Bubble Sort, Insertion Sort, and Selection Sort

How they work:

Bubble Sort - Bubble sort works by using two nested for loops that sort the array by repeatedly stepping through the array. It compares each pair of adjacent items and swaps them if they are in the wrong order.

Insertion Sort - Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. For example, to build a sorted array it starts at the front looking for an element smaller than what it has already sorted. It then places it in its correct spot.

Selection Sort - The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array. For example, it would find the smallest number at place it at index 0. Then it would find the next smallest number and place it at index 1 and so on.

Time Complexity:

Bubble Sort = O(N^2)

Insertion Sort = O(N^2)

Selection Sort = O(N^2)

Bubble Sort Implementation:

```
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]);
}
```

Insertion Sort Implementation:

```
void sort(int iArray[10])
{
    int sort, temp;

    for (int count = 0; count < 10; count++)
    {
        sort = count;
        while (sort > 0 && iArray[sort] < iArray[sort - 1])
        {
                temp = iArray[sort];
                iArray[sort] = iArray[sort - 1];
                iArray[sort - 1] = temp;
                sort--;
            }
    }
}
```

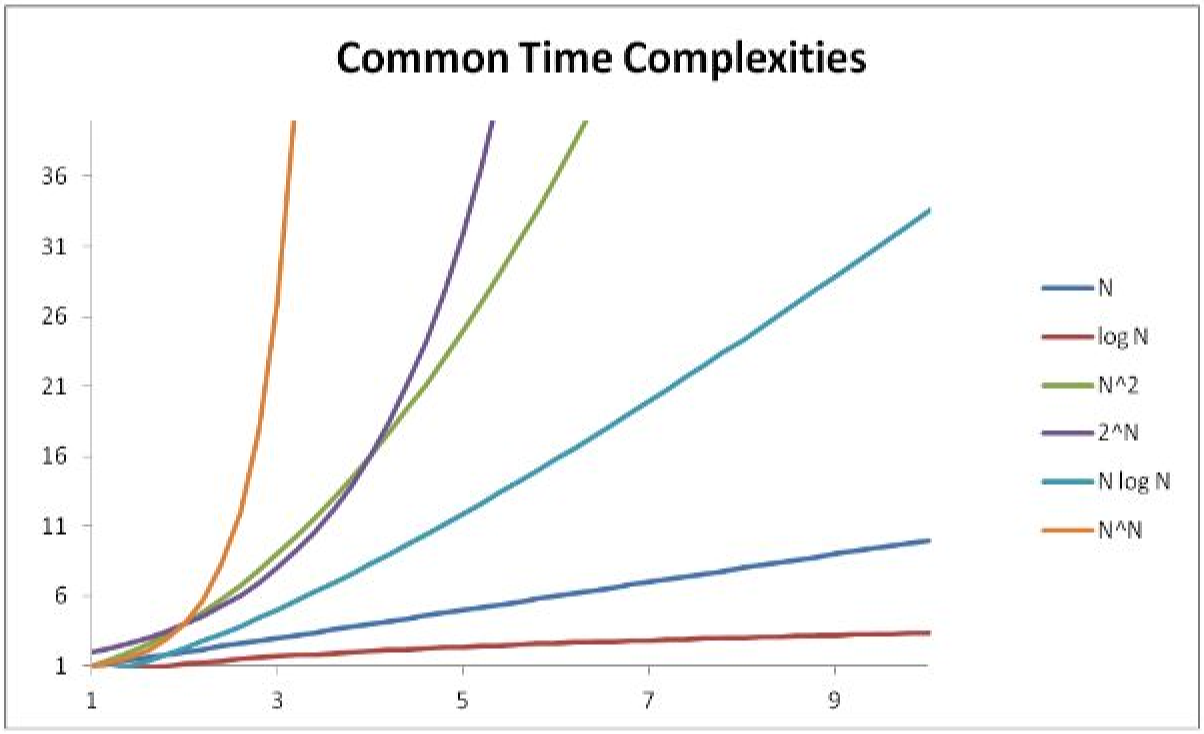
Selection Sort Implementation:

```
void sort(int iArray[10])
{

    int biggest = 0;
    int temp;
    int counter = 9;

    bool swapNeeded = false;

    for (int index = 0; index < 10; index++)
    {
        for (int count = 0; count <= counter; count++)
        {
                if (iArray[biggest] < iArray[count])
                {
                        biggest = count;
                        swapNeeded = true;

                }
        }
        if (swapNeeded = true)
        {
                temp = iArray[counter];
                iArray[counter] = iArray[biggest];
                iArray[biggest] = temp;
                counter--;
                biggest = 0;
        }
        swapNeeded = false;
    }
}
```

Time Complexity

In Computer Science we analyze the time complexity of algorithms to help us better understand the certains implications and tradeoffs between execution speed and memory size. It essentially allows us to determine how “good” an algorithm is.

Common BigO Time Complexities:



Example Algorithms with certain time complexities:

O(N) - A for loop that executes N times.
O(N^2) - A for loop nested under another for loop that executes N^2 times.
O(2^N) - A recursive function that calls 2 instances of itself every time it calls itself. Fibonacci.
Olog(N) - An algorithm that divides a number by 2 until it gets to less than or equal to 1.
ONlog(N) - An algorithm that divides a data set in half and processes each half again independently.

Arithmetic Series

$$\sum_{i=1}^{N} i = 1 + 2 + 3 + \ldots + N \qquad \sum_{i=1}^{N} = \frac{N(N+1)}{2}$$

```
int sum = 0;
for(i=1; i <=3; i++)
{
     sum = sum + i;
}
```

Geometric Series

$$\sum_{i=1}^{N} A^i = A^1 + A^2 + A^3 + \ldots + A^N \qquad \sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$$

```
int sum = 0;
for(i=0; i <= 3; i++) {
  term = 1;
}
for(j=0; j < i; j++) {
  term = term * 2;
  sum = sum + term;
}
```

Determining Time Complexity

Empirical Analysis:

Done by actually running the algorithm and placing a counter in it. See N vs execution time. Typical execution time is proportional to the number of operations performed. So we typically use number of operations as a proxy of actual time.

Rule of Thumb Analysis:

Analyze an algorithm using our learned rules of thumbs. They are…

- Ignore initialization and always assume worst case, find where the work is being done.
- Loop that executes N times = O(N)
- Nested loops = O(N^(number of nested loops))
- Biggest time complexity dominates.

Recursive Rules of Thumb:

- If N is proportional to # of boxes = O(N)
- If N is on top of tree # of boxes = O(2^N)

Queens & Maze Algorithms

How it works:

Queens - This algorithm works by placing a “queen” on a two-dimensional array that represents a chessboard. It places the queen and then checks to see if there are any collisions with any already placed queens. If so, it moves the queen to the next location. If not, it places the next queen or if all queens are placed spits out the solution.

Maze - This algorithm is given a starting locations in a maze. It determines if it can go north, south, east, or west. Moving in the first available locations it finds. It then repeats until it can longer move in which case it would “rewind” until it can move again or it has found the end of the maze.

Queens Implementation:

```
int sol[5]; //global solution "stack". Ignore cell 0
void printsolution(void)     //Solution for 4x4 Chessboard
{
       for(int i=1;i<5;i++) {
              printf("%d ,", sol[i]);
              printf("\n");
              }
}
bool cellok(int n)
{
       int i;
       // check for queens on other rows
       for(i=1;i<n;i++)
              if(sol[i] == sol[n])
                     return false;
       // check for queens on diagonals
       for(i=1;i<n;i++)
              if((sol[i] == (sol[n]-(n-i))) || (sol[i] == (sol[n] + (n-i))))
                     return false;
       return true;
}
void build(int n)
{
       int p = 1;
       // loop while there are more possible moves
       while (p <= 4) {
              // Store this move
              sol[n] = p;
              // Check if cell is okay
              if(cellok(n))
                     // is this the last column?
                     if(n == 4)
                            printsolution();
                     else
                     build(n + 1); // get the next move
              P++;
       }
}
void main(void)
{
       build(1);
}
```

Maze Implementation:

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

#define SIZE 10

class Cell {
public:
    int row = 0;
    int col = 0;
    int dir = 0;
};

Cell sol[SIZE*SIZE];

int maze[SIZE][SIZE] = {
    1,1,1,1,1,1,1,1,1,1,
    1,0,0,0,1,1,0,0,0,1,
    1,0,1,0,0,1,0,1,0,1,
    1,0,1,1,0,0,0,1,0,1,
    1,0,0,1,1,1,1,0,0,1,
    1,1,0,0,0,0,1,0,1,1,
    1,0,0,1,1,0,0,0,0,1,
    1,0,1,0,0,0,0,1,1,1,
    1,0,0,0,1,0,0,0,0,1,
    1,1,1,1,1,1,1,1,1,1
};

void build(int);
void printSolution(int);
int cellok(int);
int getNextCell(int);

void main(void)
{
    sol[0].row = 1;
    sol[0].col = 1;
    sol[0].dir = 0;

    build(0);
}

void build(int n)
{
    int p = 0;

    //cout << sol[n].row << sol[n].col << "\n"; //Line used to debugg

    while (p < 4)
    {
        int temp = getNextCell(n);

        if (temp == 0)
        {
               return;
        }

        else
        {
               if (cellok(n))
               {
                      build(n + 1);
               }

        }
        if (sol[n].row == 8 && sol[n].col == 8)
        {
               printSolution(n);
               return;
        }
        p++;
    }
}

void printSolution(int n)
{
    int i;
    cout << "\nA solution was found at:\n";
    for (i = 0; i <= n; i++)
    {
        cout << "(" << sol[i].row << ", " << sol[i].col << ")";
    }
    cout << endl << endl;
}

int getNextCell(int n)
{

    //Set initial position and direction for the next cell.
    sol[n + 1].row = sol[n].row;
    sol[n + 1].col = sol[n].col;
    sol[n + 1].dir = 0;

    //Try all positions; east, south, west, north.
    //Increment direction of current cell.
    //Increment position of next cell.

    switch (sol[n].dir)
    {
    case 0:
            sol[n].dir = 'e';
            sol[n + 1].col++;
            return 1;

        case 'e':
            sol[n].dir = 's';
            sol[n + 1].row++;
            return 1;

        case 's':
            sol[n].dir = 'w';
            sol[n + 1].col--;
            return 1;

        case 'w':
            sol[n].dir = 'n';
            sol[n + 1].row--;
            return 1;

        case 'n':
            return 0;
        }

        return 0;
    }

    int cellok(int n)
    {
        int i;

        if (maze[sol[n + 1].row][sol[n + 1].col] == 1)
        {
            return 0;
        }

        for (i = 0; i < n; i++)
        {
            if (sol[n + 1].row == sol[i].row && sol[n + 1].col == sol[i].col)
            {
                    return 0;
            }
        }
        return 1;
    }
```

Recursive Examples

Factorial by Recursion:

```
int factorial(int N) //Time complexity O(N)
{
    if (N == 0)
        return 1;
    else
        return N * factorial(N - 1);
}
```

Fibonacci by Recursion:

```
int fibonacci(int N) //Time complexity O(N^2)
{
    if (N == 0)
        return 0;
    if (N == 1)
        return 1;
    else
        return fibonacci(N - 1) + fibonacci(N - 2);
}
```

O(logN) Example:

```
void simple(int N)
{
    if (N > 0)
        simple(N / 2);
}
```

Towers of Hanoi:

```
void Hanoi(int N, int Start, int Goal, int Spare) //Time complexity O(2^N)
{
    if (N == 1)
    {
        //Move disk to goal peg.
        cout << "Move disk from peg " << Start << " to peg " << Goal << endl;
    }

    else
    {
        //Move the disks above the target disk to the spare peg.
        Hanoi(N - 1, Start, Spare, Goal);
        //Move the target disk to the goal peg.
        cout << "Move disk from peg " << Start << " to peg " << Goal << endl;
        //Move the disks from the spare peg to the goal peg.
        Hanoi(N - 1, Spare, Goal, Start);
    }
}
```

Variations of Common Linked List Algorithms

Linked List Push:

```
//Recieves a char element and appends it to the head of the list.
void push(char d)
{
    //Make a new node.
    node* p = new node;
    p->next = 0;
    p->d = d;

    //List is empty.
    if (!head)
    {
        head = tail = p;
    }

    //Append to the head end.
    else
    {
        p->next = head;
        head = p;
    }
}
```

Linked List Traverse:

```
//Traverses the list from the head to the tail, and prints out each char element.
void traverse(void)
{
    node* p = head;

    cout << "The list contains: ";
    while (p)
    {
        cout << p->d << " ";
        p = p->next;
    }
    cout << endl;
}
```

Linked List Dequeue:

```
char dq(void)
{
    node* p;
    char temp;

    //Return null of the list is empty.
    if (!head)
    {
        return -1;
    }

    //One node.
    if (head == tail)
    {
        //Remove and destroy head node.
        temp = head->d;
        delete head;
        head = tail = 0;
        return temp;
    }

    //More than one node. Remove and destroy head node.
    p = head;
    head = head->next;
    temp = p->d;
    delete p;
    return temp;
}
```

Linked List Pop:

```
//Removes a char element from the head of the list and returns it.
//Returns -1 if the list is empty.
char pop(void)
{
    node* p;
    char temp;

    //Return null of the list is empty.
    if (!head)
    {
        return -1;
    }

    //One node.
    if (head == tail)
    {
        //Remove and destroy head node.
        temp = head->d;
        delete head;
        head = tail = 0;
        return temp;
    }

    //More than one node. Remove and destroy head node.
    p = head;
    head = head->next;
    temp = p->d;
    delete p;
    return temp;
}
```

Linked List Priority Insert:

```
//Inserts an element into the list in a greater to less fashion.
void insert(char d)
{
    node* c;
    node* pc;

    //Create a new node.
    node* p = new node;
    p->d = d;

    //List is empty.
    if (!head)
    {
        head = tail = p;
        p->next = 0;
        return;
    }
    //One node.
    if (head == tail)
    {
        if (p->d >= head->d)
        {
               p->next = head;
               head = p;
               return;
        }
        else
        {
               head->next = p;
               p->next = 0;
               tail = p;
               return;
        }
    }
    //Two or more nodes.
    pc = head;
    c = head->next;

    //Found at the head.
    if (pc->d <= p->d)
    {
        p->next = head;
        head = p;
        return;
    }
    //Look at nodes after the head node.
    while (c)
    {
        //Place at tail node.
        if (c == tail)
        {
               tail->next = p;
               tail = p;
               tail->next = 0;
               return;
        }
        if (c->d <= p->d)
        {
               pc->next = p;
               p->next = c;
               return;
        }
        pc = c;
        c = c->next;
    }
    return;
}
```

Stack and Queue Algorithms

Stack - Reversing a Word:

```
//Read into stack.
while (c = readCharacter)
{
    read(c);
    push(c);
}
//Pop out of stack and print.
while (the stack is not empty)
{
    c = pop();
    print(c);
}
```

Stack - Balancing Parenthesis:

```
while (c = readCharacter)
{
    bool balanced = true;

    if (c == '(')
        push(c);

    if (c == ')')
        if (!empty)
               balanced = false;
        else
                pop();

    if (balanced == true && empty())
        return true;
    else
        return false;
}
```

Queue - Recognizing Palindromes:

```
//A word that says the same thing when read forwards or backwards.
while (c = readCharacter)
{
    push(c);
    enqueue(c);
}

while (queue is not empty && stack is not empty)
{
    if (!pop() = dequeue())
    {
        isPalindrome = false;
        return;
    }
    else
    {
        isPalindrome = true;
        return;
    }
}
```

Trees

Advantages and Disadvantages:

The primary advantage of trees is their super-efficient search time of O(logN). This is due to the fact that a data ordered or binary search tree is arranged in such a way so that data is arranged in descending or ascending values. This makes searches extremely fast since the computer only has to search one branch of the tree. One of the disadvantages can be the tree can grow lopsided if you don't have an algorithm dealing with balancing the tree. Another one is the size of some trees is huge and can take up lots of memory.

Binary Search Tree:

A tree where every parent has a maximum of 2 children. The data is ordered in such a way that the child on the left is less than the parent while the child on the right is greater than the parent or vice versa.

AVL Tree:
Also known as a height-balanced tree. In this type of tree, rotations are performed in such a way that the data is kept balanced. These rotations are performed after every insertion if necessary. Otherwise, this tree performs exactly like a binary search tree until a rotation needs to be performed. Each node has a balance of factor which equals the length of the left tree - the length of the right tree from that node. A rotation is performed if the balance on the pivot node is +-2. The pivot point is the first node up from the insertion point with a non-zero balance factor.

Rotations:

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - - ->    /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

Left, left case

```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - - ->  / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

Right, right case

```
     z                               z                          x
    / \                            /   \                       /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - - ->  /  \      - - - - - - - - -> / \    / \
T1   x                          y    T3                   T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

Left, right case

```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - - ->   /  \   - - - - - - - - -> / \    / \
   x   T4                      T2   y                  T1  T2  T3  T4
  / \                              /  \
T2   T3                          T3   T4
```

Right, left case

B-Trees:

A self-balancing tree that minimizes height. Each node of the tree is like an array.

Rules **//N is the order of the tree.**

- Root has 1 to 2N keys.
- Each child has N to 2N keys.
- For each node, the number of child nodes must be 0 or 1 more than the number of keys in the parent.

Preorder, Postorder, Inorder

Type:

Preorder - processes root, left, right or root, right, left.

Postorder - processes left, right, root or right, left, root.

Inorder - processes left, root, right or right, root, left.

Recursive Solution

```
//Preorder Traversal
void traverse(root)
{
	if(root!=NULL) {
		process(root);
		traverse(root->left);
		traverse(root->right);
	}
}
//Note just change the order of the code around to process using a different order.
```

Merge Sort, Quick Sort, Heap Sort

Merge Sort:

A divide and conquer algorithm. The array is recursively subdivided into two subarrays until the subarray reaches a size of 1. Then these subarrays are recombined in order leaving the smallest to the left and the greatest to the right or vice versa.

Merge Sort Implementation:

```
void main (void)
{
	msort(1,4);
}
void msort(int f, int l)
{
	int m;
	if (f<l) {
```

```
                m = (f+l)/2;
                msort(f,m);
                msort(m+1,l);
                merge(f,m,l);
        }
}
void merge(int f, int m, int l)
{
        int t1, t2, t3;
        t1=f; t2=m+1; t3=f;
        // compare elements in smaller arrays,
        // copy smaller element to tmp,
        // advance tmp and the copied element's array
        while(t1<=m && t2 <= l) {
                if(A[t1]<A[t2])
                        tmp[t3++]=A[t1++];
                else
                        tmp[t3++]=A[t2++];
        }
        // we are at the end of one of the arrays,
        // so copy the remaining elements in the other array to tmp
        // (only one loop will execute)
        while(t1<=m)
                tmp[t3++]=A[t1++];
        while(t2<=l)
                tmp[t3++]=A[t2++];
        //copy the sorted tmp back to A
        for(t1=f;t1<=l;t1++)
                A[t1]=tmp[t1];
}
```

Quick Sort:

A divide and conquer algorithm. The array is recursively subdivided into two subarrays. Each sort subdivides the array by finding the final position of the middle element. The subarrays on each side of the middle element are then sorted. The recursion continues until a subarray of 1 is obtained.

Quick Sort Implementation:

```
void qsort(int f, int l)
{
    int m;
    if (f <= l) {
        m = part(f, l);
        qsort(f, m-1);
        qsort(m + 1, l);
    }
}
```

```
int part(int f, int l)
{
    int P1, P2, PV;
    PV = A[f];
    P1 = f + 1;
    P2 = l;
    while (P1 <= P2) {
        while (A[P1] <= PV && P1 <= l)
               P1++;
        while (A[P2] > PV)
               P2--;

        if (P1 < P2)
               swap(P1, P2);
        else
               swap(f, P2);
    }
    return P2;
}
```

Heap Sort:

This algorithm works by dumping the array to be sorted into a binary heap. Which always contains the smallest element on top (depending on implementation). To perform a heap sort you merely fill up the heap and then empty it. The elements will then be sorted.

Time Complexity:

Quick Sort - Best case O(NlogN), Worst-case O(N^2).
Merge Sort - All cases O(NlogN).
Heap Sort - All cases O(NlogN).

Hashing

What is it:

Hashing is a constant time O(1) algorithm which is extremely efficient for insertions and searches. The only problem being it takes up a large space of memory. In hashing a key is generated for each set of data that is unique to that data. It is then placed in the corresponding location in the array where it is stored. Unfortunately. Hashing is not perfect and collisions can take place.

Collision Avoidance - Chaining:

This collision avoidance procedure chains data onto the node where the data should have gone if that spot in the array was already occupied. It works very much like a linked list.

Collision Avoidance - Linear Rehashing:
This collision avoidance procedure steps data by 1 if a collision would have occurred. This prevents two pieces of data being stored in the same locations. Unfortunately, this method has a tendency to group data which is unhealthy for hashing.

Collision Avoidance - Generalized Linear Rehashing:
This collision avoidance procedure steps by some arbitrary value other than 1. This helps prevent data from being clumped but is still not the best solution.

Collision Avoidance - Double Hashing:
This collision avoidance procedure produces another key from the generated key that defines the step value if a collision would have occurred. This is by far the most efficient method of dealing with collisions in hashing.

Perfect Hashing:
This type of hashing can only take place when the data that is going to be received is already known. This allows the programmer to create a perfect hashing algorithm or lookup table that gives each piece of data a unique key so that collisions are impossible.

Difference Between table-driven Hash and Algorithmic Hash:
A table-driven hash relies on a look-up table to determine the keys for a given set of data while the algorithmic hash uses a special algorithm or function that does not rely on a look-up table but instead produces a unique key for every piece of data just via the algorithm. The pros of a look-up table based perfect hashing tools versus an algorithmic are that it is simpler to devise. The con is that it requires an additional set of data to work properly. An algorithmic hash has the con of being difficult to devise but ultimately superior since it does not rely on another set of data.

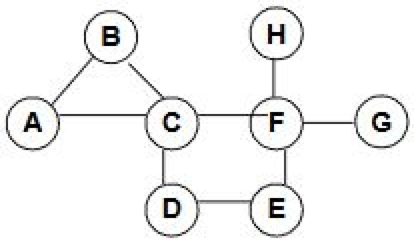
Graphs

What is it:
A graph is a generalized data structure. Most other data structures are just special cases of graphs. A graph consists of nodes and links (aka vertices and edges), organized in varying ways.

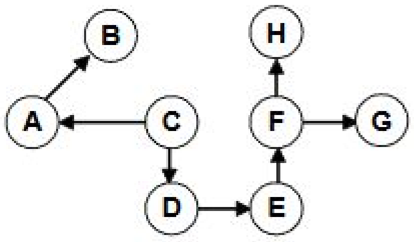
Types:
Undirected Graphs - No directions associated with the links between nodes.
Directed Graphs - A direction associated with the links between nodes.
Weighted Graphs - There is a cost (or weight) associated with each edge.

Stack Traversal:

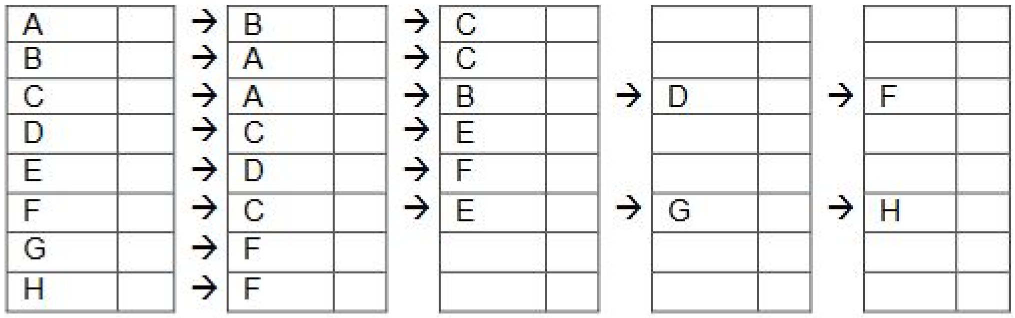
A stack traversal is a depth first traversal.



Graph



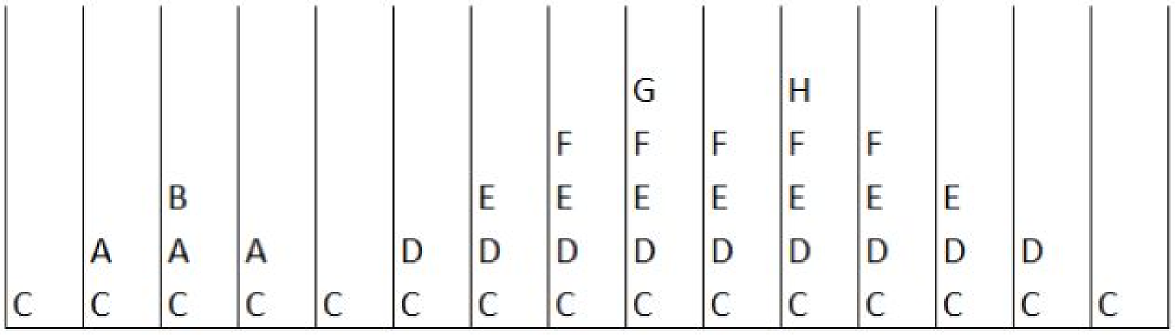
Resulting Tree



A		→	B		→	C							
B		→	A		→	C							
C		→	A		→	B		→	D		→	F	
D		→	C		→	E							
E		→	D		→	F							
F		→	C		→	E		→	G		→	H	
G		→	F										
H		→	F										

Rules:

- Pick any starting node.
- Push node.
- Select a neighbor of the top of the stack, ignoring previously selected neighbors.
- Push neighbor.
- If at the end of a list, goto 6, otherwise goto 3.
- Pop
- If stack is empty, we are done, otherwise, goto 3.

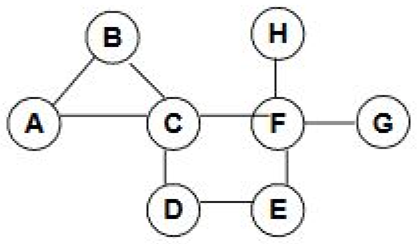


								G		H				
							F	F	F	F	F			
		B				E	E	E	E	E	E	E		
	A	A	A		D	D	D	D	D	D	D	D	D	
C	C	C	C	C	C	C	C	C	C	C	C	C	C	C

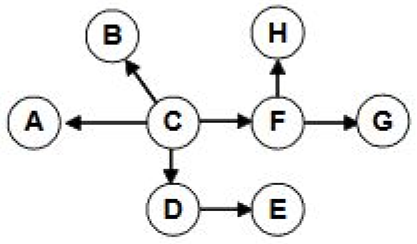
Queue Traversal:
A queue traversal is a width first traversal.

Rules:

- Pick a starting target node.
- Enqueue target node.
- Enqueue any unmarked neighbors of the target node.
- Dequeue a node.
- If queue is empty, we are done, otherwise, make the next node in the queue the target and goto 3.



Graph



Resulting Tree

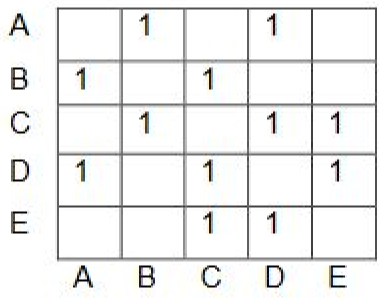
												C
								F	D	B	A	C
								F	D	B	A	
								F	D			
							E	F	D			
							E	F				
					H	G	E	F				
					H	G	E					
					H	G						
					H							

Stack Traversal Implementation:

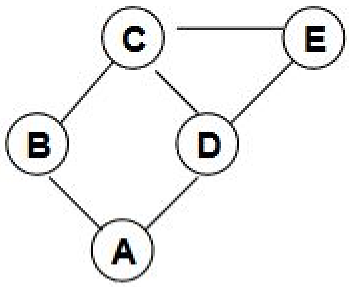
```
//Based on code by Professor Ross. Modified by Quinn Roemer.
#include <iostream>
#include <string>

using namespace std;

int graph[8][8] = {
    0, 1, 1, 1, 0, 0, 1, 0,         //A
    1, 0, 1, 0, 1, 0, 1, 0,         //B
    1, 1, 0, 0, 0, 1, 0, 1,         //C
    1, 0, 0, 0, 0, 1, 0, 0,         //D
    0, 1, 0, 0, 0, 0, 0, 1,         //E
    0, 0, 1, 1, 0, 0, 1, 1,          //F
    1, 1, 0, 0, 0, 1, 0, 1,          //G
    0, 0, 1, 0, 1, 1, 1, 0           //H
};
//    A B C D E F G H

// where I've been
bool visited[] = { false, false, false, false, false, false, false, false };

// the resulting tree.  Each node's parent is stored
int tree[] = { -1, -1, -1, -1, -1, -1, -1, -1 };

//Function Prototypes.
void traverse(int);
void printNode(int);
void printTree();

int main()
{
    cout << "Traversal path:" << endl;

    //Calling traverse: Starter node = A.
    traverse(0);

    //Printing the resulting tree.
    printTree();
}

void traverse(int startValue)
{
    int nextNode = 0;

    visited[startValue] = true;

    //Printing the current node.
    printNode(startValue);

    //Finding an unvisited node to go to next.
    while (nextNode < 8)
    {
        if (visited[nextNode] == false && graph[startValue][nextNode] == 1)
        {
               //Recording the parent of this node.
               tree[nextNode] = startValue;

               //Recursively calling traverse.
               traverse(nextNode);
        }
        nextNode++;
    }
    //Debug line
    //cout << "Returning from node #" << startValue << endl;

}

void printTree()
{
    char letter = 'A';
    char parent;

    cout << "\nThe resulting tree:" << endl << endl;
    cout << "Node \t\t Parent" << endl;

    for (int i = 0; i < 8; i++)
    {
        parent = tree[i] + 'A';

        if (parent == '@')
        {
                parent = ' ';
        }

        cout << " " << letter << "\t\t   " << parent << endl;
        letter++;
    }
}

void printNode(int startValue)
{
    char letter = startValue + 'A';
    cout << letter << endl;
}
```

<u>Adjacency Matrix:</u>
One of the two common ways to represent a graph in code or on paper. This version contains a set of edges on a table with those the nodes is connected too.

Example:



A		1		1	
B	1		1		
C		1		1	1
D	1		1		1
E			1	1	
	A	B	C	D	E

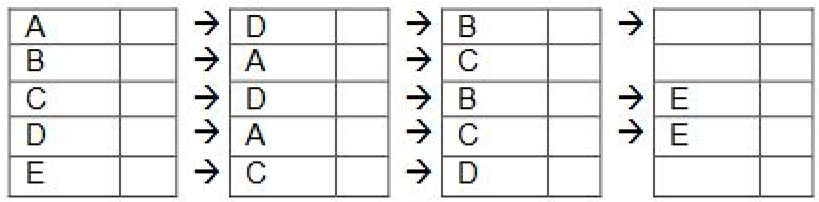


Note: If you wanted to add weights to the graph merely make the weight a subscript of the edge.

Time Complexity: The worst case time is O(N^2) when each cell must be visited.

<u>**Adjacency List:**</u>
This is the second common way to represent a graph on paper. This contains several linked lists once for each node connected together in such a way so that it describes the graph.

Example:



A	→	D	→	B	→	
B	→	A	→	C		
C	→	D	→	B	→	E
D	→	A	→	C	→	E
E	→	C	→	D		

Time Complexity: For 6 edges there are 12 steps required. So the worst case time complexity is O(2N) = O(N).

Which is better:
A matrix is good for dense (lots of edges) graphs.
A list is good for sparse graphs.

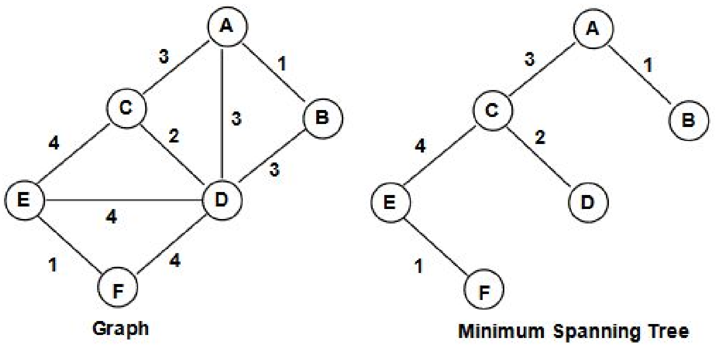
<u>Prim's, Dijkstra's (OSPF), Distance Vector (RIP) Algorithms</u>

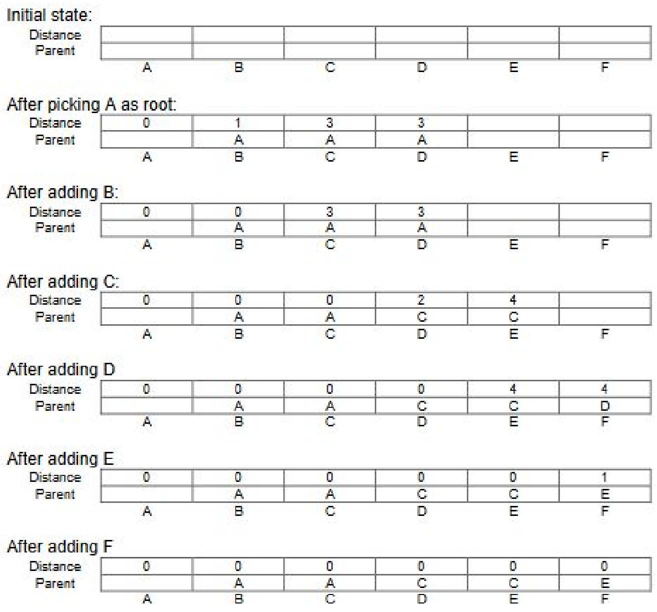
<u>Prim's Algorithm:</u>
This algorithm can convert a weighted graph into a minimum spanning tree. There are many solutions for which the total weight will be minimized. This algorithm is given a node on which

it starts. Once the node is in the tree the distance is set to 0. This is known as a greedy algorithm.

Rules:

- Pick an arbitrary root to add to the tree.
- Identify the candidate nodes and for each candidate…
  - Update the distance to the nearest tree node.
  - Update the nodes parent.
- Pick a new node having the shortest distance and add it to the tree.
- If all distance are zero, we are done, otherwise, goto 2.





Initial state:

	A	B	C	D	E	F
Distance						
Parent						

After picking A as root:

	A	B	C	D	E	F
Distance	0	1	3	3		
Parent		A	A	A		

After adding B:

	A	B	C	D	E	F
Distance	0	0	3	3		
Parent		A	A	A		

After adding C:

	A	B	C	D	E	F
Distance	0	0	0	2	4	
Parent		A	A	C	C	

After adding D

	A	B	C	D	E	F
Distance	0	0	0	0	4	4
Parent		A	A	C	C	D

After adding E

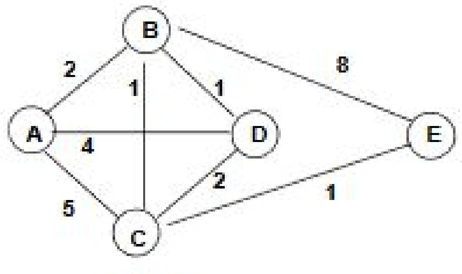
	A	B	C	D	E	F
Distance	0	0	0	0	0	1
Parent		A	A	C	C	E

After adding F

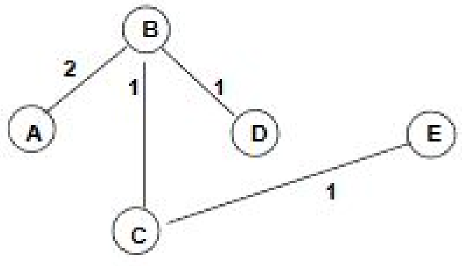
	A	B	C	D	E	F
Distance	0	0	0	0	0	0
Parent		A	A	C	C	E

Dijkstra's (OSPF) Algorithm:

In this algorithm each node knows all the link costs to every node in the network. This algorithm creates a routing table for the sent node that is the most efficient (cheapest) path to each other node in the tree. This is also known as a greedy algorithm.



Graph



Shortest Path Tree for node A

Data structure for Dijkstra's algorithm for node A

InTree	x				
Distance	0				
NextHop	A				
	A	B	C	D	E

Rules:
1) Put node with the cheapest distance into the tree
2) Update Distances and NextHop
3) Continue until all nodes are in tree

Update distances to neighbors, and the next hop

InTree	x				
Distance	0	2	5	4	
NextHop	A	B	C	D	
	A	B	C	D	E

Put B in tree, update distances and hops

InTree	x	x			
Distance	0	2	3	3	10
NextHop	A	B	B	B	B
	A	B	C	D	E

Put C in tree, update

InTree	x	x	x		
Distance	0	2	3	3	4
NextHop	A	B	B	B	B
	A	B	C	D	E

Put D in tree, update

InTree	x	x	x	x	
Distance	0	2	3	3	4
NextHop	A	B	B	B	B
	A	B	C	D	E

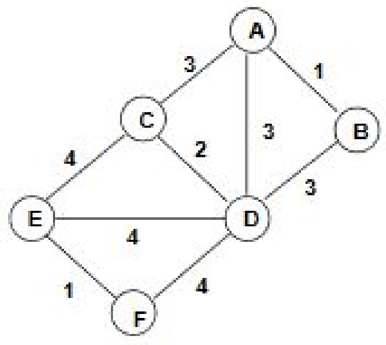
Put E in tree, done

InTree	x	x	x	x	x
Distance	0	2	3	3	4
NextHop	A	B	B	B	B
	A	B	C	D	E

<u>Distance Vector (RIP) Algorithm:</u>

Initially in this algorithm each node only knows the distance to its immediate neighbors. Periodically, each node sends info it knows about the network to its immediate neighbors. Each node then uses this info to expand and optimize its own routing table.

Eventually (hopefully) each node will create the most efficient table to navigate the network.



Graph

Rules:
1) Begin with the Distance and NextHop info for immediate neighbors
2) Use info from each immediate neighbor to check for cheaper paths to other nodes
3) If one of the nodes' path info changes, send this info to nodes' neighbors, goto 2.
4) Done

- assume that the system is "synchronized timer driven" – that is, each node uses data from the *previous cycle* to update its table
- in reality, the system may be event driven, with the latest data always being used, but this is harder to do a clean hand execution on

Information Stored at Each Node (initially)

Information stored at node:	NextHop and Distance to reach node:					
	A	B	C	D	E	F
A	0	B 1	C 3	D 3	-	-
B	A 1	0	-	D 3	-	-
C	A 3	-	0	D 2	E 4	-
D	A 3	B 3	C 2	0	E 4	F 4
E	-	-	C 4	D 4	0	F 1
F	-	-	-	D 4	E 1	0

After each node looks at its neighbor's info

Information stored at node:		NextHop and Distance to reach node:					
		A	B	C	D	E	F
A	X	0	B 1	C 3	D 3	C 7	D 7
B	X	A 1	0	A 4	D 3	D 7	D 7
C	X	A 3	A 4	0	D 2	E 4	E 5
D		A 3	B 3	C 2	0	E 4	F 4
E	X	C 7	D 7	C 4	D 4	0	F 1
F	X	D 7	D 7	E 5	D 4	E 1	0

Done. System is stable, because if you continued to solve for each cell, you would find that all paths are optimal (cell values stop changing).