have often yielded good algorithms. By mastering these design strategies, it will become easier for you to devise new and useful algorithms. Algorithm is a method for solving a computational problem and algorithm design is identified in many solution theories of operation research, such as divide and conquer, dynamic programming and greedy algorithm.

The Techniques for designing and implementing algorithm design is based on template method patterns, data structures etc. A Design technique is often expressed in pseudocode as a template that can be particularized for concrete problems [3]. This template name is algorithm schemas. Algorithm schemas consist on identifying structural similarities among algorithms that solve different problems. Programming language such as algorithmic language, COBOL, FORTRAN, PASCAL, SAIL are computing tools to

implement an algorithm design. But algorithm design is not a programming tool. Algorithm design basically mathematical process of writing a finite set of steps each of which may require one or more operations.

4.3 How to Express Algorithms

Algorithm design can be expressed in many kinds of notation like flowcharts, programming language, rational rose tool, computer aided design applications and pseudocode etc. flowcharts and rational rose are expressed in structured way and avoid the ambiguities in language statements. Computer-Aided Design (CAD), also known as Computer-Aided Drafting, is the use of computer software and systems to design and create 2D and 3D virtual models of goods and products for the purposes of testing. It is also sometimes referred to as computer assisted drafting. Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

4.4 Fundamental Concepts of Algorithm

There are two fundamental concepts of algorithm

- Functional correctness
- Proof of correctness

4.4.1 Functional Correctness

Functional correctness is depend on the following

- **Precondition:** Algorithm is correct if every data that satisfy some condition that is called precondition of the algorithm
- **Post condition:** The out put data satisfy a certain predefined condition that is called post condition of the algorithm.

4.4.2 Proofs of Correctness of Algorithms

Correctness of algorithm is depends on the two issues

- Given an algorithm prove that it is correct. It is always achieves the intended result

- Design an algorithm with intended properties from scratch. This is even more difficult.

Proof of correctness also depends on the mathematical proof. Whenever algorithm is run on a set of inputs that satisfy the problems precondition is expected to hold before the method is executed. Post condition what holds after the method is executed. A proof that a program is correct often has two pieces (that can be developed separately)

- **Proof of partial correctness:** This is a proof that, whenever an algorithm is run on a set of inputs satisfying the problem's precondition, either
 - The algorithm halts, and the outputs (and inputs) satisfy the problem's post condition, or
 - The algorithm does not halt at all.
- **Proof of termination:** This is a proof that the algorithm always halts, whenever it is run on a set of inputs that satisfy the precondition.

4.5 FUNDAMENTAL ALGORITHMS STRATEGIES

4.5.1 Brute Force Algorithms

Definition: An algorithm that inefficiently solves a problem, often by trying every one of a wide range of possible solutions

Main Approach

- Generate and evaluate possible solutions until
 - Satisfactory solution is found.
 - Best solution is found.
 - All possible solutions found
 - Return best solution
 - Return failure if no satisfactory solution.
- Generally most expensive approach.

Description

A brute force algorithm simply tries all possibilities until a satisfactory solution is found such an algorithm can be:

- **Optimizing:** Find the best solution. This may require finding all solutions, or if a value for the best solution is known, it may stop when any best solution is found
- **Satisfying:** Stop as soon as a solution is found that is good enough

Brute force algorithm is require exponential time and used in various heuristics and optimizations.

Heuristic: A rule of thumb that helps you decide which possibilities to look at first.

Optimization: In this case, to eliminate certain possibilities without fully exploring them.

The C code

```
Void BF (char *x, int m, char *y, int n) {

    int i, j;
    /* Searching */
    for (j = 0; j <= n - m; ++j)
    {
        for (i = 0; i < m && x[i] == y[i + j]; ++i);
        if (i >= m)
            OUTPUT(j);
    }
}
```

Strengths:

- Wide applicability
- Simplicity
- Yields reasonable algorithms for some important problems
 - Searching, string matching, matrix multiplication
- Yields standard algorithms for simple computational tasks
 - Sum or product of n numbers, finding max or min in a list

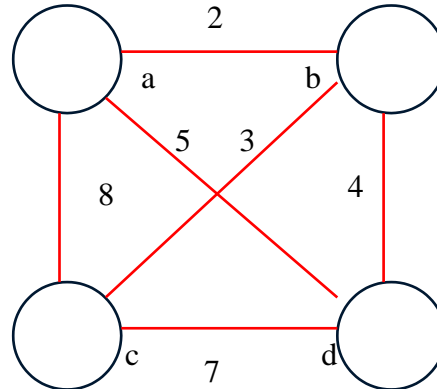
Weaknesses:

- Rarely yields efficient algorithms
- Some brute force algorithms unacceptably slow
- Not as constructive/creative as some other design techniques

Example 1: Traveling salesman problem

Question: Given n cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city.

Example:



Tour

a→b→c→d→a
a→b→d→c→a
a→c→b→d→a
a→c→d→b→a
a→d→b→c→a
a→d→c→b→a

Cost

$2+3+7+5 = 17$
 $2+4+7+8 = 21$
 $8+3+4+5 = 20$
 $8+7+4+2 = 21$
 $5+4+3+8 = 20$
 $5+7+3+2 = 17$

4.5.2 Greedy Algorithm

The greedy algorithm is perhaps the most straightforward design technique. It can be applied to a wide variety of problem. Most though not all of these problems have n inputs and require us to obtain a subset that satisfies some constraints. Any need to find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called an optimal solution.

Note: greedy algorithm avoid backtracking and exponential time $O(2^n)$

Greedy algorithms work in phases. In each phase, a decision is made that appears to be good, without regard for future consequences E.g. Kruskal's MST algorithm, Dijkstra's algorithm.

Definition

Greedy algorithm work in phases. In each phase, a decision is made that appears to be good, without regard for future consequences. Generally, this means that some local optimum is chosen. Greedy algorithm to find minimum spanning tree. Want to find set of edges.

Note: Prim's algorithm and Kruskal's algorithm are greedy algorithms that find the globally optimal solution, a minimum spanning tree. In contrast, any known greedy algorithm to find a Hamiltonian cycle might not find the shortest path, that is, a solution to the traveling salesman problem. If there is no greedy algorithm that always finds the optimal solution for a problem, one may have to search (exponentially) many possible solutions to find the optimum. Greedy algorithms are usually quicker, since they don't consider the details of possible alternatives

Type of Greedy Algorithm

There are three type of greedy algorithms

- Pure Greedy Algorithms
- Orthogonal Greedy Algorithms
- Relaxed Greedy Algorithms

General Characteristics of Greedy Algorithms

Commonly, greedy algorithms and the problems they can solve are characterized by most or all of the following features.

- To construct the solution of our problem, a set (or list) of candidates is required: the coins that are available, the edges of a graph that may be used to build a path, the set of jobs to be Scheduled, or whatever.
- As the algorithm proceeds, two other sets are accumulated. One contains candidates that have already been considered and chosen, while the other contains candidates that have been considered and rejected.
- There is a function that checks whether a particular set of candidates provides a solution to our problem, ignoring questions of optimality for the time being. For instance, do the coins add up to the amount to be paid? Do the selected edges provide a path to the node to reach? Have all the jobs been scheduled?
- A second function checks whether a set of candidates is feasible, that is, whether or not it is possible to complete the set by adding further candidates so as to obtain at least one solution to our problem. Here too, the time being concerned is not with optimallty.
- Yet another function, the selection function, indicates at any time which of the remaining candidates, that have neither been chosen nor rejected, is the most promising.
- Finally an objective function gives the value of a solution found: the number of coins used to make change, the length of the path constructed, the time needed to process all the jobs in the schedule, or whatever other values are trying to optimize. Unlike the three functions mentioned previously, the objective function does not appear explicitly in the greedy algorithm.

Example: - Scheduling

Given jobs $j_1, j_2, j_3, \dots, j_n$ with known running times $t_1, t_2, t_3, \dots, t_n$. what is the best way to schedule the jobs to minimize average completion time?

| Job | Time |
|-----|------|
| J1 | 16 |
| J2 | 8 |
| J3 | 3 |
| J4 | 14 |

Scheduling

| | | | |
|----|----|----|----|
| J1 | J2 | J3 | J4 |
| 16 | 24 | 27 | 41 |

Average completion time = $(16+24+27+41)/4 = 27$

| | | | |
|----|----|----|----|
| J3 | J2 | J4 | J1 |
| 3 | 11 | 25 | 41 |

Average completion time = $(3+11+25+41)/4 = 20$

Description

- Greedy-choice property: if shortest job does not go first, the y jobs before it will complete 3 time units faster, but j_3 will be postponed by time to complete all jobs before it
- Optimal substructure: if shortest job is removed from optimal solution, remaining solution for $n-1$ jobs is optimal

Optimality Proof

•Total cost of a schedule is

$$\sum_{k=1}^N (N-k+1)t_{ik}$$

$$t_1 + (t_1+t_2) + (t_1+t_2+t_3) \dots (t_1+t_2+\dots+t_n)$$

$$(N+1)\sum t_{ik} - \sum k*t_{ik}$$

$$k=1$$

•First term independent of ordering, as second term increases, total cost becomes smaller

Suppose there is a job ordering such that $x > y$ and $t_{ix} < t_{iy}$. Swapping jobs (smaller first) increases second term decreasing total cost

Show: $xt_{ix} + yt_{iy} < yt_{ix} + xt_{iy}$

$$xt_{ix} + yt_{iy} = xt_{ix} + yt_{ix} + y(t_{iy} - t_{ix})$$

$$= yt_{ix} + xt_{ix} + y(t_{iy} - t_{ix})$$

$$< yt_{ix} + xt_{ix} + x(t_{iy} - t_{ix})$$

$$= yt_{ix} + xt_{ix} + xt_{iy} - xt_{ix} = yt_{ix} + xt_{iy}$$

4.5.3 Divide and Conquer

Divide and conquer algorithm suggests splitting the inputs into distinct subsets. These sub problems must be solved and then a method must be found to combine sub solutions into a solution of the whole. If the sub problems are still relatively large, then the divide and conquer strategy can be possibly be reapplied. Often the sub problems resulting from a divide and conquer design are of the same type as the original problem. For those cases the reapplication of the divide and conquer principle is naturally expressed by a recursive algorithm. This algorithm technique is the basis of efficient algorithms for all kinds of the problems, such as quick sort, merge sort and discrete Fourier transform. Its application to numerical algorithms is commonly known as binary splitting .

Divide-and-conquer algorithm works as follows:

- Divide and conquer algorithm are divided into several smaller instances of the same problem and same size.
- The smaller instances are solved by recursively.
 - Sometimes, a different algorithm is applied when instances become small enough.
- The smaller instances are combined and to get a solution to the original problem.
 - No necessary to combine in some cases.

- Divide-and-conquer technique is ideally suited for parallel computers, in which each sub problem can be solved simultaneously by its own processors.
- Common case: Dividing a problem into two smaller problems

Algorithm of Divide and Conquer

1. Algorithm D-and-C (n: input size)
2. If $n \leq n_0$ /* small size problem*/
3. Solve problem without further sub-division;
4. Else
5. Divide into m sub-problems;
6. Conquer the sub-problems by solving them
7. Independently and recursively; /* D-and-C (n/k) */
8. Combine the solutions;

Examples:

- Binary search
- Powering a number
- Fibonacci numbers
- Matrix multiplication
- Strassen's algorithm
- VLSI tree layout

Divide and Conquer

Divide: $P \Rightarrow P_1, \dots, P_k$

Conquer: $S(P_1), \dots, S(P_k)$

Merge: $S(P_1), \dots, S(P_k) \Rightarrow S(P)$

Examples: Sorting (merge sort and quick sort), searching (binary search), closest pair (the $O(n \log n)$ algorithm), and selection (the linear-time algorithm).

Algorithm template:

- Function $P(n)$
- if $n \leq c$
- Solve P directly
- Return its solution

- Else P => P1, ..., Pk //divide
- For i = 1 to k
- Si = P(ni) //conquer
- S1, ..., Sk => S //merge
- return S

Time complexity:

$$T(n) = \begin{cases} 1 & n \leq c \\ \sum_{i=1}^k T(n_i) + D(n) + M(n), & n < c \end{cases}$$

Strassen's algorithm

-Given $A=(a_{ij})_{n \times n}$ and $B=(b_{ij})_{n \times n}$.

$$\text{Let } C = A \times B = (c_{ij})_{n \times n}, \text{ for } C_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

– First algorithm:

for i = 1 to n
 for j = 1 to n
 c[i,j] = 0
 for k = 1 to n
 c[i,j] = c[i,j] + a[i,k] * b[k,j]

Time complexity: $O(n^3)$

- Second algorithm:

$$A_{n \times n} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$$B_{n \times n} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C_{n \times n} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$C_{11} = A_{11} B_{11} + A_{12} B_{21}$$

$$C_{12} = A_{11} B_{12} + A_{12} B_{22}$$

$$C_{21} = A_{21} B_{11} + A_{22} B_{21}$$

$$C_{22} = A_{21} B_{12} + A_{22} B_{22}$$

So the multiplication of two $n \times n$ matrices becomes eight multiplications of two $n/2 \times n/2$ matrices, giving us $T(n) = 8T(n/2) + O(n^2)$. By iterating, we have $T(n) = O(n^3)$.
No improvement!

| | |
|-----|--------------------------------------|
| M1 | $(A_{12} - A_{22})(B_{21} + B_{22})$ |
| M2 | $(A_{11} + A_{22})(B_{11} + B_{22})$ |
| M3 | $(A_{11} - A_{21})(B_{11} + B_{12})$ |
| M4 | $(A_{11} + A_{12})B_{22}$ |
| M5 | $A_{11}(B_{12} - B_{22})$ |
| M6 | $A_{22}(B_{21} - B_{11})$ |
| M7 | $(A_{21} + A_{22})B_{11}$ |
| C11 | $M_1 - M_2 - M_4 + M_6$ |
| C12 | $M_4 + M_5$ |
| C21 | $M_6 + M_7$ |
| C22 | $M_2 - M_3 + M_5 - M_7$ |

Using the above idea in the algorithm, we get

$$T(n) = 7T(n/2) + O(n^2), \text{ thus } T(n) = O(n^{\log_2 7}) =$$

$O(n^{2.81})$ By iterating

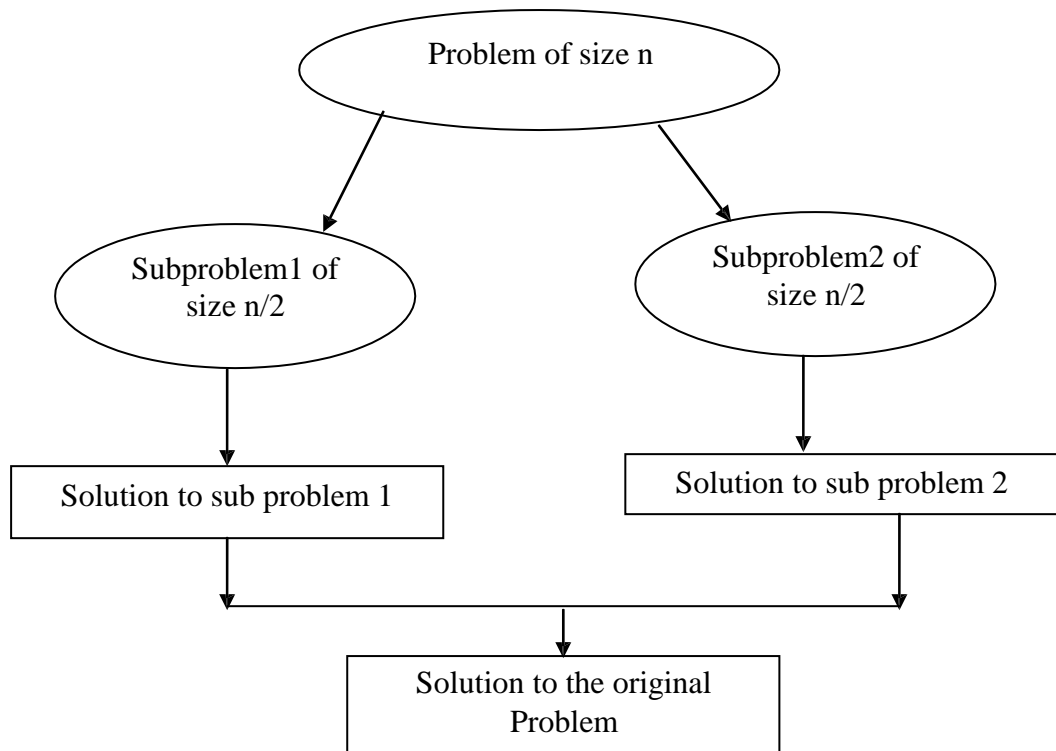


Figure 2.1 Divide and Conquer

4.5.4 Dynamic programming

Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions. Dynamic programming is similar to divide and conquer algorithm. It expresses solution of a problem in terms of solutions to sub problems. The key difference between dynamic programming and divide and conquer is that while sub problems in divide and conquer are independent, sub problems in dynamic programming may themselves share sub problems. This means that if these were treated as independent sub problems, the complexity would be higher. Dynamic programming is typically used to solve optimization problems. In bioinformatics, the most common use of dynamic programming is in sequence matching and alignment.

- To begin, the word programming is used by mathematicians to describe a set of rules, which must be followed to solve a problem.
- Thus, linear programming describes sets of rules which must be solved a linear problem.
- In our context, the adjective dynamic describes how the set of rules works.
- In this course, a number of examples of recursive algorithms are seen.

- The run time of these algorithms may be found by solving the recurrence relation itself.
- The first example of a dynamic program is a technique for solving the following recurrence relation [9].

$$F(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

- You will recall that this defines the Fibonacci sequence of integers:

1, 1, 2, 3, 5, 8, 13, 21, 33, 54,...

Example

// Calculate the nth Fibonacci number

```
Double F( double n ) {
    if ( n <= 1 ) {
        return 1.0;
    } else {
        return F( n - 1 ) + F( n - 2 );
    }
}
```

- Recall definition of Fibonacci numbers: $f(0) = 0$
 $f(1) = 1$
 $f(n) = f(n-1) + f(n-2)$
- Compute the n^{th} Fibonacci number recursively (top-down)

$$\begin{array}{ccccccc} & & & & f(n) & & \\ & & & & + & & \\ & f(n-1) & & & & f(n-2) & \\ & + & & & & + & \\ f(n-2) & + & f(n-3) & & f(n-3) & + & f(n-4) \end{array}$$

Example: Fibonacci numbers (2)

Compute the n^{th} Fibonacci number using bottom-up iteration:

1. $F(0) = 0$
2. $F(1) = 1$
3. $F(2) = 0+1 = 1$

4. $F(3) = 1+1 = 2$
5. $F(4) = 1+2 = 3$
6. $F(n-2) =$
7. $F(n-1) =$
8. $F(n) = f(n-1) + f(n-2)$

Example: Computing binomial coefficients

Algorithm Based On Identity

- Algorithm Binomial (n,k)
 - for i ← 0 to n do
 1. for j ← 0 to min(j,k) do
 2. if j=0 or j=i then $C[i,j] \leftarrow 1$
 3. else $C[i,j] \leftarrow C[i-1,j-1]+C[i-1,j]$
 4. return C[n,k]
- Pascal's Triangle

4.5.5 Backtracking Algorithm

Backtracking algorithm represents one of the most general techniques. Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation. Many of the problems being solved using backtracking require that all the solutions satisfy a complex set of constraints. For any problem these constraints can be divided into two categories explicit and implicit.

- View the problem as a sequence of decisions
- Systematically considers all possible outcomes for each decision
- Backtracking algorithms are like the brute-force algorithms
- However, they are distinguished by the way in which the space of possible solutions is explored
- Sometimes a backtracking algorithm can detect that an exhaustive search is not needed

Example: - Solving a maze

- Given a maze, find a path from start to finish

- At each intersection, you have to decide between three or fewer choices:
 - Go straight
 - Go left
 - Go right
- You don't have enough information to choose correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many types of maze problem can be solved with backtracking

Solving a puzzle

- In this puzzle, all holes but one are filled with white pegs
- You can jump over one peg with another
- Jumped pegs are removed
- The object is to remove all but the last peg
- You don't have enough information to jump correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many kinds of puzzle can be solved with backtracking

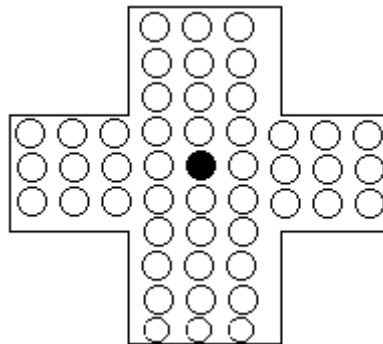


Figure 2.2 Puzzle Problem

4.5.6 Branch and Bound Algorithms

Branch and Bound Algorithm based on limiting search using current solution. It means this is a general search method. This method considering the root problem and lower bounding and upper bounding procedures are applied to the root problem. Branch and bound algorithm is applied recursively to the sub problem. If an optimal

solution is found to a sub problem, it is a feasible solution to the full problem, but not necessarily globally optimal.

Branch and Bound Algorithm Approach

- Firstly try to track best current solution found
- The partial current solutions that can't be improved that should be eliminated.
- Reduces amount of backtracking

Note: Not guaranteed to avoid exponential time $O(2^n)$

Basic features of Branch and Bound Algorithm

Best solution is only compared with a nodes bound values only if the bound value is not better than the best solution so far there are following reasons

- The value of the node bound is not better than the other
- Node does not represent the feasible solutions
- The node consists of a single point represent the subset of feasible solutions.

Example: Assignment Problem

Assigning n people to n jobs so that the total cost is minimized. Each person does one job and each job is assigned to one person.

Read the assignments as $\langle \text{Job 1, Job 2, Job 3, Job 4} \rangle$:

$\langle c, b, a, d \rangle$ assigns Person c Job 1, Person b Job 2, *etc.*

| | Job 1 | Job2 | Job3 | Job4 | |
|----|--------------|-------------|-------------|-------------|----------|
| C= | 3 | 2 | 7 | 8 | Person a |
| | 6 | 4 | 3 | 7 | Person b |
| | 5 | 8 | 1 | 8 | Person c |
| | 7 | 6 | 9 | 4 | Person d |

$$\langle a, b, c, d \rangle \text{ cost} = 3 + 4 + 1 + 4 = 12$$

$$\langle a, b, d, c \rangle \text{ cost} = 3 + 4 + 9 + 8 = 24$$

$$\langle a, d, b, c \rangle \text{ cost} = 3 + 6 + 4 + 8 = 21$$

$$\langle d, a, b, c \rangle \text{ cost} = 7 + 2 + 3 + 8 = 20$$

$\langle d, c, b, a \rangle$ cost = 7+8+3+8=26

Etc. totaling 4! Permutations.

Permutations: Generate $n!$ Permutations. The following prints all the costs of the $n!$ Job assignments

All permutations algorithm - this is a simple algorithm just to generate all $n!$ Permutations

Assumes: person $\leftarrow a + 1 \Rightarrow$ person = b

Initially, $X[a..d]$ is unassigned any Job.

Permutations(X[a..d], person)

1. **if** person = d **then** print cost(X) -- Bottom of space
2. **else**
3. **for** Job $\in \{ 1, 2, 3, 4 \}$ **do**
4. **if not** assigned(X, Job)
5. X[person+1] \leftarrow Job -- Assign person a job
6. Permutations(X[a..d], person+1)
7. X[person+1] $\leftarrow \Phi$ -- Unassign job

Cost (X) returns cost of assigning Job 1..4 to person a..d

Assigned (X, Job) returns true if Job is assigned person a..d

The resulting state-space for assigning Jobs { 1, 2, 3, 4 } to each person { a, b, c, d } is:

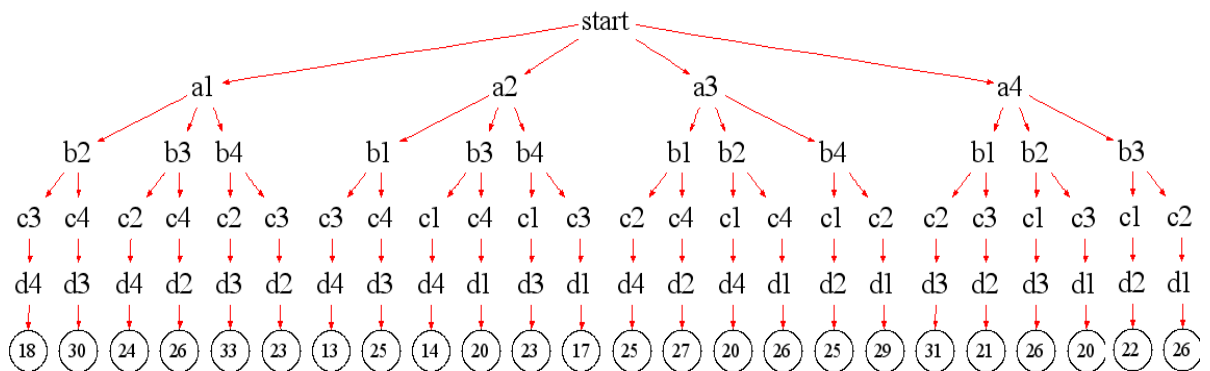


Figure 2.3 Branch and bound Assignment problem

| | Job 1 | Job2 | Job3 | Job4 | |
|----|-------|------|------|------|----------|
| C= | 3 | 2 | 7 | 8 | Person a |
| | 6 | 4 | 3 | 7 | Person b |
| | 5 | 8 | 1 | 8 | Person c |
| | 7 | 6 | 9 | 4 | Person d |

From the table above, the rightmost branch $\langle d, c, b, a \rangle$, $\text{cost} = 7^d 1 + 8^c 2 + 3^b 3 + 8^a 4 = 26$

Example: 4-queens problem

Examples: a) Longest Common Subsequence (LCS)

Given two sequences $x[1 \dots m]$ and $y[1 \dots n]$, find a longest subsequence common to them both.

b) Optimal Substructure

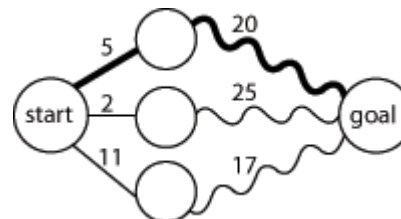


Figure 2.4 Optimal Substructure

4.5.7 Decrease-And-Conquer algorithm

Decrease-and-conquer is an approach to solving a problem by:

- Change an instance into one smaller instance of the problem.
- Solve the small instance.
- Convert the solution of the small instance into a solution for the large instance.

Decrease by a Constant

Decrease-by-a constant decreases the instance size by 1 (or some other constant), e.g., $210 = 2 * 29$

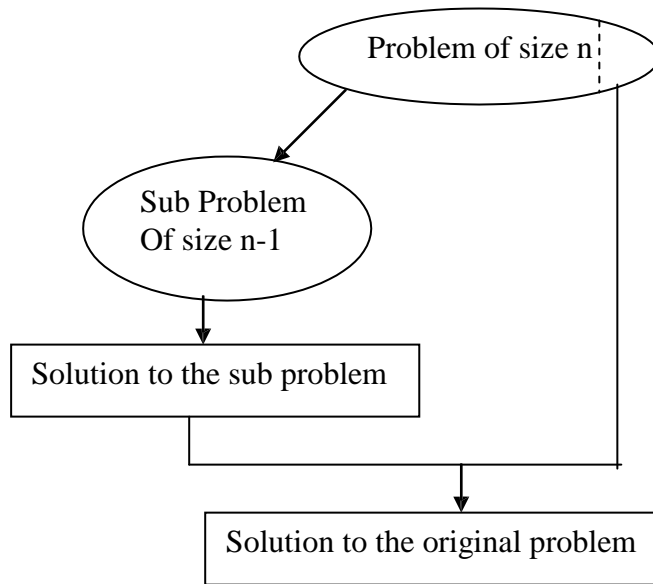


Figure 2.5 Decrease-And-Conquer algorithm

Decrease by a Constant Factor

Decrease-by-a constant-factor decreases the instance size by half (or some other fraction), e.g., $210 = 25 * 25$.

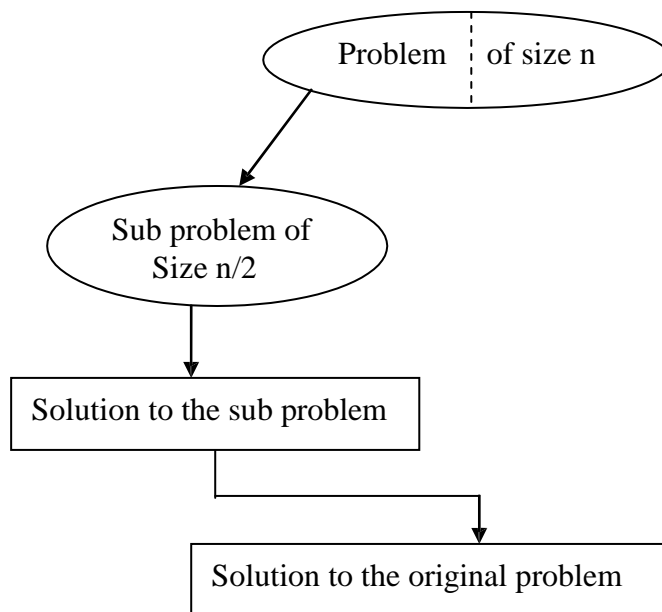


Figure 2.6 Decrease by a constant factor

Comments on Insertion Sort

- Insertion sort ensures $A[0] \leq A[1] \leq \dots \leq A[i - 1]$.
- Insertion sort looks for correct position for $A[i]$.

- Insertion sort shifts values at and above correct position.

- Worst Case: The number of comparisons

$$\sum_{i=1}^{n-1} i = n(n-1)/2 \in O(n^2).$$

- Best Case: $n-1 \in \Omega(n)$ comparisons if array is already sorted.
- Average Case $\approx n^2/4$ comparisons.

4.6 GRAPH AND TREE ALGORITHMS

Graph Traversal

Graph traversal algorithms process all the vertices of a graph in a systematic fashion.

- They are useful for many graph problems such as checking connectivity, checking a cyclicity, connected components, finding articulation points, and topological sorting.
- First all the vertices are marked as unvisited.
- Then an unvisited vertex is selected, marked as visited, and all unvisited vertices reachable from that vertex are marked as visited.
- Repeat above step until all vertices are visited.


4.6.1 Depth-first search (DFS) for undirected graphs


Depth-first search, or **DFS**, is a way to traverse the graph. Initially it allows visiting vertices of the graph only, but there are hundreds of algorithms for graphs, which are based on DFS. Therefore, understanding the principles of depth-first search is quite important to move ahead into the graph theory. The principle of the algorithm is quite simple: to go forward (in depth) while there is such possibility, otherwise to backtrack.

Algorithm

In DFS, each vertex has three possible colors representing its state:

 white: vertex is unvisited;

 gray: vertex is in progress;

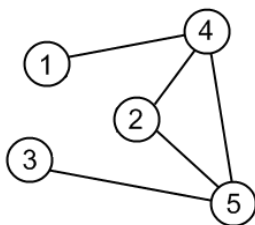
 black: DFS has finished processing the vertex.

NB. For most algorithms boolean classification *unvisited* / *visited* is quite enough, but we show general case here.

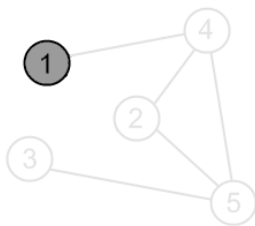
Initially all vertices are white (unvisited). DFS starts in arbitrary vertex and runs as follows:

1. Mark vertex **u** as gray (visited).
2. For each edge (**u**, **v**), where **u** is white, run depth-first search for **u** recursively.
3. Mark vertex **u** as black and backtrack to the parent.

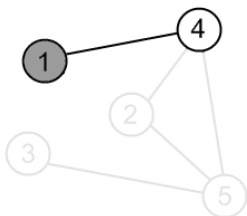
Example. Traverse a graph shown below, using DFS. Start from a vertex with number 1.



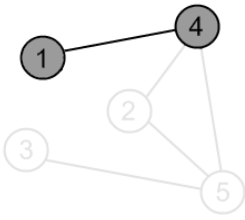
Source graph.



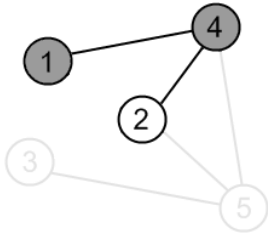
Mark a vertex **1** as gray.



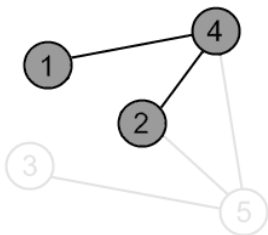
There is an edge (**1**, **4**) and a vertex **4** is unvisited. Go there.



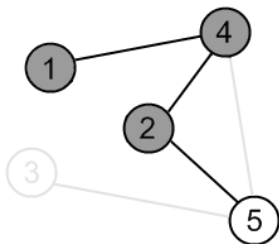
Mark the vertex **4** as gray.



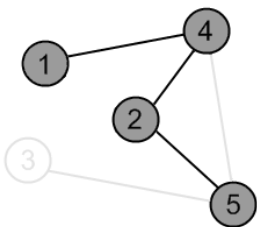
There is an edge **(4, 2)** and vertex **2** is unvisited. Go there.



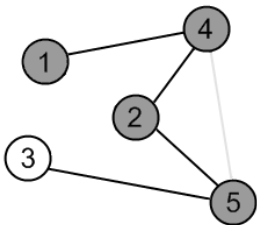
Mark the vertex **2** as gray.



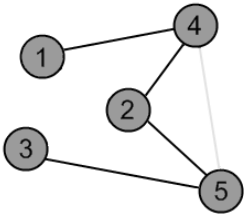
There is an edge **(2, 5)** and a vertex **5** is unvisited. Go there.



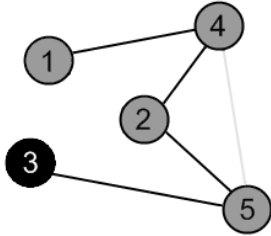
Mark the vertex **5** as gray.



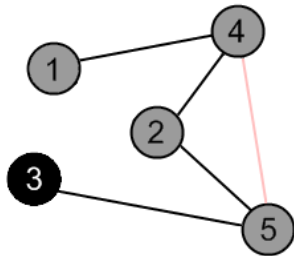
There is an edge **(5, 3)** and a vertex **3** is unvisited. Go there.



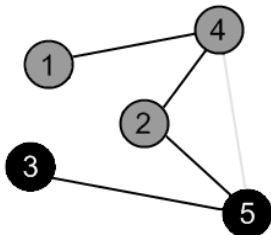
Mark the vertex **3** as gray.



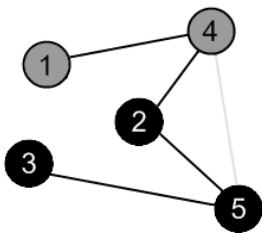
There are no ways to go from the vertex **3**. Mark it as black and backtrack to the vertex **5**.



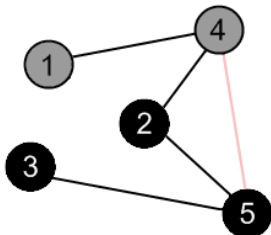
There is an edge **(5, 4)**, but the vertex 4 is gray.



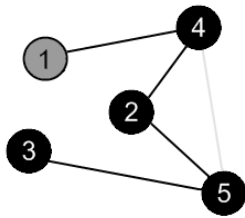
There are no ways to go from the vertex **5**. Mark it as black and backtrack to the vertex **2**.



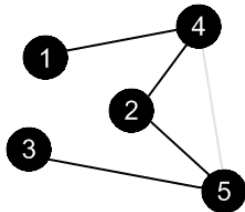
There are no more edges, adjacent to vertex **2**. Mark it as black and backtrack to the vertex **4**.



There is an edge **(4, 5)**, but the vertex 5 is black.



There are no more edges, adjacent to the vertex **4**. Mark it as black and backtrack to the vertex **1**.



There are no more edges, adjacent to the vertex **1**. Mark it as black. DFS is over.

As you can see from the example, DFS doesn't go through all edges. The vertices and edges, which depth-first search has visited is a **tree**. This tree contains all vertices of the graph (if it is connected) and is called *graph spanning tree*. This tree exactly corresponds to the recursive calls of DFS.

If a graph is disconnected, DFS won't visit all of its vertices. For details, see *finding connected components algorithm*.

Complexity analysis

Assume that graph is connected. Depth-first search visits every vertex in the graph and checks every edge its edge. Therefore, DFS complexity is $O(V + E)$. As it was mentioned before, if an adjacency matrix is used for a graph representation, then all edges, adjacent to a vertex can't be found efficiently, that results in $O(V^2)$ complexity.

C++ code

```
enum VertexState { White, Gray, Black };
```

```
...
```

```
void Graph::DFS() {
```

```
    VertexState *state = new VertexState[vertexCount];
```

```

    for (int i = 0; i < vertexCount; i++)

        state[i] = White;

runDFS(0, state);

delete [] state;

}

void Graph::runDFS(int u, VertexState state[]) {

    state[u] = Gray;

    for (int v = 0; v < vertexCount; v++)

        if (isEdge(u, v) && state[v] == White)

            runDFS(v, state);

    state[u] = Black;

}

```

4.6.2 Breadth-First Search Traversal Algorithm

Breadth-first search is a way to find all the vertices reachable from the a given source vertex, s . Like depth first search, BFS traverse a connected component of a given graph and defines a spanning tree. Intuitively, the basic idea of the breath-first search is this: send a wave out from source s . The wave hits all vertices 1 edge from s . From there, the wave hits all vertices 2 edges from s . Etc. We use FIFO queue Q to maintain the wavefront: v is in Q if and only if wave has hit v but has not come out of v yet.

Overall Strategy of BFS Algorithm

Breadth-first search starts at a given vertex s , which is at level 0. In the first stage, we visit all the vertices that are at the distance of one edge away. When we visit there, we paint as "visited," the vertices adjacent to the start vertex s - these vertices are placed into level 1. In the second stage, we visit all the new vertices we can reach at the distance of two edges away from the source vertex s . These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2, and so on. The BFS traversal terminates when every vertex has been visited.

To keep track of progress, breadth-first-search colors each vertex. Each vertex of the graph is in one of three states:

1. Undiscovered;
2. Discovered but not fully explored; and
3. Fully explored.

The state of a vertex, u , is stored in a color variable as follows:

1. $\text{color}[u] = \text{White}$ - for the "undiscovered" state,
2. $\text{color}[u] = \text{Gray}$ - for the "discovered but not fully explored" state, and
3. $\text{color}[u] = \text{Black}$ - for the "fully explored" state.

The $\text{BFS}(G, s)$ algorithm develops a breadth-first search tree with the source vertex, s , as its root. The parent or predecessor of any other vertex in the tree is the vertex from which it was first discovered. For each vertex, v , the parent of v is placed in the variable $\pi[v]$. Another variable, $d[v]$, computed by BFS contains the number of tree edges on the path from s to v . The breadth-first search uses a FIFO queue, Q , to store gray vertices.

Algorithm: Breadth-First Search Traversal

BFS(V, E, s)

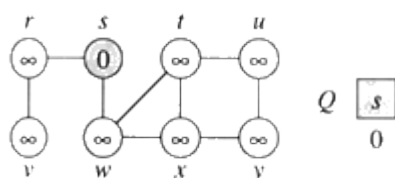
```

1.  for each  $u$  in  $V - \{s\}$            // for each vertex  $u$  in  $V[G]$  except  $s$ .
2.      do  $\text{color}[u] \leftarrow \text{WHITE}$ 
3.           $d[u] \leftarrow \text{infinity}$ 
4.           $\pi[u] \leftarrow \text{NIL}$ 
5.   $\text{color}[s] \leftarrow \text{GRAY}$            // Source vertex discovered
6.   $d[s] \leftarrow 0$                    // initialize
7.   $\pi[s] \leftarrow \text{NIL}$              // initialize
8.   $Q \leftarrow \{\}$                    // Clear queue  $Q$ 
9.  ENQUEUE( $Q, s$ )
10 while  $Q$  is non-empty
11.     do  $u \leftarrow \text{DEQUEUE}(Q)$        // That is,  $u = \text{head}[Q]$ 
12.         for each  $v$  adjacent to  $u$        // for loop for every node along with edge.
13.             do if  $\text{color}[v] \leftarrow \text{WHITE}$  // if color is white you've never seen it before
14.                 then  $\text{color}[v] \leftarrow \text{GRAY}$ 
15.                      $d[v] \leftarrow d[u] + 1$ 
16.                      $\pi[v] \leftarrow u$ 
17.                     ENQUEUE( $Q, v$ )
18.     DEQUEUE( $Q$ )
19.      $\text{color}[u] \leftarrow \text{BLACK}$ 

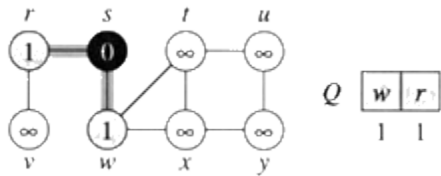
```

Example: The following figure (from CLRS) illustrates the progress of breadth-first search on the undirected sample graph.

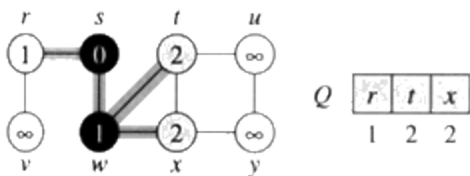
a. After initialization (paint every vertex white, set $d[u]$ to infinity for each vertex u , and set the parent of every vertex to be NIL), the source vertex is discovered in line 5. Lines 8-9 initialize Q to contain just the source vertex s .



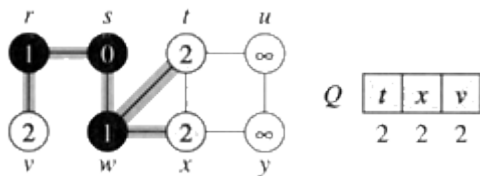
b. The algorithm discovers all vertices 1 edge from s i.e., discovered all vertices (w and r) at level 1.



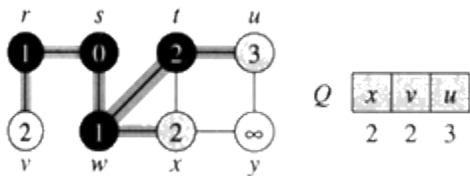
c.



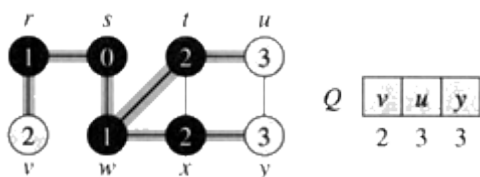
d. The algorithm discovers all vertices 2 edges from s i.e., discovered all vertices (t , x , and v) at level 2.



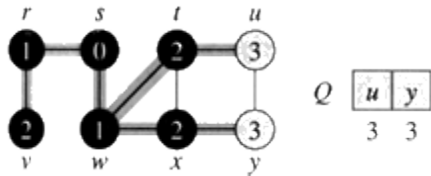
e.



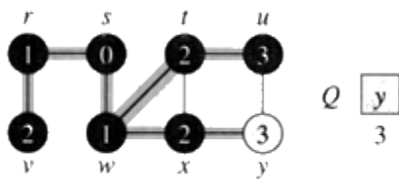
f.



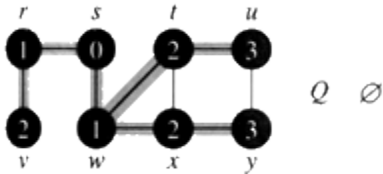
g. The algorithm discovers all vertices 3 edges from s i.e., discovered all vertices (u and y) at level 3.



h.



i. The algorithm terminates when every vertex has been fully explored.



Analysis

- The while-loop in breadth-first search is executed at most $|V|$ times. The reason is that every vertex is enqueued at most once. So, we have $O(V)$.
- The for-loop inside the while-loop is executed at most $|E|$ times if G is a directed graph or $2|E|$ times if G is undirected. The reason is that every vertex is dequeued at most once and we examine (u, v) only when u is dequeued. Therefore, every edge is examined at most once if directed, at most twice if undirected. So, we have $O(E)$.

Therefore, the total running time for breadth-first search traversal is $O(V + E)$.

Lemma 22.3 (CLRS) At any time during the execution of BFS suppose that Q contains the vertices $\{v_1, v_2, \dots, v_r\}$ with v_1 at the head and v_r at the tail. Then $d[v_1] \leq d[v_2] \leq \dots \leq d[v_r] \leq d[v_1] + 1$.

Let v be any vertex in $V[G]$. If v is reachable from s then let $\delta(s, v)$ be the minimum number of edges in $E[G]$ that must be traversed to go from vertex s to vertex v . If v is not reachable from s then let $\delta(s, v) = \infty$.

Theorem 22.5 (CLRS) If BFS is run on graph G from a source vertex s in $V[G]$ then for all v in $V[G]$, $d[v] = \delta(s, v)$ and if $v \neq s$ is reachable from s then one of the shortest paths from s to v is a shortest path from s to $\pi[v]$ followed by the edge from $\pi[v]$ to v .

BFS builds a tree called a breadth-first-tree containing all vertices reachable from s . The set of edges in the tree (called tree edges) contain $(\pi[v], v)$ for all v where $\pi[v] \neq \text{NIL}$.

If v is reachable from s then there is a unique path of tree edges from s to v . $\text{Print-Path}(G, s, v)$ prints the vertices along that path in $O(|V|)$ time.

$\text{Print-Path}(G, s, v)$

```

if  $v = s$ 
    then print  $s$ 
else if  $\pi[v] \leftarrow \text{NIL}$ 
    then print "no path exists from "  $s$  "to"  $v$ "
    else  $\text{Print-Path}(G, s, \pi[v])$ 
        print  $v$ 

```

Algorithms based on BFS

Based upon the BFS, there are $O(V + E)$ -time algorithms for the following problems:

- Testing whether graph is connected.

- Computing a spanning forest of graph.
- Computing, for every vertex in graph, a path with the minimum number of edges between start vertex and current vertex or reporting that no such path exists.
- Computing a cycle in graph or reporting that no such cycle exists.

4.6.3 Dijkstra's Algorithm

Dijkstra's shortest path algorithm

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

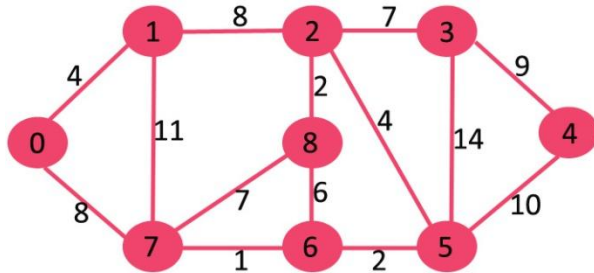
Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

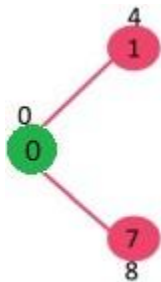
- 1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
- 2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
- 3) While *sptSet* doesn't include all vertices
 - ...a) Pick a vertex *u* which is not there in *sptSet* and has minimum distance value.
 - ...b) Include *u* to *sptSet*.
 - ...c) Update distance value of all adjacent vertices of *u*. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex *v*, if sum of distance value of *u* (from

source) and weight of edge $u-v$, is less than the distance value of v , then update the distance value of v .

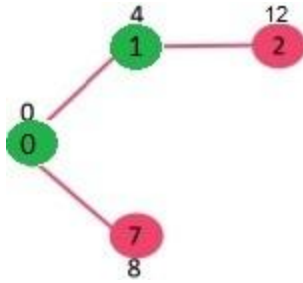
Let us understand with the following example:



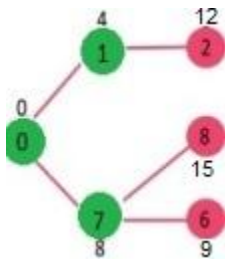
The set $sptSet$ is initially empty and distances assigned to vertices are $\{0, INF, INF, INF, INF, INF, INF, INF, INF\}$ where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in $sptSet$. So $sptSet$ becomes $\{0\}$. After including 0 to $sptSet$, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green color.



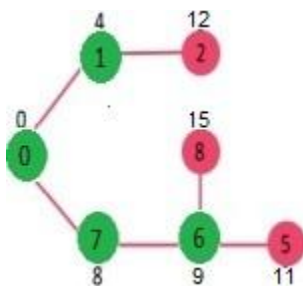
Pick the vertex with minimum distance value and not already included in SPT (not in $sptSet$). The vertex 1 is picked and added to $sptSet$. So $sptSet$ now becomes $\{0, 1\}$. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



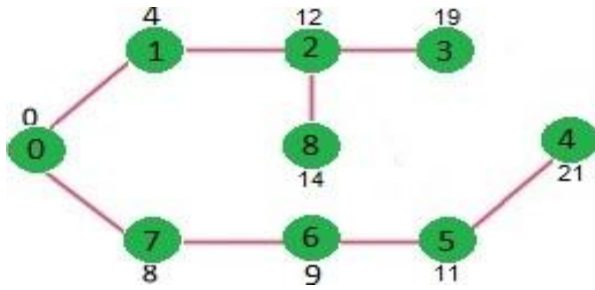
Pick the vertex with minimum distance value and not already included in SPT (not in *sptSet*). Vertex 7 is picked. So *sptSet* now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in *sptSet*). Vertex 6 is picked. So *sptSet* now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



How to implement the above algorithm?

We use a boolean array `sptSet[]` to represent the set of vertices included in SPT. If a value `sptSet[v]` is true, then vertex `v` is included in SPT, otherwise not. Array `dist[]` is used to store shortest distance values of all vertices.

// A C / C++ program for Dijkstra's single source shortest path algorithm.

// The program is for adjacency matrix representation of the graph

#include <stdio.h>

#include <limits.h>

// Number of vertices in the graph

#define V 9

// A utility function to find the vertex with minimum distance value, from

// the set of vertices not yet included in shortest path tree

int minDistance(int dist[], bool sptSet[])

{

// Initialize min value

int min = INT_MAX, min_index;

for (int v = 0; v < V; v++)

if (sptSet[v] == false && dist[v] <= min)

min = dist[v], min_index = v;

```

    return min_index;
}

// A utility function to print the constructed distance array
int printSolution(int dist[], int n)
{
    printf("Vertex  Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}

// Funtion that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // The output array. dist[i] will hold the shortest
                // distance from src to i

    bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest
                  // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V-1; count++)
    {

```

```

// Pick the minimum distance vertex from the set of vertices not
// yet processed. u is always equal to src in first iteration.
int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed
sptSet[u] = true;

// Update dist value of the adjacent vertices of the picked vertex.
for (int v = 0; v < V; v++)

// Update dist[v] only if is not in sptSet, there is an edge from
// u to v, and total weight of path from src to v through u is
// smaller than current value of dist[v]
if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
    && dist[u]+graph[u][v] < dist[v])
    dist[v] = dist[u] + graph[u][v];
}

// print the constructed distance array
printSolution(dist, V);
}

// driver program to test above function
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                       {4, 0, 8, 0, 0, 0, 0, 11, 0},
                       {0, 8, 0, 7, 0, 4, 0, 0, 2},
                       {0, 0, 7, 0, 9, 14, 0, 0, 0},
                       {0, 0, 0, 9, 0, 10, 0, 0, 0},

```



```

    {0, 0, 4, 0, 10, 0, 2, 0, 0},
    {0, 0, 0, 14, 0, 2, 0, 1, 6},
    {8, 11, 0, 0, 0, 0, 1, 0, 7},
    {0, 0, 2, 0, 0, 0, 6, 7, 0}
};

```

```

dijkstra(graph, 0);

```

```

return 0;

```

```

}

```

4.6.4 Kruskal's Algorithm

This minimum spanning tree algorithm was first described by Kruskal in 1956 in the same paper where he rediscovered Jarnik's algorithm. This algorithm was also rediscovered in 1957 by Loberman and Weinberger, but somehow avoided being renamed after them. The basic idea of the Kruskal's algorithms is as follows: scan all edges in increasing weight order; if an edge is safe, keep it (i.e. add it to the set A).

Overall Strategy

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

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

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).

- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

Data Structure

- **Make_SET(v):** Create a new set whose only member is pointed to by v . Note that for this operation v must already be in a set.
- **FIND_SET(v):** Returns a pointer to the set containing v .
- **UNION(u, v):** Unites the dynamic sets that contain u and v into a new set that is union of these two sets.

Algorithm

Start with an empty set A , and select at every stage the shortest edge that has not been chosen or rejected, regardless of where this edge is situated in the graph.

KRUSKAL(V, E, w)

$A \leftarrow \{ \}$ \triangleright Set A will ultimately contains the edges of the MST

for each vertex v in V

do MAKE-SET(v)

sort E into nondecreasing order by weight w

for each (u, v) taken from the sorted list

do if FIND-SET(u) = FIND-SET(v)

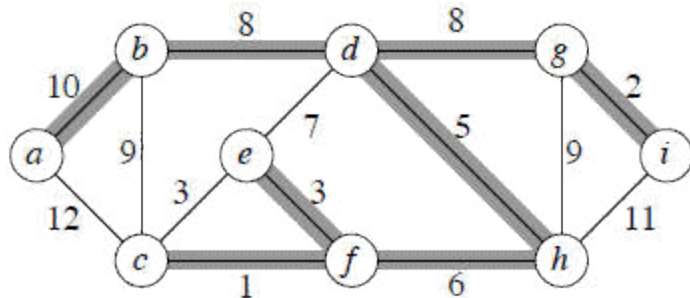
then $A \leftarrow A \cup \{(u, v)\}$

 UNION(u, v)

return A

Illustrative Examples

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



We get the shaded edges shown in the above figure.

Edge (c, f) : safe

Edge (g, i) : safe

Edge (e, f) : safe

Edge (c, e) : reject

Edge (d, h) : safe

Edge (f, h) : safe

Edge (e, d) : reject

Edge (b, d) : safe

Edge (d, g) : safe

Edge (b, c) : reject

Edge (g, h) : reject

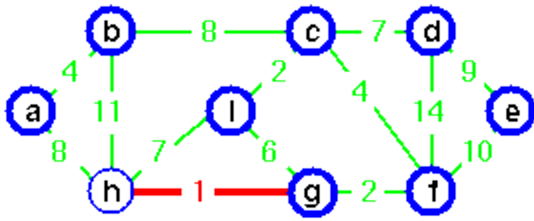
Edge (a, b) : safe

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

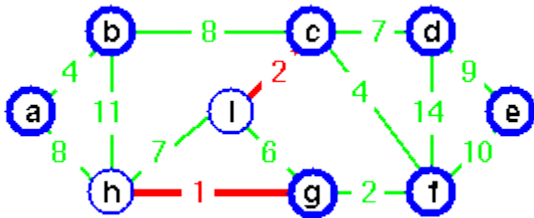
Note Carefully: Suppose we had examined (c, e) before (e, f). Then would have found (c, e) safe and would have rejected (e, f).

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

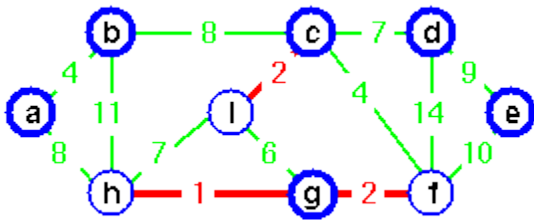
Step 1. In the graph, the Edge(g, h) is shortest. Either vertex g or vertex h could be representative. Lets choose vertex g arbitrarily.



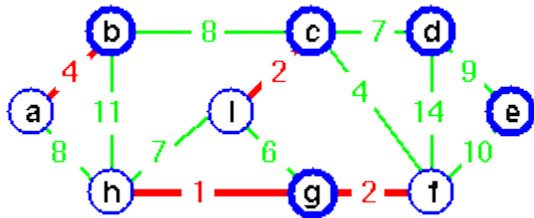
Step 2. The edge (c, i) creates the second tree. Choose vertex c as representative for second tree.



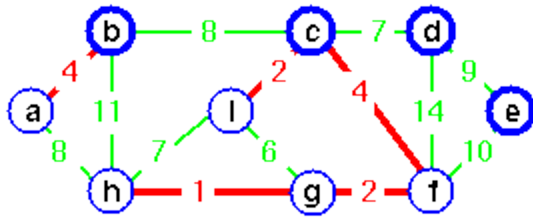
Step 3. Edge (g, f) is the next shortest edge. Add this edge and choose vertex g as representative.



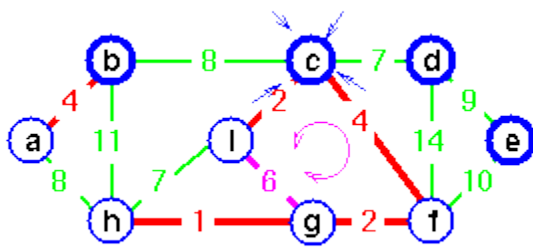
Step 4. Edge (a, b) creates a third tree.



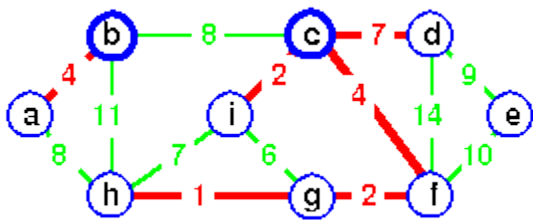
Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.



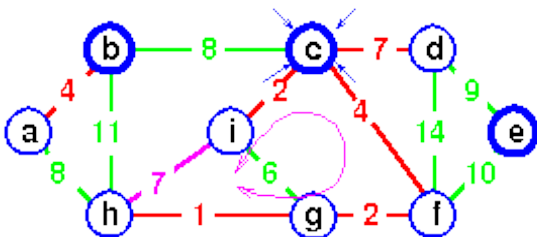
Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.



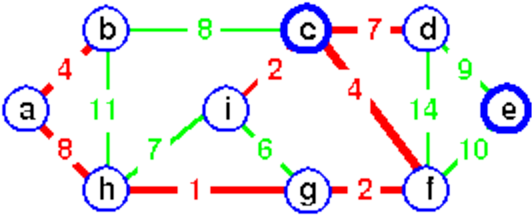
Step 7. Instead, add edge (c, d).



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



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



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

