MTH 307/417/515: Programming and data structures
Semester 2, 2017-18

April 28, 2018

Contents

1 C Programming Basics

The notes in this section are based on the books [7] and [9].

1.1 Basic structure

Example 1.

```
#include <stdio.h>
/*A program that prints the words "hello, world".*/
main()
{
    printf("hello, world\n");
}
```

1. `#include <stdio.h>` tells the compiler to include information about the standard input/output library.
2. `main()` is the function that calls all other functions. The function `main()` does not expect any arguments or return values.
3. The `printf` function prints a string of characters enclosed within quotations. `\n` is the newline character, which is also known as an *escape sequence*.
4. Anything enclosed within `/*...*/` is a comment that is not evaluated by the compiler.

1.2 Variables and expressions

Example 2.

```
#include <stdio.h>
/* print Fahrenheit-Celsius table
for fahr = 0, 20, ..., 300 */
main()
{
    int fahr, celsius;
    int lower, upper, step;
    lower = 0; /* lower limit of temperature scale */
    upper = 300; /* upper limit */
    step = 20; /* step size */
    fahr = lower;
    while (fahr <= upper)
    {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

1. `int fahr..` is a *declaration* that announces properties of the variable, which include typename (`int`) followed by a list of variables (`fahr, celsius` etc.). The general syntax for a variable declaration is:

   ```
   Typename var1, var2,..., varn;
   ```

   The commonly used variable typenames are:

Typename	Description	Range	Memory
`char`	Character	-128 to 127 or 0 to 255	1 byte
`int`	Integer	-32,768 to 32,767	2 bytes
`long`	Long integer	-2,147,483,648 to 2,147,483,647	4 bytes
`float`	Floating point	1.2E-38 to 3.4E+38 (6 dec. places)	4 bytes
`double`	Floating point	2.3E-308 to 1.7E+308 (15 dec. places)	8 bytes

2. Statements of the form `lower = 0;` are called *assignment statements*. Assignment statements are used to set initial values to variables. They have the general syntax:

```
var = value;
```

3. The mathematical expression on the right of the statement `celsius = 5*(fahr-32)/9;` will yield an integer value, which is then stored to the variable `celsius`. In general, if an arithmetic operator has integer operands, then an integer operation is performed. However, if an operator has at least one floating point operand, then a floating point operation is performed. Alternatively, one could declare `celsius` as a floating point variable and replace the arithmetic expression with `celsius = 5*(fahr-32)/9.0;`.

1.3 Formatted output and input

1.3.1 The `printf` function.

In `C`, the `printf` function (which is predefined in `stdio.h`) is used for formatted output. Any call of the `printf` function has the following general format:

```
printf("format string", expr1, expr2, ...);
```

The *format string* may consist of simple characters and *conversion specifications*, which begin with the character `%`. Each conversion specification in the format string is paired with a corresponding expression (which could also be a variable). In Example 2, the `\t` in the statement

```
printf("%d \t %d\n", fahr, celsius);
```

ensures that each pair of values of `fahr` and `celsius` is printed with a tab space between, and the `\n` ensures that successive pairs are printed one below the

other. The codes `\t` and `\n` used in `printf` format strings are called *escape sequences*, which enable strings to contain characters that would otherwise cause problems for the compiler. For the floating point version, a suitable printing option would be:

```
printf("%d \t %6.1f\n", fahr, celsius);
```

Some commonly used conversion specifications are listed below:

Conversion specification	**Printing format**
`%kd`	Integer value right justified in a field of `k` characters.
`%-kd`	Integer value left justified in a field of `k` characters.
`%k.pf`	Float value right justified in a field of `k` characters overall with `p` digits after the decimal.
`%-k.pf`	Float value left justified in a field of `k` characters overall with `p` digits after the decimal.

1.3.2 The `scanf` function.

The `scanf` function (also predefined in `stdio.h`) is used for formatted input, and its call has the following general syntax:

```
scanf("format string", & var1, & var2,..., & varn);
```

For each conversion specification in the format string, `scanf` tries to find an item of the appropriate type in the input data, skipping *white space characters* (spaces, horizontal and vertical tabs, form-feeds, and new-line character). It repeatedly reads white space characters (the number of white space characters is irrelevant), until it reaches a non-white space character (in the format string), which it then compares with the next input character. If the character matches, it discards the input character and continues to process the format string. We will now give an example for a `scanf` function call.

Example 3.

```
scanf("%d %d %f %f", &i, &j, &x, &y);
/* Input:
                1
                _ 20 _ _ _ .3
                _  _ _ _ _ _ -4.0e3 */
/* scanf stores  1 to i, 20 to j, .3  to x,
   and -4.0e3 to y */
```

When scanning an integer, `scanf` first searches for a digit, a + sign, or a - sign, reads digits until it reaches the first nondigit. When reading a floating point, `scanf` looks for + or - sign, followed by a series of digits (possibly containing decimal), followed by an optional exponent (`e` or `E`), followed by an optional sign, and one or more digits.

1.4 Symbolic constants

In *C*, the `#define` line is used to define a *symbolic name or a symbolic constant* to be a particular string of characters. A symbolic declaration has the general format:

```
#define name replacement text
```

Here, `name` refers to a variable name, and the `replacement text` could be any sequence of characters, including numbers. Once declared, the value of a symbolic constant is accessible globally (i.e. anywhere in the program) and more importantly, its value cannot be altered during the course of the program. Hence, a symbolic constant cannot be declared like a variable. We will now provide an alternative for Example 2 using symbolic constants.

Example 4.

```
#define LOWER 0 /* lower limit of table */
#define UPPER 300 /* upper limit */
#define STEP 20 /* step size */
/* print Fahrenheit-Celsius table */
main()
{
    int fahr;
    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Here `LOWER`, `UPPER` and `STEP` are symbolic constants.

1.5 Character input and output

In *C*, the predefined functions `getchar` and `putchar` (included in `stdio.h`) can be used for reading and writing (resp.) one character at a time. The `getchar` reads the next character that is inputted (using a keyboard or other input device) and returns its value. It has the general format:

```
c = getchar()
```

Here, the next input character is read and stored to the variable `c`. The `puchar` function prints a character each time it is called, and has the syntax:

```
putchar(c)
```

In both cases, `c` could be declared as an `int` or a `char` variable. The following example, illustrates the use of these functions.

Example 5.

```
#include <stdio.h>
/* copy input to output*/
main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

The function `getchar` returns a distinctive value when there is no more input (that cannot be confused with any real character) called `EOF`. `EOF` is a integer defined in `stdio.h`.

1.6 Operators

1.6.1 Assignment operators

(a) **Basic assignment operator.** In `C`, the basic assignment operator is =, which is used for storing the value of an expression to a variable. It has the general format:

```
var_name = expression
```

The following is basic example for the use of = operator, as we have already seen.

Example 6.

```
int i;
float f;
i = 72.99; /* i = 72 */
f = 136; /* j = 136.0 */
```

Several = operators can be chained together in a single statement, and such statements are evaluated from right to left (i.e = is right associative). Consider the expressions in the following examples.

Example 7.

```
i = j = k = 0; /* Same as i = (j = (k = 0)))*/
```

Example 8.

```
i = 1;
k = 1 + (j = i);
printf("%d %d %d\n",i,j,k); /* Prints 1 1 2 */
```

(b) **Increment and decrement operators.** Increment/decrement operators have the general format

```
var_name++ (or) var_name--
++var_name (or) --var_name
```

In general, the ++ operator is used to increment the value of a variable by 1, while the - is used to increment the value of a variable by 1. The first two operators above are called post increment (and decrement) operators (resp.), and the last two operators are called pre-increment (and pre-decrement) operators (resp.) These operators can have different meanings depending on whether they appear before or after a variable name. We will now illustrate the meaning of these operators using the following example.

Example 9.

```
i = 1;
j = 1;
printf("%d", i++); /*  Prints 1 */
printf("%d", --j); /*  Prints 0 */
```

In the example above, the first `printf` statement means

```
printf("%d", i);
i = i + 1;
```

while the second `printf` statement means

```
j = j - 1;
printf("%d", j);
```

(c) **Compound assignment operator.** The compound assignment operator has the general format

```
var_name op= expression,
```

where '`op`' refers to one of the binary operators: `+, -, *, /, %`. The statement above has the same meaning as:

```
var_name = var_name op expression
```

For example,

Example 10.

```
i += 1; /* means i = i + 1 */
i /= 2; /* means i = i / 2 */
```

Compound assignment operators are also right associative and possess the same properties as the = operator.

1.6.2 Arithmetic operators

The binary arithmetic operators in *C* are:

Operators	Meaning
+	+
-	−
*	×
/	÷
%	mod

The % cannot be applied to `float` or `double` data types. All arithmetic operators are evaluated from left to right, and their order of precedence is:

Operators	Precedence
Unary + and -	1
Binary *, /, %	2
Binary +, -	3

Assignment and arithmetic operators together follow the following rules of precedence.

Precedence	Operators	Associativity
1	a++, a--	Left
2	++a, --a, +a, -a	Right
3	*, /, %	Left
4	+, -	Left
5	=, op=	Right

We illustrate this precedence using the following examples.

Example 11.

```
i = 1; j = 1; k = 2;
i += j += k; /* Means i += (j += k) */
printf("%d %d %d",i,j,k); /* Prints 4 3 2 */
```

Example 12.

```
a = b += c++ - d + --e / -f;
/* Means a = (b += (((c++) - d) + ((--e) / (-f)))) */
```

1.6.3 Relational and logical operators

The *relational operators* in C are:

Operators	Meaning
> and >=	> and ≥
< and <=	< and ≤
==	Equal to
!=	Not equal to

The *logical operators* in C are:

Operators	Meaning
&&	and
||	or

Relational and logical operators are evaluated from left to right, and have lower precedence than arithmetic operators. For example, here is set of conditional statements for determining whether a given year is a leap year, or not.

Example 13.

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
printf("%d is a leap year\n", year);
else
printf("%d is not a leap year\n", year);
```

1.7 Loops

1.7.1 The `while` loop

A **while** loop has the general form:

```
while(loop condition}
{
    Body of the loop
}
```

The body of a `while` loop may comprise either one statement (in which case the curly brackets are redundant) or more generally, collection of statements. The compiler keeps executing the body of the loop while the loop condition

remains satisfied. The following program determines whether a given number is prime or composite.

Example 14.

```
#include<stdio.h>
void main()
{
    int i,n,flag=1;
    printf("\n Enter a positive integer : ");
    scanf("%d",&n);
    i = 2;
    while(i <= n/2)
    {
        if (n%i == 0)
        {
            flag = 0;
            break; /* This will exit the loop */
        }
        i = i+1;
    }
    if (flag == 1) printf("\n The number %d is prime",n);
    else printf("\n The number %d is composite",n);
}
```

1.7.2 The `for` loop

A **for** loop has the general form:

```
for(initialization; loop condition; increment)
{
    Body of the loop
}
```

As in case of the `while` loop, here too the compiler keeps executing the body of the loop, as long as the loop condition remains satisfied.

The following example gives an alternative for Example 2 can using the `for` loop.

Example 15.

```
#include <stdio.h>
/* print Fahrenheit-Celsius table */
main()
{
    int fahr;
    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

1.7.3 The `do-while` loop

The `do-while` loop has the general form:

```
 do
 {
     ---
     Body of the loop
     ---
}while(loop condition);
```

The `do-while` loop (unlike the `while` loop) is a post-evaluation loop in the which the loop condition is checked after evaluating the body of the loop at least once. We will now given an alternative for Example 6 using the `do-while` loop.

Example 16.

```
#include<stdio.h>
void main()
{
    int i,n,flag=1;
    printf("\n Enter a positive integer : ");
    scanf("%d",&n);
    i = 2;
    do
    {
        if (n%i == 0)
        {
            flag = 0;
            break; /* This will exit the loop */
        }
        i = i+1;
    }while(i <= n/2);
    if (flag == 1) printf("\n The number %d is prime",n);
    else printf("\n The number %d is composite",n);
}
```

In the example above, if the user enters 2 for `n`, then the loop body will execute once, even though the loop condition is violated.

1.7.4 The `break` and `continue` statements

The `break` statements transfers control just past the end of a loop (or a `switch` sequence), while the `continue` statement transfers control to a point just before the end of the loop body. The `continue` statement can be used only with loops (i.e `for`, `while` and `do-while`). The following example illustrates how the break statement can be effectively used in a loop.

Example 17.

```
/* Determines whether a given number is prime or composite */
#include<stdio.h>
void main()
{
    int n,d,flag=1;
    printf("\nEnter a number : ");
    scanf("%d",&n);
    for(d=2; d<= n/2; d++)
    {
        if (n%d == 0)
        {
            flag = 0;
            print("\n%d is a composite number",n);
            break;
        }
        /* continue would transfer control here */
   }
   /* break transfers control here */
   if (flag==1) printf("\n%d is a prime number",n);
}
```

1.8 Arrays

1.8.1 One-dimensional arrays

An **array** is a data structure containing a number of data values, all of which have the same type. A typical array declaration has the format:

`Typename varname[k];`

Array subscripts always start with 0 in *C*. For example, consider the following program:

Example 18.

```
#include <stdio.h>
/* count digits, white space, others */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];
    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
    ndigit[i] = 0;
    while ((c = getchar()) != EOF)
    {
        if (c >= '0' && c <= '9')
        ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
        ++nwhite;
        else
        ++nother;
    }
printf("digits =");
for (i = 0; i < 10; ++i)
printf(" %d", ndigit[i]);
printf(", white space = %d, other = %d\n",
nwhite, nother);
```

Here, the elements of the integer array `ndigit[10]` are:

`ndigit[0], ..., ndigit[9].`

1.8.2 Multidimensional arrays

An array can have multiple dimensions in C. In the following example, we declare a 2-dimensional integer array.

Example 19.

```
int a[4][5]; /* a has 20 elements - 4 rows and 5 columns */
```

Multidimensional arrays are stored sequentially in row-major order i.e row 1 followed by row 2, and so on. Accessing the element in the i^{th} row and j^{th} column would use the subscripts: `a[i-1][j-1]`.

Nested `for` loops are ideal for processing multi-dimensional arrays. In the following example, we write a function that created an $n \times n$ identity matrix.

Example 20.

```
#define N 10
void identity_mat(int a[N][N}, int n)
{
    int i,j;
    for(i = 0; i< N; i++)
    {
        for(j = 0; j< N; j++)
        {
            if (i == j) a[i][j] = 1;
            else a[i][j] = 0;
        }
    }
}
/* Function call:  identity_mat(b, N) */
```

Multidimensional arrays can also be initialized by nesting one-dimensional initializers, for example:

```
a[3][3] = {{1,0,0},{0,1,0},{0,0,1}};
```

1.9 Conditional statements

1.9.1 The `if else` statement

A general sequence of `if`,`else if`,and `else` statements has the following format:

```
if (condition 1)
{
    Statement block 1
```

```
}
else if (condition 2)
{
    statement block 2
}
...
...
else
{
    statement block n
}
```

It is not mandatory to pair an `if` statement with an `else if` (or `else`) statement, but a standalone `else` statement is illegal. In the above format, the statement block i ($1 \leq i < n$) is executed only if the corresponding condition i is satisfied. The last `else` statement is called a *default* `else` statement, whose associated statement block is executed only if conditions $1, \ldots, n-1$ are violated (see Example 4 above for such a sequence).

1.9.2 The `switch` statement

The `switch` statement is a multi-way decision statement that tests whether an expression matches one of a number of constant integer values, and branches accordingly. Each case is labeled by one or more integer-valued constants or constant expressions. The general syntax of a `switch` statement is as follows:

```
switch(expression)
{
     case const-expr1: statements
     case const-expr2: statements
     .
     .
     .
     case const-exprn: statements
     default: statements; /* optional*/
}
```

When the `const-expri` matches the `expression` value, then the corresponding statements that follow it after `:` are executed. The statements that follow

the `default` case (which is optional) are executed if none of the other cases are satisfied. The following example gives an alternative for Example 7 using the `switch` statement.

Example 21.

```
#include <stdio.h>
main() /* count digits, white space, others */
{
    int c, i, nwhite, nother, ndigit[10];
    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
        ndigit[i] = 0;
        while ((c = getchar()) != EOF) {
        switch (c)
        {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8':
            case '9': ndigit[c-'0']++;
            break;
            case ' ': case '\n':case '\t': nwhite++;
            break;
            default: nother++;
            break;
        }
}
printf("digits =");
for (i = 0; i < 10; i++)
    printf(" %d", ndigit[i]);
printf(", white space = %d, other = %d\n",nwhite, nother);
return 0;
}
```

1.9.3 The conditional operator

The *conditional operator* consists of two symbols (`?` and `:`) and three operands. It has the general format

```
test condition ?  expr1 :  expr 2
```

The statement above is equivalent to the `if` sequence:

```
if (test condition)
    expr1;
else
    expr 2;
```

The following example illustrates the use of the conditional operator:

Example 22.

```
int i,j,k;
j = 1;
k = 2;
k = i > j ? i : j; /*k is now 2*/
k = (i  >= 0 ? i : 0) + j; /*k is now 3*/
```

1.10 Functions

In C, a function (which is predefined or user-defined) provides a convenient way to encapsulate a collection of computations (or more generally, tasks) that may be required to be executed at multiple parts of the program. (For example, `printf, getchar and putchar` are predefined functions in C that are defined in the library `stdio.h`.) Once an appropriate function is defined by the user, it could then be repeatedly invoked from any part of the program that might require these tasks to be executed. The general format of a user-defined function is:

```
return-typename function-name(parameter declarations, if any)
{
    local declarations
    statements
}
```

It is not mandatory for a function in C to either have parameters or a return data type (or typename). However, a function has no return data type, it is a good practice to designate the function return type as `void`, that is:

```
void function-name(parameter declarations)
```

In general, it is a good practice to define all functions before the `main` function appears in the program. However, if a function is defined after the `main` function, it will require a *prototype declaration*, that precedes the `main` function, which has the general form:

```
return-typename function-name(parameter declarations);
```

To invoke (or call) a user-defined function (anywhere in the program), we will use the format:

```
function-name(arguments);
```

While calling a function, care should be taken to ensure that the number of arguments and their data types match the parameter declarations in its definition. Consider the following example.

Example 23.

```
#include <stdio.h>
    int power(int m, int n);
/* test power function */
main()
{
    int i;
    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}
/* power: raise base to n-th power; n >= 0 */
int power(int base, int n)
{
    int i, p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Here, the user-defined function `power` has two integer parameters m and n, and returns an integer m^n when called. For example, the function call `power(2,i)` in the `main()` function will return the value 2^i. Also, note that as `power` is defined after `main`, there is a prototype declaration for it right before `main` (i.e `int power(int m, int n);`).

2 Advanced C programming

2.1 Pointers

The memory address of variables in `C` are stored in special variables called *pointers*.

2.1.1 Pointer declaration

Pointer variables can be declared using the *indirection operator* `*`. A pointer declaration has the general format:

```
Typename *var1, *var2, . . ., *vark;
```

Example 1.

```
int *p, i; /* p points to an integer's address. */
```

2.1.2 Indecision operator and pointer initialization

Pointer variables can be initialized using the `&` operator. A pointer declaration has the general format:

```
pvar = &var;
```

Example 2.

```
int *p, i;
p = &i; /* p points to the address of i. */
```

Pointer can also be initialized during declaration.

Example 3.

```
int *p = &i;
```

The indirection operator can be used to access what is stored in an address a pointer is pointing to. For example:

Example 4.

```
int *p, i = 2;
p = &i;
printf("%d", *p); /* Prints 2. */
```

The indirection operator `*` can also be thought of as an "inverse" of the operator `&`. For example:

Example 5.

```
j = *&i; /* Same as j = i. */
```

Moreover, if a pointer `p` points to a variable `i`, then `*p` is an alias for `i`. In particular, changing the value of `*p` also changes the value of `i`. For example:

Example 6.

```
p = &i;
i = 1;
printf("%d\n", *p); /* Prints 1. */
*p = 2;
printf("%d\n", i); /* Prints 2. */
```

The value of pointer variable can also be assigned to another pointer variable. For example:

Example 7.

```
int i, *p, *q;
p = &i;
q = p;
*p = 1; /* i = 1 and p, q both point to i */
*q = 2; /* i = 2 and p, q both point to i */
```

Caution: The statements `q = p` and `*q = *p` have different meanings. For example:

Example 8.

```
int i, j, *p, *q;
p = &i;
q = &j;
i = 1;
*q = *p;  /* i = j = 1 and p points to i, and q to j */
/*Here, q = p would mean both p,q point to i. */
```

2.1.3 Pointers and arrays

A pointer `p` can be made to point to the address of the i^{th} element of an array `a` using the following initialization:

```
p = &a[i];
```

Example 9.

```
int a[5], p;
p = &a[0]; /* p points to address of a[0].*/
*p = 5; /* Same as a[0] = 5. */
```

If a pointer `p` points to `a[i]` (i.e. `p = & a[i]`,) then for a positive integer `k`, `p ± k` would point to the `a[i ± k]` (provided that an $(i \pm k)^{th}$ element of the array `a` exists. For example:

Example 10.

```
int a[10], *p, *q;
p = &a[3]; /* p points to address of a[3]. */
q = p + 5; /* q points to address of a[8]. */
p = p - 2; /* p points to address of a[1]. */
```

Two pointers pointing to different elements of the same array can be compared using the operators: `<`, `<=`, `>`, `>=`, `==`, and `!=`. For example:

Example 11.

```
p = &a[5];
q = &a[1];
printf("%d %d", p <= q, p >= q); /* Prints 0 1 */
```

These properties of pointers can also be exploited for array processing. For example:

Example 12.

```
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
    sum += p;
printf("%d", sum); /* Prints the sum of elements of a. */
```

The increment and decrement operators (`++` and `--`) can also be combined with the indirection operator. For example:

Example 13.

```
p = &a[i];
*p++ = j; /* Same as a[i++] = j. */
```

Here, the post increment operator has higher precedence than `*`.

Example 14.

```
p = &a[i];
*p++ = j; /* Same as *(p++) = j. */
```

The following table gives some variants of the expression above and their meaning.

Expression	Meaning
`*p++ or *(p++)`	Value is `*p` before increment; Increments `p` later
`(*p)++`	Value is `*p` before increment; Increments `*p` later
`*++p or *(++p)`	Incerement `p` first; Value is `*p` after increment.
`++*p or ++(*p)`	Increment `*p` first; Value is `*p` after increment.

The meanings are analogous for the corresponding expressions with $--$.

2.1.4 Array names and array arguments

The name of an array can be used as a pointer to the first element of the array (without declaring a separate pointer variable). For example:

Example 15.

```
int a[10];
*a = 7; /* Same as a[0] = 7 */
*(a+1) = 12; /* Same as a[1] = 12 */
```

An alternative for Example 12 could be:

Example 16.

```
sum = 0;
for (p = a; p < a + N; p++)
    sum += *p;
printf("%d", sum); /* Prints the sum of elements of a. */
```

In general, `a+i` points to the same address as `a[i]`, while `*(a+i)` is the same as `a[i]`.

When an ordinary variable is passed into a function, its value is copied, and any changes to the corresponding parameter (within the function) does not change the value of the variable. In contrast, when an array is used as an argument, it is not protected against change, since no copy is made of the array itself. In fact, when an array is passed it is treated simply as an pointer. Consider the following function that computes the largest element of a an array.

Example 17.

```
int find_largest(int a[], int n)
{
    int i, max;
    max = a[0];
    for (i = 0; i < n; i++)
        if (a[i] >  max) max = a[i];
    return max;
}
/* Function call: largest = find_largest(b, N); */
```

The function call in the example above causes a pointer to the first element of the array `b` to be assigned to `a`. However, the array `b` itself is not copied to `a`. The following example illustrates how an array argument is not protected against change.

Example 18.

```
void zero_mat(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
         a[i] = 0;
}
/* Function call: zero_mat(b, N); */
```

The function call in the example above stores `0` into every element of the array `b`.

2.1.5 Pointers as arguments

The C programming languages passes most arguments by value. So, a variable supplied as an argument is protected against change. This limitation can be circumvented by passing a pointer to the variable, instead of the variable itself. For example:

Example 19.

```
void decompose(double x, long *int_part, double *frac_part)
{
    *int_part = (long)x; /*
    *frac_part = x - *int_part;
}
/* Function call: decompose(3.14159, &i, &d)*/
```

The function call in the example above makes `int_part` point to `i` and `frac_part to d`, and stores value `3 to i` and `.14159 to d`.

In C, there are also predefined functions, such as `scanf` function, which need pointer arguments. For example:

Example 20.

```
int i, *p;
scanf("%d", &i); /* scanf is supplied with pointer to i. */
p = &i;
scanf("%d", p); /* Reads p and stores to i. */
```

The following program, which prints the largest and smallest elements of an integer array uses pointers as arguments.

Example 21.

```
#include <stdio.h>
# define N 10
void max_min(int a[], int *max, int *min);
int main(void)
{
    int b[N], i, big, small;
    printf("\n Enter %d numbers : ", N);
    for(i = 0; i < N; i++)
        scanf("%d", &b[i]);
    max_min(b, N, &big, &small);
    printf("\n Largest: %d", big);
    printf("\n Smallest: %d", small);
    return 0;
}
void max_min(int a[], int *max, int *min)
{
    int i;
    *max = *min = a[0];
    for (i = 1; i < N; i++)
    {
        if (a[i] > max)
            max = a[i];
        else if (a[i] < min)
            min = a[i];
     }
}
```

2.1.6 The `const` data type

The `const` data type can be used to ensure that a function does not change the value of an object whose address is passed into the function. It has the general format:

```
const typename var = value
```

This declaration documents that the program will not modify the value of the variable `var`. For example:

Example 22.

```
void func(const int *p); /* Parameter declaration */
/* func can't change the integer p points to */
void func(const int *p)
{
    int j;
    *p = 0; /* wrong */
    p = &j; /* legal */
}
```

2.2 Structures

In C, structures are used for storing a collection of related data items.

2.2.1 Structure declaration

The general syntax of a structure declaration is as follows:

```
 struct tagname /* tagname is optional */
 {
     typename1 mem_var1; /* Member variable 1 */
     typename2 mem_var2;
     ...
     ...
     ...
     typenamek mem_vark; /* Member variable k */
}str_var1, str_var2, ..., str_varn; /*The structure variables */
```

In following example, we define a structure that stores the details of each spare part in a workshop.

Example 23.

```
struct
{
    int number; /* Part number.*/
    char name[LEN+1]; /* Name of part. */
    int on_hand; /* Available number. */
} part1, part2;
```

In the example above, the *structure variables* are `part1 and part2`. Each structure structure variable has three *members*: `number`, `name`, and `part`. The members of a structure are stored in memory sequentially in the order of declaration.

Each structure represents a new scope (or a separate *name space*) for its members. Any names declared within that scope will not conflict with other names in the program. For example, one can declare the following structure (in which two members names are the same) right the structure declared in Example 23:

Example 24.

```
struct
{
    int number; /* Employee number.*/
    char name[LEN+1]; /* Name of employee. */
    int age; /* Age of employee. */
} emp1, emp2;
```

2.2.2 Initializing structure variables

Structure variables (like other variables) can also be initialized at the time of declaration, for example:

Example 25.

```
struct
{
    int number; /* Part number */
    char name[LEN+1]; /* Name of part */
    int on_hand; /* Available number */
} part1 = {528, "Tire", 10},  part2 = {400, "light", 20};
```

A member variable `mem_var` within a structure can only be accessed one of its structure variables `str_var` using an expression of the form:

```
str_var.mem_var
```

For example, an alternative way of initializing the structure variable `part1` in Example 25 is:

Example 26.

```
struct
{
    int number; /* Part number */
    char name[LEN+1]; /* Name of part */
    int on_hand; /* Available number */
} part1;
part1.number = 528;
part1.name = "Tire";
part1.on_hand = 10;
```

We could also accept values for the members variables of a structure (from the user) using the `scanf` function. For example:

Example 27.

```
struct
{
    int number; /* Part number */
    char name[LEN+1]; /* Name of part */
    int on_hand; /* Available number */
} part1;
scanf("%d", &part1.number);
scanf("%s", part1.name);
scanf("%d", part1.on_hand);
```

One could also use assignment operators to assign values to the member variables of a structure, for example:

Example 28.

```
struct
{
    int number; /* Part number */
    char name[LEN+1]; /* Name of part */
    int on_hand; /* Available number */
} part1;
part1.number = 528;
part1.on_hand = 10;
part1.on_hand++;
```

In fact, structures can also be copied (as a whole) using the standard assignment operator =, for example:

Example 29.

```
struct
{
    int number; /* Part number */
    char name[LEN+1]; /* Name of part */
    int on_hand; /* Available number */
} part1 = {528, "Tire", 10},  part2;
part2 = part1; /* Copies values of part2 to part1 */
```

Since arrays in C cannot be copied this way (using the = operator), structures provide a viable alternative for copying arrays. For example:

Example 30.

```
struct
{
    int a[10];
} a1, a2;
a1 = a2; /* Copies a2 to a1*/
```

2.2.3 The structure tag and type

A *structure tag* is a name used to identify a particular kind of structure. For example:

Example 31.

```
struct part /* A structure tag named part */
{
    int number;
    char name[LEN+1];
    int on_hand;
} ; /* Semicolon required here to end declaration */
```

Once a structure tag is created, it can be use to declare structure variables, for example:

Example 32.

```
struct part /* A structure tag named part */
{
    int number;
    char name[LEN+1];
    int on_hand;
} ;
struct part part1; /* Declaration of a structure variable */
```

Structure tags are not recognized unless they are preceded by the word `struct`. For example, the following declaration is illegal.

Example 33.

```
part part1, part2; /* Wrong */
```

The declaration of a structure tag cam also be combined with the declaration of structure variables, for example:

Example 34.

```
struct part /* A structure tag named part */
{
    int number;
    char name[LEN+1];
    int on_hand;
} part1, part2;
```

Moreover, all structure variables declared to have the same type are compatible with each other, for example:

Example 35.

```
struct part
{
    int number;
    char name[LEN+1];
    int on_hand;
};
struct part part1 = {528, "Tire, 10}, part2;
part2 = part1;
```

As an alternative to creating a structure tag name, we can use `typedef` to define a genuine structure type name. For example:

Example 36.

```
typedef struct
{
    int number;
    char name[LEN+1];
    int on_hand;
}Part;
Part part1 = {528, "Tire, 10}, part2;
part2 = part1;
```

Here, `Part` is a `typedef` name, so using `struct Part` is illegal.

2.2.4 Structures as arguments and return values

Functions can have structures as arguments and return values. Foe example:

Example 37.

```
struct part
{
    int number;
    char name[LEN+1];
    int on_hand;
};
struct part part1 = {528, "Tire, 10};
void print_part(struct part p)
{
    printf("\nPart number: %d", p.number);
    printf("\nPart name: %s", p.name);
    printf("\nPart quantity: %d", p.on_hand);
}
/* Function call:  print_part(part1); */
```

The following function returns a `part` structure that it constructs from its arguments.

Example 38.

```
struct part build_part(int number, const char *name, int on_hand)
{
    struct part p;
    p.number = number;
    strcpy(p.name, name);
/* strcpy(s1,s2) copies s2 to s1; needs library string.h */
    p.on_hand = on_hand;
    return p;
}
/* Function call: part1 = build_part(528, "Tire", 10); */
```

Caution. Passing and returning structures (particularly large structures) can impose a fair amount of overhead on the program. To avoid this, it is sometimes advisable to pass a pointer to a structure rather than the structure itself.

2.2.5 Nested structures

We can declare a structure variable of one type as a member of another structure. This is called *nesting*.

Example 39.

```
struct person_name
{
    char first_name[LEN+1];
    char initial;
    char last_name[LEN+1];
}
struct student
{
    struct person_name name;
    int id, age;
    char sex;
} student1, student2;
```

If one needs to access, the first name of `student1`, they will have to use `student1.name.first_name`.

2.2.6 Arrays of structures

It is often useful to declare an array whose elements are structures. An array of this kind can serve as a simple database. Building on our earlier example, we could declare a `student` array having 100 elements:

Example 40.

```
struct student
{
    struct person_name name;
    int id, age;
    char sex;
} student[100];
```

Initialization of an array of structures is analogous to the initialization of a multi-dimensional array. Consider the following example.

Example 41.

```
struct dialing_code
{
    char *country_name;
    int code;
};
const struct dialing_code codes = {{"India", 91}, {"USA", 1}};
```

3 Data structures

The notes and examples in this section are based on the books [5] and [7], and the lecture notes available at [1], [2], and [3].

3.1 Abstract data types (ADTs)

In a programming language, the *data type* of a variable is the set of values it may assume. For example, in `C`, we have the data types `int float, char etc.` An *abstract data type (ADT)* is a mathematical model , together with various operations defined on that model. We can write algorithms in terms of ADTs, but ti implement an algorithm in a given programming language, one must find some way of representing the ADTs in terms of the data types and operations supported by the language.

To represent the mathematical model underlying an ADT, we use *data structures*, which are collections of variables, possibly of different data types connected in various ways. The basic building block of a data structure is called a *cell*, which is usually pictures as a box capable of holding a value. Data structures are created by giving names to aggregates of cells and interpreting the values of some cells as representing connections (eg. pointers) among cells. The simplest aggregating mechanism in most programming languages in an *array* (for eg. `int a[10] in C`), which is simply a sequence of cells of a given type, often referred to as a *sequence*.

3.2 Basic ADTs and their implementation

3.2.1 Lists and linked lists

Abstractly speaking, a *list* is a sequence of elements of a given type written as $a_1, a_2 \ldots, a_n$, where $n \geq 0$. The number n is called the *length* of a list, and if $n = 0$, the list said to be *empty*. Lists are flexible, as they can grow and shrink on demand, and elements can be accessed, inserted, or deleted at any position within a list. Lists can also be concatenated or split into *sublists*. The elements of a list are linearly ordered according to their positions in the list. Below is a representative set of abstract list operations (Please note that the implementations of these operations may vary from language to language).

1. `INSERT(x, p , L)` - Insert `x` at position `p` in list `L`.
2. `LOCATE(x, L)` - Returns the position of `x` in `L`.
3. `RETRIEVE(p, L)` - Returns the element at position `p` in `L`.
4. `DELETE (p, L)` - Delete element at position `p` in `L`.

5. `NEXT(p, L) and PREV(p, L)` - Returns the positions following and preceding `p` in `L`.

6. `MAKENULL(L)` - Makes `L` an empty list.

7. `FIRST(L) and LAST(L)` - Returns fist and last positions in `L`.

8. `PRINTLIST(L)` - Prints elements of a list `L`.

A list that is made up of cells, where each cell contains an element of the list, and a pointer to the next cell on list is called a *linked list.* For example, a cell holding the a_i of a list a has a pointer to the cell holding a_{i+1}.

3.2.2 Implementing linked lists in C.

In `C`, a linked list consists of a sequence of structures (called *nodes*) with each node containing a pointer to the next node in the chain. The last node contains a null pointer. Declaring a node in `C` involves the declaration of a basic structure as follows:

Example 1.

```
struct node
{
    int value; /* data stored in node*/
    struct node *next; /*pointer to next node*/
 };
```

Node creation. When we create a node, we will need a variable that can point to the new node temporarily, for example in our case:

Example 2.

```
struct node *new_node;
```

Creation of a node in `C` involves the following three steps:

1. *Allocate memory for node.* To allocate memory in `C`, we use the `malloc` function, which allocates a block of memory without initializing it. The `malloc` function has the following prototype:

```
void *malloc(size-type size);
```

The function `malloc` allocates a block of `size` bytes and returns the pointer to it. Here, `size` has type `size-type`. For example, to allocate space for a string of `n` characters, we have:

Example 3.

```
p = malloc(n+1); /* char occupies 1 byte */
```

Here, the `n+1` is for accommodating the null character, which cannot be forgotten. `malloc` uses the library `stdlib.h`.

When `malloc` is called, there is a possibility that it won't be able to locate a block of memory as large as requested. In such cases, it returns the *null pointer*, which points to nothing. The null pointer is represented by the macro `NULL`.

In order to determine how much memory is required to store values of a particular type, we use the `sizeof` operator, which has the format:

```
sizeof(type_name)
```

In general, `sizeof` returns an `unsigned` integer, for example:

Example 4.

```
sizeof(char); /* returns 1 */
sizeof(int); /* returns 4 in a 32-bit machine */
```

In the following example, we show how `malloc` and `sizeof` can be used to allocate memory for our node (based on Example 1).

Example 5.

```
new_node = malloc(sizeof(struct node));
```

2. *Storing data in a node.* Storing data in a node is tantamount to assigning a value to a member of the structure that represents the node. In continuation with Examples 1 and 5, we store data in the `value` member of the `new_node`.

   Example 6.

   ```
   *(new_node).value = 10;
   ```

   As an alternative to Example 6, we could also use the *right arrow selection operator* `->`, which is a combination of the `*` and `.` operators. Using this operator, Example 6 can be written as:

   Example 7.

   ```
   new_node -> value = 10;
   ```

3. *Inserting a node in a linked list.* The advantage of a linked list is that nodes can be added at any point in the list. However, we will focus on adding a node at the beginning. If `new_node` is pointing to the node to be inserted, and `first` is pointing to the first node in the linked list, then we will need the following two statements to insert a node at the beginning of the list.

   Example 8.

   ```
   new_node -> next = first;
   /* Modifies new node's next member to point to the node
   previously at the beginning of the list */
   first = new_node;
   /* makes first point to new node */
   ```

   We will now give an example of function that adds a node to the beginning of a linked list.

Example 9.

```
struct node *add_to_list(struct node *list, int n)
{
    struct node *new_node;
    new_node = malloc(sizeof( struct node));
    new_node -> value  = n;
    if (new_node == NULL)
    {
        printf("Error");
        exit(EXIT_FAILURE);
    }
    new_node->next = list;
    return new_node;
}
```

This function returns a pointer to the newly added node. Note that `EXIT_SUCCESS` (for normal termination) and `EXIT_FAILURE` (for abnormal termination) are macros in `stdlib.h`. For calling the above function, we could use for example:

Example 10.

```
first = add_to_list(first, 10);
first = add_to_list(first, 20);
/* adds 10 and 20 to the list pointed by ‘first’ */
```

We can generalize the above function to add a node to an arbitrary position in a linked list.

Example 11.

```
struct node *add_to_list(struct node *list, int n, int pos)
{
    struct node *new_node, *prev = list, *curr;
```

```
    int i = 1;
    new_node = malloc(sizeof(struct node));
    new_node -> value  = n;
    new_node -> next = NULL;
    if (list == NULL) return new_node;
    if (pos == 0)
    {
        new_node -> next = list;
        return new_node;
    }
    else
    {
        while((prev -> next != NULL) && ( i ! = pos))
        {
            i++;
            prev = prve -> next;
        }
        if(i == pos)
        {
            curr = prev -> next;
            prev -> next = new_node;
            new_node -> next = curr;
        }
        else prev -> next = new_node;
    }
    return new_node;
}
```

Searching a node. Searching a node in a linked list could involve a function as follows:

Example 12.

```
struct node * search( struct node *list, int n)
{
    struct node *p;
    for(p = list; p ! = NULL; p = p->next)
    {
        if (p -> value == n)
            return p;
    }
    return NULL;
}
```

Deleting a node. Deleting a node in a linked list could involve a function as follows:

Example 13.

```
struct node *delete_from_list( struct node *list, int n)
{
    struct node *curr, *prev;
    for(curr = list, prev = NULL; curr!= NULL
     && cur -> value != n; prev = curr, curr = curr -> next)
    {
        if (curr == NULL)
            return list;
        if (prev == NULL)
            list = list -> next;
        else
            prev -> next = curr -> next;
    }
    return list;
}
```

3.2.3 Double linked list

A *double linked list* is a sequence of nodes in which every node has links to its previous and next element in the sequence. The first node of a double linked list is always pointed by *head*. A typical node of a double linked list can be implemented in C using a structure such as follows:

Example 14.

```
 struct Node
{
   int data;
   struct Node *previous, *next;
}*head = NULL;
```

A complete implementation will involve the following functions

```
void insertAtBeginning(int value);
void insertAtEnd(int values);
void insertAfter(int value, int location);
void deleteBeginning();
void deleteEnd();
void deleteSpecific(int deValue);
```

3.2.4 Stack

A *stack* is a special kind of list in which we insert (*push*) and delete (*pop*) at only one end called the *top* of the stack. It is also known as a *lifo* or *push-down* list. The intuitive model for a stack is a pile of poker chips on a table or books on a floor. A stack involves the following abstract operations:

1. `MAKENULL(S)` - Makes stack `S` empty.
2. `TOP(S)` - Returns element at the top of stack `S`.
3. `POP(S)` - Deletes the element at the top of stack `S`.
4. `PUSH(x, S)` - Pushes element `x` into top of stack `S`.

5. EMPTY(S) - Returns True if S is empty or False, otherwise.

The following is an implementation of the PUSH and POP operations on a stack in C, using linked lists.

Example 15.

```
 struct Node
{
   int data;
   struct Node *next;
}*top = NULL;
void push(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(top == NULL)
      newNode->next = NULL;
   else
      newNode->next = top;
   top = newNode;
   printf("\nInsertion was successful!!!\n");
}
void pop()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      printf("\nDeleted element: %d", temp->data);
      top = temp->next;
      free(temp);
   }
}
```

3.2.5 Queue

A *queue* is another kind of list where nodes are inserted (or *enqueued*) at one end called the back and deleted at another end (or *dequeued*) called the *front*. A queue involves the following abstract operations:

1. `MAKENULL(Q)` - Makes the queue `Q` empty.
2. `FRONT(S)` - Returns element at the front of the queue `Q`.
3. `ENQUEUE(x, Q)` - Inserts element `x` at the end of queue `Q`.
4. `DEQUEUE(Q)` - Deletes the element at the front of the queue `Q`.
5. `EMPTY(Q)` - Returns `True` if `Q` is empty or `False`, otherwise.

The following is an implementation of the `ENQUEUE` and `DEQUEUE` operations on a queue in `C`, using linked lists.

Example 16.

```
 struct Node
{
   int data;
   struct Node *next;
}*front = NULL,*rear = NULL;

void enqueue(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode -> next = NULL;
   if(front == NULL)
      front = rear = newNode;
   else{
      rear -> next = newNode;
      rear = newNode;
   }
   printf("\nInsertion was successful!!!\n");
}
```

```
void dequeue()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      front = front -> next;
      printf("\nDeleted element: %d\n", temp->data);
      free(temp);
   }
}
```

3.2.6 Double-ended queue

A *double ended queue* is a queue in which insertion and deletion can be performed at both ends (i.e in the front and back). A complete implementation of a double ended queue in C will involve the following functions.

```
void enQueueRear(int value);
void enQueueFront(int value);
void deQueueRear();
void deQueuFront();
```

3.2.7 Priority queue

A *priority queue* is a queue in which each node comes with an associated *priority* that is normally an integer. Nodes with higher priority get preference over nodes with lower priority during the insertion process. Below is an implementation of the `insertion and deletion` operations in a priority queue using linked lists.

Example 17.

```
typedef struct node
{
    int priority;
    int info;
```

```
    struct node *link;
}NODE;
NODE *front = NULL;

void insert(int item,int priority)
{
    NODE *tmp,*q;
    tmp = (NODE *)malloc(sizeof(NODE));
    tmp->info = item;
    tmp->priority = priority;
    /*Queue is empty or item to be added has priority more than first item*/
    if( front == NULL || priority < front->priority )
    {
        tmp->link = front;
        front = tmp;
    }
    else
    {
        q = front;
        while( q->link != NULL && q->link->priority <= priority )
        q=q->link;
        tmp->link = q->link;
        q->link = tmp;
    }
}

void del()
{
    NODE *tmp;
    if(front == NULL)
    printf("Queue Underflow\n");
    else
    {
        tmp = front;
        printf("Deleted item is %d\n",tmp->info);
        front = front->link;
        free(tmp);
    }
```

```
}
```

3.2.8 Trees

A *tree* is a collection of elements called nodes, one of which is a distinguished node called a *root*, along with a relation called *parenthood* that places a hierarchical structure on the nodes. A tree can be defined recursively as follows:

1. A single node by itself is a *tree* called the *root of a tree*.

2. Suppose n is a node, and $T_1, \ldots, T_k$ are trees with roots $n_1, \ldots, n_k$, respectively. We can construct a new tree by making n be the parent of the nodes $n_1, \ldots n_k$. In this tree, n is the root, and $T_1, \ldots, T_k$ are the subtrees of the root. Nodes $n_1, \ldots, n_k$ are called the children of node n.

A sequence $n_1, \ldots, n_k$ of nodes in a tree such that n_i is the parent of n_{i+1}, for $1 \leq i \leq k-1$, is called a *path* from n_1 to n_k. If there is a path from a node a to a node b in a tree, then a is called an *ancestor* of b, and b is called a *descendant* of a. In a tree, a root is the only node with no proper ancestor. A *leaf* is a node with no proper descendant.

The *height* of a node is the length of the longest path from a node to a leaf. The *depth* of a node is the length of the unique path from the root to that node. The descendants of a node of depth one are its *children*. The children of a node are ordered from left to right. The descendant of a node of same depth are called *siblings*. The left-right ordering of siblings can be extended to compare any two nodes that are not related by the ancestor-descendant relationship. In this regard, there are two key rules that a tree data structure must follow:

1. If nodes a and b are siblings, and a is to the left of b, then all descendants of a are to the left of all descendants of b.

2. Moreover, given a node n , finding nodes to its left and those to its right is done by drawing a a path from the root to n. All nodes branching off to the left (resp. right) of this path, and all descendants of such nodes are to the left (resp. right) of n.

Binary trees. A tree in which every node has a maximum of two children is called a *binary tree*. A binary tree in which every node has either two or zero

number of children is called a *strictly binary tree*. The process of displaying or visiting a particular node in a binary tree is called *binary tree traversal*. There are three types of binary tree traversals:

1. *In-order traversal*. In this traversal, the root node is visited between the left and right child nodes.
2. *Pre-order traversal*. In this traversal, the root node is visited before the left and right child nodes.
3. *Post-order traversal*. In this traversal, the root node is visited after the left and right child nodes.

Below is an implementation of a binary tree in C that uses the in-order traversal for display.

Example 18.

```
// C program to demonstrate insert operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
```

```
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d \n", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80 */
    struct node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
```

```
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // print inorder traversal of the BST
    inorder(root);
    return 0;
}
```

4 Algorithms

The notes in this section were prepared based on the study material available at [1], [2], [3], and [4].

4.1 Sorting algorithms

We will discuss four basic sorting algorithms.

4.1.1 Bubble sort

This is a simple comparison-based sorting algorithm in which each pair of adjacent elements in a list (or an array) are compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items. Below is a simple C pseudo code for the implementation of bubble sort.

Example 1.

```
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void bubbleSort(int arr[], int n)
{
```

```
    int i, j;
    for (i = 0; i < n-1; i++)

        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
               swap(&arr[j], &arr[j+1]);
}
```

4.1.2 Insertion sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands. It follows the following algorithm:

1. Assume that the first element in the list is in the sorted portion of the list, and remaining elements are in unsorted portion of the list.
2. Consider the first element from the unsorted portion of the list and insert that element into the sorted portion of list in order specified.
3. Repeat the above process until all the elements from the unsorted list are moved into the sorted list.

Below is a simple routine for implementing this algorithm in C.

Example 2.

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i-1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
```

```
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
```

4.1.3 Selection sort

In selection sort, the first element in the list is selected and it is compared repeatedly with remaining elements in the list. If any element is smaller (than the selected element), then the two elements are swapped. Then we select the element at second position in the list and repeat the above procedure. This is repeated until the entire list is sorted. The selection sort algorithm is performed using the following steps:

1. Select the element at the first position in the list.
2. Compare the selected element with all other elements in the list.
3. During each comparison, if any element is smaller than the selected element (or larger), then swap the two elements.
4. Repeat the same procedure with next position in the list until the entire list is sorted.

To sort a unsorted list with n elements it takes $n(n-1)/2$ number of comparisons in the worst case scenario, and so this algorithm has a worst case complexity of $O(n^2)$.

4.1.4 Quick Sort

Quick sort is a highly efficient sorting algorithm that is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, called a *pivot*, based on which the partition is made, and another sub-array holds values greater than the pivot value. Quick sort partitions an array and then calls itself recursively

twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of $O(n^2)$, and an average complexity of $O(n\log(n))$, where n is the size of the list or the array. The quick sort algorithm has the following steps:

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

4.2 Basic searching algorithms

Searching is a process of finding a value in a list of values. We discuss two basic search algorithms for lists.

4.2.1 Linear Search

This search process starts comparing of search element with the first element in the list. If both are matching then the algorithm terminates, otherwise search element is compared with next element in the list, and the same process is repeated until the search element is compared with last element in the list. Linear search algorithm finds given element in a list of elements with $O(n)$ time complexity, where n is total number of elements in the list. We now give a formal algorithm for this search process.

1. Compare the search element with the first element in the list.
2. If both are matching, then display "Element found!!!" and terminate the search.
3. If both the two elements do not match, then compare search element with the next element in the list.
4. Repeat steps 2 and 3 until the search element is compared with the last element in the list.

5. If the last element in the list does not match the search element, then display "Element not found!!!" and terminate the search process.

4.2.2 Binary search

That means, binary search can be used only in a sorted list. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the search process terminates. Otherwise, it checks whether the search element is smaller or larger than the middle element in the list, and repeats the same process for the sublist to the left (or to the right) of the middle element. We repeat this process until we find the search element in the list, or until we are left with a sublist of size one. The Binary search algorithm finds given element in a list of elements with $n\log(n)$ complexity, where n is total number of elements in the list. We given a simple algorithm describing this search process.

1. Find the middle element in the sorted list.
2. Compare, the search element with the middle element in the sorted list.
3. If both are matching, then display "Given element found!!!" and terminate the function
4. If both are not matching, then check whether the search element is smaller or larger than middle element.
5. If the search element is smaller than the middle element, then repeat steps 1 - 4 for the sublist to the left of the middle element.
6. If the search element is larger than middle element, then repeat steps 1 - 4 for the sublist to the right of the middle element.
7. Repeat the same process until we find the search element is found in the list, or until we are left with a sublist of size one.
8. If the search is not in the sublist of size one, then display "Element not found in the list!!!" and terminate the search process.

4.3 Graphs and traversal algorithms

4.3.1 Introduction to graphs

Abstractly, a *finite undirected simple graph* G is a pair (V,E), where $V = \{v_1,\ldots,v_n\}$ is called a set of *vertices* and $E \subset \{\{v,w\} \in V \times V : v \neq w\}$ is called the set of *edges*. We define the *size* of a graph $G = (V,E)$ as above by $|G| := |V|$. Graphs are widely used to model communication networks. In network science, a vertex of a graph is also referred to as a *node*. For practical purposes, we will only consider connected graphs which have at most one edge connecting any pair of vertices.

The *adjacency matrix* $A(G)$ associated with a graph $G = (V,E)$ as above is defined as the matrix $A(G) := (a_{ij})_{n\times n}$, where

$$a_{ij} = \begin{cases} 1, & \text{if } \{v_i, v_j\} \in E, \text{ and} \\ 0, & \text{otherwise.} \end{cases}$$

the *valency* of a vertex v in a graph $G = (V,E)$ is the number of verticed adjacent to it, that is, $|\{\{v,w\} : \{v,w\} \in E\}|$. If every vertex of a graph G has valency $k > 0$, then the graph is said to be *k-regular*. It is known [6] that for a k-regular graph G, $A(G)$ always has k has a eigenvalue, and in fact, any eigenvalue of λ of $A(G)$ satisfies $|\lambda| \leq k$.

The difference between the largest and second largest eigenvalues of a graph G is called its *spectral gap*. Graphs with large spectral gaps (called *spectral expanders*) are known to have high connectivity [8], and hence such graphs are ideally suited for adaptation to communication networks. Constructing infinite families of spectral expander graphs (or *expander families*) is a now a huge area of research that lies at the interface of various fields of mathematics which include algebraic graph theory, representation theory, number theory, and combinbatorics.

A graph that no cycles (or closed paths) is called a *tree*. A *spanning tree or maximal tree* of a graph G is a tree that contains every vertex of the graph G. It is not hard to see that every connected graph has at least one maximal tree.

4.3.2 Graph traversal algorithms

Graph traversal is a technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit during the search process. We will discuss two well known graph traversal algorithms that find a maximal tree in a given graph G.

1. **DFS (Depth First Search).** We use the Stack data structure to implement a DFS traversal, and it had the following algorithm:

    1. Define a Stack of size $|G|$.
    2. Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
    3. Visit any one of the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.
    4. Repeat step 3 until there are no new vertices to be visited from the vertex at the top of the stack.
    5. When there is no new vertex to be visited, then use backtracking and pop one vertex from the stack.
    6. Repeat steps 3 -5 until stack becomes empty.
    7. When stack becomes empty, then produce final spanning tree by removing unused edges from the graph.

2. **BFS (Breadth First Search).** We use the queue data structure to implement a DFS traversal, and it has the following algorithm:

    1. Define a Queue of size $|G|$.
    2. Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
    3. Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.
    4. When there is no new vertex to be visited other than the vertex at the front of the Queue, then delete that vertex from the Queue.
    5. Repeat step 3 - 4 until queue becomes empty.
    6. When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

References

[1] Bsc btech smart class - c programming language. `http://btechsmartclass.com/CP/c-home.htm`.

[2] geeksforgeeks.org - priority queue using linked lists. https://www.geeksforgeeks.org/priority-queue-using-linked-list/.

[3] Includehelp.com - data structure using c and c++ programming. http://www.includehelp.com/ds/.

[4] tutorialspoint.com - quick sort. https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm.

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data structures and algorithms*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publishing Co., Reading, Mass., 1983.

[6] Chris Godsil and Gordon Royle. *Algebraic graph theory*, volume 207 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 2001.

[7] Kim N King. *C programming: a modern approach*. WW Norton & Company, 2008.

[8] Mike Krebs and Anthony Shaheen. *Expander families and Cayley graphs*. Oxford University Press, Oxford, 2011. A beginner's guide.

[9] Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.