Part -2
Data Structures

Memory

Arrays (1D, 2D, 3D, etc.)
  Array Variables
  Length
  Operations (Insertion, Deletion, Print, Search, Sort, etc.)
  Implementation

Linked List
  Data structure
  Operations
  Implementation

Stacks
  Data structure
  Operations
  Implementation

Queues
  Data structure
  Operations
  Implementation

Doubly Linked List
  Data structure
  Operations (insert, delete, search, print, traversal, etc.)

Recursion

Binary Trees
  Data structure
  Operations (insert, search, print, traversal, etc.)
  Implementation

Hash Tables
  Data structure
  Hash Function
  Collision resolution
Memory

- The main memory is an electronic solid-state random access device.

```
0 1
```

- Memory can be viewed as a series of sequential bytes, in which data and/or code can be stored.

Variables:

- A variable is the name given to a memory location. (the memory location may hold data items such as numbers, characters, etc..)

Example:

```
Declaring Variables:
    int x;
    float y;
    char z;

Using the Variables:
    x = 5;
    y = 4.8;
    z = 'A';
```
The Array Data Type:
- A homogeneous aggregate of data elements.
- A collection of objects of the same type.
- A collection of similar variables which are identified under the same name.

Creating an Array:

```c
int test_scores[20]; // create an array of 20 integers
```

```
test_scores

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 0 | 1 | 2 | 3 | 4 | ... | ... | ... | ... | ... | 19
```

- Creates **20 integer variables** under the name `test_scores`.
- In order to access each variable (array element) we must use an array index.

```c
test_scores[0] = 2;
test_scores[1] = 5;
test_scores[4] = 9;
```

```
test_scores

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 2 | 5 | 9 |   |   |   |   |   |   |   | 19
```

<table>
<thead>
<tr>
<th>Array</th>
<th>Array Element of</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Index</td>
</tr>
<tr>
<td>or Subscript</td>
<td>array</td>
</tr>
</tbody>
</table>
Array as an Object

Viewing the data (integer, char, string, array, etc.) and its operations (+, -, /, *, initialize, print, read, sort, etc.) as an atomic unit.

Operation (methods) of an Array:

- Initializing the Test_scores:
  
  ```
  for (index = 0; index < 20; index++)
  test_scores[index] = 0;
  ```

- Reading and Inserting data in the array:
  
  ```
  cin >> test_scores[index];
  ```

- Displaying the cell contents of the array:
  
  ```
  cout << test_scores[index];
  ```

- Searching an Array:
  - Sequential search
  - Binary search (shown below)

  ```
  cin >> value; // value to search for
  for (index = 0; index < 20; index++)
  if (test_scores[index] == value)
  cout << “Found the value at location” << index << endl;
  ```

- Sorting an Array
  - Selection Sort (Simple and slow, good for small number of items)
  - Bubble Sort (Simple and slow, ..)
'Overloaded Search Function (Two Parameters)

Search for a value in an array and return the index, or return -1
to indicate that value was not found.

Private Function Search(ByVal value As Integer, ByRef TheArray() As Integer) As Integer
    Dim Index As Integer
    For Index = 0 To TheArray.GetUpperBound(0)
        If TheArray(Index) = value Then
            Return Index
        End If
    Next Index
    Return (-1) 'return -1 to indicate the value was not found
End Function

' BinarySearch()

' Perform a Binary Search for a value in an array and return
'the index, or return -1 to indicate that value was not found.
'NOTE: A binary Search requires the array to be sorted.

Private Function BinarySearch(ByVal value As Integer, ByRef TheArray() As Integer) As Integer
    Dim Low, High, Middle As Integer
    Low = 0
    High = TheArray.GetUpperBound(0)

    Do While Low <= High
        Middle = (Low + High) \
        If value = TheArray(Middle) Then 'It’s a match!!
            Return Middle
        ElseIf value < TheArray(Middle) Then 'Search the low end of array
            High = Middle - 1
        Else
            Low = Middle + 1
        End If
    Loop
    Return (-1) 'return -1 to indicate the value was not found
End Function
Selection Sort:

- Given an array of numbers, use **selection sort** to sort the array in ascending order.

- **Algorithm:**
  
  **Step 1:** Start from the first location of the array. (Start_Location = 0)

  **Step 2:** Go through the array (from Start_Location to Array_Size) and find the smallest element and replace it with the first element of the array. (Now you have the smallest number at the beginning of the array)

  **Step 3:** Increment the Start Location. Goto Step 2. Stop, when Start_Location reaches Array_Size - 1
Bubble Sort:

- Given an array of numbers, use **bubble sort** to sort the array in ascending order.
- **Algorithm:**

  Step 1: Start from the first location of the array. (Start_Location = 0)

  Step 2: Compare each two neighboring pair of elements in the array and swap them so that the smaller number appears first.

  Step 3: Repeat the process until the list is sorted.

```vbnet
' Bubble Sort()
'
Private Sub BubbleSort(ByRef TheArray() As Integer)
    Dim Pass, Index, Hold As Integer
    For Pass = 1 To TheArray.GetUpperBound(0)
        For Index = 0 To TheArray.GetUpperBound(0) - 1
            If TheArray(Index) > TheArray(Index + 1) Then
                'swap(TheArray(Index), TheArray(Index+1))
                Hold = TheArray(Index)
                TheArray(Index) = TheArray(Index + 1)
                TheArray(Index + 1) = Hold
            End If
        Next Index
    Next Pass
End Sub
```
#include <iostream.h>
#include <conio.h>
#include <math.h>

const int ARRAY_SIZE = 10;

typedef int One_D_Type[ARRAY_SIZE]; // Create a new type

void Initialize_One_D_Array(One_D_Type One_D_Array, int array_size);
void Print_One_D_Array(One_D_Type One_D_Array, int array_size);
double Calculate_Mean(One_D_Type One_D_Array, int array_size);
double Calculate_Std_Dev(One_D_Type One_D_Array, int array_size);
void Read_One_D_Array(One_D_Type One_D_Array, int array_size);

void main()
{
    One_D_Type One_D_Array;
    Initialize_One_D_Array(One_D_Array, ARRAY_SIZE);
    Print_One_D_Array(One_D_Array, ARRAY_SIZE);
    Read_One_D_Array(One_D_Array, ARRAY_SIZE);
    Print_One_D_Array(One_D_Array, ARRAY_SIZE);
    cout << "The Mean = " << Calculate_Mean(One_D_Array, ARRAY_SIZE) << endl;
    cout << "The Std Dev = " << Calculate_Std_Dev(One_D_Array, ARRAY_SIZE) << endl;
    getch();
}

void Initialize_One_D_Array(One_D_Type One_D_Array, int array_size)
{
    int i;
    cout << "Initialize One-D Array" << endl;
    for (i= 0; i< array_size; i++)
    {
        One_D_Array[i] = -1;
        cout << ".";
    }
    cout << endl;
}

void Print_One_D_Array(One_D_Type One_D_Array, int array_size)
{
    int i;
    cout << "Print One_D Array:" << endl;
    for (i= 0; i< array_size; i++)
        cout << One_D_Array[i];
    cout << endl;
}

I308 - Part 2 (Data Structures ) 8
// void Read_One_D_Array(One_D_Type One_D_Array, int array_size)
void Read_One_D_Array(One_D_Type One_D_Array, int array_size)
{
    int i;
    cout << "Read One_D Array: " << endl;
    for (i= 0; i< array_size; i++)
    {
        cout << "Enter Array[" << i << "]: ";
        cin >> One_D_Array[i];
    }
    cout << endl;
}

// double Calculate_Mean(One_D_Type One_D_Array, int array_size)
double Calculate_Mean(One_D_Type One_D_Array, int array_size)
{
    int i;
    double sum =0;
    for(i=0; i< array_size; i++)
    {
        sum = sum + One_D_Array[i];
    }
    cout << sum << endl;
    return (sum / array_size);
}

// double Calculate_Std_Dev(One_D_Type One_D_Array, int array_size)
double Calculate_Std_Dev(One_D_Type One_D_Array, int array_size)
{
    double the_std_dev;
    int index;
    double term = 0.0;
    double summation = 0.0;
    double the_mean = Calculate_Mean(One_D_Array, array_size);
    for(index = 0; index < ARRAY_SIZE; index++)
    {
        term = pow(One_D_Array[index] - the_mean, 2);
        summation = summation + term;
    }
    the_std_dev = sqrt(summation / (ARRAY_SIZE -1));
    return(the_std_dev);
}

// In probability and statistics, the standard deviation of a
// probability distribution, random variable, or population or
// multiset of values is a measure of the spread of its values.
// It is usually denoted with the letter σ (lower case sigma).
// It is defined as the square root of the variance. To understand
// standard deviation, keep in mind that variance is the average of
// the squared differences between data points and the mean. Variance
// is tabulated in units squared. Standard deviation, being the square
// root of that quantity, therefore measures the spread of data about
// the mean, measured in the same units as the data.

// Said more formally, the standard deviation is the root mean square
// (RMS) deviation of values from their arithmetic mean.

I308 - Part 2  (Data Structures )
Two Dimensional Arrays:

```cpp
int My_Two_D_Array[5][5];
```

- **Initializing the 2D Array to all '.'**
  ```cpp
  for (row = 0; row < 5; row++)
  for (col = 0; col < 5; col++)
    My_Two_D_Array[row][col] = 0;
  ```

- **Reading a 2D Array:**
  ```cpp
  for (row = 0; row < 5; row++)
  for (col = 0; col < 5; col++)
    cin >> My_Two_D_Array[row][col];
  ```

- **Printing a 2D Array:**
  ```cpp
  for (row = 0; row < 5; row++)
  for (col = 0; col < 5; col++)
    cout << My_Two_D_Array[row][col];
  ```

- **Searching a 2-D Array:**
  ```cpp
  cin >> value; // value to search for
  for (row = 0; row < 5; row++)
  for (col = 0; col < 5; col++)
    if (My_Two_D_Array[row][col] == value)
      cout << "Found the value at row " << row << " , column" << col << endl;
  ```
TWO Dimensional Arrays

#include <iostream.h>
#include <iomanip.h> // Used for setw()
#include <conio.h> // used for getch()
#include <stdlib.h> // used for srand() and rand()
#include <time.h> // used for time()

const int ROW = 5;
const int COL = 5;
typedef int Two_D_Type[ROW][COL]; // Create a new type

void Initialize_Two_D_Array(Two_D_Type Two_D_Array, int row, int col);
void Print_Two_D_Array(Two_D_Type Two_D_Array, int row, int col);
void Randomize_Two_D_Array(Two_D_Type Two_D_Array, int row, int col);

void main()
{
    Two_D_Type Two_D_Array;
    Initialize_Two_D_Array(Two_D_Array, ROW, COL);
    Print_Two_D_Array(Two_D_Array, ROW, COL);
    Randomize_Two_D_Array(Two_D_Array, ROW, COL);
    Print_Two_D_Array(Two_D_Array, ROW, COL);
    getch();
}
// Initialize Two-D Array
void Initialize_Two_D_Array(Two_D_Type Two_D_Array, int row, int col)
{
    int i, j;
    cout << "Initialize Two-D Array" << endl;
    for (i= 0; i< row; i++)
    {
        for (j= 0; j< col; j++)
            Two_D_Array[i][j] = -1;
        cout << endl;
    }
}

// Print Two-D Array
void Print_Two_D_Array(Two_D_Type Two_D_Array, int row, int col)
{
    int i, j;
    cout << "Print Two-D Array:" << endl;
    for (i= 0; i< row; i++)
    {
        for (j= 0; j< col; j++)
        {
            cout << setw(10) << Two_D_Array[i][j];
            cout << endl;
        }
        cout << endl;
    }
}

// Randomize Two-D Array
void Randomize_Two_D_Array(Two_D_Type Two_D_Array, int row, int col)
{
    int i, j;
    cout << "Load the Two-D Array with Random Numbers:" << endl;
    srand(time(NULL)); // Initialize the seed for the random number generator
    cout << RAND_MAX << endl;
    for (i= 0; i< row; i++)
    {
        for (j= 0; j< col; j++)
        {
            Two_D_Array[i][j] = rand()%100; // Rand return a random value between 0 and RAND_MAX
            cout << "Randomize is complete..." << endl;
        }
    }
    cout << endl;
}
Linked List

- A linked list is a dynamically allocated data structure.
- Data structure
- Operations

Diagram:
- Head
- LINKED LIST
- Object
- etc()
- Insert()
- Search()
- Delete()
- Print()
```cpp
#include <iostream>

#define TRUE 1
#define FALSE 0

using namespace std;

class ListNode {
public:
    char data;
    ListNode *next;

    ListNode(char mydata) {
        data = mydata;
        next = NULL;
    }
};

class LinkedList {
public:
    ListNode *head;
    int size;

    LinkedList() {
        head = NULL;
        size = 0;
    }

    int isEmpty();
    int getSize();
    void insertAtBegining(char item);
    void insertAtEnd(char item);
    void remove(char item);
    void printList();
};
```
void LinkedList::insertAtBegining(char item) {
    ListNode *newNode = new ListNode(item);
    if (isEmpty())
        head = newNode;
    else {
        newNode->next = head;
        head = newNode;
    }
    size++;
}

void LinkedList::insertAtEnd(char item) {
    ListNode *newNode = new ListNode(item);
    ListNode *tempPtr;
    if (isEmpty())
        head = newNode;
    else {
        tempPtr = head;
        while (tempPtr->next != NULL) {
            tempPtr = tempPtr->next;
        }
        tempPtr->next = newNode;
    }
    size++;
}
void LinkedList::remove(char item) {
    ListNode *tempPtr, *lastPtr, *currentPtr;
    int Found;

    if (isEmpty())
        cout << "Warning: The list is empty, item was not removed" << endl;
    else if (head->data == item) {
        tempPtr = head;
        head = head->next;
        delete(tempPtr);
        size--;
    }
    else {
        lastPtr = currentPtr = head;
        Found = FALSE;
        // while item not found and not end of list
        while (Found != TRUE && currentPtr != NULL) {
            if (currentPtr->data == item) {
                Found = TRUE;
            } else {
                lastPtr = currentPtr;
                currentPtr = currentPtr->next;
            }
        }
        if (Found == TRUE) { // Delete the item
            lastPtr->next = currentPtr->next;
            delete currentPtr;
            size--;
        } else
            cout << "Warning: The item is not found in the list" << endl;
    }
}

void LinkedList::printList() {
    ListNode *tempNode;
    if (isEmpty())
        cout << "the list is empty" << endl;
    else {
        tempNode = head;
        while (tempNode != NULL) {
            cout << tempNode->data;
            tempNode = tempNode->next;
        }
    }
    cout << endl;
}
// LinkedList::isEmpty() { 
//    return (head == NULL); 
// }

// LinkedList::getSize() { 
//    return size; 
// }

int main() { 
    LinkedList mylist;
    mylist.printList();
    cout << mylist.getSize() << endl;
    mylist.insertAtBeginning('a');
    mylist.insertAtBeginning('b');
    mylist.insertAtBeginning('c');
    mylist.printList();
    cout << mylist.getSize() << endl;
    mylist.insertAtEnd('A');
    mylist.insertAtEnd('B');
    mylist.insertAtEnd('C');
    mylist.printList();
    cout << mylist.getSize() << endl;
    mylist.remove('c');
    mylist.printList();
    cout << mylist.getSize() << endl;
    mylist.remove('A');
    mylist.printList();
    cout << mylist.getSize() << endl;
    mylist.remove('b');
    mylist.printList();
    cout << mylist.getSize() << endl;
    mylist.remove('C');
    mylist.printList();
    cout << mylist.getSize() << endl;
    mylist.remove('B');
    mylist.printList();
    cout << mylist.getSize() << endl;
    mylist.remove('Z');
    mylist.printList();
    cout << mylist.getSize() << endl;
    getchar();
    return 0;
}
Stacks

- A data structure in which elements are added and removed from only one end (LIFO). Elements are inserted and deleted from the top of the stack.

- Operations
  
  ```
  void Push(TYPE & value); // Placed an element on top of the stack.
  TYPE Pop(void); // Remove the top element of the stack.
  TYPE Peek(void); // Peek at the top element but don't POP it.
  int Empty_Stack(void); // Returns TRUE if empty
  void Print_Stack(void);
  void Reset_Stack(void);
  ```

- Implementation
  Stacks can be implemented as an array or a linked list.
// Stack.h
// ---------
// An implementation of a Stack template.
//------------------------------------------------------------------------
// Programmer: Hossein Hakimzadeh
// Date: 1/25/96
// Compiler: Borland C++ 3.1
// Last Modified: Same
// Version: 1.0
//------------------------------------------------------------------------
// Note:
// This template class expects the following operators:
//
// "=" and ">
// to be defined by the instantiating object. In addition
// the instantiating object must also define a copy constructor.
//------------------------------------------------------------------------
#include <iostream.h>
#define TRUE 1
#define FALSE 0
//------------------------------------------------------------------------
template <class TYPE>
class Stack {    // Each node can hold an element of type TYPE
    struct node {    // Each node can hold an element of type TYPE
        TYPE  Element;
        node  *Next;
    };
    node  *Top;
    int  Stack_Size;
public:
    Stack(void) { Top = NULL; Stack_Size = 0; }    // Each node can hold an element of type TYPE
    ~Stack(void); // Free the Stack
    void Push(TYPE & value);
    TYPE Pop(void);
    TYPE Peek(void);
    int  Empty_Stack(void);    // Returns TRUE if empty
    void Print_Stack(void);
    void Reset_Stack(void);
};
template <class TYPE>
Stack<TYPE>::~Stack(void)
{
    cout << "Entering Stack Destructor" << endl;
    Reset_Stack();
    cout << "Exiting Stack Destructor " << endl;
}

//------------------------------------------------------------------------
//template <class TYPE>
void Stack<TYPE>::Reset_Stack(void)
{
    cout << "Reset Stack" << endl;
    while (Empty_Stack() != TRUE) {
        Pop();
    }
}
//------------------------------------------------------------------------
//template <class TYPE>
int Stack<TYPE>::Empty_Stack(void)
{
    if (Stack_Size == 0)
        return(TRUE);
    else
        return(FALSE);
}
//------------------------------------------------------------------------
//template <class TYPE>
void Stack<TYPE>::Push(TYPE & value)
{
    cout << "Push: " << value << endl;
    node *Temp_Node = new node;
    if (Temp_Node == NULL)
        { cout << "Push: Memory Allocation Error"<< endl; exit(1);}  
    Temp_Node->Element = value;
    Temp_Node->Next = Top;
    Top = Temp_Node;
    Stack_Size++;
}
template <class TYPE>
TYPE Stack<TYPE>::Pop(void)
{
    TYPE Return_Element;
    if(Top != NULL) {
        node *Temp_Node = Top;
        Return_Element = Top->Element;
        Top = Top->Next;
        delete Temp_Node;      // delete the node
        Stack_Size--;
    }
    cout << "Pop: " << Return_Element<< endl;
    return(Return_Element);
}

template <class TYPE>
TYPE Stack<TYPE>::Peek(void)
{
    TYPE Return_Element;
    if(Top != NULL)
    {
        Return_Element = Top->Element;
        cout << "Peek: " << Return_Element<< endl;
        return(Return_Element);
    }
}

template <class TYPE>
void Stack<TYPE>::Print_Stack(void)
{
    cout << "Inside Print_Stack" << endl;
    cout << "Stack Size = " << Stack_Size << endl;
    node *Temp_Node = Top;
    while(Temp_Node != NULL) {
        cout << " " << Temp_Node->Element << endl;
        Temp_Node = Temp_Node->Next;
    }
}

Queues

- A data structure in which elements are added from one end (tail) and removed from the other end (head) of the structure (FIFO).

- Operations
  
  ```c
  void En_Q(TYPE & value); // Insert an element into the queue
  TYPE De_Q(void);       // Remove an element from the head of the queue
  int Empty_Q(void);     // Returns TRUE if queue is empty
  void Print_Q(void);
  void Reset_Q(void);
  ```

- Implementation
  Queues can be implemented as an array or a linked list.
// Queue.h
// An implementation of a queue template.

// Programmer: Hossein Hakimzadeh
// Date: 1/23/96
// Compiler: Borland C++ 3.1
// Last Modified: Same
// Version: 1.0

#include <iostream.h>
#define TRUE 1
#define FALSE 0

class Queue
{
private:
    struct node
    {
        TYPE Element;
        node *Next;
    };
    node *Q_Front;
    node *Q_Rear;
    int Q_Size;
public:
    Queue(); // Free the Queue
    ~Queue();
    void En_Q(TYPE & value);
    TYPE De_Q(void); // Returns TRUE if queue is empty
    void Print_Q(void);
    void Reset_Q(void);
};
template <class TYPE>
Queue<TYPE>::~Queue(void)
{
    cout << "Entering Queue Destructor" << endl;
    Reset_Q();
    cout << "Exiting Queue Destructor" << endl;
}

//------------------------------------------------------------------------
template <class TYPE>
void Queue<TYPE>::Reset_Q(void)
{
    cout << "Reset Queue" << endl;
    while (Empty_Q() != TRUE) {
        De_Q();
    }
}

//------------------------------------------------------------------------
template <class TYPE>
int Queue<TYPE>::Empty_Q(void)
{
    if (Q_Size == 0)
        return(TRUE);
    else
        return(FALSE);
}
template <class TYPE>
void Queue<TYPE>::En_Q(TYPE & value)
{
    cout << "En_Q: " << value << endl;
    if(Q_Front == NULL) { 
        Q_Front = new node;
        if (Q_Front == NULL) {
            cout << "En_Q: Memory Allocation Error" << endl; exit(1);
        }
        Q_Front->Element = value;
        Q_Front->Next = NULL;
        Q_Rear = Q_Front;
    }
    else {
        node *Temp_Node = new node;
        if (Temp_Node == NULL) {
            cout << "En_Q: Memory Allocation Error" << endl; exit(1);
        }
        Temp_Node->Element = value;
        Temp_Node->Next = NULL;
        Q_Rear->Next = Temp_Node;  // Make Temp_Node the last node
        Q_Rear = Temp_Node;
        Q_Size++;
    }
}

template <class TYPE>
void Queue<TYPE>::Print_Q(void)
{
    cout << "Inside Print_Q" << endl;
    cout << " Queue Size = " << Q_Size << endl;

    node *Temp_Node = Q_Front;
    while(Temp_Node != NULL) {
        cout << " " << Temp_Node->Element << endl;
        Temp_Node = Temp_Node->Next;
    }
}
template <class TYPE>
TYPE Queue<TYPE>::De_Q(void)
{
    TYPE Return_Element;
    if(Q_Front != NULL) {
        node *Temp_Node = Q_Front;
        Return_Element = Temp_Node->Element;
        Q_Front = Q_Front->Next; // delete the node
        delete Temp_Node;
        Q_Size--; // Just Dequeued the last node!
    }
    cout << "De_Q: " << Return_Element << endl;
    return(Return_Element);
}
Doubly Linked List

- Data structure
- Operations (insert, search, print, traversal, etc.)
- Implementation

![Diagram of Doubly Linked List](image-url)
Binary Trees
- Tree terminology
- Data structure
- Operations (insert, search, print, traversal, etc.)
- Implementation (Array vs. Linked)
- Traversing the tree using recursion

Let’s discuss recursion.......  We’ll come back to trees later.

Recursion
- A technique of defining a process in term of itself.

Example:
Factorial function is recursively defined as follows:

\[
\text{factorial (n)} = \begin{cases} 
1, & \text{if } n = 0 \\
 n \times \text{factorial (n-1)}, & \text{if } n > 0 
\end{cases}
\]

\[
0! = \text{factorial (0)} = 1 \\
3! = \text{factorial (3)} = 3 \times \text{factorial(2)} \\
= 3 \times 2 \times \text{factorial(1)} \\
= 3 \times 2 \times 1 \times \text{factorial(0)} \\
= 3 \times 2 \times 1 \times 1
\]

Advantage of Recursion:
- Power for solving certain types of problems (tree traversal, factorial, power, Fibonacci numbers).
- Reduces the complexity of the algorithms needed to solve problems.
- Hides the details.
Methods for verifying if a recursive functions works properly:

- The three-question method.

Ask yourself the following questions:

1) Base question

   Is there a non-recursive way out of the procedure or function, and does the routine work correctly for this “base” case? In other words can we stop the recursion?

2) Smaller Caller question:

   Does each recursive call reduces the original problem? In other words are we getting closer to a solution?

3) General Case question

   Assuming that the recursive calls are working proper, does the whole function work correctly.
Recursive Functions:

Example 1:
- A Recursive Function to determine if a value is in an array.

Boolean valueInArray ( array, value, startIndex, endIndex)
{
    if (array[startIndex] == value) // base case, value is found.
        Return (true);
    else
    {
        If (startIndex == endIndex) // base case, value is not in the list
            Return (false);
        Else
            Return(valueInArray( array, value, startIndex+1, endIndex));
    }
}

- The yellow highlight is the recursive call which reduces the problem by reducing the array size by one each time it is called.
Example 2:
- A Recursive Function to calculate factorial.

```c
int factorial ( int n)
{
    if (n == 0)   // base case
        return (1);
    else
        return(n * factorial(n-1));
}
```

- How does recursion work?
The following items are placed in the runtime stack every time the recursive function is called. 
  1) The return address 
  2) the number (n) 
  3) the result of the function call

<table>
<thead>
<tr>
<th>factorial (4)</th>
<th>Runtime Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st. Call</td>
<td>R1 (1st Recursive call)</td>
</tr>
<tr>
<td></td>
<td>4 (n)</td>
</tr>
<tr>
<td></td>
<td>? (result)</td>
</tr>
<tr>
<td>2nd. Call</td>
<td>R2 (2nd Recursive call)</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>?</td>
</tr>
<tr>
<td>3rd. Call</td>
<td>R3</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>?</td>
</tr>
<tr>
<td>4th. Call</td>
<td>R4</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>?</td>
</tr>
<tr>
<td>5th. Call</td>
<td>R5</td>
</tr>
<tr>
<td></td>
<td>0 N is equal 0, stop the recursion.</td>
</tr>
<tr>
<td></td>
<td>(Return 1)</td>
</tr>
</tbody>
</table>
Returning from recursive calls:

<table>
<thead>
<tr>
<th>factorial (4)</th>
<th>Runtime Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st. Call</td>
<td>R1</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>(Return 4 * 6)</td>
</tr>
<tr>
<td>2nd. Call</td>
<td>R2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(Return 3 * 2)</td>
</tr>
<tr>
<td>3rd. Call</td>
<td>R3</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>(Return 2*1)</td>
</tr>
<tr>
<td>4th. Call</td>
<td>R4</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(Return 1*1)</td>
</tr>
<tr>
<td>5th. Call</td>
<td>R5</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>(Return 1)</td>
</tr>
</tbody>
</table>
Binary Trees

- Tree terminology
  - **Root** is a node at the top of the tree. (Has no parent nodes)
  - Each node in a binary tree can have at most 2 children.
  - Each node in a binary tree can have only one parent.
  - **Leaf node** is a node that has no children (terminal node)
  - Two nodes are **siblings** if they have the same parent.
  - A node is an **ancestor** of another node if it is its parent or the parent of some ancestor of that node.
  - A node is an **descendant** of another node if it is a child of that node or the child of some other descendent of that node.
Level of a Node

- Level of the node is its **distance from the root**.
- Root is at level zero (0).

- The maximum number of nodes at any level (N) is \(2^N\)
  
  Max nodes at level 0 = \(2^0\)
  Max nodes at level 1 = \(2^1\)
  Max nodes at level 2 = \(2^2\)
  Max nodes at level 3 = \(2^3\)

- The maximum total number of nodes in a tree with (N) levels is \(2^{N+1} - 1\)

**Definition of a Binary Search Tree**

- A binary search tree is a binary tree in which the left child, if any, contains a smaller value than does the parent node and the right child if any, contains a larger value than does the parent node.
### Data structure

```cpp
//class TreeNode {
public:
    int data;
    TreeNode *left; // pointer to the left child
    TreeNode *right; // pointer to the right child

    TreeNode(char mydata) {
        data = mydata;
        left = NULL;
        right = NULL;
    }
};

//class BinaryTree {
public:
    TreeNode *Root; // pointer to the Head of the tree

    BinaryTree() {
        Root = NULL;
    }

    void Insert(int value);
    void Delete(int value);
    bool Search(int value);
    void Print(TRAVERSAL_ORDER); // Traversal Order can be Inorder, Preorder, or Postorder

};
```
Searching the Tree

Boolean Search (int searchValue)
{
    TreeNode *Ptr;
    Boolean ValueInTree;

    Ptr = Root;
    ValueInTree = FALSE;

    While (Ptr != NULL and ! ValueInTree)
    {
        If (Ptr.value == searchValue)
            ValueInTree = TRUE;
        else
        {
            If (Ptr.value > searchValue)
                Ptr = Ptr.left; // Go down the left subtree
            else
                Ptr = Ptr.right; // Go down the right subtree
        }
    }

    Return(ValueInTree) ;
}
Can you think of a recursive algorithm for performing the search?

Boolean Search (TreeNode * Root, int searchValue)
{
    ValueInTree = FALSE;
    If (Root.value == searchValue)
        ValueInTree = TRUE;
    else
    {
        If (Root.value > searchValue)
            Search(Root.left, searchValue);  // Recursively go down the left subtree
        else
            Search(Root.right, searchValue);  // Recursively go down the right subtree
    }
    Return(ValueInTree);
}
Inserting into a Binary Tree

1) Create a node,
2) Find the place for insertion
3) Connect the link.

- Insertion in a binary search tree is always done at a leaf node.

- The order of insertion of values determines the shape of the tree.

- If the values being inserted are sorted, the tree will be skewed. (We get a linked list instead of a tree)

- A random mix of elements will produce a shorter (bushy) tree.

- Height of the tree (max level of nodes in a tree) determines the number of comparisons in a binary search tree. In a binary tree the shape is very important!
insert (value)
{
    TreeNode *Ptr, *Back;

    TreeNode NewNode (value);

    Ptr = Root;
    Back = NULL;
    if (search(value) == FALSE) // If the value is not already in the tree, insert it.
    {
        While (Ptr != NULL) // If the value is not already in the tree, insert it.
        {
            Back = Ptr;
            if (Ptr.data > value)
                Ptr = Ptr.left;
            else
                Ptr = Ptr.right;
        }

        If (Back == NULL) // tree is empty
            Root = NewNode;
        else
            If (Back.data > value)
            {
                Back.left = NewNode;
            }
            else
            {
                Back.right = NewNode;
            }
    }
}
Can you think of a recursive algorithm for performing the insert?

Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value. In other words, we examine the root and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root.

/* Inserts the node pointed to by "newNode" into the subtree rooted at "treeNode" */
void InsertNode(Node* &treeNode, Node *newNode)
{
    if (treeNode == NULL)
        treeNode = newNode;
    else if (newNode->key < treeNode->key)
        InsertNode(treeNode->left, newNode);
    else
        InsertNode(treeNode->right, newNode);
}

http://en.wikipedia.org/wiki/Binary_search_tree
Exercise:

Insert the following into the binary search tree. Draw the tree at each step.

5, 9, 7, 3, 8, 12, 6, 4, 20
Deleting a node from a Binary Search Tree

- Deleting from the tree can be more complex than inserting, because if we delete a node with children we have to find the proper parent for the children.

- There are 3 cases which must be considered:
  1) Deleting a leaf node (no children).
  2) Deleting a node with only one child.
  3) Deleting a node with two children.
1) Deleting a leaf node (no children)
   - Simplest form of delete.
   - **Solution:** First search for the node, and then dispose of it.

2) Deleting a node with only one child.
   - **Solution:** Reconnect the links so that the node’s parent is now the one pointing to the grandchild or the child of the current node.
3) Deleting a node with two children.

- Most complicated (because we can not have a grand parent point to both its grand children.)

**Solution:** Find the node closest in value to the node being deleted and replace it with the deleted node. (e.g. find the immediate predecessor or immediate successor of the node being deleted.)

- To find the immediate predecessor, we move once to the left and then as far as we can to the right.

- To find the immediate successor, we move once to the right and then as far as we can to the left.

- After finding the immediate predecessor/successor:
  1) Replace the value in the deleted node with the replacement node.
  2) Dispose the replacement node.
Example:

Delete the node with the value 8.

1) find the immediate predecessor (7)
2) copy the value into the node being deleted.
3) delete the replacement node.
DeleteNode(value, ByRef SubtreeRoot)
{
    If (SubTreeRoot == NULL)
        Cout << “noting to delete”;
    Else {
        if (value < SubTreeRoot.data)
            DeleteNode(value, SubTreeRoot.left); // Go left
        else if (value >SubTreeRoot.data)
            DeleteNode(value, SubTreeRoot.right); // Go right
        else { // Found the element to be deleted
            if (SubTreeRoot.left == NULL) { // Only a right child
                TempPtr = SubTreeRoot;
                SubTreeRoot =SubTreeRoot.right;
                delete(TempPtr); // dispose the node
            } else if (SubTreeRoot.right == NULL) { // Only a left child
                TempPtr = SubTreeRoot;
                SubTreeRoot =SubTreeRoot.left;
                delete(TempPtr); // dispose the node
            } else { // There are two children, replace with largest value in the
                // left subtree.
                TempPtr = FindMax(SubTreeRoot.left);
                SubTreeRoot.data = TempPtr.data;
                DeleteNode(SubTreeRoot.data, SubTreeRoot.left); // delete the
                // replaced
                // value.
            }
        }
    }
}
struct Binary_Search_Tree *Bst_Delete(struct Binary_Search_Tree *node, int val)
{
    struct Binary_Search_Tree *successor, *node_delete;
    if(node)
    {
        if(node->val > val)
            node->left = Bst_Delete(node->left, val);
        else if(node->val < val)
            node->right = Bst_Delete(node->right, val);
        else
        {
            if(node->left == NULL)
            {
                node_delete = node;
                node = node->right;
                free(node_delete);
            }
            else if(node->right == NULL)
            {
                node_delete = node;
                node = node->left;
                free(node_delete);
            }
            else /* use inorder_predecessor every alternate delete for better tree balancing */
            {
                successor = inorder_successor_Bst(node->right);
                node->val = successor->val;
                node->right = Bst_Delete(node->right, successor->val);
            }
        }
    }
    return node;
}
else
{
    printf("\nNode to be deleted not found\n");
    return NULL;
}
}
Three Method of Traversing the Binary Tree:
- Inorder (LNR) (Produces a sorted list)
- Preorder (NLR)
- Postorder (LRN)

Inorder Traversal (LNR) (Produces a sorted list):

Called inorder because each node is visited in between its left and right subtrees.

Inorder (TreeNode *ptr)
{
    If (ptr != NULL)
    {
        inorder(ptr.left);  // Traverse the left subtree to print all the smaller values.
        PrintNode(ptr.data); // Print the value of this node
        inorder(ptr.right); // Traverse the right subtree to print all the larger values.
    }
}

To print the tree:

inorder(root);

Produces: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
**Preorder Traversal (N LR):**

1) Visit the root  
2) Traverse the left subtree in preorder  
3) Traverse the right subtree in preorder

```c
Preorder(TreeNode *ptr)  
{  
    If (ptr != NULL)  
    {  
        PrintNode(ptr.data); // Print the value of this node  
        preorder(ptr.left);  // Traverse the left subtree to print all the smaller values.  
        preorder(ptr.right); // Traverse the right subtree to print all the larger values.  
    }  
}
```

BINARY Search TREE

Produces: 5, 3, 2, 1, 4, 8, 6, 7, 9, 10
Postorder Traversal (LRN):

1) Traverse the left subtree in postorder
2) Traverse the right subtree in postorder
3) Visit the root

Postorder (TreeNode *ptr)
{
    If (ptr != NULL)
    {
        Postorder(ptr.left);  // Traverse the left subtree to print all the smaller values.
        Postorder(ptr.right); // Traverse the right subtree to print all the larger values.
        PrintNode(ptr.data); // Print the value of this node
    }
}

Produces: 1, 2, 4, 3, 7, 10, 9, 8, 5

BINARY Search TREE

Produces: 1, 2, 4, 3, 7, 10, 9, 8, 5
Hash Tables
Data structure
Hash Function
Collision resolution

Hashing:

- Technique used to place and/or access elements in a list in a relatively constant amount of time. \( O(1) \)

- A hashing function is a function which maps a large range of key values to a smaller range of physical addresses.

Using a hashing function to produce a unique key (most of the time).

- A method to directly transform a data element into a location where it will be stored, by computing some arithmetic function. The data structure for storing the elements is called a HASH TABLE, and the arithmetic function is called a HASH FUNCTION.
Characteristics of a Good Hash Function:

1) **Maps a key value to a home address** (a probable address) where the key may be stored or retrieved from.

2) **Evenly distributes the keys** among the addresses. (reduces the number of collisions.)

3) **Executes efficiently**
   - Easy to calculate.
   - Minimizes storage and retrieval time.

Common Hash Functions:

A function used to produce a **Semi-unique address** for a given **key**. The idea is to find a function which is **easy to compute** (takes less time), **minimizes the size of the hash table** and **minimizes collisions**.

1) Division Method

2) Folding Method

3) Truncation Method

4) Mid-Square Method

5) Radix Conversion
Division Method

**Key MOD N**

Hash value is computed by using the modulo ( % ) operator. **N is the table size.**

\[ Hash(key) = key \mod N \]

**Key MOD P**

Hash value is computed by using modulo ( % ) operation. **P is the smallest prime number >= Table_Size.** (P is then selected as the new table size)

\[ Hash(key) = key \mod P \]

Selecting The Table Size:

- Often chosen with one of the following characteristics:
  - **Power of 2** (makes the division easier, i.e. shift left instead of division)
  - **Prime number** (dividing by a prime number is more likely to produce a unique number.)
Folding Method

H is computed by partitioning the data_item into several parts, all but the last being the same length. These parts are then folded and added together to obtain the address.

<table>
<thead>
<tr>
<th>Key to be hashed:</th>
<th>123</th>
<th>456</th>
<th>789</th>
</tr>
</thead>
</table>

**Folding by Shifting:**

\[
\text{Hash}(123456789) = \begin{array}{c}
123 \\
456 \\
+789 \\
\hline
258
\end{array}
\]

(Add with no carry)

**Another Alternative:**

\[
\text{Hash(value)} = \text{v+a+l+u+e}
\]

or

\[
\text{Hash(value)} = (\text{v}^\text{e}) \oplus (\text{a}^\text{u}) \oplus \text{l} \quad // \text{fold the value onto itself}
\]
Truncation

- **AKA Sub-stringing**

- Given a key of length \( n \) extract a desired number of characters or digits from the key.

- Example:

  \[
  \text{Hash(123456789)} = 2345 \quad \text{(4 digit address extracted from the key)}
  \]
Mid-Square Method

- Hash value is computed by squaring the key and then using an appropriate number of bits (or digits) from the middle of the square to obtain the bucket address.

\[
\text{Hash}(45) = ((45 \times 45) \& 0x0ff0) \gg 4 \quad // \text{pick the middle 8 bits}
\]
Radix Conversion

- The key is considered to be in a base other than 10 and is then converted into a number in base 10.

- Example: (key = 1234 in base 11)

\[
\text{Hash}(1234) = (1 \times 11^3) + (2 \times 11^2) + (3 \times 11^1) + (4 \times 11^0) \\
= 1331 + 242 + 33 + 4 \\
= 1610
\]

- Example: (key = 129 in base 16)

\[
\text{Hash}(129_{16}) = (1 \times 16^2) + (2 \times 16^1) + (9 \times 16^0) \\
= 256 + 32 + 9 \\
= 297
\]

- After the conversion, the truncation or sub-string method can be used to select the desired number of digits.
Collisions

- When two or more keys produce the same hash address. (i.e. they are synonym keys)

Primary Clustering:

- A hash function with a large number of collision is said to exhibit primary clustering.

The Importance of Good Hash Functions:

- The goal is to find the record in one try. (Reduce collisions)

Reducing Collisions:

- Change the Hashing function
- Change the Packing Factor

Packing Factor:

- AKA Packing Density or Load Density
- The ratio of number of items stored in a file to the capacity of the file.

\[
Packing\ factor = \frac{\text{Number of records stored}}{\text{Total storage locations}}
\]

- There is a direct relationship between Packing factor and the number of collisions. (they both increase and decrease at the same time.)

- A small packing factor will result in wasted space in the file.
Collision Resolution:

- **Goal:**
  - To find a collision resolution method with the least number of probes.

1) **Chaining:**

- Allow multiple keys to hash to the same address but use a chain (linked list) to hold the overflow.

- The additional **pointer** field will **waste space**.
2) Open Addressing:

- Allow multiple keys to hash to the same address but uses no chaining. Instead the overflow records are stored after the current home address. (using a linear or quadratic probing)

- **Linear Probing**
  - **sequential probing with a wrap around.**
  - **Collisions are placed in the next available space.**
  - **Uses a linear function** $f(i) = i / \text{table_size}$
  - **First collision = 1 cell away**
  - **Second collision = 2 cells away**
  - **Third collision = 3 cells away**

- **Quadratic Probing** = Quadratic probing with a wrap around to find an empty space.
  - **Uses a Quadratic function** $f(i) = i^2 / \text{table_size}$
  - **First collision = 1^2 away**
  - **Second collision = 2^2 away**
  - **Third collision = 3^2 away**

- **No wasted space**, however, it may take many probes to find an empty space.