Data Structures

LECTURE NOTES

by
Dan Ross

10th Edition

Table of Contents

Table of Contents **2**

Introduction **4**

What is data structures? *4*

Mathematics Review **5**

Exponents and Logarithms *5*
Series *5*

Algorithm Analysis **7**

Why Analyze Algorithms? *7*
Math Background *7*
Empirical Testing *7*
Running Time Analysis *7*

Recursion **14**

Introduction *14*
Examples *15*

Linked Lists **23**

Fundamentals of Linked Lists *23*
A Linked List Toolkit *24*
Static Arrays, Dynamic Arrays, Linked Lists, Doubly Linked Lists *27*
Cursors *27*

Stacks **30**

Introduction *30*
Stack Implementation *30*
Stack Applications *30*

Queues **33**

Introduction *33*
Queue Implementation *33*
Queue Applications *33*
Infix Evaluator Lab *33*

Trees **39**

Introduction *39*
Tree Implementations *39*
Tree Traversals *39*
Data Ordered Binary Tree (aka Binary Search Tree) *41*
AVL Tree (aka Height-Balanced 1-Tree) *42*
B- Trees of order N *43*
Binary Heaps *44*

Sorting **45**

Introduction *45*
BubbleSort *45*
SelectionSort *46*
InsertionSort *47*
MergeSort *48*
QuickSort *50*
HeapSort *52*
Summary *53*

Searching/Hashing **54**

Introduction *54*
Hashing *54*
Collisions *55*
Perfect Hashing *57*

Graphs **58**

Definitions *58*
Graph Applications *58*
Graph Implementations *59*
Graph Traversals *60*
Minimum Spanning Tree *64*
Internet Routing Table Algorithms (Cheapest Paths) *69*

Encryption **74**

Basics *74*
Private Key Encryption *77*
Public Key Encryption *78*

State Machines **80**

Introduction

What is data structures?

- A recipe book - the study of the historical ways to organize data and the corresponding algorithms that manipulate such data
- understand complexity of algorithms - in machine terms (time and space) not in human terms
- learn analysis and hand-execution techniques that allow us to overcome the limits of our raw intelligence and allow us to solve more difficult problems
- opens up a whole new world of techniques and analysis skills - makes one a better programmer
- seperates the “Computer Scientist” from the “Programmer” or the “Engineer Who Can Code a Little” - gets you into the “club”

Mathematics Review

Exponents and Logarithms

Definition:
$X^A = B$ if and only if $\log_X B = A$

in CS, all logs are base 2 unless otherwise stated
skip the fancy stuff…we can do a lot of hand waving in CS

Series

Arithmetic Series

$$\sum_{i=1}^{N} i = 1 + 2 + 3 + ... + N$$

Formula for sum of the first N terms of an arithmetic series:

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

example:

$$\sum_{i=1}^{3} = 1 + 2 + 3 = 6 \ = 1$$

$$\sum_{i=1}^{3} i = \frac{3(3+1)}{2} = \frac{3(4)}{2} = \frac{12}{2} = 6$$

the above example in code:

```
int sum = 0;

for(i=1; i <=3; i++)
      sum = sum + i;
```

Geometric Series

$$\sum_{i=0}^{N} A^i = A^1 + A^2 + A^3 + ... + A^N$$

Formula for the sum of the first N terms of a geometric series:

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$$

example:

$$\sum_{i=0}^{3} 2^i = 2^0 + 2^1 + 2^2 + 2^3 = 1 + 2 + 4 + 8 = 15$$

$$\sum_{i=0}^{3} 2^i = \frac{2^{3+1} - 1}{2 - 1} = \frac{2^4 - 1}{1} = \frac{16 - 1}{1} = 15$$

the above example in code:

```
int sum = 0;

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

So...
N represents the “size of the problem”

In general, the geometric series will give a larger answer than the arithmetic series will for a given N. This is even more obvious for larger values of N.

We say that the geometric series **“grows faster”** and has a **“bigger time complexity”** than the arithmetic series.

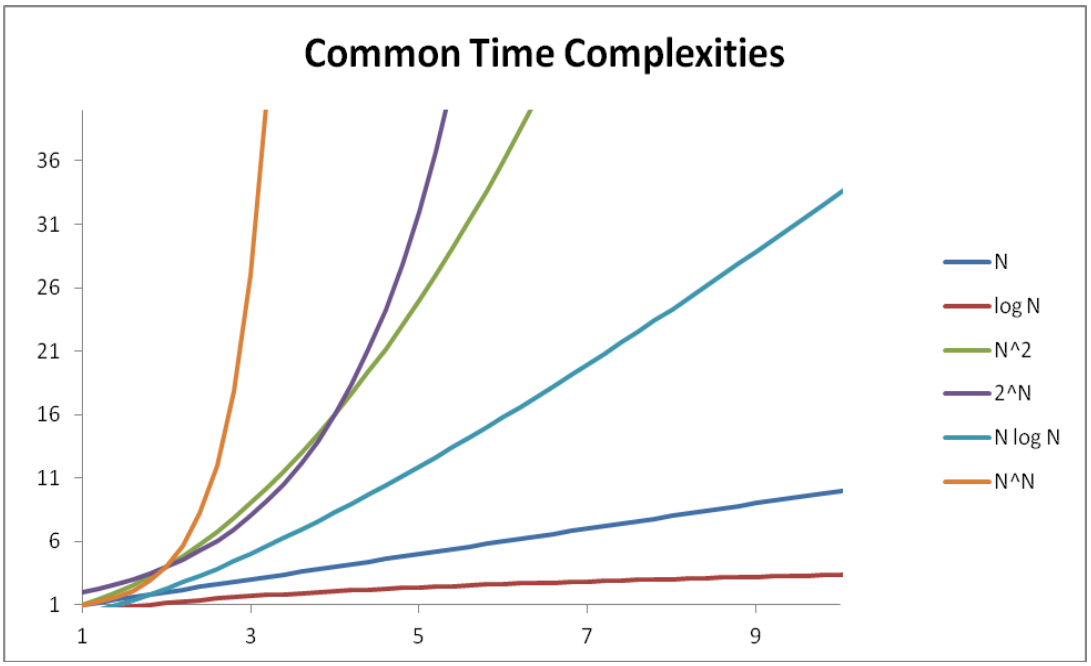
Algorithm Analysis

Why Analyze Algorithms?

- To better understand the implication and tradeoffs between **Time** (execution speed) and **Space** (memory size)
- Help us determine how “good” an algorithm is
- So we can use analysis instead of empirical methods. Analogy: So we don't have to make the 2 hour drive to San Francisco - use a map to determine the time instead

Math Background

We are usually interested of the upper bound of the rate of growth of an algorithm
"Big Oh" notation indicates the "upper bound" of the rate of growth



Empirical Testing

- Actually run the algorithm and measure how long it takes to execute for various input values, N.
- Graph N vs “Execution Time” and compare the shape of the curve to known shapes.
- Typically, execution time is proportional to the number of operations performed, so we typically use number of operations as a proxy for the actual time
- Demonstrate a 2^n algorithm such as Towers of Hanoi or Fractal Dragon
- Problem: It takes a looooooong time! In fact, some algorithms are designed to **intentionally** take an “infinite” amount of time, practically speaking (you will be dead before they finish executing.) This is the basis for many encryption techniques.

Running Time Analysis

Do not actually have to run the code, just analyze it instead.
Takes some practice and experience.

O(n) example:

```
int sum = 0;

for(i=1; i <=3; i++)
      sum = sum + i;
```

O(n^2) example:

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

Shortcuts

Ignore the simple constant time O(1) lines, such as
- initializations
- lines that execute only a few times
- tests that are constant time
- etc

Rules (Recursively defined)

1) For loops:
   the time for the statements inside the loop times the loop size (number of iterations)
   note the recursive nature: the stuff inside may also be loop(s)
2) consecutive statements
   only the largest counts
3) If/else
   the time of the test plus the time of the largest branch

Recursion

- sometimes recursions are really thinly veiled loops – the next example is simply **O(N)**

```
long int
Factorial(int N)
{
      if(N<=1)
            return 1;
      else
            return N * Factorial(N-1);
}
```

- we typically need more sophisticated analysis to determine the time complexity or recursive statements. We will spend some time on this.

- other times recursion requires **iterative expansion** math techniques – beyond the scope of this course

Some Example Algorithms and Their Time Complexities

- Here are 3 different solutions to the same problem, each solution has a different time complexity

Analysis of Maximum Subsequence Sum problem

```
#include <stdio.h>

/* Define one of CubicAlgorithm, QuadraticAlgorithm, NlogNAlgorithm,
 * or LinearAlgorithm to get one algorithm compiled */

#define NlogNAlgorithm

#ifdef CubicAlgorithm  O(N^3)

/* START: fig2_5.txt */
        int
        MaxSubsequenceSum( const int A[ ], int N )
        {
            int ThisSum, MaxSum, i, j, k;

/* 1*/      MaxSum = 0;
/* 2*/      for( i = 0; i < N; i++ )
/* 3*/          for( j = i; j < N; j++ )
                {
/* 4*/              ThisSum = 0;
/* 5*/              for( k = i; k <= j; k++ )
/* 6*/                  ThisSum += A[ k ];

/* 7*/              if( ThisSum > MaxSum )
/* 8*/                  MaxSum = ThisSum;
                }
/* 9*/      return MaxSum;
        }
/* END */

#endif
```

```
#ifdef QuadraticAlgorithm  O(N^2)

/* START: fig2_6.txt */
        int
        MaxSubsequenceSum( const int A[ ], int N )
        {
            int ThisSum, MaxSum, i, j;

/* 1*/      MaxSum = 0;
/* 2*/      for( i = 0; i < N; i++ )
            {
/* 3*/          ThisSum = 0;
/* 4*/          for( j = i; j < N; j++ )
                {
/* 5*/              ThisSum += A[ j ];

/* 6*/              if( ThisSum > MaxSum )
/* 7*/                  MaxSum = ThisSum;
                }
            }
/* 8*/      return MaxSum;
        }
/* END */



#endif
```

```
#ifdef NlogNAlgorithm  O(NlogN)

        static int
        Max3( int A, int B, int C )
        {
            return A > B ? A > C ? A : C : B > C ? B : C;
        }

/* START: fig2_7.txt */
        static int
        MaxSubSum( const int A[ ], int Left, int Right )
        {
            int MaxLeftSum, MaxRightSum;
            int MaxLeftBorderSum, MaxRightBorderSum;
            int LeftBorderSum, RightBorderSum;
            int Center, i;

/* 1*/      if( Left == Right )  /* Base case */
/* 2*/          if( A[ Left ] > 0 )
/* 3*/              return A[ Left ];
                else
/* 4*/              return 0;

/* 5*/      Center = ( Left + Right ) / 2;
/* 6*/      MaxLeftSum = MaxSubSum( A, Left, Center );
/* 7*/      MaxRightSum = MaxSubSum( A, Center + 1, Right );

/* 8*/      MaxLeftBorderSum = 0; LeftBorderSum = 0;
/* 9*/      for( i = Center; i >= Left; i-- )
            {
/*10*/          LeftBorderSum += A[ i ];
/*11*/          if( LeftBorderSum > MaxLeftBorderSum )
/*12*/              MaxLeftBorderSum = LeftBorderSum;
            }

/*13*/      MaxRightBorderSum = 0; RightBorderSum = 0;
/*14*/      for( i = Center + 1; i <= Right; i++ )
            {
/*15*/          RightBorderSum += A[ i ];
/*16*/          if( RightBorderSum > MaxRightBorderSum )
/*17*/              MaxRightBorderSum = RightBorderSum;
            }

/*18*/      return Max3( MaxLeftSum, MaxRightSum,
/*19*/                   MaxLeftBorderSum + MaxRightBorderSum
);
        }

        int
        MaxSubsequenceSum( const int A[ ], int N )
        {
            return MaxSubSum( A, 0, N - 1 );
        }
/* END */


#endif
```

```
#ifdef LinearAlgorithm  O(N)
/* START: fig2_8.txt */
        int
        MaxSubsequenceSum( const int A[ ], int N )
        {
            int ThisSum, MaxSum, j;

/* 1*/      ThisSum = MaxSum = 0;
/* 2*/      for( j = 0; j < N; j++ )
            {
/* 3*/          ThisSum += A[ j ];

/* 4*/          if( ThisSum > MaxSum )
/* 5*/              MaxSum = ThisSum;
/* 6*/          else if( ThisSum < 0 )
/* 7*/              ThisSum = 0;
            }
/* 8*/      return MaxSum;
        }
/* END */


#endif

main( )
{
    // the answer is 20, elements 11 through 13
    static int A[ ] = { -2, 11, -4, 13, -5, -2 };

    printf( "Maxsum = %d\n",
            MaxSubsequenceSum( A, sizeof( A ) / sizeof( A[ 0 ] ) )
);
    return 0;
}
```

Recursion

Introduction

Recursion

- a function that calls itself
- sometimes done by accident, (so we need to be able to recognize it)
- sometimes done on purpose - can be used to simplify complex

Infinite Recursion
- function calls itself in an infinite loop.
- will get "stack overflow" or "out of resources" error from the operating system.

Ex) Infinite Recursion

```
void main(void)
{
        f1();
}
void f1()
{
        f1();
}
```

Mutual Recursion
f1 calls f2 calls f1 calls f2 ...

Ex) Mutual Recursion

```
// A not so obvious case of recursion
// This is easy to do by accident
void main(void)
{
        f1();
}
void f1()
{
        f2();
}
void f2()
{
        f1();
}
```

Memory Usage
- there is only one copy of the function's instructions
- a new "instance" of the function is created in memory. (like starting notepad several times)
- a set of all local variables are created on the stack for each instance
- therefore infinite recursion uses up the whole stack

Intentional uses of recursion
Function MUST have a conditional to control whether the function will call itself again or not.

Examples

Ex) Simple Example1

//A simple recursion example with a conditional

```
void f(int n)      // 𝒪(N)
{
      n--;
      if(n>0)
            f(n);
}
```

Ex) Simple Example2

```
void f(int n)       // 𝒪(log N)
{
      if(n>0) {
            f(n/2);
            cout << n%2 << endl;
      }
}

int main(void)
{
      f(5);
}
/*
1
0
1
*/
```

Ex) Factorial (Trivial, a loop in disguise)

```
int fac(int n)     // you determine 𝒪( ) ☺
{
      if(n==0)
            return 1;
      else
            return n * fac(n-1);
}

int main(void)
{
      cout << fac(3);
}
```

Ex) Fibonacci Numbers (Bad use of recursion)

```
int fib(int n)     // you determine 𝒪( ) ☺
{
      if(n==0)
            return 0;
      if(n==1)
            return 1;
      else
            return fib(n-1) + fib(n-2);
}

int main(void)
```

```
{
	cout << fib(6);
}
```

Ex) Efficient Exponentiation

- the “obvious” algorithm in which we simply multiply a var times itself in a loop is O(N)
- this alternative algorithm is O(logN), for example:

$$X^{62} = (X^2)^{31}, X^{31} = (X^2)^{15} X, X^{15} = (X^2)^7 X, X^7 = (X^2)^3 X, X^3 = (X^2)^1 X$$

```
#include <stdio.h>

#define IsEven( N ) ( ( N ) % 2 == 0 )

/* START: fig2_11.txt */
        long int
        Pow( long int X, unsigned int N )
        {
/* 1*/      if( N == 0 )
/* 2*/          return 1;
/* 3*/      if( N == 1 )
/* 4*/          return X;
/* 5*/      if( IsEven( N ) )
/* 6*/          return Pow( X * X, N / 2 );
            else
/* 7*/          return Pow( X * X, N / 2 ) * X;
        }
/* END */

main( )
{
    printf( "2^21 = %ld\n", Pow( 2, 21 ) );
    return 0;
}
```

Ex) msort algorithm

```
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);
      }
}
```

Ex) Fractal Dragon

```
main()
{
   int Depth;

   cout << "Enter the depth of recursion for your fractal (0..18):
";
```

```
   cin >> Depth;

   // get started with two lines between three points
   Dragon(600, 200, 400, 100, 200, 200, Depth);
}

void Dragon (double X1, double Y1, double X2, double Y2, double X3,
double Y3, int Level)
{
   if (Level <= 0) {
      line(X1, Y1, X2, Y2);
      line(X2, Y2, X3, Y3);
   }
   else {
      // find poke in point
      double X4 = (X1 + X3) / 2;
      double Y4 = (Y1 + Y3) / 2;
      // subdivide line
      Dragon(X2, Y2, X4, Y4, X1, Y1, Level - 1);

      // find poke out point
      double X5 = X3 + X2 - X4;
      double Y5 = Y3 + Y2 - Y4;
      // subdivide line
      Dragon(X2, Y2, X5, Y5, X3, Y3, Level - 1);
   }
}
```

Ex) Towers of Hanoi

```
void main()
{
      int Rings;

      cout << "Enter the number of rings and I'll play Towers of
Hanoi: ";
      cin >> Rings;

      // start the solution here
      Hanoi(Rings, 1, 3, 2);

}

void Hanoi (int N, int Start, int Goal, int Spare)
{
      if (N == 1)
            // move the disk to the 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);
        }
    }
```

Ex) The N-Queens problem

```
    #include "stdio.h"

    int sol[9];

    void printsolution(void)
    {
        for(int i=1;i<9;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 <= 8) {

        // Store this move
        sol[n] = p;

        // Check if cell is okay
        if(cellok(n))
            // is the next possibility the end of the maze?
            if(n == 8)
                printsolution();
            else
                build(n + 1);               // get the next move
        p++;
        }
    }

    void main(void)
```

```
{
      build(1);
}
```

Ex) The Maze problem

- use similar algorithm as above

```
#include "stdio.h"
#define SIZE 10
//#define DEBUG

// GLOBAL DATA

// a cell type containing a row and column position
// and the compass direction most recently moved
struct cell_type {
      int row;
      int col;
      int dir;    // the most recent compass direction, 'n', 's',
'e', 'w', 0
};
typedef struct cell_type Cell;

Cell sol[SIZE*SIZE];          // the solution represented as an
array of Cells

int maze[SIZE][SIZE] = {      // the maze
      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
};

// FUNCTION PROTOTYPES
void build(int);
void printSolution(int);
int cellOk(int);
int getNextCell(int);

void main(void)
{
      // set starting position and direction
      sol[0].row = 1;
      sol[0].col = 1;
      sol[0].dir = 0;

      // start recursive solution
      build(0);
}

/*
```

```
A recursive function to determine a path through the maze.
Called for each cell in the solution array.  Finds the next
valid cell to move to, then calls itself for the next cell.
Inside the function, all possible moves for the current cell
are tried before the function returns.
*/
void build(int n)
{
      // *** YOU WRITE THIS FUNCTION ***
      // ...


// a handy debug function
#ifdef DEBUG
printf("Iteration: %d\tAt: (%d, %d)\t Trying: (%d, %d)\n",
         n, sol[n].row, sol[n].col, sol[n+1].row, sol[n+1].col);
#endif

      // ...

}

/*
      Outputs the current solution array.
*/
void printSolution(int n)
{
      int i;

      printf("\nA solution was found at:\n");
      for(i=0;i<=n;i++)
            printf("(%d, %d) ", sol[i].row, sol[i].col);
      printf("\n\n");
}

/*
Determines the next cel to try.  Increments the position
of the next cell and increments the direction of current cell.
Directions are tried in the order east, south, west, north.
*/
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 postion 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;                     // all directions have
been tried
      }
      return 0;                           // make compiler happy
}

/*
Checks if a cell is a valid move.  Invalid moves are cells
that are blocked with a wall, or with a cell that is
part or the present     path.
*/
int cellOk(int n)
{
      int i;

      // check if cell is a border cell
      if (maze[sol[n+1].row][sol[n+1].col])
            return 0;

      // check if we are attempting to cross our own path
      for(i=0;i<n;i++)
            // if where we want to go is somewhere we've been...
            if(sol[n+1].row == sol[i].row && sol[n+1].col ==
sol[i].col)
                  return 0;

      return 1;
}
```

Linked Lists

Fundamentals of Linked Lists

- dynamically allocate memory
- but different than dynamic arrays

Nodes
- each piece of data is contained in a node
- each node contains a piece of data and a pointer to the next node

```
struct Node {
        int data;
        Node *link;
}
```

- use a struct when all members are public
- a list can be constructed by linking together nodes
  - the first node is called the **head**
  - the last node is called the **tail**, and points to **NULL**

Head Pointers and Tail Pointers
- two pointers are used to access the list.
  - the **head pointer** points to the head node
  - the **tail pointer** points to the tail node

```
Node *head_pointer, *tail_pointer;
```

- an empty list has a NULL head and a NULL tail pointer

Member Selection Operator
- the dot operator has a higher precendence than the * operator.
- therefor, use:

```
(*head_ptr).data                    // OR
head_ptr->data
```

- to access the data member of the node pointed to by head_ptr

Dereferencing the Null Pointer
- always check that a pointer is not NULL before dereferencing it!!

Ex) Check for NULL pointer before dereferencing

```
if(head_ptr!=NULL) cout << (*head_ptr).data;
```

A Linked List Toolkit

Computing the length of a linked list

```
size_t list_length(Node* head_ptr)
{
   Node *cursor;
   size_t answer;

   answer = 0;
   for (cursor = head_ptr; cursor != NULL; cursor = cursor->link)
      answer++;

   return answer;
}
```

- cursor = cursor->link copies the link field of the node into the cursor
- common algorithm to traverse a linked list
- be careful not to dereference a NULL pointer
- be sure to handle empty lists correctly -- as this code does
- uses a pointer **passed by value** - i.e., the original pointer **cannot** be set to point to a different node. Changing the parameter's value has no effect on the original pointer.
- could be rewritten without cursor - use head pointer instead

Inserting a node at the head of the list

```
void list_head_insert(Node*& head_ptr, const Node::Item& entry)
{
   Node *insert_ptr;

   insert_ptr = new Node;                  // allocate a new node
   insert_ptr->data = entry;               // put the item in the new node
   insert_ptr->link = head_ptr;            // hook the new node to the front of the list
   head_ptr = insert_ptr;                  // move the head pointer to the new front
}
```

- uses a pointer **passed by reference** - i.e., the original pointer **can** be set to point to a different node. Changing the parameter's value changes the original pointer.

Inserting a node that is not at the head of the list

```
void list_insert(Node* previous_ptr, const Node::Item& entry)
{
   Node *insert_ptr;

   insert_ptr = new Node;                        // allocate a new node
   insert_ptr->data = entry;                     // put the item in the new node
   insert_ptr->link = previous_ptr->link;        // hook the new node after it
   previous_ptr->link = insert_ptr;              // hook the previous node to the new node
}
```

- requires a pointer to a the node just before the the intended location of the new node (because we can traverse in only one direction)
-

Searching for an Item in a linked list

```
Node* list_search(Node* head_ptr, const Node::Item& target)
{
   Node *cursor;

   for (cursor = head_ptr; cursor != NULL; cursor = cursor->link)
      if (target == cursor->data)
         return cursor;
   return NULL;
}
```

- returns a pointer to a Node
- returns NULL if Item is not found

Finding a Node by its Position in the linked List

```
Node* list_locate(Node* head_ptr, size_t position)
{
   Node *cursor;
   size_t i;

   assert (0 < position);              // bail if position is zero
   cursor = head_ptr;
   for (i = 1; (i < position) && (cursor != NULL); i++)
      cursor = cursor->link;
   return cursor;
}
```

- returns a pointer to a Node
- returns NULL if Item is not found

Copying a Linked List

```
void list_copy(Node* source_ptr, Node*& head_ptr, Node*& tail_ptr)
{
   head_ptr = NULL;
   tail_ptr = NULL;

   // Handle the case of the empty list
   if (source_ptr == NULL)         // assume head and tail were previously set to NULL
      return;

   // Make the head node for the newly created list, and put data in it
   list_head_insert(head_ptr, source_ptr->data);
   tail_ptr = head_ptr;

   // Copy the rest of the nodes one at a time, adding at the tail of new list
   for (source_ptr = source_ptr->link; source_ptr != NULL; source_ptr = source_ptr->link)
   {
      list_insert(tail_ptr, source_ptr->data);
      tail_ptr = tail_ptr->link;
   }
```

- creates an entire new linked list
- source pointer passed by value - the original pointer will not be changed
- head and tail pointers passed by reference - the original pointers will be changed

Copying a piece of a list

```
void list_piece(Node* start_ptr, Node* end_ptr, Node*& head_ptr, Node*& tail_ptr)
{
   head_ptr = NULL;
   tail_ptr = NULL;

   // Handle the case of the empty list
   if (start_ptr == NULL)
      return;

   // Make the head node for the newly created list, and put data in it
   list_head_insert(head_ptr, start_ptr->data);
   tail_ptr = head_ptr;
   if (start_ptr == end_ptr)
      return;

   // Copy the rest of the nodes one at a time, adding at the tail of new list
   for (start_ptr = start_ptr->link; start_ptr != NULL; start_ptr = start_ptr->link)
   {
      list_insert(tail_ptr, start_ptr->data);
      tail_ptr = tail_ptr->link;
      if (start_ptr == end_ptr)
         return;
   }
}
```

Removing a Head Node

```
void list_head_remove(Node*& head_ptr)
{
   Node *remove_ptr;

   remove_ptr = head_ptr;
   head_ptr = head_ptr->link;
   delete remove_ptr;
}
```

Clearing a Linked List - (delete all nodes)

```
void list_clear(Node*& head_ptr)
{
   while (head_ptr != NULL)
      list_head_remove(head_ptr);
}
```

- head passed by reference - it will be changed
- terminated when the head points to NULL
- Note: tail pointer still points to a non-existent node!!

Static Arrays, Dynamic Arrays, Linked Lists, Doubly Linked Lists

- which to use depends on many factors

Statics Arrays
Good:
- when the maximum size of the array is known at design time
- random access – accessing any element takes the same amount of time
- can be easily saved to disk

Bad:
- wastes space when there are many arrays which could each be partially full

Indiferent:
- insertions – each insertion takes O(n) time if the items must all be moved
- each insertion takes O(1) time if an idex is used to keep the items stationary

Dynamic Arrays
Good:
- when the maximum size of the array is unknown at design time
- random access

Bad:
- requires added complexity
- resizing can be inefficient if it occurs frequently

Linked Lists
Good:
- when the maximum size of the array is unknown at design time
- insertions – each insertion takes the same amount of time provided the insertion point (cursor location) is known
- Resizing is very efficient – just add a new node!
- more efficient use of space when there are many sets of data as compared to arrays
- there are some very efficient sorting algorithms for linked lists

Bad:
- random access can be inefficient – each operation takes O(i) time
- more difficult to save to and retrieve from disk
- extra mem space req'd for the pointer data

Doubly linked List
Good:
- when it is necessary to be able to move backwards and forwards through the list

Cursors

- there is a special type of data structure known as a cursor, which is sort of a cross between a static array and a linked list

Reason for using cursor:
- static arrays can be wasteful when there are many partially filled arrays
- static arrays can be wasteful when they are created and destroyed frequently while the program is running, because they each require a contiguous block of memory, which the OS or program will have to gather up.
- linked lists are not as easy to store and retrieve to disk
- linked lists require memory to be allocated each time a node is created

What is a cursor?
- a static (or dynamic) array of elements.
- each element contains a data member, and a link to the next element
- one cursor can contain several “linked lists”, optimizing memory usage

To set up and use a cursor:
- the cursor is initialized
- the index (an integer value, not a pointer) to the next free element is kept
- a newcell() function is written to retreive an index to the next available element in the “free store”
- a deletecell() function is written to return an element to the “free store”

To create linked lists using a cursor:
- the index of the top of the list is kept
- linked list access functions are written to create new “nodes” by calling newcell or deletecell, then updating the link member of the related elements as required

Ex) Cursor implementation file
```
static int free;                // index of the first free element
Element heap[MAXCELLS];         // the cursor
int newcell(void)
{
        int i;
        i = free;
        free = heap[free].link;
        return i;
}

void disposecell(int e)
{
        heap[e].link = free;
        free = e;
}
```

Ex) A Linked list stack implementation
```
static int top;
void push(Token t)
{
        int old_top;

        // add a new cell to top of stack
        old_top = top;
        top = newcell();
        heap[top].link = old_top;

        // stick data in new cell
        heap[top].data = t;
}

int pop(Token * t)
{
        // Note: returns 1 if successful, 0 if not successful
        int new_top;

        // bail if stack is empty
```

```
        if(top==-1) return 0;

        // get data from top of stack
        *t = heap[top].data;

        // remove the top cell
        new_top = heap[top].link;
        disposecell(top);
        top = new_top;

        // successful, return true
        return 1;
}
```

Stacks

Introduction

Stack
- very common in programming
- an ordered, controled data structure
- a LIFO list
- all access is at the top of the stack
- analogous to a stack of papers of coins
- access controlled mainly through push() and pop() operations
- aka "a pushdown store"

Stack Implementation

- a typical stack will have the following functions
  - void push(data);
  - data pop();

Stack Applications

Some applications of stacks
- Used by a program to keep track of return addresses and local variables when a function is called
- Used by a compiler to analyze expressions
- To search a maze or other branching structure (backtracking)

Reversing a Word

```
while (c=readcharacter) {
        read(c)
        push(c)
}

while(the stack is not empty) {
        c = pop()
        print(c)
}
```

Balancing Parenthesis

```
while (c=readcharacter) {
        if (c == '(' ) push(c);
        if (c == ')' ) {
                if (empty) incorrect;
                else pop();
        }
}

if (not empty) incorrect;
```

Evaluating Infix Arithmetic Expressions (Fully Parenthesized)

- we need two stacks, one for operands, and one for operators
- we will assume that the expression is fully parenthesized

```
while (c=readcharacter) {
        if (c is a number) nums.push(c);
        if (c is an operand) ops.push(c);
        if (c is a right parenthesis) {
                num1=nums.pop();
                num2=nums.pop();
                op=ops.pop();
                ans=num2 op num1;
                nums.push(ans);
        }
        if (c is a left parenthesis) do_nothing;
}

ans = pop();
```

Evaluating Postfix Arithmetic Expressions (No Parens)

- can use one stack

```
while (c=readcharacter) {
        if (c is a number) push(c);
        else {                          // c must be an operand
                num1=pop();
                num2=pop();
                ans=num2 c num1;
                push(ans);
        }
}

ans = pop();
```

Translating Infix to Postfix (Fully Parenthesized)

- we can just move the operands to the matching right paren, then remove all parens

Ex)

(((A + 7) * (B / C)) - (2 * D))

A 7 + B C / * 2 D * -

```
while (c=readcharacter) {
        if (c is a left paren) push(c);
        elseif (c is an operand) print(c);
        elseif (c is an operator) push(c);
        elseif (c is a right paren) {
                print( pop() );
                pop();              //remaining left paren
        }
}
```

Translating Infix to Postfix (NOT Fully Parenthesized)

- requires precedence rules for operators
- parens considered operators
- from higher to lower: ( ), / *, + -

```
while (c=readcharacter) {
        if (c is a left paren) push(c);
        elseif (c is an operand) print(c);
        elseif (c is an operator) {
                // pop and print while there is an operator
                // with a lower precedence at top of stack,
                // or until stack is empty
                c2 = peek()
                while (stack is not empty AND prec(c2) < prec(c) {
                        print(c2)
                        c2 = pop()
                }
                // push the last read operator
                push(c)
        }
        elseif (c is a right paren) {
                // pop and print until a left paren is popped
                c2 = pop()
                while (c2 != left paren) {
                        print(c2)
                        c2 = pop()
                }
        }
}

// pop n print remaining stack
while (not empty) {
        print(pop());
}
```

Queues

Introduction

Queue
- very common in programming
- an ordered, controled data structure
- a FIFO list
- items are added to one end and removed from the other
- analogous to a line at the grocery store
- access controlled mainly through enqueue() and dequeue() operations

Queue Implementation

- a typical queue will have the following functions
  - void q(data);
  - data dq();

Queue Applications

Some general applications of queues
- used in operating systems to keep track of processes and resources
- many general applications

Recognizing Palindromes

- need a stack and a queue

```
while (c=readcharacter) {
        push(c);
        enqueue(c);
}

while (q is not empty and stack is not empty) {
        if (pop!=dequeue) {
                is_palidrome = false;
                return;
        }
}
```

Infix Evaluator Lab

Some Hints
atoi() library function converts strings to numbers

Translating Infix to Postfix (NOT Fully Parenthesized) then evaluating the postfix (Lab 4)

- uses a Token struct
- uses a stack and a queue
- the stack is used to translate the infix expression
- the translated expression is stored in the queue

- the translated expression is evaluated using the stack

```
struct Token_type {
        int value;              // the value of the token
        int token_type;         // operator (0) or operand (1)
};
typedef struct Token_type Token;
```

Algorithm I: Convert infix expression to postfix

```
initialize stack and queue to empty;
push('#');      // a dummy operator having a precedence < any other operator (*/, +-, #)
while (not at end of expression) {
        get a token;
        if (token is an operand)
                enqueue(token);
        else if (token=='(')
                push('(');
        else if (token==')') {
                tmp = pop;
                while (tmp != '(') {
                        enqueue(tmp); tmp = pop;
                }
        }
        else {
                while (precedence(token) <= precendence(topofstack)) {
                        tmp = pop; enqueue(tmp);
                }
                push(token);
        }
}
while (topofstack != '#') {
        tmp = pop; enqueue(tmp);
}
```

Algorithm II: Evaluate postfix expressions

```
initialize stack to empty;
while (queue not empty) {
        token = dequeue;
        if (token is an operand)
                push(token)
        else {
                operand1=pop; operand2=pop;
                result=perform(operand1,operand1,token);
                push(result);
        }
}
```

Some Code

```
/*
        FILE: postfix.cpp

        CLASS:  CIS 40

        INSTRUCTOR: Dan Ross
```

```
        ASSIGNMENT:  Lab #4  Infix Evaluator

        DESCRIPTION:
        Implements an infix evaluator. Input of the form "12*(34-5)" is
        converted into its postfix equivalent "12 34 5 - *" by decomposing
        the expression into tokens and moving the tokens onto a stack and
        a queue.  The postfix form and the value of the expression are
        displayed.

*/

#include "iostream.h"
#include "stdio.h"
#include "queue1.h"
#include "stack1.h"
#include "string.h"
#include "stdlib.h"

// The token type definition
struct Token_type {
   int token;
   int token_type;
};
typedef struct Token_type Token;

Stack<Token> st;
Queue<Token> qu;

// FUNCTION PROTOTYPES
void get_token(char * str, Token & t);
void print_token(Token *);
int prcdnc(Token t);
Token perform(Token,Token,Token);
void bufferize(char * s, Token t);
void calc_postfix(void);
int get_input(void);

// GLOBAL DATA
char infix_str[80] = "";     //"12+45*(67-89+70)/34";  // for debug
char postfix_str[80] = "";
int ichar=0;                 // pointer to chars in infix_str

void main(void)
{
        // get infix input from user and convert to postfix
        while(get_input())
                calc_postfix();
}

/*
Gets a string from the user.  Returns false if the user
enters a null string.
*/
int get_input(void)
{
        // get an infix expression from the user
```

```
	cout << "Please enter an infix expression (RETURN to exit):\n";
	cin.getline(infix_str,80);
	postfix_str[0]=0;
	return infix_str[0];
}

/*
Converts a string containing a valid infix expression
into the corresponding postfix expression.  Calculates
and displays the value of the expression.
*/
void calc_postfix(void)
{
	// *** YOU WRITE THIS ***

	Token t, tmp, op1, op2;

	// push on a dummy operator having lowest precedence
	// move infix tokens into stack and queue

	// pop and enqueue the remaining operators

	// EVALUATE POSTFIX EXPRESSION

	// DISPLAY ANSWER

	printf("\nThe infix expression is: %s\n", infix_str);
	printf("The postfix equivalent is: %s\n", postfix_str);
	printf("The value of the expression is: %d\n\n\n", t.token);
}

/*
Converts a token into a string and concatenates the result
onto s.
*/
void bufferize(char * s, Token t)
{
	char buf[10];

	if(t.token_type) {
		itoa(t.token,buf,10);
		strcat(s,buf);
		strcat(s," ");
	}
	else {
		buf[0]=t.token;
		strncat(s,buf,1);
		strcat(s," ");
	}
}

/*
Performs an arithmetic operation on two tokens using
the operator contained in the third token.
*/
Token perform(Token op1,Token op2,Token t)
```

```
{
	Token answer;

	switch (t.token) {
		case '*' :
			answer.token = op2.token * op1.token;
			break;
		case '/':
			answer.token = op2.token / op1.token;
			break;
		case '+':
			answer.token = op2.token + op1.token;
			break;
		case '-':
			answer.token = op2.token - op1.token;
			break;
	}
	answer.token_type = 1;  // just for consistency
	return answer;
}

/*
Returns the precedence of an operator
3)	*	/
2)	+	-
1)	#
0)	(
*/
int prcdnc(Token t) {

	if(t.token_type) return 0;

	switch (t.token) {
	case '*':
	case '/':
		return 3;
	case '-':
	case '+':
		return 2;
	case '#':
		return 1;
	}
	return 0;
}

/*
A handy debug function.
*/
void print_token(Token * t)
{
	if(t->token_type)
		printf("Token: %d\tType: %d\n", t->token ,t->token_type);
	else
		printf("Token: %c\tType: %d\n", t->token ,t->token_type);
}
```

```
/*
Parses a string into tokens.  Reference parameters are used to
receive the string and return the corresponding token.
*/
void get_token(char * str, Token & t)
{
	//NOTE:
	//Expressions may only contain the chars "0123456789/*-+". No Spaces!

	char buf[2] = {0,0};			// a buffer used with atoi

	switch (str[ichar]) {
	case '*':
	case '/':
	case '-':
	case '+':
	case '#':
	case ')':
	case '(':					// token must be an operator
		t.token = str[ichar];
		t.token_type = 0;
		ichar++;
		return;
	}

	buf[0]=str[ichar];
	t.token = atoi(buf);			// token must be an operand
	t.token_type = 1;
	ichar++;
}
```

Trees

Introduction

Tree

- have nodes linked together in branches
- each node may have children
- the first node is called the root
- trees have edges, consisting of exterior nodes
- last node is called a leaf
- height of a tree is number of nodes along the longest branch
- trees with ordered data are efficient for searching and sorting

Binary Tree
- each parent has a maximum of two child nodes

Height, h	Min n	Max n
1	1	1
2	2	3
3	3	7
4	4	15
…		
h	h	$2^h - 1$

- thus, for a given n, the maximum height is $\log_2 (n + 1)$ (draw graph)
- this means that the height of the tree grows very slowly as more nodes are added, which makes it fast to find a given node

Tree Implementations

Array Implementations
- can go both ways

left child = 2n+1
right child = 2(n+1)
parent = (n-1)/2

0	A
1	B
2	C
3	D
4	E
5	F
6	G

Pointer Implementations
- can only go parent to child
- this is the typical implementation

```
struct node {
        Item data;
        struct node * left, * right;
}
```

- over half the pointers can are wasted!!

Tree Traversals

- we often need to visit the nodes in a tree one-by-one

- we can use a recursive or a non-recursive algorithm

Three kinds of tree traversals:

a) preorder NLR or NRL
b) inorder LNR or RNL
c) postorder LRN or RLN

// non recursive preorder traversal with a stack

```
cur = root;
while(cur!=NULL) {
        process(cur);
        if(cur->right!=NULL)
                push(cur->right);
        if(cur->left!=NULL)
                cur=cur->right);
        else
                if(stack_not_empty)
                        cur=pop();
                else
                        cur=NULL
}
```

// recursive preorder traversal

```
void traverse(root)
{
        if(root!=NULL) {
                process(root);
                traverse(root->left);
                traverse(root->right);
        }
}
```

// recursive inorder traversal

```
void traverse(root)
{
        if(root!=NULL) {
                traverse(root->left);
                process(root);
                traverse(root->right);
        }
}
```

// recursive postorder traversal

```
void traverse(root)
{
        if(root!=NULL) {
                traverse(root->left);
                traverse(root->right);
                process(root);
        }
}
```

Data Ordered Binary Tree (aka Binary Search Tree)

- binary search trees are fast

A linked list can be searched in O(n) time
A fully-filled tree can be searched in $O(\log_2 n)$ time minimum, which also happens to be the average time

- data is ordered by a key field in each node
- keys in left subtree are < parent key
- keys in right subtree are > parent key

Tree Insertions
1) make a new node and initialize it
2) if this is the first node, hook up node and bail
3) find the insertion point
4) hook up the new node

Tree Deletions
Find the node to be deleted (traverse)
a) if the node is a leaf, remove it
b) if the node has only one child, make the child the parent, delete the parent
c) if the node has two children, replace the parent with
    i. the smallest node on the right
    ii. the largest node on the left

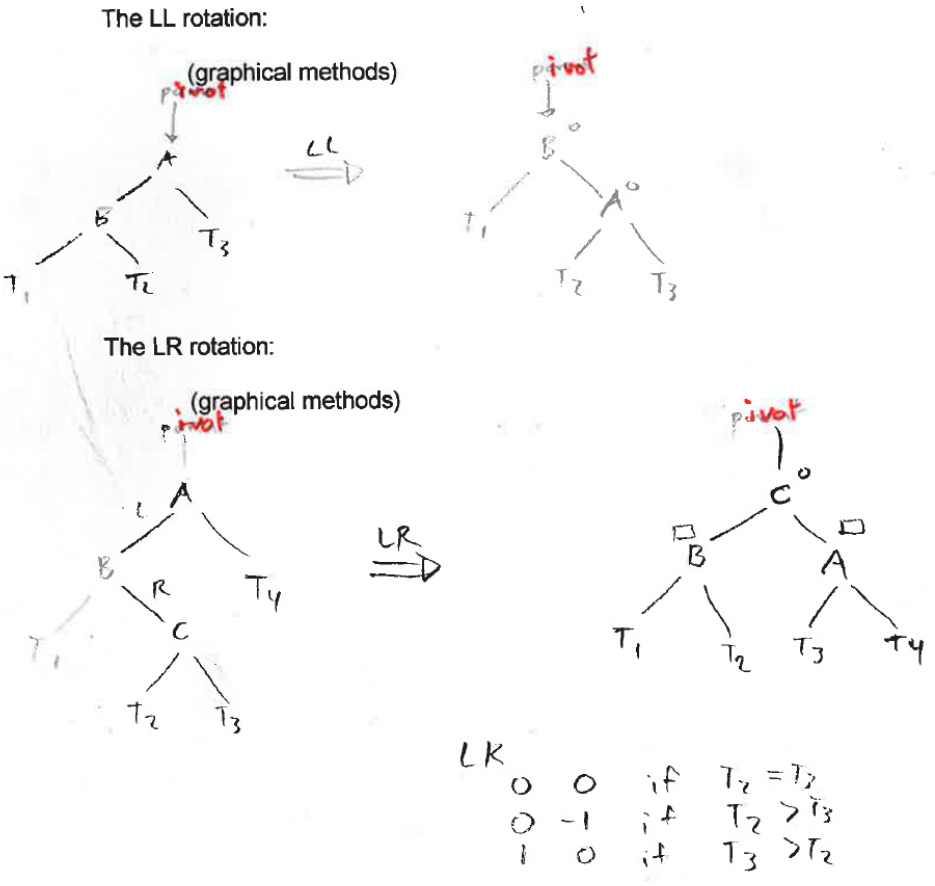
AVL Tree (aka Height-Balanced 1-Tree)

Problem with binary search Tree
- they are usually not fully filled
- they are usually not balanced
- this makes searches inefficient

Solution: AVL Trees
- the tree is adjusted (rotated) after each insertion
- the difference in the height of any two subtrees is <= 1
- we need to maintain a balance factor in each node.

  $BF = H_L - H_R$

AVL Tree Insertions:
1) Find the insertion point and the pivot node
   the pivot node is the first node up from the new node with a non-zero balance factor
2) Update the balance factors for nodes between the new node and the the pivot (including the pivot node)
3) Rotate if the balance factor for the pivot is +-2



B- Trees of order N

What is it?
- NOT a binary tree!
- each node may have SEVERAL keys

B-Trees are another solution to unbalanced binary trees
- the tree is adjusted (rotated) after each insertion (if required)
- the difference in the height of any two subtrees is zero

Uses for B Trees:
- to represent file directories

Implementation:
- typical implementations may use nodes.
- each node contains a partially filled array of keys and links to other nodes
- this implementation makes disk access of portions of a large B-Tree easier because the data in each node is sequential

B-Tree Rules:
- the root may have 1 to 2N keys
- each child may have N to 2N keys
- for each node, the number of child nodes must be zero or one more than the number of keys in the parent
- all leaves are at the same level (the tree is balanced)

B-Tree Insertions
- new keys are added to the leaves
- when an insertion causes a node to have 1 extra key, the node is split and the middle key is passed up to the parent node

  (graphical methods)

Binary Heaps

What is it?
- a fully-filled binary tree
- the keys are only partially ordered
- each parent node is smaller (min heap) than either of it's children
- don't confuse with the area of a program used for dynamic memory allocation, also called a heap

Uses for Binary Heaps:
- for quickly finding a minimum of maximum value
- NOT used for searching
- can be used for sorting

Implementation:
- typical implementations use an array. Why? cuz tree is fully filled and does not need to be rotated
- access and manipulation is easy ans efficient
- can easily go both ways (up and down) in the tree/array

left child = 2n+1
right child = 2(n+1)
parent = (n-1)/2

0	A
1	B
2	C
3	D
4	E
5	F
6	G

Binary Heap Insertion Rules:
1) always insert at the next available position in the tree
2) swap the child with the parent until the parent is smaller than the child

- time for insertion is height of the tree:
  $h = \log_2 n$

  (graphical methods)

Binary Heap Deletion Rules:
1) always delete the root
2) replace the root with the last node
3) starting at the root, swap the parent node with the smaller child node until the parent is smaller than either child

- time for deletion is height of the tree:
  $h = \log_2 n$

  (graphical methods)

Sorting

Introduction

Non-Recursive Algorithms
- Bubblesort
- SelectionSort
- InsertionSort
- Heapsort

Recursive aka "Divide and Conquer" Algorithms
- MergeSort
- Quicksort

BubbleSort

- each pass compares two elements and swaps them if required
- n complete passes are required

60	50	30	40	90	80	20	10	70	100
Compare and swap these two if required									

50	60	30	40	90	80	20	10	70	100
	Compare and swap these two if required								

etc…

Time Analysis
- the time for each pass is O(n)
- there are n passed required, thus
- total time is n * n = **$O(n^2)$**

SelectionSort

- each pass finds the largest element and swaps it with the last element
- n complete passes are required

Initial

60	50	30	40	90	100	20	10	70	80

After swap 80 with 100

60	50	30	40	90	80	20	10	70	100

After swap 90 with 70

60	50	30	40	70	80	20	10	90	100

After swap 80 with 10

60	50	30	40	70	10	20	80	90	100

After swap 70 with 20

60	50	30	40	20	10	70	80	90	100

etc…

Time Analysis

- the time for each pass is O(n)
- there are n passed required, thus
- total time is n * n = **$O(n^2)$**

InsertionSort

- each element in the unsorted array is inserted into the correct location in the sorted array.
- None, some, or all of the elements in the sorted array must be scooted down one to make room for the new element.
- We can use one array, by using the front of the array for the sorted part, and the rear of the array for the unsorted part.

Initial

60	50	30	90	40	100	20	10	70	80

Insert 60

60	50	30	90	40	100	20	10	70	80

Insert 50. Must move 60 down

50	60	30	90	40	100	20	10	70	80

Insert 30. Must move 50 and 60 down

30	50	60	90	40	100	20	10	70	80

Insert 90. Don't have to move anything down

30	50	60	90	40	100	20	10	70	80

Insert 40. Must move 50, 60, and 90 down

30	40	50	60	90	100	20	10	70	80

Time Analysis

- the time for each pass is O(n)
- there are n passed required, thus
- total time is n * n = **$O(n^2)$**

MergeSort

- a divide and conquer algorithm
- the array is recursively subdivided into two subarrays
- each (sorted) subarray is merged with the other (sorted) subarray to yield a sorted array
- the recursion continues until a small enough subarray is obtained (usually a 1 element subarray)
- the subarrays may also be sorted by another sorting algorithm
- Note: Text says that each recurse must allocate a temp array. This is not always required. Instead, each recurse may share one section of the same global array. However, we still need two arrays.

Graphical Methods
- this array will be subdivided
- each subarray will be sorted
- then, the subarrays will be merged

50	60	30	40	90	80	20	10	70	100

split here

After sorting each subarray

30	40	50	60	80	10	20	70	80	100

After merging the subarrays

10	20	30	40	50	60	70	80	90	100

Implementation
- requires two primary functions, merge() and msort()
- most of the work is done by the merge() algorithm

```
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];
}
```

Time Analysis

- merge subdivides the array, so it is called $\log_2 n$ times
- the *total* work done at each *level* is proportional n
- thus, total time is $n * \log_2 n$ = **$O(n \log_2 n)$**

QuickSort

- 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 either side of the middle element are then sorted
- the recursion continues until a small enough subarray is obtained (usually a 1 element subarray)
- the subarrays may also be sorted by another sorting algorithm

Graphical Methods

- this array will be subdivided by finding the final position of the middle element for each array
- each subarray will be sorted

Initial

40	20	60	80	10	70	30	50

After sort 1

10	20	30	40	80	70	60	50

After sorts 2 and 3

10	20	30	40	50	70	60	80

After sorts 4 and 5

10	20	30	40	50	70	60	80

After sorts 6 and 7

10	20	30	40	50	60	70	80

After sort 8

10	20	30	40	50	60	70	80

Implementation

- requires two primary functions, partition() and qsort()
- most of the work is done by the partition() algorithm

```
Private Sub Qsort(f As Integer, l As Integer)

   Dim m As Integer

   If (f <= l) Then
      m = Part(f, l)
      Qsort f, m - 1
      Qsort m + 1, l
   End If

End Sub
```

```
Private Function Part(f As Integer, l As Integer) As Integer

   Dim P1 As Integer, P2 As Integer, PV As String

   PV = A(f)
   P1 = f + 1
   P2 = l

   Do While (P1 <= P2)
      Do While (A(P1) <= PV And P1 <= l)
         P1 = P1 + 1
      Loop
      Do While (A(P2) > PV)
         P2 = P2 - 1
      Loop
      If (P1 < P2) Then
         Swap P1, P2
      Else
         Swap f, P2
      End If
   Loop

   Part = P2

End Function
```

Time Analysis

- partition finds the middle element
- partition is called n times for the *worst case*
- however, partition is called $\log_2 n$ times on *average*
- the *total* work done at each *level* is proportional n
- thus, total time is n * $\log_2 n$ = **$O(n \log_2 n)$** **on average**
- thus, total time is n * n = **$O(n^2)$** **worst case**

HeapSort

- a heap always contains the smallest element on the top
- fill up a heap
- empty a heap
- the elements are now sorted!!

- we can use one array for a heapsort, use the front part of the array for the heap, and the rear part of the array for the list to be sorted

Insertions

Initial

60	50	30	90	40	100	20	10

After inserting 60 into heap

60	50	30	90	40	100	20	10
heap	list						

After inserting 50 into heap

50	60	30	90	40	100	20	10
	heap	list					

After inserting 30 into heap

30	60	50	90	40	100	20	10
		heap	list				

After inserting 90 into heap

30	60	50	90	40	100	20	10
			heap	list			

After inserting 40 into heap

30	40	50	90	60	100	20	10
				heap	list		

After inserting 100 into heap

30	40	50	90	60	100	20	10
					heap	list	

After inserting 20 into heap

20	40	30	90	60	100	50	10
						heap	list

After inserting 10 into heap

10	20	30	40	60	100	50	90
							heap

Deletions:

After deleting 10

20	40	30	90	60	100	50	10
						heap	list

After deleting 20

30	40	50	90	60	100	20	10
					heap	list	

After deleting 30

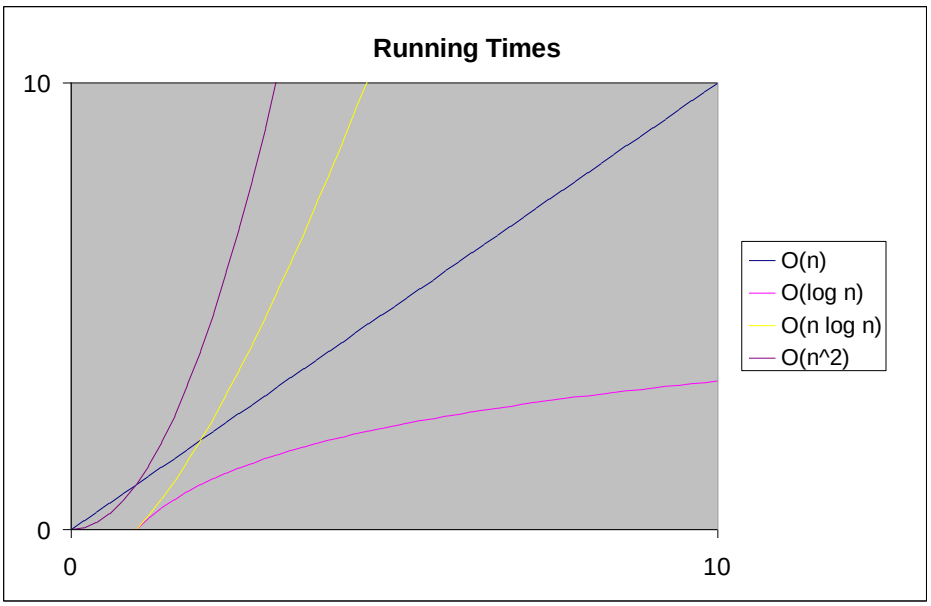
40	60	50	90	100	30	20	10
				heap	list		

etc…

Time Analysis

- one insertion requires $\log_2 n$ time
- we must perform n insertions, requiring $n \log_2 n$ time
- one deletion requires $\log_2 n$ time
- we must perform n deletions, requiring $n \log_2 n$ time
- thus, total time is insertions is $2 * n \log_2 n$ = **$O(n \log_2 n)$**

Summary



Sorting Algorithm Running Time Analysis			
Algorithm	**Best Case**	**Average Case**	**Worst Case**
BubbleSort			$O(n^2)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$
MergeSort		$O(n \log_2 n)$	$O(n \log_2 n)$
QuickSort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
HeapSort		$O(n \log_2 n)$	$O(n \log_2 n)$

Searching/Hashing

Introduction

Searching Lists

There are two ways to a list
- sequential search
- binary search

Binary Search
- used with sorted lists
- the list is bisected at each step
- similar to searching a binary tree
- requires **$O(\log_2 n)$** time in the worst case

Sequential Search
- used with unsorted lists
- each element is looked at in sequence
- requires **O(n)** time in the worst case

Hashing

Direct addressing
- we can avoid searching if we simply use a large array with the same number of addresses as key values
- this wastes memory space if the range of addresses is large but incomplete
- many of the elements are empty

Hashing
- a technique used to directly calculate the address of an item from its key.
- allows smaller arrays to be used than with direct addressing
- a hashing algorithm allows the address to be directly calculated **O(c)**
- the hashing algorithm in effect “compresses” the range
- there are all kinds of hashing algorithms that manipulate the key in some way to produce an address
- most common hashing algorithms are
  - digit selection - selecting specific digits of the key
  - division - using the modulo of the key

Ex) Simple hash table

Consider entering the following keys using hash table and hashing algorithm below:

h(key) = key % 7

35		44	3	25	26	13
0	1	2	3	4	5	6

- the keys can be instantly retrieve by simply calculating the hash address

Entries	
key	h(key)
25	4
3	3
35	0
26	5
13	6
44	2

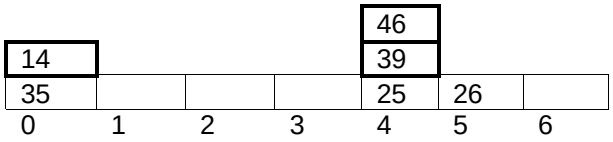
Collisions

- collisions are the main problem with hashing
- occur when two different keys produce the same hashed address
- strategies to avoid them include:
  - chaining
  - rehashing (aka open addressing)
  - using a perfect hash function (prevents any collisions)

External Chaining
- the hash table contains a data element and a link element
- collisions are resolved by attaching a node at the collision location
- when searching, all chain elements are searched until the end of the chain is encoutered

Ex) External chaining
Consider entering the following keys using hash table and hashing algorithm below:

h(key) = key % 7



Entries	
key	h(key)
25	4
39	4
35	0
26	5
14	0
46	4

- when searching, all entries in a chain must be searched until either the item is found, or an empty location is found

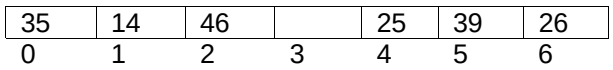
Rehashing
- linear rehashing
- generalized linear rehashing
- psuedo-random rehashing
- quadratic rehashing
- double hashing

Ex) Linear rehashing:

- collisions spill over to the next available location
- address sequence is:

  h(key), h(key)+1, h(key) + 2, h(key) + 3, etc…

- problem is clustering decreases performance



- must maintain at least one empty location so that we know when to stop searching

Ex) Generalized linear rehashing

- uses a step size to avoid physical clustering of entries
- address sequence is:

  h(key), h(key)+c, h(key) + 2c, h(key) + 3c, etc…

- hash table with a step size of 3:

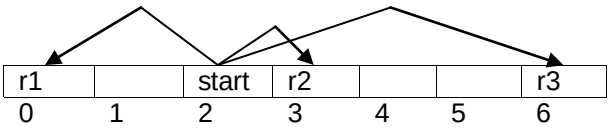
35	46	14		25	26	39
0	1	2	3	4	5	6

- still have a secondary clustering problem
- items are not physically clustered, but are *logically* clustered instead (items with collisions are all moved over by the same step)

Ex) Psuedo-random rehashing

- uses a psuedo-random sequence of numbers to avoid secondary clustering
- each key in the sequence has a "random" step size
- must generate the same "random" sequence for each table access
- address sequence is:

  h(key), h(key)+r1, h(key) + r2, h(key) + r3, etc…



- avoids physical clustering
- *still* have logical clustering
- also requires some computation time to calculate psudo-random steps

Ex) Quadratic rehashing

- similar to psuedo-random, but cheaper
- each key in the sequence has almost a "random" step size
- address sequence is:

  h(key), h(key)+1^2, h(key) - 1^2, h(key) + 2^2, h(key) - 2^2, h(key) + 3^2, etc…

35	14		46	25	39	26
0	1	2	3	4	5	6

- avoids physical clustering
- *still* have logical clustering

Ex) Double hashing

- avoids logical and physical clustering
- each key has an associated step size generated by a second hash function
- address sequence is:

  h1(key), h1(key)+c, h1(key) - 2c, h1(key) + 3c, etc…

  where;

  c = h2(key)

Entries		
key	h1(key)	h2(key)
25	4	
39	4	4
35	0	
26	5	
14	0	3
46	4	5

- consider h2(key) = (key % 6) +1

35	39	46	14	25	26	
0	1	2	3	4	5	6

Perfect Hashing

- it is possible to devise a hashing algorithm that does not generate duplicate addresses for a specific set of keys
- algorithmic perfect hash functions are very difficult to devise
- look-up table based perfect hash algorithms are simpler

Consider the finite set of three digit keys:
916, 209, 559, 503, 707, 800

We can define a perfect hash algorithm that uses digit selection and a 10 element lookup table:

Entries	
key	h(key)
916	6
209	1
559	9
503	7
707	4
800	8

h(key) = (g[first digit] + g[second digit] + g[third digit]) % 10

Value	0	1	2	2	4	5	6	7	8	9
index	0	1	2	3	4	5	6	7	8	9

- the hash algorithm will generate addresses to store the keys in a 10 element hash table.
- we could also devise a look-up table that will work with 6 element hash table, but this is more difficult
- note that we changed the value in index 3 to avoid a collision between 503 and 800
- Note: the lookup table and the hash table are coincidentally both 10 elements. Only the lookup table needs to be 10 elements.

Graphs

Definitions

- a graph is a generalized data structure
- most other simpler data structures are special cases of a graph
- a graph consists of nodes and links (aka vertices and edges), organized in many ways

Undirected Graphs
- there is no direction associated with the links between nodes

Directed Graphs
- there is a direction associated with the links between nodes

Loops
- a node with a link to itself

Multiple edges
- two nodes with more than one link between them

Simple Graph
- a graph with no loops or multiple edges
- many problems can be solved with simple undirected graphs
- in a simple graph there are n(n – 1)/2 edges max

Weighted graphs
- there is a cost (or weight) associated with each edge

Graph Applications

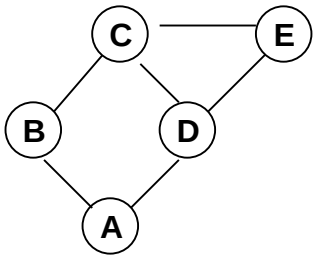
State Graphs
- can represent a problem by representing each state as a node, drawing all possible states, and connecting adjacent states
- a solution can be found by finding the a path from the beginning state to the final state

Networks
- useful for airlines, mail, and computer networks
- may want to finding the shortest path
- may want to check if two nodes are connected (can I get there from here?)

Map coloring problem
- two countries on a map may not have the same color

Nueral nets
- can be implemented with a directed, weighted graph
- a set of input will filter through the graph to produce an output

Graph Implementations



There are two common ways to represent graphs:
- Adjacency matrix
- Adjacency list (aka edge list)

Adjacency matrix
- contains a set of edges
- worst case time complexity is **O(n^2)** when each cell must be visited

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

Adjacency List
- several linked lists, one for each node
- each list represents all the nodes connected to the given node
- for 6 edges, there are 12 steps required, so the worst case time complexity is O(2m) = **O(m)**

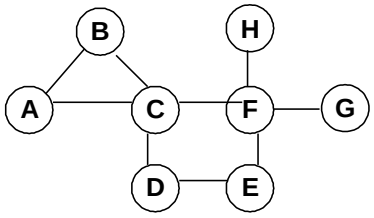
A		→	D		→	B		→			
B		→	A		→	C					
C		→	D		→	B		→	E		
D		→	A		→	C		→	E		
E		→	C		→	D					

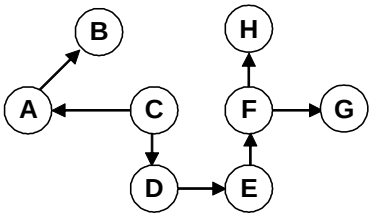
Which representation is better?
- matrix is good for dense (lots of edges) graphs
- list is good for sparse graphs

Graph Traversals

There are two common ways to traverse a graph:
- a stack traversal (aka depth first traversal)
- a queue traversal (aka width first traversal)

Stack 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:
1) Pick any starting node (C in the example below)
2) Push node
3) Select a neighbor of the top of the stack, ignoring neighbors previously selected
4) Push neighbor
5) If at the end of a list, goto 6, otherwise, goto 3
6) Pop
7) If stack is empty, we are done, otherwise, goto 3

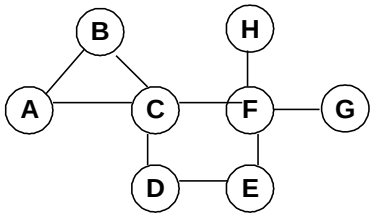
								G		H				
							F	F	F	F	F			
		B				E	E	E	E	E	E	E		
	A	A	A		D	D	D	D	D	D	D	D	D	
C	C	C	C	C	C	C	C	C	C	C	C	C	C	C

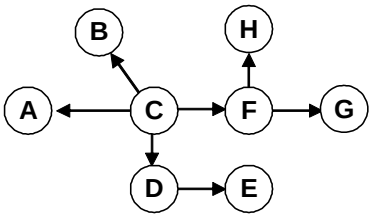
```
// A recursive solution for a stack traversal of a graph

/*
Each instance of traverse keeps track of one node's adjacency list.
The call to and return from traverse, substitute for push and pop.
*/

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


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

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

static void Main()
{
    // "Push" C
    traverse(2);

    printtree();
}

void traverse(node)
{
    mark node as visited
    print(node)

    init target to 0(A)
    while(not at end of node's list)
    {
        if(target is a neighbor and target is unvisited)
        {
            who's target's daddy?
            traverse(target)  // push
        }
        next target
    }
    return  // pop
}
/*
The stack traversal path:
C
A
```

```
B
D
E
F
G
H

The resulting tree:

Node    Parent
A       C
B       A
C       @
D       C
E       D
F       E
G       F
H       F
*/
```

Queue Traversal

Rules:
1) Pick a starting target node (C in the example below)
2) Enqueue target node
3) Enqueue any unmarked neighbors of the target node
4) Dequeue a node
5) If queue is empty, we are done, otherwise, make the next node 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							

Minimum Spanning Tree

- a weighted graph can be converted into a tree.
- not necessarily a binary tree
- there are many solutions for a given graph
- there will be at least one solution for which the total weight of all edges will be minimized
- useful for determining how to wire a house for electricity in the cheapest way (Cannot be cyclical like a graph!

Prim's Algorithm

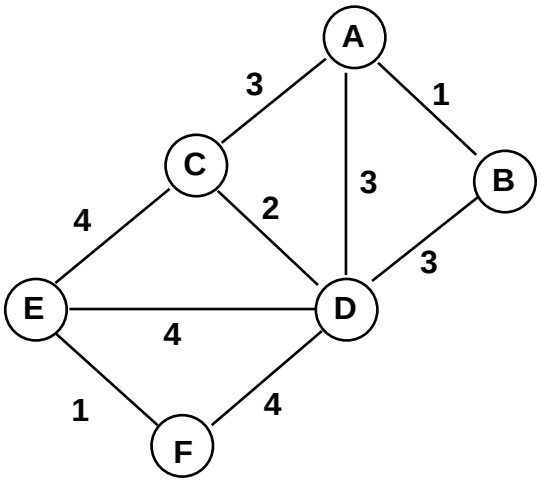
- for each node we must remember:
  - Distance: the *smallest* distance from the node to the tree (Nodes in the tree have a distance of zero)
  - Parent: each node's parent
- a candidate node is a node *adjacent* to the tree, but not yet *in* the tree
- the root has no parent

Data structure for Prim's algorithm

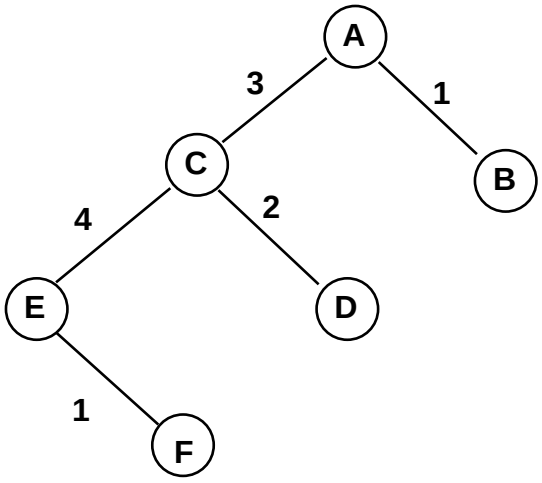
Distance						
Parent						
	A	B	C	D	E	F

Rules:

1) Pick an arbitrary root to add to the tree
2) Identify the candidate nodes. For each candidate node:
   a) update the distance to the nearest tree node
   b) update the node's parent
3) pick a new node having the shortest distance and add it to the tree
4) if all distances are zero, we are done, otherwise, goto 2



Graph



Minimum Spanning Tree

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

```
// A solution for Prim's minimum spanning tree algorithm

// the graph adjacency matrix
static int[,] graph = {
    {0,1,3,3,0,0},        //A
    {1,0,0,3,0,0},        //B
    {3,0,0,2,4,0},        //C
    {3,3,2,0,4,4},        //D
    {0,0,4,4,0,1},        //E
    {0,0,0,4,1,0},        //F
    //A B C D E F
};

// the resulting tree.  Store each node's distance to the tree and parent.
struct dp
{
    public int distance;
    public int parent;
    public dp(int d, int p)
    {
        distance = d; parent = p;
    }
};
static dp[] tree = {
                        new dp(9999, -1),
                        new dp(9999, -1),
                        new dp(9999, -1),
                        new dp(9999, -1),
                        new dp(9999, -1),
                        new dp(9999, -1)
                    };
// Make A the first target to put into the tree
int iTarget = 0;

while(!done())
{
    // add the target to the tree

    // update each candidate node (neighbors of the new node)
    for(each_neighbor_of_the_new_node)
    {
        if(distance_to_the_tree_is_shorter_for_this_candidate)
        {
            // update info for this candidate
        }
    }

    // find the next target (candidate with the shortest distance)
}

/*
The target is A.

The tree after inserting the target:

Node    Distance     Parent
A       0            @
```

```
B       1           A
C       3           A
D       3           A
E       9999           @
F       9999           @
The target is B.

The tree after inserting the target:

Node    Distance     Parent
A       0            @
B       0            A
C       3            A
D       3            A
E       9999           @
F       9999           @
The target is C.

The tree after inserting the target:

Node    Distance     Parent
A       0            @
B       0            A
C       0            A
D       2            C
E       4            C
F       9999           @
The target is D.

The tree after inserting the target:

Node    Distance     Parent
A       0            @
B       0            A
C       0            A
D       0            C
E       4            C
F       4            D
The target is E.

The tree after inserting the target:

Node    Distance     Parent
A       0            @
B       0            A
C       0            A
D       0            C
E       0            C
F       1            E
The target is F.

The tree after inserting the target:

Node    Distance     Parent
A       0            @
B       0            A
C       0            A
```

```
D       0           C
E       0           C
F       0           E
*/
```

Internet Routing Table Algorithms (Cheapest Paths)

- we are interested in determining the tree having branches that are each the cheapest path to a particular node
- typically, Internet routers are only concerned with how to get packets to the next router along the path to the packet's final destination
- this information is stored in tables in each router
- two important algorithms for generating this information are:
  - Distance Vector (RIP)
  - Dijkstra's (Open Shortest Path First (OSPF)

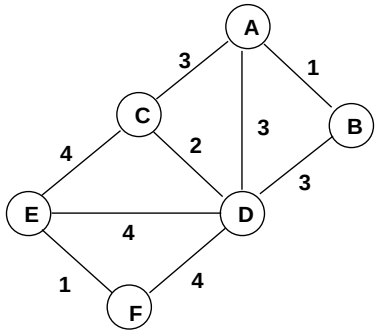
Distance Vector (RIP) Algorithm

- for each destination node we must remember:
  - the shortest distance from a given node to every other known node
  - the first hop in each of these paths

- initially, each node only knows:
  - the distance to it's immediate neighbors

- periodically, each node sends the information it knows about the network to it's immediate neighbors
- each node then uses this information to expand and optimize it's routing table

- a distributed algorithm - each node processes information as it is received. there is no way to predict the order
- eventually, the system will stabilize (usually)



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

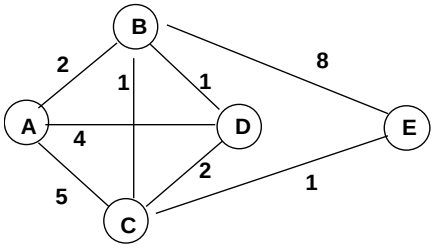
After each node looks at its neighbor's info

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

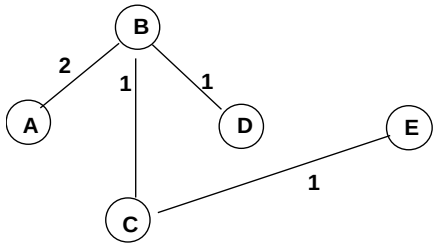
Done. System is stable, because if you continued to solve for each cell, you would find that all paths are optimal (cell values stop changing).

Dijkstra's (OSPF) Algorithm

- each node knows all the link costs for **every** node in the network
- a greedy algorithm - the shortest link is added to the path
- one tree can be built for each node
- distance = total path distance from a particular node to the root



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

```
// A solution for Dykstra's OSPF


// the graph adjacency matrix
graph = new int[,] {
    {0,2,3,0},        //A
    {2,0,0,4},        //B
    {3,0,0,1},        //C
    {0,4,1,0},        //D
    //A B C D
};
size = 4;

tree = new dp[] {
    new dp(false, 9999, -1),
    new dp(false, 9999, -1),
    new dp(false, 9999, -1),
    new dp(false, 9999, -1),
};

// Make A the first target to put into the tree

// store the root

while(!done())
{
    // printtarget

    // add the target to the tree

    // update each candidate node
    for(each_neighbor_of_the_target)
    {
        // calculate path distance for this candidate
        if(calculated_path_distance_for_this_candidate < distance_in_tree)
        {
            // update new distance for this candidate

            if(target_is_the_root)
                //nexthop is the neighbor itself
            else
                //nexthop is the target's nexthop
                // i.e., the hop that gave us the shorter distance
        }
    }

    printtree();

    // find the next target (candidate with the shortest distance in the
tree)
    for(each_candidate_element_of_the_tree_array)
    {
        if(this_distance_is_shorter)
        {
            store next target
        }
    }
```

```
}

/*
The target is A.

The tree after inserting the target:

Node    InTree  Distance        NextHop
A       True    0               A
B       False   2               B
C       False   3               C
D       False   9999            @
The target is B.

The tree after inserting the target:

Node    InTree  Distance        NextHop
A       True    0               A
B       True    2               B
C       False   3               C
D       False   6               B
The target is C.

The tree after inserting the target:

Node    InTree  Distance        NextHop
A       True    0               A
B       True    2               B
C       True    3               C
D       False   4               C
The target is D.

The tree after inserting the target:

Node    InTree  Distance        NextHop
A       True    0               A
B       True    2               B
C       True    3               C
D       True    4               C
*/
```

Encryption

Basics

- an encryption, or cipher can be though of as a reversible hash function
- input text is referred to as “text” or “data” or “plaintext”
- output is referred to as “cipher” or “etext” or “ciphertext”

Substitution Ciphers

Ceasar Cipher
- characters are substituted with other characters
- one of the simplest substitution cipher is a Ceasar cipher, which simply offsets each letter of the alphabet by the value of the key
- For example, a Ceasar Cipher with a key of 3 has the following mapping:

in	a	b	c	d	e	f	G	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
out	d	e	f	g	h	i	J	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c

t: dan
e: gdq

Every Nth Letter
- a more complex mapping can be obtained using a more complex algorithm on the key
- for example, picking every 3rd letter yields the following mapping:

in	a	b	c	d	e	f	G	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
out	a	d	g	j	m	p	S	v	y	b	e	h	k	n	q	t	w	z	c	f	i	l	o	r	u	x

- this example steps by 3, in general the scheme has 25 possible step sizes or “keys”
- still more complex mappings can be generated

Frequency Analysis of Substitution Ciphers

- substitution ciphers can be broken by analying the cipher text
- for example, for a large section of English text, there is a **frequency distribution pattern** for the letters as shown in the table below. The letter “e” is the most common letter, followed by the letter “t”
- the etext can be analyed using these frequencies to determine the text

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M
Freq	8.23	1.26	4.04	3.4	12.32	2.28	2.77	3.94	8.08	0.14	0.43	3.79	3.06
Letter	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Freq	6.81	7.59	2.58	0.14	6.67	7.64	8.37	2.43	0.97	1.07	0.29	1.46	0.09

Polyalphabetic Substitution Ciphers

- text data has the **frequency patterns** shown above
- a simple permutation can be done on text to hide the letter occurrence frequencies:
- for example, one can **hide the frequency of e’s** by using the permutation partially defined in the following table:

e: elephant
t: 1234

In	el	Ep	ha	nt	.	.	.
Out	1	2	3	4	.	.	.

- note that this mapping requires $26^2 = 676$ output mappings
- which would require groups of 10 bits because $2^{10} = 1024$
- so, in binary,

e: elephant
t: 0000000001 0000000010 0000000011 0000000100

Frequency Analysis of Polyalphabetic Substitution Ciphers

- polyalphabetic substitution ciphers essentially are redefining a much larger alphabet for the language
- some “letters” for example “it”, occur more frequently than others, so ga frequency analysis can still be done, given a large enough ciphertext sample

Transposition (Permutation) Ciphers
- rearranging the data in some way
- for example, if one simply swaps every-other letter:

Example:
t: peanutbutter
e: epantuubttre

- another type of transposition can be done using a matrix:

t: ilikepeanutbutterandjelly
e: ieunylatdintjkueeetrlpbal

i	e	u	n	y
l	a	t	d	
i	n	t	j	
k	u	e	e	
e	t	r	l	
p	b	a	l	

Limitations of Substitution and Transposition Ciphers

- there are more elaborate substitution and transposition ciphers

Patterns
- substition ciphers for text have frequency data patterns
- polyalphabetic substition ciphers for text also have frequency data patterns
- other kinds of data also have their own characteristic patterns
- the main problem with these ciphers is that ***patterns*** in the ciphertext remain, allowing crackers to gain a foothold in breaking the code

- a **good cipher** hides these patterns, making the cyphertext appear as random as possible

Keyspace

- of course, if the cipher algorithm is known, we can always just **guess the key**!!
- a **good cipher** has a large number of possible keys (a big keyspace)
- example: a 128 bit key has 2^128 possible keys, this would take a VERY long time to guess
- however, keys are themselves generated by algorithms, so studiing the key generation algorithm can help bad guys guess the key – analogous to human-generated passwords which typically have certain patterns which can be guessed.

Private Key Encryption

- a private key algorithm uses a single key to encrypt and decrypt
- the key is known to both the sender and the receiver, and **ONLY** by the sender and the receiver

XOR

- XOR can be used for a simple kind of private key encryption
- XOR has the following property:

a	b	XOR(a, b)
0	0	0
0	1	1
1	0	1
1	1	0

e = XOR(key, t)
t = XOR(key, e)

- for example, to encrpyt:

```
t  11001100
k  01010101
e  10011001
```

- and to decrypt:

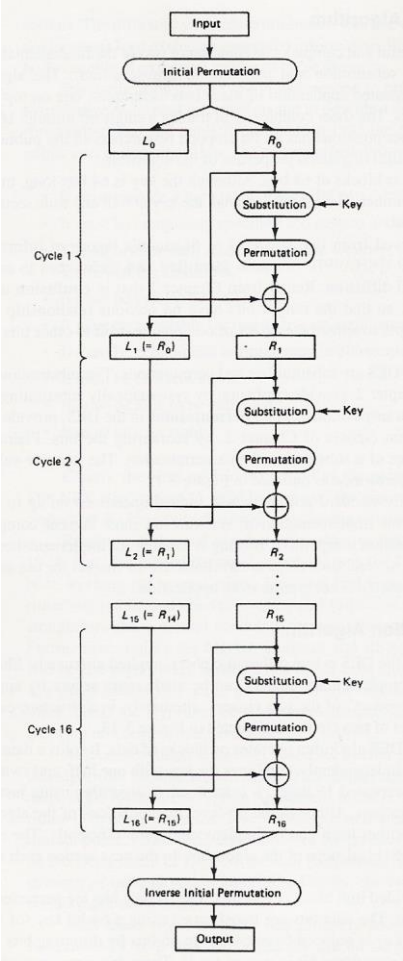
```
e  10011001
k  01010101
t  11001100
```

- note that this simple algorithm does not hide the **frequency patterns** in the original data, even if more then 8 bits are used for text. That is, even if 32 bit chunks were encrypted, the same frequency distribution for e's would occur in the sets of data consisting of the first 8 bits out of all the 32 bit chunks.
- some kind of permutation is still required to hide the frequency data

DES and 3DES

- uses a combination of **substitution** and **permutation**, both of which are a function of the key
- the combination of substitutions and permutations effectively **hide any patterns** in the data
- 3DES simply does DES three times
- the key is large enough (162 bits for 3DES) that quessing the key with brute force takes 2^{162} = 5.84e+48 guesses. This many guesses would take more time than is available before the universe ends, even with a really really really fast computer.

- How it works
  1) 64 bits are divided into two 32 bit chunks.
  2) The old right chunk becomes the new left chunk for the next cycle.
  3) The new right chunk is derived by using the key to permute and substitute the old right chunk, then combining the processed chunk with the old left chunk
  4) The cycle repeats 16 times for DES, 16*3 for 3DES



Public Key Encryption

RSA Algorithm

- uses two keys
- keys are generated based on the factoring of large prime numbers

- either key can be used to encrypt, and the other key must be used to decrypt
- typically, one key is kept private, and the other is made public, this is sometimes called "public and private keypairs"
- can be used for a variety of purposes, such as:
  - authentication – are you who you say you are
  - certification – are you recognized by others as being "OK"
  - the exchange of private keys

Session key exchange using RSA

- 3DES is a more efficient algorithm than RSA
- however, the problem with 3DES is how to do the secret exchange of the private key between sender and receiver

- typically, RSA is used initially to exchange a 3DES key – known as the "session key"
- the session key is used for the rest of the encrypted communication
- Example Key Exchange Protocol:

1) B generates a session key, encrypts it using A's public key, and sends it to A
2) A uses its private key to decrypt the session key

- typically, this exchange is combined with some form of authentication, to protect against a bad guy pretending to be B (spoofing).

- Example Key Exchange Protocol with Authentication:

1) A encrypts a random number using B's public key
2) B decrypts A's number using B's private key, combines the number with a session key, encrypts the whole message using A's public key, and sends it to A
3) A decrypts the message using A's private key, if the random number matches the message must be from B.

- Authentication gets more complex. Solutions involve certificates and certificate issuing authorities such as Verisign, etc…

State Machines

A String Processing example

```
/*
	A state machine to parse a sequence of known strings.
*/
#include <iostream>
#include <fstream>
using namespace std;

// State Table for:
//		bat bob can cat
char NextState[6][6] = {
	// 0  1  2  3  4  5
	  -1, 3,-1,-1, 5,-1,  //0
	   1,-1, 0,-1,-1,-1,  //1
	   4,-1,-1,-1,-1,-1,  //2
	  -1,-1,-1,-1,-1, 0,  //3
	  -1, 2,-1,-1,-1,-1,  //4
	  -1,-1,-1, 0,-1, 0,  //5
};

		//a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z
char map[26] = {0,1,2,0,0,0,0,0,0,0,0,0,0,3,4,0,0,0,0,5,0,0,0,0,0,0};

char state = 0;

process(char ch)
{
	// print current char
	cout << ch;

	// move to next state
	state = NextState[map[ch - 'a']][state];

	// output a space if we are back to initial state
	if(!state) cout << " ";
}

main()
{
	char ch;
	ifstream in("in.txt");
	if(!in){
		cout << "Error opening file";
		return;
	}

	// read and process characters
	while(in){
		in.get(ch);
		if(in)
			process(ch);
```

```
        }

    }
```