CSC 181F Lecture Notes

These lecture notes are provided for the personal use of students taking CSC181F in the Fall term 1999/2000 at the University of Toronto

Copying for purposes other than this use and all forms of distribution are expressly prohibited.

Reading Assignment

```
Supplementary reading

        S. McConnell      Chapter  26, 25
```


Software Debugging and Testing

- Debugging is the process of finding errors in a program under development that is not thought to be correct
- Testing is the process of attempting to find errors in a program that is thought to be correct. Testing attempts to establish that a program satisfies its Specification[a]
- Exhaustive testing is not possible for real programs due to combinatorial explosion of possible test cases. Amount of testing performed must be balanced against the cost of undiscovered errors
- *Regression Testing* is used to compare a modified version of a program against a previous version

[a] **Testing can establish the presence of errors but cannot guarantee their absence** (E.W. Dijkstra)

Program Correctness

- A program is *correct* if it compiles without errors and when executed produces output that satisfies the specification for the program.
- **Correctness is more important than efficiency** (or anything else)
- Levels of Correctness:

1. No syntax errors
2. No semantic errors
3. There exists some test data for which program gives a correct answer
4. Program gives correct answer for reasonable or random test data
5. Program gives correct answer for difficult test data
6. For all legal input data the program gives the correct answer
7. For all legal input and all likely erroneous input the program gives a correct or reasonable answer
8. For all input the program gives a correct or reasonable answer

Testing Strategy

- Try simple cases first
  so you can hand compute answer
- Try boundary conditions & special cases
- Try reasonable & random input
- Try input containing errors
- Try *really hard* input
- **Be really cruel**
  What is the worst thing you can do to the program?
- Try to test all parts of the program

Testing & Bad Attitude

- The goal in testing software is to find as many errors as possible in the program under test with the least effort expended
- Testing efficiency is measured in the number of errors discovered per hour of testing
- When testing your attitude should be
  What is the absolutely worst thing I can do to the program?
  and **not**
  What can I do to make this program look good?
- Test case selection is one key factor in successful testing
- Insight and imagination are essential in the design of good test cases

Sources for Test Cases

- Requirements and Specification for the program
- General knowledge about the application area
- Program design and user documentation
- Specific knowledge of the program source code (White Box Testing)
- Specific knowledge of the programming language and implementation techniques
- Test at and near (inside and outside) the boundaries of the programs applicability
- Test with *random* data
- Test for response to probable errors (e.g. invalid input data)
- **Think nasty when designing test cases.**
  Try to destroy the program with your test data

Testing Based on the Source Program

- *Basic Path Testing* - design test cases to guarantee that every path through the program is executed at least once
  (i.e both branches of every **if** , every loop, all function calls)
  Derive test cases from examination of the program
- *Condition Testing* - design test cases to test all possible outcomes for each condition (Boolean expression) in the program.
- *Branch testing* - design test cases to cause each condition in an **if** to evaluate to true and false. Test every case and default in each **switch** statement.

- *Definition-Use Testing* - design tests to link definition (i.e. value assignment) and use of variables in the program
  Try to execute every definition-use chain at least once
- *Simple Loop Testing* - design test cases to exercise every loop in the program
  - Loop not executed at all - tests code after loop for correct handling of null case
  - Loop executed once - tests initialization and exit condition
  - Loop executed twice - tests passing of values between iterations
  - Loop executed random legal number of times
  - Loop executed one less than allowable maximum
  - Loop executed exactly allowable maximum
  - Loop executed one more than allowable maximum

Testing Example - Quadratic Program

- Easy quadratics with two real roots
- Easy quadratics with complex roots
- Degenerate cases, a, b and/or c = 0.0
- Hard quadratics
  very large or very small coefficients
  $b^2 \sim 4ac$
  $-b \sim \sqrt{b^2 - 4ac}$
  $x_1 = x_2$ or $x_1 \sim x_2$

Testing - Example

Program — Search an array for a given value

int Search(**int** *Ar*[ ] , **const int** ArSize , **const int** *val*)

Specification — if *val* is an element of the array *Ar* then Search returns its index in Ar. Otherwise Search returns -1

Test Data for Search

Each of these tests is designed to catch a specific kind of error.

Array with zero elements
Array with one element
    val in, not in below, not in above
Array with random even size
    val not in, in random, in first, in last, in middle $\pm$ 1
Array with random odd size
    val not in, in random, in first, in last, in middle, in middle $\pm$ 1
Array with two elements
    val not in below, not in above, in first, in last
Arrays containing ordered, reverse ordered and unordered data
Array random size containing all one value, equal, not equal to val
Array of maximum allowable size
Array with upper bound of largest allowable integer
Array containing largest and smallest integers

Uninitialized Variable Errors

- An *uninitialized variable error* occurs when the *value* of a variable is *used* (e.g. the variable occurs in an expression) **before** a value has been assigned to the variable.
- Except for some rare pathological cases, it is an **ERROR** to use a variable before it has a value. GARBAGE IN implies GARBAGE OUT.
- Any incorrect program behavior may be a symptom of an uninitialiazed variable error.
- Uninitialized variable errors are often vary hard to find.
  - Symptoms may vary from one run to another. Different Garbage.
  - *Heisenbug Effect* – adding debugging code may change or obscure the error.
  - The program "looks" OK. Unitialized variable errors are hard to see.

- Eliminating unitialized variable errors by Inspection is *much more* effective than tracing and debugging a running program.
- At each place where a variable is used in a program it should be possible to give an (informal) argument that the variable *always has* a value.
- If you can't make the argument then you have
  - an ERROR in your program (99.36% probability).
  - a rare pathological case that needs a special comment.
- **Example:**

```
float  sum , A[ ASIZE ];
int  K ;
/* Assume A is given a value here */
for( K = 0 ; K < ASIZE ; K++ )
   sum += A[K] ;
```


Program Inspection to Improve Quality

- Program inspection is the process of examining a program in fine detail to find errors.
- **Much more effective in terms of programmer effort than testing.**
- Read through the program a few (<= 3) lines at a time. Try to describe in words *exactly* what the lines do.
- Program inspection was pioneered by Bell Northern Research. It is widely used in industry by real programmers since it's by far the most cost and time effective way to find errors in programs.
- *With careful inspection it* **is** *possible to write programs that will:*
  - *compile without errors the first time they are compiled*
  - *run without errors the first time they are executed*

How to inspect Programs[a]

- Check that every variable **will always** have a value before it is used.
- Check all expressions to make sure the correct value is being computed.
  Check that all subscript expressions will be valid.
- Check conditions in all **if** statements
  Do they partition between the true and false cases correctly?
  Are all possibilities covered? Check cases in **switch** statements.
- Check all **for** , **while** and **do** statements
  Is the exit condition correctly specified?
  Beware of *off-by-one* errors
- Check all function calls for the correct type and order of parameters.
- *Learn from your mistakes!* If you consistently make one kind of error, inspect extra hard for that error.

[a] B.W. Kernighan and P.J. Plauger, Elements of Programming Style, McGraw-Hill, 1978

Program Inspection Example

```
/* Read in year and day, output month, day, year. */
const char * monthName[13] =        /* monthName[0] unused */
    { "", "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December" } ;
short monthLength[13] =        /* monthLength[0] unused */
    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 } ;
int year, dayInYear, preceedingDays, month ;
while ( true ) {        /* Infinite loop to read then output dates */
    while ( true ) {        /* Process one correct input */
        printf ("Enter year and number of day in year\n" ) ;
        if ( scanf ("%d%d", & year, & day ) == 2 )
            if ( 1900 <= year && year <= 3000 && dayInYear >= 1 )
                if ( year % 4 == 0 ) {
                    monthLength[2] = 29 ;
                    if ( dayInYear <= 366 )
                        break ;
                } else {
                    monthLength[2] = 28 ;
                    if ( dayInYear <= 365 )
                        break ;
                }
    }
    precedingDays = 0 ;
    while ( dayInYear < precedingDays + monthLength[month] ) {
        precedingDays = precedingDays + monthLength[month] ;
        month++ ;
    }
    printf ( "The Date is %s %d , %d \\n" ,
        monthName[month], daysInYear - precedingDays, , year
}
```

Reading Assignment

K.N. King

Sections 7.1, 7.2, 7.5
Sections 16.5, 20.1 18.2

Some More Details about C

- All about Integer Types
- Float and Double Types
- Explicit Type Conversion (casting)
- Enumerations
- Bitwise Operators
- Storage Classes

The Integer Types

type-name	Size	Range of Values
unsigned char	8 bits	$0 : 2^8 - 1$
signed char	8 bits	$-2^7 : 2^7 - 1$
short	16 bits	$-2^{15} : 2^{15} - 1$
short int	16 bits	$-2^{15} : 2^{15} - 1$
unsigned short	16 bits	$0 : 2^{16} - 1$
int	32 bits	$-2^{31} : 2^{31} - 1$
signed	32 bits	$-2^{31} : 2^{31} - 1$
unsigned int	32 bits	$0 : 2^{32} - 1$
unsigned	32 bits	$0 : 2^{32} - 1$
long	32 bits	$-2^{31} : 2^{31} - 1$
long int	32 bits	$-2^{31} : 2^{31} - 1$
unsigned long	32 bits	$0 : 2^{32} - 1$

Float and Double

type-name	Size	Range of Values (±)	Precision
float	32 bits	$1.17 \cdot 10^{-38}$ **..** $3.40 \cdot 10^{38}$	6 digits
double	64 bits	$2.22 \cdot 10^{-308}$ **..** $1.79 \cdot 10^{308}$	15 digits
long double	80/128 bits	varies	varies

- **float** and **double** constants are written in form of scientific notation.
  The constant consists of a mantissa followed by an optional exponent part.
  A float or double constant must contain a decimal point *or* an exponent part to distinguish it from an integer constant.
- The mantissa is a sequence of decimal digits. The manitssa may optionally include one decimal point. Example mantissas: 0.1 .23 45. 678.9
- The optional exponent part consists of an upper or lower case letter E followed by an optional sign and one or more exponent digits. **Examples:** E12 E-4 E+145 e-67 e89
- Constants are represented internally as type double unless they are followed by the letter F (float) or L (long double).
- **Examples:** .0123 12.34 1234. 123.456e+7 123.456E-12F

Casting - Explicit Type Conversion

(*typeName*) *expression*

Explicit type conversion forces *expression* to be treated as if it were the type specified by *typeName*

Effect as if *expression* was assigned to a variable of type *typeName*

Examples:

```
int I ;
float X ;
...
I = ( int ) X ;
X = ( float ) I ;
```


Enumerations

enum *enumTag* { *identifierList* } **;**

- The **enum** declaration specifies a list of symbolic names (the *identifierList*)
- The *optional* enumTag is an identifier which provides a *name* for the enumeration type.
- Usually the compiler assigns an internal representation to the identifiers in the list. The programmer can specify the values used by including assignments in the identifier list as in

  enum Numbers { two = 2 , three, four, eight = 8, nine } **;**

  The default representation starts at zero and gives each identifier a value one greater than the identifier that precedes it.

HOW TO Use Enumerations

- Enumerations are a mechanism that allows you to declare a set of related *symbolic names* in cases when you don't care about the internal representation.
- Use enumerations to represent the state of variables that take on a small number of values. The symbolic names make the program easier to read.
- The same effect could be achieved using **#define** but **enum** is more elegant and makes the program easier to read.
- Almost all enumeration declarations should have an enumeraton tag (unless they appear in a **typedef** declaration.
- **Examples:**

```
enum directions { south, southWest, west, northWest,
        north, northEast, east, southEast } ;
enum StopLight { green, flashingGreen, yellow, red } ;
typedef enum { Clubs, Diamonds, Hearts, Spades } Suit ;
```

Bitwise Operators

&	Bitwise and
|	Bitwise or
˜	Bitwise not
ˆ	Bitwise exclusive or
<<	Left shift n bits
>>	Right shift n bits

Bitwise operations are used to manipulate the pattern of *bits* in an expression, e.g extracting or combining information.

WARNING: BE CAREFUL, don't confuse

& **and** && **,** | **and** || **,** ˜ **and** !

A & B could be zero (false) even if A and B are non-zero (true).

Bitwise Operators Defintions

A	B	˜ A	A | B	A & B	A ˆ B
0	0	1	0	0	0
0	1	1	1	0	1
1	0	0	1	0	1
1	1	0	1	1	0

A	A >> 1	A >> 2	A << 1	A << 2
11010	01101	00110	110100	1101000
00101	00010	00001	001010	0010100

Bitwise Operator Examples

```
unsigned short A, B, C, D ;     /* 16-bit variables */
                          /* Value in Binary (base 2)    */
A = 03567 ;               /* 0000011101110111  */
B = 255 ;                 /* 0000000011111111  */
C = 0x35AF ;              /* 0011010110101111  */

D = ˜ C ;                 /* 1100101001010000  */
D = B & C ;               /* 0000000010101111  */
D = ˜ B & C ;             /* 0011010100000000  */
D = A | C ;               /* 0011011111111111  */
D = A ˆ C ;               /* 0011001010111000  */
D = B << 3 ;              /* 0000011111111000  */
D = C >> 7 ;              /* 0000000001101011  */
```


HOW TO Use Bitwise Operators

- Bitwise operators can be used for several purpose
  - To pack data into less space and to extract packed data.
  - To access packed information in hardware registers.
  - To emulate higher level data structures, e.g. sets
- Bitwise operations **should not** be used if they make the program hard to understand and there is a simpler alternative.
  To use bitwise operators you need to understand how information is represented internally in the computer. See previous slide.
- It is usually slower to access packed information so packing should only be used when the space saving is really important.
- The & operator can be used to extract information and to create a hole to put information into. The | operator can be used to insert information into a large item.

Data Packing Example

Assume that six 5-bit integers (values 0 .. 31) are to be packed in a 32-bit **unsigned** variable. The six subfields are called A, B, C, D, E, F .
This example shows how to access one field (D) of the packed information.

```
#define DMASK    ( 0x00007C00 )  /* 0..0111110000000000 */
#define FIVEBITS ( 0x1F )        /* 0   ..  011111 */
#define DSHIFT   ( 10 )   /* # bits to the right of D   */

unsigned  x ;   /* Variable containing A,B,C,D,E,F */

short   dtmp , dtmp1 ;   /* Variable to hold D  */

x = 0xCAE1DB75 ;

dtmp1 = 0x35  ;
```



```
/*  Extract D from x and store in dtmp */
dtmp = ( x & DMASK ) >> DSHIFT ;
----------------------------------------------------------------
11001010111000011101101101110101        x
00000000000000000111110000000000        DMASK
00000000000000000101100000000000        x & DMASK
00000000000000000000000000010110       ( x & DMASK ) >> DSHIFT

/*  Extract D using only shift operators */
dtmp = ( ( x << 5+5+5+2 ) >> 32-5 ) ;
----------------------------------------------------------------
11001010111000011101101101110101        x
10110110111010100000000000000000        x << 17
00000000000000000000000000010110        ( x << 17 ) >> 27
```



```
/*  Insert new D value (dtmp1) into x, trim to fit.       */
x =  ( x & ˜DMASK ) | ( ( dtmp1 & FIVEBITS ) << DSHIFT ) ;
----------------------------------------------------------------
11001010111000011101101101110101        x
11111111111111111000001111111111        ˜ DMASK
11001010111000011000001101110101        ( x & ˜DMASK )
                0000000000110101        dtmp1
                0000000000011111        FIVEBITS
                0000000000010101        ( dtmp1 & FIVEBITS )
                0101010000000000        ( dtmp1 & FIVEBITS )
                                                   << DSHIFT
11001010111000011101011101110101        ( x & ˜DMASK ) |
                                            ( ( dtmp1 & FIVEBITS
                                              << DSHIFT )
```


Storage Classes

- A storage class is associated with every declared object.
  This storage class determines the extent (lifetime) of the storage associated with the object.
  In some cases the storage class also affects the visibility of the object.
- The storage classes in C

auto	locally created storage **this is the default**
static	for variables indicates statically created permanent storage also restricts visibility to file of declaration
extern	static extern but name is visible outside file of declaration
register	hint to compiler to store variable in a hardware register

Examples

static double RandomSeed = 123456.789 **;**
register int I **,** K **;**
extern long sharedData **;**

Reading Assignment

```
        K.N. King           Section  9.6

Supplementary reading

        S. McConnell        Chapter 16

Also Recommended
        E. Roberts,  Thinking Recursively
             SHORT-TERM LOAN - ENGINEERING LIBRARY

E. Roberts,  Programming Abstractions in C
             Chapters  4, 5, 6
             SHORT-TERM LOAN - ENGINEERING LIBRARY
```


Recursion

- **Extremely important programming technique**
- Based on
  Divide and Conquer
  Induction
- Think of recursion when a problem involves embedded instances of itself
- **You should become proficient in using recursion as a problem solving technique**

Why Recursion ?

- Recursive solutions are frequently simpler than non-recursive solutions
- Recursive programs are easier to make correct
- Use of recursion often leads to simpler, more elegant algorithms
- Recursion divides a large problem into smaller, easier to solve pieces

Simple Example - Factorial

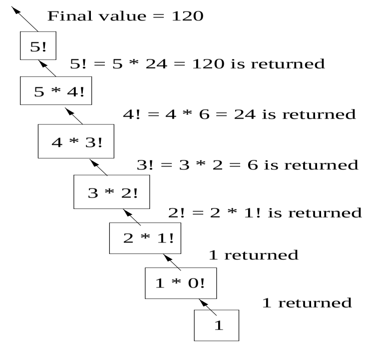
- The factorial function is a very simple example of a function that can be computed using recursion.
- Mathematical Definition

$$N! = \begin{cases} 1 & \text{if } N = 0 \\ N \cdot (N-1)! & \text{otherwise} \end{cases}$$

- Key insights
  $1!$ is really easy to compute
  if $N > 0$ then $N - 1$ approaches 1 in the limit
  $N!$ can be defined in terms of $(N - 1)!$

Example - Factorial

```
/* Computing factorial */
int factorial( int N)
{
  if ( N == 0 )
    return 1;    /* basis */
  else
    return N * factorial (N - 1);
}
```



Trace of call factorial(5)

HOW TO Use Recursion

- Analyze the Problem
  - Identify simple cases
  - Identify ways to divide problem
    - Simple cases
    - Rest of problem same/similar form as the problem

- Select data structure to represent problem
- Write recursive functions & procedures
  - Handle simple cases directly
  - Use decomposition and recursion on the rest

Similar to mathematical induction

Generic Recursive Model

type-name Func (*parameters*)
{
  if *simpleCase* {
    /* Handle Simple Case */
    return ..
  }
  else {
    /* Decompose problem into parts */
    /* Call the function *Func* recursively */
    $Solution_{part}$:= *Func* ($Problem_{part}$)
    $Solution_{rest}$:= *Func* ($Problem_{rest}$)
    /* Combine $Solution_{part}$ and $Solution_{rest}$ */
    return ..
  }
}

The Towers of Hanoi Problem

In the great temple at Benares beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles each a cubit high and as thick as the body of a bee. On one of these needles at the creation, God placed 64 disks of pure gold, the largest disk on the brass plate and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and Night unceasingly, the priests transfer the disks from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disk at a time and that he must place this disk on a needle so that there is no smaller disk below it. When all the 64 disks have been thus transfered from the needle on which at the creation God placed them to one of the other needles, tower, temple and Brahmins alike will crumble into dust and with a thunderclap the world will vanish.[a]

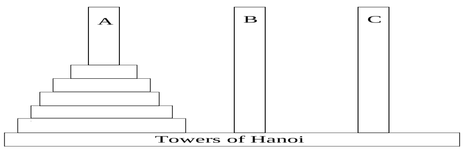
[a] W.W.R. Ball as quoted by E. Roberts, Programming Abstractions in C, page 196

Example - Towers of Hanoi

Move N disks from one peg to another by moving one disk at a time subject to the constraint that a larger disk may never be placed on a smaller disk

Analysis:

- Simple cases: move one disk
- Division: Move top N -1 disks out of the way.
  Move bottom disk to final destination.
- Composition: Move the remaining N -1 disks to the destination.



Example - Hanoi

```
/* Solution to Towers of Hanoi Puzzle */
void hanoi( const char source, const char dest, const char temp, const int N )
{
    /* Move N disks from source peg to dest peg, using temp peg as temporary storage */
    if ( N == 1 ) {
        printf ("Move a disk from %c to %c\n", source, dest) ;
        return ;
    }
    else {
        /* Move top N-1 disks out of the way to temp peg */
        hanoi( source, temp, dest, N - 1 ) ;
        /* Move bottom source disk to destination */
        printf ("Move a disk from %c to %c\n", source, dest) ;
        /* Move remaining N-1 disks from temp peg to destination */
        hanoi( temp, dest, source, N - 1) ;
    }
}
```


HOW TO Find a Recursive Strategy

- Any problem you want to solve using recursion must satisfy the conditions

  1. There **must** be one or more **simple cases**. i.e. cases that can be done directly.
  2. It must be possible to break the problem down into simpler subproblems **of the same form**.
  3. Solution of the subproblems must somehow help to solve the larger problem.

- Decomposition and recombination are often the hardest parts of the strategy.
- Once you've designed a recursive strategy, you should *validate* the strategy by working through a few simple examples by hand.

HOW TO Avoid Pitfalls in Recursive Solutions[a]

- Are simple cases checked for *first*?
  Are the simple cases solved correctly?
- Does the recursive decomposition make the problem *simpler*?
  Each recursion should make progress toward reaching one of the simple cases.
- Will the recursion *always* terminate?
  Does the simplification process *always* reach the simple cases?
  Have some simple cases been overlooked?
- Do the recursive calls represent subproblems that are truly identical in form to the original problem?
  Do the solutions to the recursive subproblems provide a complete solution to the original problem?

[a] Adapted from E.Roberts, Programming Abstractions in C, Chapter 4

Example - Detecting Palindromes

A *palidrome* is a string that reads identically forward and backward.

Examples: ”level” ”Madam I am Adam”

Design a recursive strategy to determine if a given string is a palindrome.

Analysis:

- Simple cases: empty string is a palindrome,
  a string containing one character is a palindrome .
- Division: If a string is a palindrome then the first and last characters in the string must be the same.
  If a string is a palindrome than the string formed by removing the first and last characters must also be a palindrome.
- Composition: if first and last characters are unequal return false otherwise return palindromness of string with first and last characters removed.

Example - IsPalindrome

```
Bool IsPalindrome( const char string[] ) {
    /* Return TRUE if string is a palindrome, FALSE otherwise */
    return IsPalindrome2( string, 0, strlen( string ) - 1 ) ;
}

Bool IsPalindrome2( const char string[], const int first, const int last ) {
    if ( last - first <= 1 )
        return true ;
    else
        return ( string[first] == string[last] )
                && IsPalindrome2( string, first + 1 , last - 1 ) ;
}
```


HOW TO Use Helper Functions

- The solution to IsPalindrome uses a *helper function* IsPalindrome2 to do the real work.
- Helper functions are appropriate when you need some extra parameters to carry additional information between levels of recursive calls
- Think of using helper functions in cases where the function you need to write (i.e. it’s specifications are a given) doesn’t have all the parameters you need to compute its value efficiently.
- Try to use the minimum number of extra parameters required to solve the problem. Usually one or two.

Example - Binary Search

Search sorted table (array) for key

Return table index if found, -1 otherwise.

Analysis:

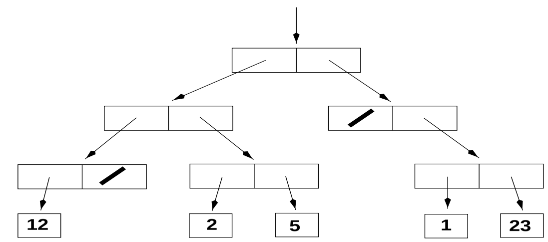
- Simple cases: empty table,
  table with one entry
- Division: Split table in half
- Composition: Result from half of table

Example - Binary Search

```
int binSearch( const int table[], const int low, const int high, const int key )
{
    /* Search sorted array table low .. high for key */
    /* Return table index if found, otherwise -1 */
    const int mid = ( low + high ) / 2 ;     /* middle of table */
    if ( high < low )     /* empty table */
        return -1 ;     /* key not found */
    else if ( table(mid) == key )
        return mid;     /* key found at table(mid) */
    else if ( table(mid) < key )
        /* search upper half of table for key */
        return binSearch(table, mid + 1, high, key) ;
    else
        /* search lower half of table for key */
        return binSearch(table, low, mid - 1, key) ;
}
```


Example - Sum Binary Tree

Binary tree:



Analysis:

- Simple cases: null tree, leaf
- Decomposiiton: left branch & right branch
- Composition: sum of left and right branches

Trees using Arrays

```
#define MAXTREE ( 10000 )
#define STUMP ( 0 )     /* null tree */
#define STUMP_VALUE ( 0 )
typedef short Tree ;
typedef enum ( branch, leaf ) treeNodeType ;
typedef long int treeLeafType ;
/* Storage for trees[a] */
treeNodeType treeNode[ MAXTREE ] ;
treeLeafType treeLeaf[ MAXTREE ] ;
Tree treeLeft[ MAXTREE ] ;
Tree treeRight[ MAXTREE ] ;
```

[a] We'll see much better ways to do trees later in the term.

Example - Tree Sum

```
treeLeafType treeSum( const Tree treePtr ) {
    /* return sum of leaves of tree */
    if ( treePtr == STUMP )
        return STUMP_VALUE ;
    if ( treeNode[ treePtr ] == leaf )
        /* process leaf */
        return treeLeaf[ treePtr ] ;
    else
        /* process branch */
        return treeSum( treeLeft[ treePtr ] )
            + treeSum( treeRight[ treePtr ] ) ;
}
```