THE COMPUTER SCIENCE TUTORING CENTER

www.cstutoring.com

C++ DATA STRUCTURES PROGRAMMING COURSE E-BOOK OUTLINE

C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 1

File:	cppdsGuideL1.doc
Date Started:	Jan 21, 2000
Last Update:	Dec 22, 2001
Status:	draft

INDUCTION, TIMING AND LOOP INVARIANTS

INDUCTION

We need to prove that theorems are true. Proof by induction is usually used. Proof by induction is a little confusing. What they are trying to do is prove a theorem in steps. They think if the first step is true, and if they can prove subsequent steps are true then the theorem is true.

step 1 prove a base case: n = known value

This step merely confirms that the theorem is true for known values

step 2 induction hypotheses: k

The theorem is assumed to be true for all cases up to some known value k. Using this assumption then the theorem is tried to be proven correct for the next value k + 1 using the induction step.

step 3 induction step: n = k+1

Prove that the theorem is true for the next value n = k + 1.

step 4 induction conclusion

If the theorem is true for k+1 then the theorem is true for n

example 1

Prove that the series $1 + 2 + 3 + 4 + \ldots + n = \frac{n(n+1)}{2}$ is true using induction for n >= 1

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

step1: <u>base case</u> prove for n = 1 the sum must be 1

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

step 2 **induction hypotheses** assume that the theorem is true when n = k

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

n = k

Wherever there was an **n** we substitute **k**

step 3 **induction step** prove that the theorem is true for the next value (k + 1)

(n is now is equal to k + 1) n = k + 1

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

$$= \frac{k^2 + k + 2k + 2}{2} = \frac{k^2 + 3k + 2}{2} = \frac{(k+1)(k+2)}{2}$$

we have substituted $\frac{k(k+1)}{2}$ for $\sum_{i=1}^{k} i$

To complete the poof we must now substitute **n** for **k + 1**

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

You should now be able to realize and see the proof now.

step 4 **induction conclusion** If the theorem is true for n+1 then the theorem is true for n

We substituted the next value (k + 1) into the theorem now n is equal to k+ 1. We solved the theorem equation for (k + 1). From the answer we substituted n for (k + 1) we got back the theorem. Thus we have concluded if the theorem can be proven true for the next value (k + 1) the theorem is true for n.

Try for various n: **n = 1** $1 = \frac{n^2 + n}{2} + \frac{1^2 + 1}{20} = 1$

$$\textbf{n = 3} \quad 1 + 2 + 3 = \frac{n^2 + n}{2} + \frac{3^2 + 3}{2} = \frac{9 + 3}{2} = \frac{12}{2} = 6$$

using mathematical notation

We can use mathematical notation to represent the above proof.

Let S represent our Theorem. (S represents summation)

We want to prove for all n >= 1 that S(n) is true

Here is our proof:

Base Case:

Prove that S(1) is true

Induction Hypotheses:

For any value k where n = k then assume S(k) is true

Induction Step:

Prove that the next value (k + 1) is true. Let n = (k + 1) Prove S(k+1) is true.

Induction Conclusion:

if (k + 1) is true then assume for all n >= 1 then S(n) is true

The mathematical approach using notation gives a proof for any theorm.

weak induction

The above inductive step uses weak induction. Weak induction means we just test the base case and the next case if they are both true then we assume our theorem is true. If the base case is true n = 1 and if the next value is true n = (k + 1) then the Theorem is true for all n.

strong induction

With strong induction you use the induction hypotheses to assume every value from the base case to k is true if 1 <= k <= n is true then the theorem is true for all n. With strong induction you are testing **every case k.**

LESSON 1 EXERCISE 1

Prove that the series

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + 4^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{2}$$ is true using induction for n >= 1

PROOF BY CONTRADICTION

Proof by contradiction works by assuming a theorem is false and then prove some known property of the theorem false. (an opposite view point) If a false theorem is false then it got to be true ! You see you contradict your self. An easy example to understand is "There is no largest integer"

step 1 : assume the theorem is false

Assume the largest integer is X

step 2: prove the theorem false

Let Y = X + 1.

Now Y > X therefore X cannot be the largest integer, because now Y is the largest integer. There fore the theorem is true. There cannot be any largest integer. You can always find another larger number.

RECURSION AND INDUCTION

A function that is defined in terms of itself is known as recursion. In the following equation **f(x)** is on the **left** side of the equation and **f(x-1)** is on the **right** side of the equation.

f(x) = f(x - 1) + x

The next value of f(x) is dependent on previous values of f(x)

Start with x = 0:

f(0) = 0

f(1) = f(0) + 1 = 1

f(2) = f(1) + 2 = 1 + 2 = 3

f(3) = f(2) + 3 = 3 + 3 = 6

f(4) = f(3) + 4 = 6 + 4 = 10

etc.

The equation uses the results from previous computations. This is called recursion.

rules of recursion

(1) base case: solved without recursion (recursion **stops** at the base case)

(2) recursive cases: each recursive call leads to the base case

proof of recursion using induction for f(x) = f(x - 1) + x

prove: **f(n) = f(n - 1) + n** correct for n >= 0

The proof of induction is identical to the algorithm description

base case: when n = 0

f(0) = f(-1) + 0 = 0 (f (-1) is assumed 0)

Induction Hypotheses:

For any value k where n = k then assume f(k) is true

Induction Step:

Prove that the next value (k + 1) is true.
Let n = (k + 1) Prove f(k+1) is true.

f(k + 1) = f((k+1) - 1) + (k + 1) = f(k) + (k + 1)

substituting n = (k + 1) for both sides

f (n) = f(n - 1) + n

Induction Conclusion:

if (k + 1) is true then assume for all n >= 1 then f(n) is true

recursive computer program

A recursive mathematical function is easily converted into a program, you just use the equation! We use the base case when n = 0. For each recursive call we decrement n each value of n is stored separately. When the recursion is finished all the separate stored values of n are added up together.

```
/*  f (n) =  f(n - 1) + n */
int f(int n)
        {
        if(n == 0)                      /* use the base case  when n = 0     f(0) = 0 */
                return 0;
        else
                return f(n-1) + n       /*  for each recursive call n decrements by 1 */
        }
```

call	n	return value
1	3	6
2	2	3
3	1	1
4	0	0

For every function call the each value of **n** is stored. When the base case is reached all the stored values of n are added to the return value of the function from bottom to top

example using n = 3

For each equation call the result value is stored separately.

```
f(n) = f(n-1) + n          f(3) = f(2) + 3
f(n-1) = f(n-2) + n-1      f(2) = f(1) + 2
f(n-2) = f(n-3) + n-2      f(1) = f(0) + 1
f(n-3) = 0;                f(0) = 0
```

add up results from each recursive call

when the base case is reached the stored return values are added up:

```
f(3) = 0 + (n-2) + (n-1) + n
f(3) = 0 +  1  +   2   + 3 = 6
```

For any n we can start with the formula

```
f(n) = f(n-1) + n
```

and repeately substitute for the next call

```
f(n) = f(n-1) + n
f(n-1) = f(n-2) + (n-1) + n
f(n-2) = f(n-3) + (n-2) + (n-1) + n
f(n-3) = f(n-4) + (n-3) + (n-2) + (n-1) + n
etc.
```

We take the upper bound and assume f(n-4) is zero. The result is the **binomial series**.

$$f(n) = (n-3) + (n-2) + (n-1) + n = \sum_{i=1}^{n} i \quad \text{(binomial series)}$$

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

LESSON 1 EXERCISE 2

Write the equations and the recursive code for the Fibonacci sequence:

```
Fib(n) = Fib(n-1) + Fib(n-2)
```

There are two base cases: Fib(0) = Fib(1) = 1

TIMING

We need to estimate how long a program will run. Every time we run the program we are going to have different input values and the running time will vary. Since the running time will vary, we need to calculate the worst case running time. The worst case running time represents the maximum running time possible for all input values. We call the worst case timing **"big Oh"** written **O(n)**. The **n** represents the worst case execution time units. How do we calculate "**Big Oh**" ???

We first must know how many time units each kind of programming statement will take:

1. simple programming statement: O(1)

```
k++;
```

Simple programming statements are considered 1 time unit

2. linear for loops: O(n)

```
k=0;
for(i=0; i<n; i++)
        k++
```

For loops are considered **n** time units because they will repeat a programming statement **n** times. The term linear means the **for** loop increments or decrements by 1

3. non linear loops: O(log n)

```
k=0;
for(i=n; i>0; i=i/2)
        k++;
```

```
k=0;
for(i=0; i<n; i=i*2)
        k++;
```

For every iteration of the loop counter **i** will divide by 2. If **i** starts is at 16 then then successive **i's** would be 16, 8, 4, 2, 1. The final value of **k** would be 4. Non linear loops are **logarithmic**. The timing here is definitely $\log_2 n$ because $2^4 = 16$. Can also works for multiplication.

4. nested for loops $O(n^2)$: $O(n) * O(n) = O(n^2)$

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

Nested for loops are considered n^2 time units because they represent a loop executing inside another loop. The outer loop will execute **n** times. The inner loop will execute **n** times for each iteration of the outer loop. The number of programming statements executed will be **n * n.**

5. sequential for loops: O(n)

```
k=0;
for(i=0; i<n; i++)
    k++;

k=0;
for(j=0; j<n; j++)
    k++;
```

Sequential for loops are not related and loop independently of each other. The first loop will execute n times. The second loop will execute n times after the first loop finished executing. The worst case timing will be:

O(n) + O(n) = 2 * O(n) = O(n)

We drop the constant because constants represent 1 time unit. The worst case timing is **O(n)**.

6. loops with non-linear inner loop: O(n log n)

```
k=0;
for(i=0;i<n;i++)
    for(j=i; j>0; j=j/2)
        k++;
```

The outer loop is **O(n)** since it increments linear. The inner loop is **O(log n)** and is non-linear because decrements by dividing by 2. The final worst case timing is:

O(n) * O(log n) = O(n log n)

7. inner loop incrementer initialized to outer loop incrementer: $O(n^2)$

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

In this situation we calculate the worst case timing using both loops. For every **i** loop and for start of the inner loop **j** will be n-1 , n-2, n-3…

O(1) + O(2) + O(3) + O(4) + ……

which is the binomial series:

$$\frac{n(n-1)}{2} = \frac{n^2 - n}{2} = \frac{n^2}{2} = \mathbf{O(n^2)}$$

i	j
0	n-0
1	n-1
2	n-2
3	n-3
4	n-4

8. power loops: $O(2^n)$

```
k = 0;
for(i=1;  i<=n; i=i*2)
    for(j=1; j<=i; j++)
       k++;
```

To calculate worst case timing we need to combine the results of both loops. For every iteration of the loop counter **i** will multiply by 2. The values for j will be 1, 2, 4, 8, 16 and k will be the sum of these numbers 31 which is 2^n- 1.

9. if-else statements

With an **if else** statement the worst case running time is determined by the branch with the largest running time.

```
/* O(n) */
if (x == 5)
    {
    k=0;
    for(i=0; i<n; i++)
       k++;
    }

/* O(n^2) */
else
   {
   k=0;
   for(i=0;i<n;i++)
      for(j=i; j>0; j=j/2)
         k++;
   }
```

choose branch that has largest delay

The largest branch has worst case timing of **$O(n^2)$**

10. recursive

From our recursive function let T(n) be the running time.

```
int f(int n)
      {
      if(n == 0)
            return 0;
      else
            return f(n-1) + n
      }
```

recursion behaves like a loop.
The base case is the termination for recursion.

For the line: **`if(n == 0) return 0;`** this is definitely: **T(1)**
For the line: **`else return f(n-1) + n`** the time would be : **T(n-1) + T(1)**

The total time will be: **T(1) + T(n-1) + T(1) = T(n-1) + 2** which is **O(n)**

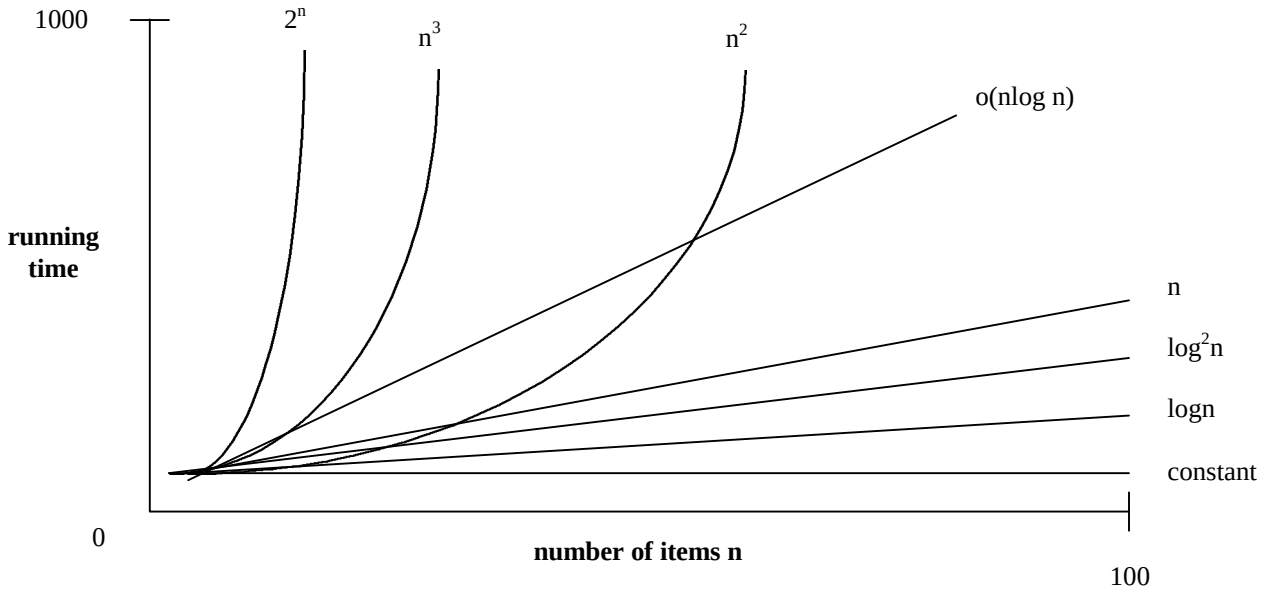
growth rates summary

The following chart lists all the possible growth rates:

c	constant
log n	logarithmic
log_2 n	log squared
n	linear
n log n	linear log squared
n^2	quadratic
n^3	cubic
2^n	exponential

growth rate graph

The growth rates are easily visualized in the following chart. The horizontal axis **x** represent **n** and the vertical **y** axis represent running time:



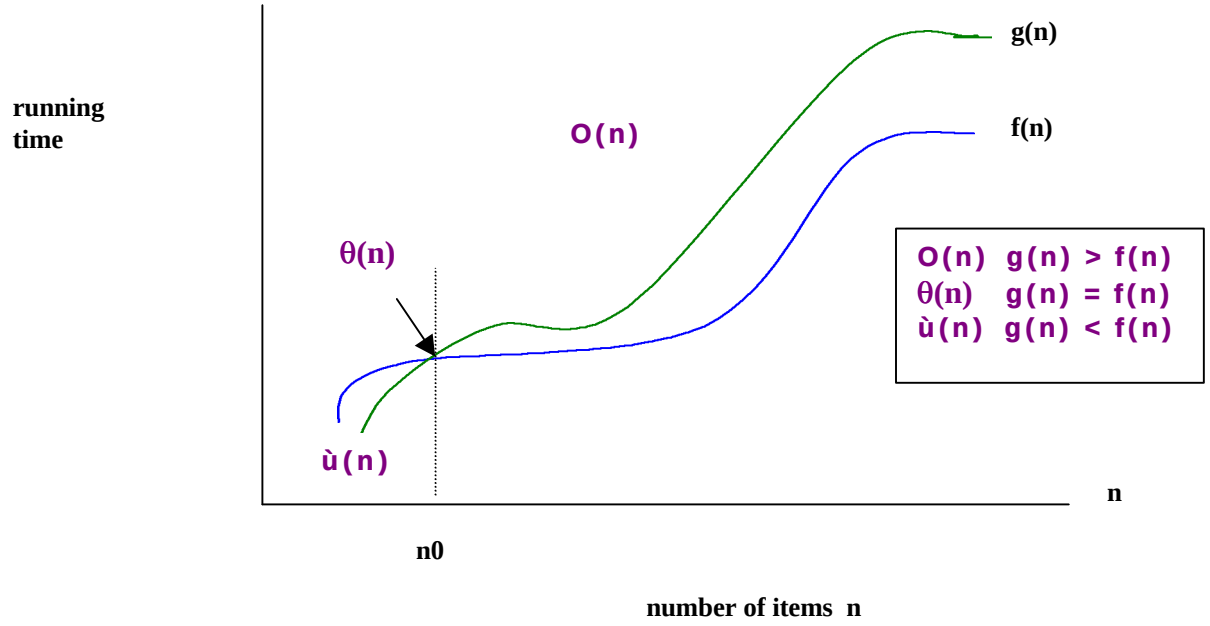
As n increases the running depends on the growth rates. For any n constant is the fastest and 2^n is the slowest. Worst case timing is also dependent on n For small values of n we can see n^2 is mush faster than n log n. For larger values of n (n log n) is much faster than n^2. You must be careful in choosing your algorithms for values of n.

$$n^2 \sim n \log n$$

MATHEMATICAL THEORY

To understand what **O(n)** is all about you need to know the following mathematics. Mathematics is trying to explain what is happening.

Consider the following graph:



We have two functions **f(n)** and **g(n)**. We are going to multiply **g(n)** by a constant **c** where c > 0. We have arbitrary constant **n0** that is greater than or equal to 1 (n0 >= 1). "**big Oh**" says that there must be some constant **c** times **g(n)** so that **f (n)** < **c * g(n)** and **n** must be grater than **n0**

We will call the running time T(n).

if cg(n) < f(n) then T(n) = ù(n)

if cg(n) == f(n) then T(n) = θ(n)

if cg(n) > f(n) then T(n) = O(n)

O(n) is always worst case timing

Rules:

T(n) + T(n) = 2 * T(n) = T(n)
T(n) * T(n) = T(n2)

(drop constant because constants are considered 1 time unit)

LESSON 1 EXERCISE 3

What is "big Oh" ? for:

(a)
```
for(i=0;i<n*n; i++)
   {
    for(j=i; j<n; j++)
        k++;
   }
```

(b)
```
for(i=0; i<n; i++)
   {
    for(j=i; j>0; j=j/2)
       k++;
   }
```

(c)
```
for(i=0; i<n; i=i*2)
   {
    for(j=i; j<n; j*j)
       k++;
   }
```

SERIES

You should know the following series

$$\boxed{\sum i} = \frac{n(n+1)}{2}$$

$$\boxed{\sum i^2} = \frac{n(n+1)(2n+1)}{6}$$

$$\boxed{\sum i^k} = \frac{n^{(k+1)}}{k+1}$$

LESSON 1 EXERCISE 4

Write down the expansion for each series up to n = 4 and k = 3.

PRECONDITIONS, INVARIANTS AND POSTCONDITIONS

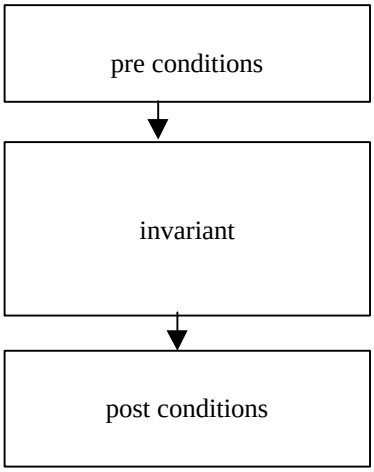
Preconditions, invariants and postconditions are used to analyze functions and loops.

The **precondition** is what is known to be true before the function or loop begins,

The **invariant** is known what is true inside the function or loop

The **post condition** is what is true after the function or loop terminates.

For the function or loop to operate correctly the **post condition** must be true after the function or loop terminates.



What is known to be true before entering

What is known to be true when inside

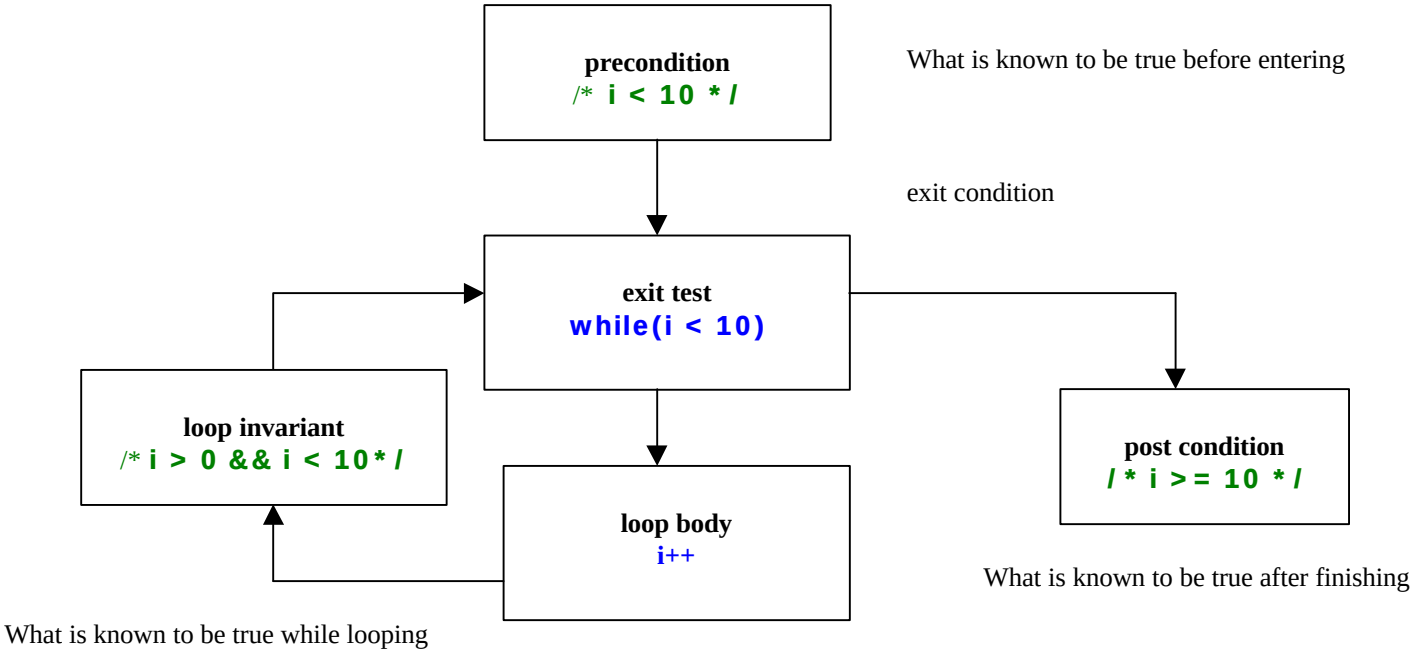
What is known to be true after finishing

A loop has 5 sections:

section	description	example
		`i = 0;`
Precondition	what is true before entering the loop	`/* i = <10 */`
exit test	what causes the loop to exit	`while(i < 10)`
Invariant	what is true for each iteration of the loop	`/* i > 0 && i < 10 */`
body	the loop statements	`i++;`
post condition	what is true after he loop conplete executiong	`/* i >= 10 */`

LOOP FLOW

It is easier to understand Preconditions, invariants and postconditions when using a loop. A loop has a precondition which is what is known to be true before we enter the loop. The loop will keep on looping while the loop invariant is true. The loop invariant must be the variables of the loop test condition that keep the loop looping. Inside the loop some operations are being performed. Every loop must have an exit condition that stops it from looping. The loop condition must test the loop invariant and exit if false. When the loop ends there are some values of variables that are true this is known as the post condition. The above preconditions, invariants and postconditions can easily be applied to functions.



Proving a loop is correct (big deal)

It's a four step process to prove that a loop is correct

step	what we have to prove
1	the precondition must be true before entering the loop
	the loop invariant must be true for the first iteration of the lop
2	the invariant is true for the next loop iteration of the loop does not exit
3	when the loop exits the post condition is true
4	the loop exits

LESSON 1 EXERCISE 4

Write a function that searches for a number in an array. Assume that the number is really in the array. The function gets the array, the length of the array and the value to search for. In your code state the **Precondition**, **exit test**, **Invariant**, **body section** and **post condition**.

USING MATHEMATICAL INDUCTION TO PROVE THAT A LOOP IS CORRECT

base case: i

Prove the invariant is true at the beginning of the loop

inductive step: i+1

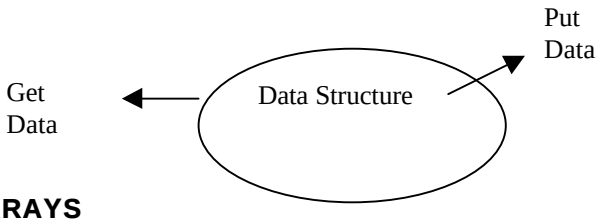
If invariant is true at the beginning of the loop then at the beginning of the next loop it will be true again as long as the loop does not exit

proving loop termination

When the number of loop iterations reaches a certain count the loop terminates.

C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 2

File:	CppdsGuideL2.doc
Date Started:	July 24,1998
Last Update:	Dec 22, 2001
Status:	draft



LESSON 2 ABSTRACT DATA TYPES USING ARRAYS

Data is needed to be stored efficiently for fast and easy insertion and retrieval. **Abstract Data Types** are used to store data in a computer. **Abstract** means the method of stored data is immaterial to the user. **A**bstract **D**ata **T**ypes are also called **ADT** for short. There a are many types of ADT's since there are many different ways to store data in memory. Each ADT will have a different data structure for storing the data. There are three tradeoffs to consider with each ADT.

implementation difficulty	Implementation difficulty indicates how difficult it is to write a program to implement the ADT.
worst case running time	The running time is the estimated worst case timing needed to do an insertion, deletion or search. Big O notation denotes worst case.
memory requirements	Memory requirements indicate how much additional memory is required to implement the ADT.

The following table lists common abstract data types indicating each trade off. We assume the lists are not ordered meaning we do not care where the elements are inserted.

ADT	Implementation Difficulty	Worst Case Running Time	Memory Requirements
vector	easy	O(n)	moderate
stack	easy	O(1)	moderate
queue	easy	O(1)	moderate
link list	not so easy	O(n)	small
double link list	not so easy	O(n)	small
hash table	moderate	O(1)	small
binary tree	difficult	O(log n)	small

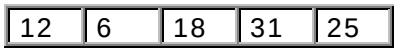
Which ADT to use ?

You choose which ADT to use by using the specifications of the project to determine which ADT is suitable for your application. Worst case expected running time can be broken down into search insertion and deletion time. Link lists have a good search time if sorted but longer search time if unsorted. Binary search trees very difficult to implement but have fast search time. The running time is O(log n) per insertion because it only has to compare per tree level. If there are 8 items then there would only be 3 levels in the tree and only 3 comparisons. (2 ** 3 = 8).

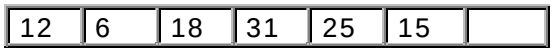
VECTOR ADT CLASS

A vector is a dynamically resizing array. The major problem when you declare an array, it is of fixed size. If you need to add more items, and your array is filled you are stuck. If you need to delete a lot of items in your array permanently then you are left with a large amount of computer memory unused. Vectors come to the rescue. In a vector ADT you allocate array memory to a reasonable **size**. If the vector gets filled it will allocate some more memory at an agreed reasonable amount called the **step**. When items are deleted the vector will shrink the allocated memory to a reasonable size accordingly to the step. We use a step because we want new items are to be added so no new memory needs to be allocated continuously.

Vector operation:
Create a vector of size 5 and step 2 and fill with 5 data items

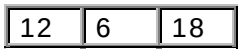


Add 1 more data item, the vector grows to 7



step by 2

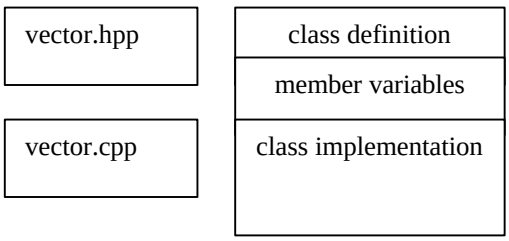
Delete 3 data items the vector shrinks to 3



decrease by 4

Implementing a Vector ADT

To implement a Vector you need functions to insert elements, remove elements and resize the array. You may want functions to insert and remove elements at the start or end of the Vector or to insert and remove at a certain location. We use a class to implement our Vector ADT where member variables are used to hold the size and vector array elements.



The header file contains the function prototypes.
The member variables holds the allocated array and variables shared between all Vector functions.

The implementation file contains the function definitions.

Here's the code for the vector ADT class. we have a small file defs.h that includes definitions for true, false, error and bool. If your compiler does not supply these for you then you need to include the defs.h file in vector.h.

```
/* defs.h */
#define true 1
#define false 0
#define error -1
typedef int bool;
```

Vector Module header file:

```
/* vector.hpp */
/* vector ADT class */
//#include "defs.h"
#ifndef __VECTOR
#define __VECTOR
/* vector data type */
typedef int VectorData;

/* vector module data structure */
class Vector
{
private:
VectorData* items; /* pointer to array element */
int size; /* vector size */
int step; /* vector step */
int last; /* last element location in vector */

public:
/* function prototypes for vector module */
/* constructor to initialize vector module variables and allocate memory */
Vector(int size,int step);
/* get number of items in vector module */
int numItem();
/* insert data item into last location of vector */
bool insertVecLast(VectorData data);
/* insert data item into first location vector */
bool insertVecFirst(VectorData data);
/* insert data item at index */
bool insertVecAt(VectorData data,int index);
/* return item value at a specified index */
VectorData getVecAt (int index);
/* find item in vector, return index location in vector */
int findVec(VectorData data);
/* search and replace a vector item with new data */
bool replaceVec(VectorData newdata,VectorData olddata);
/* replace a vector item at specified index */
bool replaceVecAt(VectorData newdata,int index);
/* search for and delete specified data item, return index */
int removeVec (VectorData data);
/* delete data item at specified index, return data value */
VectorData removeVecAt (int index);
/* print out vector items */
friend ostream& operator << (ostream& out, Vector& vec);
/* free vector items */
~Vector();
};
#endif
```

Vector class code implementation file:

```
/* vector.cpp */
#include <iostream.h>
#include "vector.hpp"
/* code implementation functions for vector ADT  class */
/* initialize vector data structure and allocate memory for vector */
Vector::Vector(int size,int step)
        {
        int i;
        /* allocate memory for vector at specified size */
        this->items = new VectorData[size];
        this->size = size; /* set size */
        this->step = step; /* set step */
        this->last = 0; /* set last item to start of vector */
        for(i=0;i<this->size;i++)this->items[i]=0; /* clear vector */
        }
/* get number of items in vector module */
int Vector::numItem()
        {
        return this->last; /* return last used location in vector */
        }
/* insert a data item into the vector at last index */
bool Vector::insertVecLast(VectorData data)
        {
        /* call to insert at last location */
        return insertVecAt(data,this->last);
        }
/* insert a data item into the vector at last index */
bool Vector::insertVecFirst(VectorData data)
        {
        /* call to insert at first location */
        return insertVecAt(data,0);
        }
/* insert a data item into the vector at index */
bool Vector::insertVecAt(VectorData data,int index)
        {
        int i;
        VectorData* newItem;
        VectorData* temp;
        if(index <0) return false; /* index out of bounds */
        /* check if enough room in vector */
        if(index < this->size)
                {
                /* shift up, no need to shift at ends of vector */
                for(int i=this->last+1;this->last!=0&&index!=this->last&&i>=index;i--)
                   this->items[i]=this->items[i-1];
                this->items[index]= data; /* insert data */
                this->last++; /* increment last */
                return true;
                }
```

```
        else
                {
                /* create a new item */
                newItem = new VectorData[ this->size + this->step];
                /* clear new vector */
                for(i=0;i<this->size+this->step;i++)newItem[i]=0;
                /* copy existing items */
                for(i=0;i<this->size;i++) newItem[i]= this->items[i];
                newItem[i++]=data;
                this->size = this->size + this->step; /* increase size */
                this->last = i; /* store last location */
                temp = this->items; /* temp point to new items */
                this->items = newItem; /* vector points to new items */
                delete(temp); /* free memory pointed to by temp */
                return true;
                }
        }
/* return item value at a specified index */
VectorData Vector::getVecAt (int index)
        {
        /* return value at vector location */
        if(index < this->last)return this->items[index];
        else return NULL;
        }
/* find data item in vector */
int Vector::findVec(VectorData data)
        {
        int i;
        /* loop through items looking for data item */
        for(i=0;i<this->last;i++)
             if(this->items[i]==data)return i; /* exit when found */
        return -1;
        }
/* replace a vector item */
bool Vector::replaceVec(VectorData newdata,VectorData olddata)
        {
        int index = findVec(olddata); /* get location of data item */
        if(index < 0) return false;
        /* if old data item found replace item with new data */
        this->items[index] = newdata;
        return true;
        }
```

```
/* replace a vector item with a new data value at a specified index */
/* return true if replaced */
bool Vector::replaceVecAt(VectorData newdata,int index)
        {
        VectorData data;
        /* index is less than last location and greater then zero */
        if((index < this->last) && (index >= 0))
                {
                data = this->items[index];
                this->items[index] = newdata; /* replace with new data */
                return true;
                }
        else return false;
        }
/* remove data item in vector */
int Vector::removeVec(VectorData data)
        {
        /* get location of data item */
        int index = findVec(data);
        if(index >= 0)removeVecAt(index); /* remove item */
        return index;
        }
/* remove a data item from the vector */
VectorData Vector::removeVecAt (int index)
        {
        int i;
        VectorData* newItem;
        VectorData* temp;
        VectorData data;
        /* check for out of bounds index */
        if((index > this->last) || (index < 0))return NULL;
        /* store value to remove */
        data = this->items[index];
        /* shift up items in vector */
        for(i=index; i<this->last; i++)this->items[i] = this->items[i+1];
        (this->last)--;
        /* check to deallocate memory */
        if(this->last <  this->size-this->step)
                {
                /* allocate memory for new vector items size - step */
                newItem =  new VectorData[this->size - this->step];
                /* copy vectors from old items to new items */
                for (i=0;i<this->last;i++)newItem[i]=this->items[i];

                temp = this->items; /* temp points to old vector */
                this->items = newItem; /* point to new vector memory */
                delete (temp); /* deallocate old vector memory */
                }
        return data;
        }
```

```
/* destructor to deallocate any allocated memory */
Vector:: ~Vector()
        {
        delete(this->items); /* deallocate memory */
        this->items = NULL; /* set all to default values */
        this->size = 0;
        this->step = 0;
        this->last = 0;
        }
/* print out vector items */
ostream& operator <<(ostream& out, Vector& vec)
        {
        int i;
        /* check for empty vector */
        if(vec.items == NULL)
                {
                out << "the vector is empty" << endl;
                return out;
                }
        out << "vector: [";
        /* loop through all items in vector */
        for(i=0;i<vec.last;i++)
                {
                out << vec.items[i]; /* print out values */
                if(i < (vec.last-1))out << ','; /* insert comma */
                }
        out << "]" << endl;
        return out;
        }

/* main program for vector module */
int main()
        {
        Vector vec(5,2); /* initialize a vector data type size,step */
        cout << vec; /* print out vector */
        /* insert 3 items into vector */
        vec.insertVecLast(6);
        vec.insertVecLast(3);
        vec.insertVecLast(9);
        cout << vec; /* print out vector */
        /* remove an item */
        vec.removeVec(8);
        cout << vec; /* print out vector */
        /* insert 5 items */
        vec.insertVecLast(8);
        vec.insertVecLast(12);
        vec.insertVecLast(4);
        vec.insertVecLast(12);
        vec.insertVecLast(7);
        cout << vec; /* print out vector */
```

```
/* remove 1 item */
vec.removeVec(6);
cout << vec; /* print out vector */
/* remove 5 items */
vec.removeVec(12);
vec.removeVec(4);
vec.removeVec(7);
vec.removeVec(6);
vec.removeVec(8);
cout << vec; /* print out vector */
/* replace items */
vec.replaceVec(13,12); /* replace 12 with 13 */
cout << vec; /* print out vector */
vec.replaceVecAt(9,1); /* replace 1 with 9 */
cout << vec; /* print out vector */
vec.replaceVec(5,vec.getVecAt(0)); /* replace item index 0 */
cout << vec; /* print out vector */
vec.replaceVecAt(17,vec.findVec(5)); /* replace 5 with 17 */
cout << vec; /* print out vector */
return 0;
} /* end main */
```

program output:

```
vector: []
vector: [6,3,9]
vector: [6,3,9]
vector: [6,3,9,8,12,4,12,7]
vector: [3,9,8,12,4,12,7]
vector: [3,9,12]
vector: [3,9,13]
vector: [3,9,13]
vector: [5,9,13]
vector: [17,9,13]
```

LESSON 2 EXERCISE 1

Edit copy paste or type in the Vector class and run it.

LESSON 2 EXERCISE 2

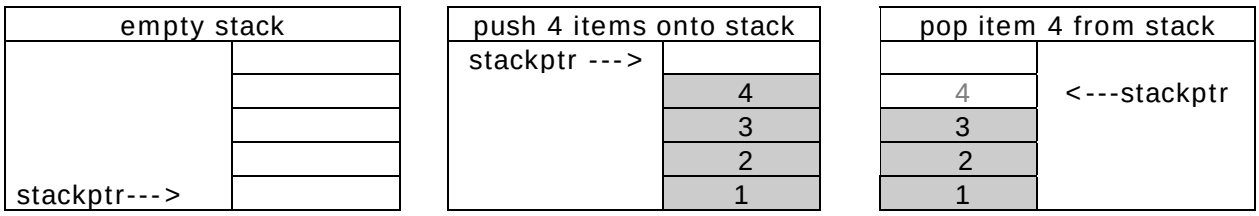
Modify the vector class so that it removes a vector item from the front of the vector. You will need a "first" pointer keeps track of the removed item location. This way we don't have to keep shifting data, when we remove an item from the Vector. When the first pointer meets the last pointer then delete memory for the vector. You must also modify the numItems function. You must modify removeVec and removeVecAt functions to allow for the first pointer modification, especially when vector items are removed at other locations.

LESSON 2 EXERCISE 3

Make the Vector class into a template class called TVector so that we can handle any data type.

STACK ADT CLASS

A **Stack** allows you to insert and retrieve items as **last in first out** (LIFO) into a list. This means if you insert the number 1, 2 then 3 you will get back 3, 2 and 1 in the reverse order. Stack operations are known as **push** and **pop**. To insert an item onto the stack a **push** operation is done. To remove items from the stack a **pop** operation is done. The location to insert the data item into the stack is pointed to by the **stack pointer**. The **stack pointer** is initially set to zero to the beginning of the stack. The beginning of the stack is known as the stack **bottom**. The end of the stack is known as the stack **top.** When an item is pushed onto the stack the stackptr is incremented **after** the operation. When a item is popped of the stack the stackptr is decremented **before** the operation.

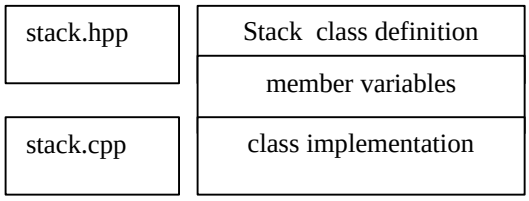


Think of a stack as a **stack of books** each being placed on top of each other as they are added. You make a stack with an array. You can use a fixed length array or a dynamically resizable array. You use a fixed length array when you know the maximum number of elements that will be pushed onto the stack. You uses an resizable array when you not know how many elements will be pushed onto the stack. Fixed stacks made from conventional arrays are prone to **overflow** and **underflow**.

Overflow occurs when you add an item and there is no room on the stack. **Underflow** occurs when you remove an item from the stack and the stack is already empty. When you use an resizable array these problems of overflow and underflow do not arise. For simplicity we will use a fixed array stack.

Implementing a Stack ADT

To implement a Stack you need functions to push elements and pop elements from the stack. You also need a stack pointer to point to the position in the array that represents the bottom of the stack. We use the modular approach to implement our Stack ADT where a structure is use to hold the pointers and stack array elements.



The header file contains the Stack class definitions
The member variables holds the stack array and stack pointer to be shared between all Stack class functions.

The code contains the class function definitions.

Here's the stack module. We have a small file defs.h that includes definitions for true, false, error and bool. If your compiler does not supply these for you then you need to include the defs.h file in stack.h.

```
/* defs.h */
#define true 1
#define false 0
#define error -1
typedef int bool;
```

```
/* stack.hpp */
#ifndef __STACK
#define __STACK
//#include "defs.h"
/* stack module data items */
typedef int StackData;
/* stack module data structure */
class Stack
{
private:
StackData* items; /* pointer to vector module */
int stkptr; /* stack pointer */
int size; /* max size of stack */
/* stack class function prototypes */
public:
/* constructor to initialize stack size */
Stack(int size);
/* put data item onto stack */
bool push(StackData data);
/* get data item from stack */
StackData pop();
/* print stack contents */
friend ostream& operator << (ostream& out, Stack& stk);
/* destructor to deallocate memory for stack */
~Stack();
};
#endif
```

Stack Implementation code file:

```
/* stack.cpp */
#include <iostream.h>
#include "stack.hpp"
/* stack class functions */
/* initialize stack  */
Stack::Stack(int size)
        {
        /* allocate memory for vector data structure */
        this->items=new StackData[size];
        this->stkptr = 0; /* set stackptr to top of stack */
        this->size=size; /* set max size of stack */
        }
/* push item into stack */
bool Stack::push(StackData data)
        {
        if(this->stkptr == this->size)return false; /* return false if stack full */
        this->items[this->stkptr]=data; /* insert into vector at stpkptr location */
        (this->stkptr)++; /* increment stack ptr */
        return true;
        }
```

```
/* get item from stack */
StackData Stack::pop()
        {
        if(this->stkptr <= 0)return 0; /* check if stack empty */
        --(this->stkptr); /* decrement stack pointer */
        return this->items[this->stkptr]; /* return element */
        }
/* print stack items */
ostream& operator << (ostream& out, Stack& stk)
        {
        int i;
        /* check for empty stack */
        if(stk.items == NULL)
                {
                out << "the stack is empty" << endl;
                return out;
                }
        /* print out stack elements */
        out << "Stack: [";
        /* loop through all items in stack */
        for(i=0;i<stk.stkptr;i++)
                {
                out << stk.items[i]; /* print out values */
                if(i < (stk.stkptr-1))out << ','; /* insert comma */
                }
        cout << "]" << endl;
        }
/* deallocate memory for stack */
Stack::~Stack()
        {
        delete (this->items);
        this->stkptr=0;
        this->size=0;
        }
/* main function to test stack module */
int main()
        {
        StackData data; /* data from stack */
        Stack stk(10); /* initialize stack to 10 elements */
        cout << stk; /* print stack contents */
        stk.push(8); /* push "8" into stack */
        cout << stk; /* print stack contents */
        data = stk.pop(); /* get item from stack */
        cout << "data from stack: "<< data << endl;
        data = stk.pop(); /* get item from stack */
        cout << "data from stack: "<< data << endl;
        stk.push(3); /* push "3" into stack */
        stk.push(7); /* push "7" into stack */
        stk.push(4); /* push "4" into stack */
        cout << stk; /* print stack contents */
        data = stk.pop(); /* get item from stack */
        cout << "data from stack: "<< data << endl;
```

```
cout << stk; /* print stack contents */
data = stk.pop(); /* get item from stack */
cout << "data from stack: "<< data << endl;
cout << stk; /* print stack contents */
data = stk.pop(); /* get item from stack */
cout << "data from stack: "<< data << endl;
cout << stk; /* print stack contents */
return 0;
}
```

program output:

```
stack: []
stack: [8]
data from stack: 8
data from stack: -1
stack: [3,7,4]
data from stack: 4
stack: [3,7]
data from stack: 7
stack: [3]
data from stack: 3
stack: []
```

LESSON 2 EXERCISE 4

Edit copy paste or type in the Stack module and run it. Add an **isEmptyStk()** function to the Stack. Push some values in the Stack and use the **isEmptyStk()** method to pop all the values from the stack till empty and print them out.

LESSON 2 EXERCISE 5

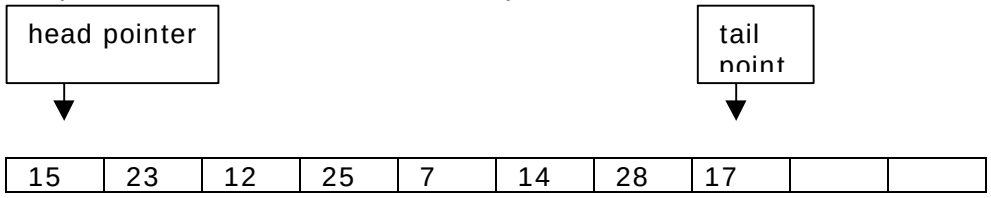
Use the Vector module to make a Stack.

LESSON 2 EXERCISE 6

Make a Stack template class called TStack so that we can handle any data type.

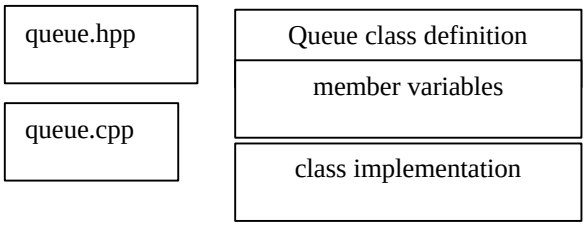
QUEUE ADT MODULE

A Queue let you add items to the end of a list and to remove from the start of a list. A Queue implements **first in first out** (FIFO). This means if you insert 1, 2, and 3 you will get back 1,2 and 3. A Queue has both a **head pointer** and a **tail pointer**. Think as a Queue as a line in a bank. People enter the bank and stand in line. People get served in the bank at the start of the line. As each person is served they are removed from the Queue. Newcomers must start lining up at the end of the line. The first people who enter the bank are the first to be served and the first to leave. When you insert an item on the Queue it is known as **enqueue** and the tail pointer increments. When the tail pointer comes to the end of the array it wraps around to the start of the array. When you remove an item from the Queue it is known as **dequeue**. . When the tail pointer comes to the end of the array it wraps around to the start of the array.



Implementing a Queue ADT

T o implement a Queue you need functions to enqueue elements and deqeue elements. You also need first and last pointers to place elements at the end of the queue and remove elements at the start of the queue. We use the modular approach to implement our Queue ADT where a structure is use to hold the pointers and queue array elements.



The header file contains the class function prototypes.
The data member variables holds the array and variables and stack pointer to be shared between all Stack class functions.
The implementation contains the class function definitions.

Here's the stack class. We have a small file defs.h that includes definitions for true, false, error and bool. If your compiler does not supply these for you then you need to include the defs.h file in stack.h.

```
/* defs.h */
#define true 1
#define false 0
#define error -1
typedef int bool;

/* queue.hpp */
/* queue module header file */
#ifndef __QUEUE
#define __QUEUE
typedef int QueueData;
/* queue module data structure */
class Queue
{
private:
QueueData* items;
int head; /* point to first element in queue */
int tail; /* point to last element in queue */
int size; /* size of queue */
/* queue class function prototypes */
public:
/* constructor to initialize stack queue */
Queue(int size);
/* put item into queue */
bool enqueue(QueueData data);
/* remove data item from queue */
QueueData dequeue();
/* destructor deallocate memory for queue */
~Queue();
/* print out contents of queue */
friend ostream& operator <<(ostream& out, Queue& que);
};
#endif
```

Implementation code file:

```
/* queue.cpp */
#include <iostream.h>
#include "queue.hpp"
//#include "defs.h"
/* queue module functions */
/* initialize queue */
Queue::Queue(int size)
{
/* allocate memory for vector data structure */
this->items= new QueueData[size];
this->head=0;
this->tail=0;
this->size=size;
}
/* insert items into the queue */
bool Queue::enqueue(QueueData data)
{
/* insert item at end of vector */
this->items[this->tail]=data;
(this->tail)++; /* increment tail */
this->tail= this->tail % this->size; /* wrap around */
return  true;
}
/* remove items from the queue */
QueueData Queue::dequeue()
{
QueueData data;
/* check if queue empty */
if(this->head==this->tail)return 0;
/* return and remove item at start of vector */
data = this->items[this->head];
(this->head)++; /* increment head */
this->head= this->head % this->size; /* wrap around */
return data;
}
/* deallocate memory for queue */
Queue::~Queue()
{
/* deallocate memory for vector */
delete (this->items);
this->items=NULL;
this->head=0;
this->tail=0;
this->size=0;
}
/* print out queue items */
ostream& operator <<(ostream& out, Queue&  que)
{
int i;
```

```
        /* check for empty vector */
        if(que.items == NULL)
                {
                cout << "the queue  is empty" << endl;
                return out;
                }
        /* print out queue elements */
        cout << "Queue: [";

        /* loop through all items in vector */
        for(i=que.head;i!=que.tail;i= (i + 1) % que.size)
                {
                cout << que.items[i]; /* print out values */
                if(i < (que.tail-1))cout <<','; /* insert comma */
                }
        cout << "]" << endl;
        }
/* main function to test queue module */
int main()
        {
        Queue que(10); /* create and initialize queue to 10 elements */
        que.enqueue(5); /* put 5 into queue */
        cout <<  que; /* print out queue */
        /* get data item from queue and print out value */
        cout << "data value: " << que.dequeue() << endl;
        cout <<  que; /* print out queue */
        que.enqueue(1); /* put 1 into queue */
        cout <<  que; /* print out queue */
        que.enqueue(2); /* put 2 into queue */
        cout <<  que; /* print out queue */
        que.enqueue(3); /* put 3 into queue */
        cout <<  que; /* print out queue */
        /* get data item from queue and print out value */
        cout << "data value: " << que.dequeue() << endl;
        cout <<  que; /* print out queue */
        /* get data item from queue and print out value */
        cout << "data value: " << que.dequeue() << endl;
        cout <<  que; /* print out queue */
        /* get data item from queue and print out value */
        cout << "data value: " << que.dequeue() << endl;
        cout <<  que; /* print out queue */
        return 0;
        }
```

program output:

```
queue: [5]
data value: 5
queue: []
queue: [1]
queue: [1,2]
queue: [1,2,3]
data value: 1
queue: [2,3]
data value: 2
queue: [3]
data value: 3
queue: []
```

LESSON 2 EXERCISE 7

Edit copy paste or type in the Queue module and run it. Add an **isEmptyQue()** function to the Queue. Enqeue some values in the Queue and use the **isEmptyQue()** method to deqeue all the values and print them out.

LESSON 2 EXERCISE 8
Make a Queue where all the elements are inserted in ascending order. This queue is called a Priority queue. Call your modules pqueue.h and pqueue.c.

LESSON 2 EXERCISE 9
Use the Vector module to make a Queue. Call your modules vqueue.h and vqueue.c.

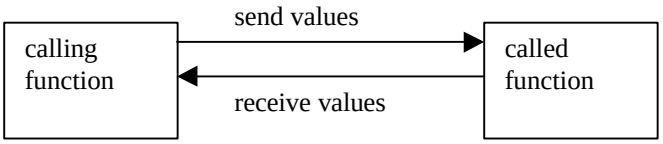
LESSON 2 EXERCISE 10
Push some values in a Stack and pop the values into a Queue. Deqeue the Queue and push into a Stack. Pop the Stack and print out the values. Make a separate main function in a file called L2Ex10.c

LESSON 2 EXERCISE 11
Make a Queue template class called TQueue so that we can handle any data type.

C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 3

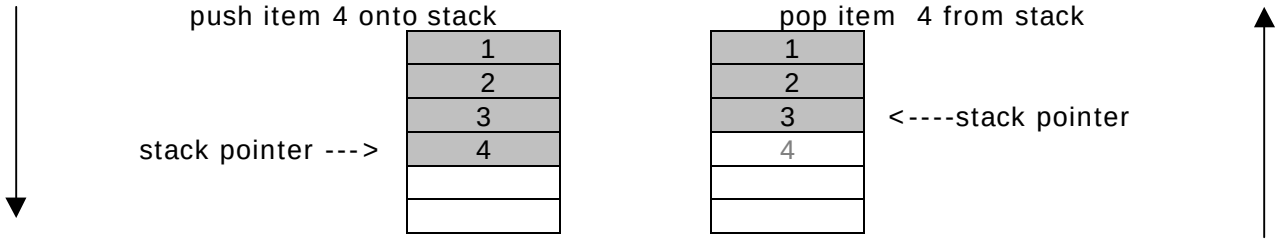
File:	CppdsGuideL3.doc
Date Started:	April 15,1999
Last Update:	Dec 22, 2001
Status:	draft

LESSON 3 RECURSION



RECURSION

Recursion is very powerful in programming. Recursion can solve very complicated problems in only a few lines of code. Recursion is very difficult to understand. It seems to work like magic. In order to understand recursion you need to know how a program calls and executes a function. Once you understand this, all your problems are over. When a function calls another function the **calling** functions passes arguments to the **called** function. There must be some mechanism that allows the function that is called to receive values. There must also be a mechanism that lets the program return values to the calling function and then return back to the calling function, once the called function finishes executing. There is such a mechanism and it is called an **execution stack**. From the lesson on **abstract data types** (ADT's) you learned that a stack provides storage for items in a last in first out arrangement (LIFO). This means the last item to be pushed onto the stack in the first item to be popped from the stack. This also means the fist item to be pushed onto the stack in the last item to be popped from the stack. Items being stored are **pushed** onto the stack and items being retrieved are **popped** from the stack An stack item is pointed to by a stack pointer Items popped from the stack are in reverse order from when they were pushed onto the stack. These stacks are hardware stack and work differently from software stacks. The stack pointer is at the top of the stack and decrements when pushed and increments when popped.



The execution stack and stack pointer is implemented by the CPU and works automatically in hardware. The direction for push and pop depends on the implementation. The stack pointer may be incremented or decremented for push and pop operations. The top of the stack is known as the **stack top.** A stack is needed when your running program calls another function. When a function calls another function it pushes its arguments onto the execution stack. The called function must keep track where the arguments are pushed on the stack. The compiler knows how many items were pushed onto the stack. The called function knows where each parameter is by using an offset from the stack pointer. Each parameter will have a hard coded offset from the stack pointer. The calling function must also push the return address onto the stack and make space for any return values. When the called function finishes executing it must know where to continue program execution in the calling function. When the return address and arguments are pushed onto the execution stack this is known as a **stack frame**. The stack frame convention is that the parameters are pushed first in listed order and then the return address is pushed onto the stack.

After the called program finishes executing the returned address is popped off the stack and program execution resumes to the program statements after where the function was called. Finally the arguments are popped off the stack and may or may not be assigned to the variables in the calling function. The stack pointer then returns to its original position.

stack frame

return value
parameter 1
parameter 2
parameter n
return address

<-------- stack pointer

example calling a function

The following program demonstrates a stack frame using a multiply function. The multiply function has two parameters x and y. If x = 5 and y = 4 then mul (5, 4) would be equal to 20.

```
/* program multiplies two numbers L3p1.c */
#include <iostream.h>

/* function prototype */
int mul(int x, int y);
/* main function */
void main()
{
int x = mul(5,4);  /* call mul function */

/* print out answer */
cout << "5 times 4 is: " << x << endl;
}
```

```
/* multiply two numbers */
int mul(int x, int y)
{
int product = 0; /* initialize product to 0 */

/* loop till y decrements to zero */
while(y > 0)
     {
     product = product + x; /* add x to product */
     y = y - 1; /* decrement y */
     }
return product; /* return answer */
}
```

The stack frame when the **mul** function is executing is:

stack frame

offset	parameter	initial value	final value
6	return value	?	20
4	x	5	1
2	y	4	5
0	return address	main	main

<-------- stack pointer

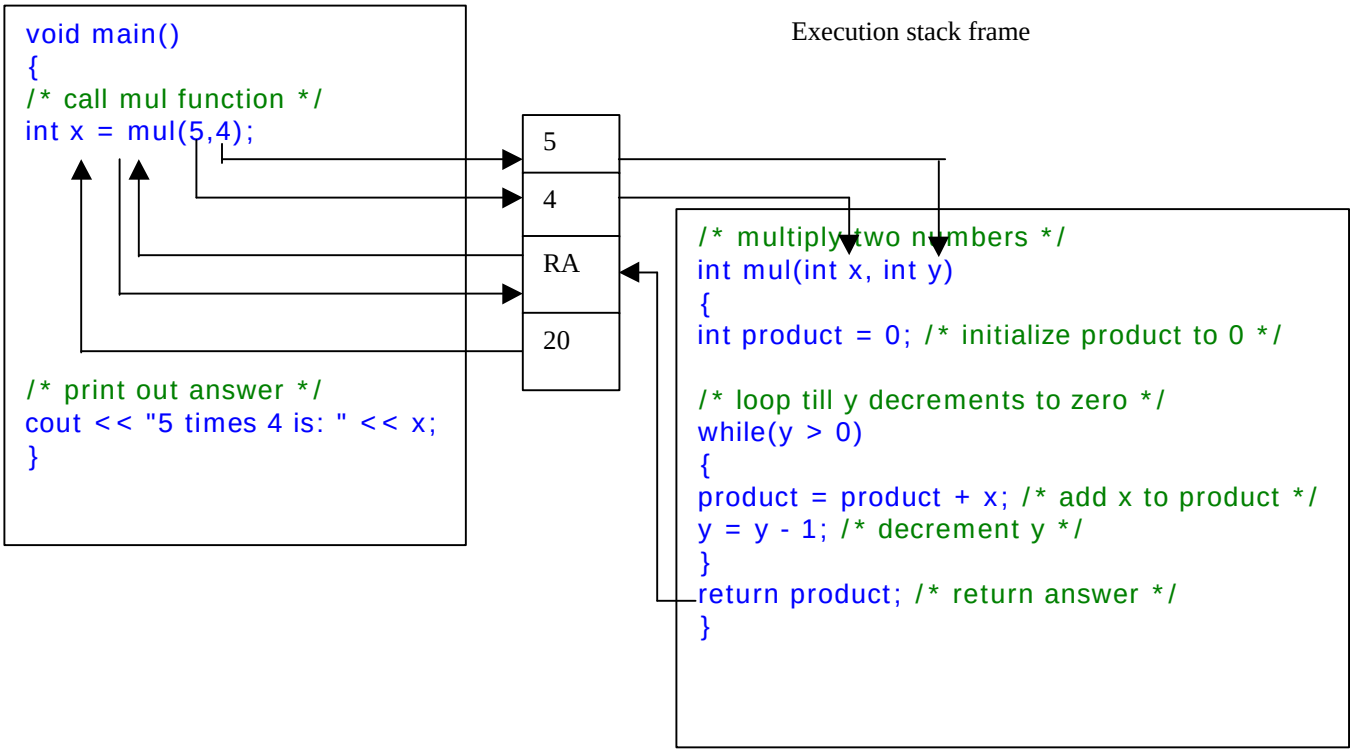
The function uses the stack frame as temporary locations to hold the parameters. The stack frame is also used to hold temporary locations for local variables declared in the function. When the function is finished executing the return address is popped off the stack and program execution resumes after the programming statement where the called function was called.

```
int x = mul(5,4);
```

returning point | calling point

Now x will get the value 20 that was popped of the stack.

Here is a data flow and operational diagram:



example using recursive function

With a recursion function the operation is the same. The only difference is that the called function calls itself. That's right the called function calls itself and it also becomes a calling function. In this situation many stack frames will be generated. For every time a function is called a stack frame is created. For every time a function returns a stack frame disappears. We can demonstrate recursion using a recursive multiply function. The multiply function has two parameters x and y. If x = 5 and y = 4 then mul (5, 4) would be equal to 20. We can rewrite the mul() function by replacing the while loop with a recursive function. The y parameter is used as a down counter just as in the while loop version.

```
/* while loop multiply function */
int void mul(int x, int y)
        {
        int sum = 0;
        /* loop till end */
        while(y > 0)
                {
                sum = sum + x; /* add x */
                y = y - 1; /* decrement y */
                }
        return sum;
        }
```

```
/* recursive multiply function */
int void mul(int x, int y)
        {
        if(y == 0)return 0;

        /* return x if end of recursion */
        else if(y == 1)return x;

        /* add x and decrement y */
        else return x + mul(x,y-1);
        }
```

The recursive method has less lines but works mysteriously. Lets figure out how it works. The main function is identical as before.

```
/* l6p2.c  lesson 3 program 2 */
/* this program multiplies two numbers recursively */
#include <iostream.h>
/* function prototypes */
mul(int x, int y);
/* main function */
void main()
        {
        int x = mul(5,4);  /* call mul function */
        cout << "5 times 4 is: " << x << endl; /* print out answer */
        }
/* multiply two numbers recursively/
int mul(int x, int y)
        {
        if(y == 0)return 0;
        else if(y == 1)return x;
        else return x + mul(x,y-1);
        }
```

push onto stack:

5
4
main

The **secret** for understanding how recursion works lies in the execution stack. Every time a recursive function calls itself a new stack frame is formed. This means there will be a separate stack frame for every recursive function call. For every stack frame the function will access the **same parameters** but **different values**. The stack pointer points to the stack frame that is presently being executed by the recursive function. You can think of a stack frame as storing separate set of parameter and variable values for every time the function calls itself. When the function returns you get the previous set of values. For a value of y equal to 4 we will have 4 stack frames.

execution stack frame 1	return value	?	mul(5,4)	
	x	5		
	y	4		
	return address	main		
execution stack frame 2	return value	?	mul (5,3)	
	x	5		
	y	3		
	return address	mul(5,4)		
execution stack frame 3	return value	?	mul (5,2)	
	x	5		
	y	2		
	return address	mul(5,3)		
execution stack frame 4	return value	?	mul (5,1)	
	x	5		
	y	1	stack pointer	
	return address	mul(5,2)	<--	

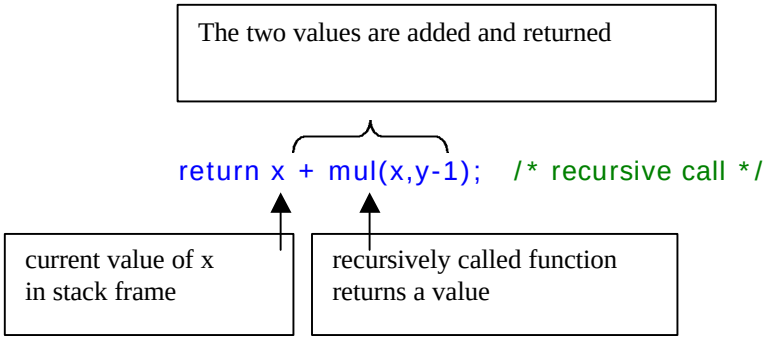
Every time a function is called a stack frame is created. For each time the recursive function terminates(returns) the stack pointer points to the last execution stack frame. Program execution is transferred to the return address stored in the stack frame. The return address can be the recursive function or the main function. When the stack pointer reaches the first stack frame then program execution returns to the calling function. This is because the first stack frame contains the return address of the original calling function main. To understand how our mul() function works we make a **chart**. A chart is an excellent way to trace how a program works. Always make a chart to see how a program works. When you write an exam make a chart, a chart will display all the variable values at different points in time. A chart will organize things for you. It is better to use a chart then try to remember every thing in your head. It is very easy to get mixed up and loose track of things in your head.

```
/* here is the recursive multiply function again for your reference: */
int void mul(int x, int y)
        {
        if(y == 0)return 0;  /* (1) return 0 if y is 0 */
        else if(y == 1)return x;  /* (2) return x when y is 1 */
        else return x + mul(x,y-1);  /* (3)call and add previous results */
        }
```

To make a chart you need to list all the variables and test conditions at the top for each column. The rows will become the stack frame that is executing. The first row contains all the initial values. RA means return address and --- means no execution.. The stack grows from left to right.

stack frame	stack contents	x	y	y==0 line (1)	y==1 line (2)	y-1 line (3)	call/ return
0		--	--	---	---	---	mul(5,4)
1	5,4,RA	5	4	F	F	3	mul(5, 3)
2	5,4,RA,5,3,RA	5	3	F	F	2	mul(5,2)
3	5,4,RA,5,3,RA,5,2,RA	5	2	F	F	1	mul(5, 1)
4	5,4,RA,5,3,RA,5,2,RA,5,1,RA	5	1	F	T	----	5
3	5,4,RA,5,3,RA,5,2,RA	5	2	---	----	----	5 + 5 = 10
2	5,4,RA,5,3,RA	5	3	---	----	----	5 + 10 = 15
1	5,4,RA	5	4	---	----	----	5 + 15 = 20
main		---	---	---	-----	----	20

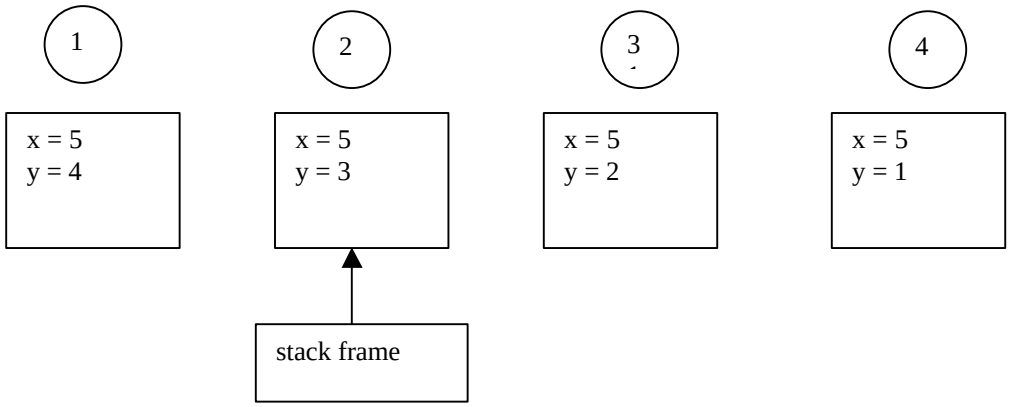
Two important things are happening here The y variable is a counter and is used to count how many recursive calls we need. For each stack frame the y and x values are pushed onto the stack. The y values are decreasing and the x value never changes. When y reaches the value 1 then the function does not call itself any more, recursion is stopped. When y is 1 then the current value of x that is in the stack frame is returned to the calling function. The calling function happens to be the function itself ! The other important thing to realize is that the following statement is returning the value x plus the return value of the recursive call.



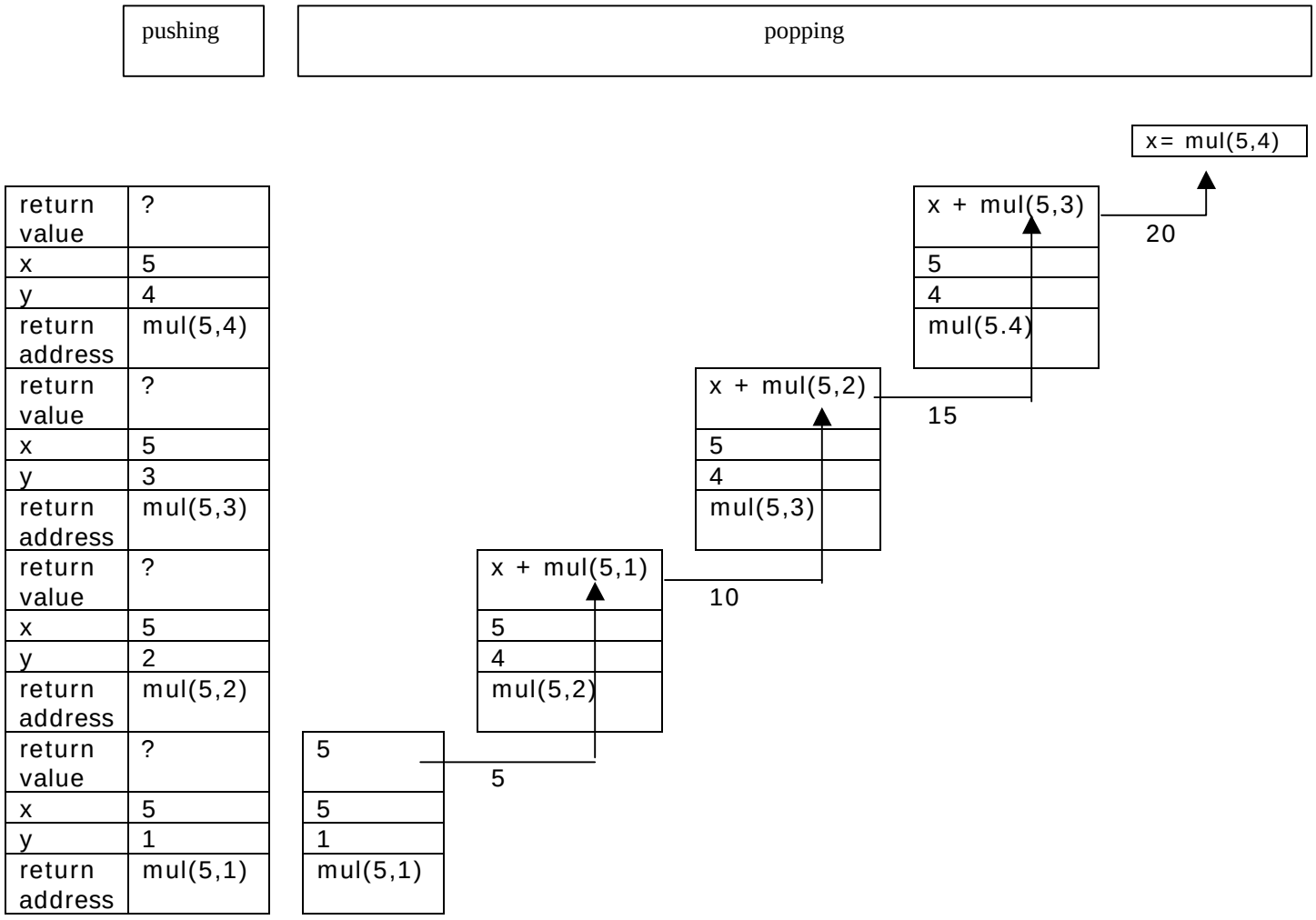
This means the recursive function is returning a value that is calculated from adding a variable value and the return value from the function calling itself. Every time the function calls itself a new stack frame is produced. When a function call itself the return value is not available until the function returns with a value. Every time a function returns the stack frame disappears and the return value given to the function in the preceding stack frame.

```
/* here is the recursive multiply function again for your reference: */
int void mul(int x, int y)
{
if(y == 0)return 0;  /* (1) return 0 if y is 0 */
else if(y == 1)return x;  /* (2) return x when y is 1 */
else return x + mul(x,y-1);  /* (3)call and add previous results */
}
```

You can also thing of recursion as storing a separate set of values for every recursive call. When the function returns it selects the preceding set of values using the stack frame pointer and discards the current set of values.



Here is a trace of what is actually happening:



Since we do not know how many functions we need we only call 1 function and use the counter y to keep track how many times the function must call itself. The counter y keeps track how many times the function mul(x,y) will be called. When y is equal to 1 then the recursive function stops calling itself and starts returning and popping values off the execution stack frame. Recursive functions uses counters allot. This is the magic secret about recursion., by solve a problem using a different approach. The most important thing to realize is that when the recursive function finishes executing it returns to the point where it was called not at the beginning of the function. This is why it only evaluates y once. As soon as y becomes 1 the function stops calling itself and returns x. When the function finishes executing it returns to the point after where it was called. not at the beginning of the function. Just remember where every a recursive function is called it returns to the calling point. Think of it like this. For every door you go in you have to come out of that door.

return x + **mul(x,y-1);**

calling and returning point

two values added and returned

A function returns at a return statement or the last bracket in a function. This is an implied return statement.

```
/* here is the recursive multiply function again for your reference: */
int void mul(int x, int y)
{
if(y == 0)return 0;  /* (1) return 0 if y is 0 */
else if(y == 1)return x;  /* (2) return x when y is 1 */
else return x + mul(x,y-1);  /* (3)call and add previous results */
}
```

IMPLEMENTING RECURSION WITH A STACK

If you still have difficulty understanding or you did not understand the trace out of the recursive program then try to think recursion is identical to a loop and a stack with push and pop operations. We will rewrite the mul(int x, int y) function using a stack ADT. The push () operation will put a value on the stack and the pop() operation will retrieve a value from the stack.

```
/* multiply x by y using a stack */
int void mul(int x, int y)
        {
        int i,product = 0;
        for(i=0;i<y;i++)push(x); /* push x onto stack */
        for(i=0;i<y;i++)product = product + pop(x); /* pop values from stack */
        return product; /* return result */
        }
```

The result using a stack is the same as recursion except there is no return address.

i	x	y	stack	sum	return	
0	5	4	5	0		pushed
1	5	4	5,5	0		
2	5	4	5,5,5	0		
3	5	4	5,5,5,5	0		
0	5	4	5,5,5	5		popped
1	5	4	5,5	10		
2	5	4	5	15		
3	5	4	------	20	20	

Hers is the complete program that you can type in and try or trace. We have made our own mini stack ADT for you.

```
/* l6p3.cpp  lesson 3 program 3 */
/* this program multiplies two numbers recursively */
#include <iostream.h>
/* data structures */
typedef int StackData;
class Stack
{
private:
StackData* items;
int size;
```

```
int stkptr;
public:
/* class functions declarations */
Stack(int size); /* initialize stack module */
bool push(StackData data);  /* push item into stack */
StackData pop();  /* get item from stack */
};

/* function prototypes */
int mul(Stack& stk, int x, int y);  /* multiply two numbers  */
/* main function */
void main()
        {
        Stack stk(100);
        int x;
        x = mul(stk,5,4);
        cout << "5 times 4 is: " << x << endl;
        }
/* multiply two numbers using a stack */
int mul(Stack& stk,int x, int y)
        {
        int i;
        int sum = 0;
        for(i=0;i<y;i++)stk.push(x);
        for(i=0;i<y;i++)sum = sum + stk.pop();
        return sum;
        }
/* stack code implementation */
/* initialize stack */
Stack::Stack(int size)
        {
        this->items= new StackData[size];
        this->stkptr = 0;
        this->size = size;
        }
/* push item into stack */
bool Stack::push(StackData data)
        {
        if(this->stkptr > this->size)return false;
        this->items[this->stkptr++] = data;
        return true;
        }
/* get item from stack */
StackData Stack::pop()
        {
        if(this->stkptr <= 0)return -1;
        else return this->items[--(this->stkptr)];
        }
```

We have examined 3 method to multiply two numbers together.

method	comment	operation
while loop	medium lines of code	faster
recursion	few lines of code	medium
stack	large overhead	slowest

The while loop is the fastest operation but needs more lines of code. The recursive method is preferred because there are fewer lines to code. The stack eliminates temporary variables. The stack method is to be avoided because the recursive method automatically uses a stack for you.

LESSON 3 EXERCISE 1

Type in the recursive mul() function and main program. use the debugger to trace through it and watch it work. Write down in order the statements it executes.

LESSON 3 EXERCISE 2

Write a recursive function to find the power of two numbers. For example 5 to the power of 4 is 5 * 5 * 5 * 5 = 625. Write the main function to test your recursive power function.

LESSON 3 EXERCISE 3

Write a recursive method to find the factorial of a number n!

```
n! = n * (n-1) * (n-2) * (n-3) ...
4! = 4*3*2*1 = 24
```

LESSON 3 EXERCISE 4

Write a recursive method to test if a string is a palindrome. A palindrome is a string that reads the same backwards as front words. For example "radar" is a palindrome. You may want to write a lop version first and then convert to a recursive function.

LESSON 3 EXERCISE 5

Write a recursive function to find the square root of a number. Use Newton's formula for finding square roots.

```
pn+1 = (2 + x/pn)/2
```

Stop the recursion when x - (pn+1 * pn+1) < .0001

APPLICATIONS TO RECURSION

We will study the most famous recursive procedures

1. fibonnaci numbers
2. greatest common denominator
3. towers of Hanoi

FIBONNACI NUMBERS

The fibonnaci numbers define a number series obeying the following relations

fn = n for n = 0 or n = 1 termination condition
fn = fn-1 + fn-2 for n > 1 recurrence relation

It is easy for us too write the recursive function for the fibonnaci numbers using the above relations We can use a chart to calculate the fibonicci numbers from 3 to 5 1 to 2 is assumed to be 1

n	f(n-1)	f(n-2)	fn = fn-1 + fn-2
1	0	0	0
2	1	0	1
3	1	1	2
4	2	1	3
5	3	2	5
6	5	3	8

```
#include <iostream.h>
/* function prototypes */
int fib(int n);
/* main function */
void main()
        {
        int x = fib(4);
        cout << "the fibonnaci number  is: " << x << endl;
        }
/* calculate fibonnaci number for  n */
int fib(int n)
        {
        if( n <= 1) return n;
        else return fib(n-1) + fib (n-2); /* recurrence relation */
        }
```

Double recursion

To see how this recursive function works lets make a chart we will use n = 4. You must understand when we return we are returning the results of the last f(n-1) + f(n-2) operations. These operations are built up or calculated from previous operations. This type of recursion is considered inefficient,

stack frame	stack contents	n	n <= 1	fib(n-1)	fib(n-2)	fib(n-1) +fib(n-2)
1	4	4	F	fib(3)	-----	-----
2	3,4	3	F	fib(2)	-----	----
3	2,3,4	2	F	fib(1)	-----	-----
4	1,2,3,4	1	T	1	-----	-----
3	2,3,4	2	T	-----	fib(0)	-----
4	0,2,3,4	0	-----	-----	0	-----
3	2,3,4	2	-----	-----	-----	1 + 0 = 1
2	3,4	3	F	-----	fib(1)	-----
3	1,3,4	1	T	-----	1	----
2	3,4	3	-----	-----	----	1 + 1 = 2

1	4	4	F	-----	fib(2)	-----
2	2,4	2	F	fib(1)	-----	-----
3	1,2,4	1	T	1	----	-----
2	2,4	2	F	-----	fib(0)	-----
3	0,2,4	0	T	-----	0	-----
2	2,4	2	----	-----	-----	1 + 0 = 1
1	4	4	-----	-----	-----	2 + 1 = 3

The result of this solution is kind of interesting it must validate all the possibilities fib (n-1) and the ne-2. The most others interesting thing is that the recursive function is coded exactly like the recurrence relation,

GREATEST COMMON DENOMINATOR

The other function is FINDING the greatest common denominator of two given numbers The greatest common denominator of two number id the largest number that bot divide evenly into. For example the greatest common denominator of 25 and 35 is 5 since 5 is divisible by both, and it is the greatest denominator. The gcd algorithm use's Euclid's method

for finding the gcd of two number m and n we use the function:

```
gcd = gcd(m,n);
```

if m == n then gcd = m otherwise replace the larger of m or n by the difference between them

```
if m > n then gcd(m-n) = gcd(m-n,n)
```

Some math whiz has just told us if m >= n then if n % m is equal to n when m is greater than n.

```
gcd(m,n) = gcd(n,m mod n)
```

Here's the recursive gcd function using both methods. The second method using the mod is more efficient because it takes less calculations by applying the short cut.

```
/* this program multiplies two numbers */
        #include <iostream.h>
        /* function prototypes */
        gcd1(int m, int n);
        gcd2(int m, int n);
        /* main function */
        void main()

        {
        int x;
        x  = gcd1(25,35);
        cout << "the gcd of 25 and 35 is: " << x << endl;
        x  = gcd2(35,25);
        cout << "the gcd of 35 and 25 is:" << x << endl;
        }
```

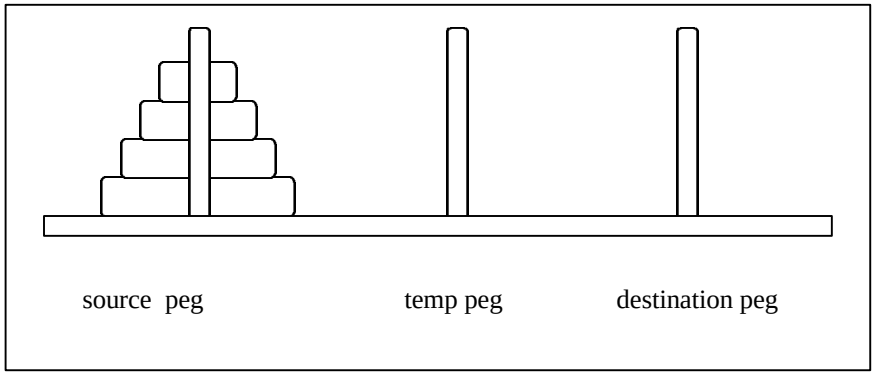
% means successive subtraction

```
int gcd(int m, int n)
        {
        if(m == n) return m;
        else if (m > n)
        return gcd(m-n,m);
        else return gcd(m,n-m);
        }
```

```
int gcd(int m, int n)
        {
        if(m == 0) return n;
        else if  (n == 0) return m;
        else return gcd(n, m % n);
        }
```

towers of hanoi

This is a famous puzzle to solve by recursion that nobody understand how it works. There are three wooden pegs. labeled 1 to 3. On the first the pegs there are 4 discs of different sizes. The disks are arranged from largest to smallest where the top is the smallest. The idea is to move the disks from peg 1 the source peg and placed on peg 3 the target peg in the same order as arranged on peg 1. Peg2 is used as a temporary place holder, the rule is you cannot have a larger disk on a smaller disk at any time.



The procedure for moving the disks are as follows

(1) move the n-1 smaller disks to the temporary peg
(2) move the n disk to the target peg
(3) move the n-1 smaller disk from the temporary peg to the target peg

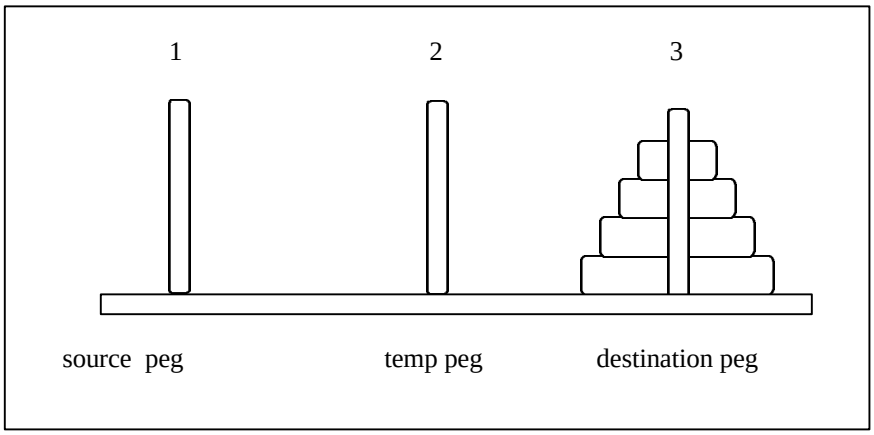
Here is the recursive function for the towers of hanoi:

```
/* towers of hanoi L3p6.c  */
#include <iostream.h>

/* function prototypes */
void hanoi(int n, int source, int temp, int target);
/* main function */
void main()
{
hanoi(4,1,2,3);
}
```

```
/* towers of hanoi */
void hanoi(int n, int source, int temp, int target)
{
if(n > 0)
     {
     hanoi(n-1,source,target,temp);
     cout << "move disk" << n
          << " from peg " << source
          << " to peg  " << target << endl;
     hanoi(n-1,temp,source,target);
     }
}
```

temp <-- target target<-- temp

source <-- temp source<-- temp



program output for n = 4

```
move disk 1 from peg 1 to peg 2
move disk 2 from peg 1 to peg 3
move disk 1 from peg 2 to peg 3
move disk 3 from peg 1 to peg 2
move disk 1 from peg 3 to peg 1
move disk 2 from peg 3 to peg 2
move disk 1 from peg 1 to peg 2
move disk 4 from peg 1 to peg 3
move disk 1 from peg 2 to peg 3
move disk 2 from peg 2 to peg 1
move disk 1 from peg 3 to peg 1
move disk 3 from peg 2 to peg 3
move disk 1 from peg 1 to peg 2
move disk 2 from peg 1 to peg 3
move disk 1 from peg 2 to peg 3
```

CONCLUSION

We have been introduced to recursion and examined four recursive functions. If you can underrated these examples completely then you are doing very well.

LESSON 3 EXERCISE 6

Type in the above recursive functions and write the main functions for them. use the debugger to trace through each function rum. Make charts for the gcd and towers of hanoi.

LESSON 3 EXERCISE 7

Write a recursive function to determine if a number is prime. If a number ha no factors then it is prime. The number 13 is prime because nothing can divide into it. a number is prime if it has no factors between 2 and square root of n**.** Need a square root function then see exercise 5 !

LESSON 3 EXERCISE 8

Write a recursive program to generate binomial coefficients C(N,K)

C(N,k) = 1 for k = 0
C(N,N) = 1 for k = N
C(N,k) = C(N-1,k) + C(N-1, k -1) for (0 < k < N)

An example output would be:

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

LESSON 3 EXERCISE 9

Given a string. write a recursive program that will generate all possible combinations of letter arrangements. Example "abc" can generate "acb" "bac" etc.

LESSON 3 EXERCISE 10

Write a recursive program that will generate all the possible words for a 7 digit phone number

2	**a b c**
3	**d e f**
4	**g h i**

5	**j k l**
6	**m n o**
7	**j r s**

8	**t u v**
9	**w x y**

LESSON 3 EXERCISE 11

Write a recursive function to find the determinant of a square matrix.. Use the equation:

det = sum of (-1) * *(i+j) *mij * Mij

where mij is a particular element in the matrix at row i and column j and Mij is a matrix not including row i and column j. Each Mij will be 1 size smaller than the previous 1. You need to keep track of each row and column used by individual rows to determine the matrix you are working on. The determinant of size 0 is 1. The determinant of a matrix of size 1 is the element itself. Every time you recurse you use the above formula to calculate the matrix and decrease the size of the matrix by 1.

The matrix of a 2 * 2 matrix

```
 4  -5
-1  -2
```

det is : 4 (-2) - (-5) (-1) = -13

LESSON 3 QUESTION 12

Write a recursive function that automatically guesses a word. When a letter is correct it remembers it and calculates the remaining letters until the word is guessed. The user types in a word and then the program guesses it. The program is only told of the correct letters when it they are correctly guessed. Keep track of the number of tries. Call your file L3ex8.c

LESSON 3 QUESTION 13

Write a recursive method to find the missing numbers in the square matrix. The columns, rows and both diagonals must add up to the given sums. The numbers in the square can only be between 1 and 9. You will have to optimize your solution so that each square can have a possible minimum values and maximum value as to speed up your solution calculations.

				12
5		4	3	15
	1	2	7	23
5	6		2	17
	1	4	3	18
12	13	20	23	17

C++ DATA STRUCTURED PROGRAMMERS GUIDE LESSON 4

File:	CppdsGuideL4.doc
Date Started:	April 15,1999
Last Update:	Dec 22,2001
Status:	draft

LESSON 4 SORTING ARRAYS

SORTING ALGORITHMS

Sorting is arranging a array or list in **ascending** or **descending** order. Ascending means the numbers are increasing 1, 2 3, 4 .... Descending means the numbers are decreasing 8,7,6... There are many sort algorithms. The goal of a sorting algorithm is to sort fast. It is almost a contest to see which algorithm is the fastest. Sort time is measured by **big O** notation. Big O notation states the **worst case** running time. An example if you had an array of n items and you wanted to search for an item, we can use big O notation to estimate the running time to find the item. If the item was the last element, and we start searching at the start of the array then the search time would take n comparisons. This would be the worst case. If the item to be found was at the start of the array then this would be the best case. Unfortunately , we always need to state the worst case. No matter where the item is the search time is the worst case n items. This is what big O notation is all about, the worst case estimate. Big O states the worst case. No matter where your item is it is still O(n). Constants are not to be included in big O notation O(2n) is the same as O(n). Most of the sort routines are O(n**2) (n squared) this because they both uses a loop inside a loop. The inner loop is O(n) and the outer loop is O(n). When you have loops inside loops then the worst case timing is a multiplication of the two separate loops. Total worst case timing is O(N) * O(N) = O(n ** 2). There are many sort algorithms. The following chart lists the sorting algorithm with worst case running time.

sort algorithm	worst case timing	comment
bubble sort	O(n**2)	simple
insertion sort	O(n**2)	simple
selection sort	O(n**2)	simple
shell sort	O(n**1.5)	not simple
merge sort	O(n log n)	recursive
quick sort	O(n log n)	recursive

BUBBLE SORT

This is the most common sorting algorithm. It works by going over a **n** array of element **n** times always exchanging or swapping 2 elements at a time. If the left element is greater than the right element then the elements are swapped. Since the array will be examined **n** times we are assured that the array will be sorted. There will be 6 iterations. For each iteration the bubble sort algorithm will scan the array swapping elements if the left element is greater than the right element. Each element pairs are scanned sequentially. This means an element could be swapped more than once. In the following table we show the swapped pairs shaded in gray for each iteration. We show complete scan results for iteration 1. At iteration 6 the array is sorted. For each iteration we scan left to right, swapping any left element greater than the right element.

scan and swap

initial array	2	6	5	4	3	1

iteration 1	2	**5**	**6**	4	3	1	swap 6 and 5
	2	5	**4**	**6**	3	1	swap 6 and 4
	2	5	4	**3**	**6**	1	swap 6 and 3
	2	5	4	3	**1**	**6**	swap 6 and 1

iteration 2	2	5	4	3	1	6
iteration 3	2	4	3	1	5	6
iteration 4	2	3	1	4	5	6
iteration 5	2	1	3	4	5	6
iteration 6	1	2	3	4	5	6

sorts left to right
bubbles right to left

Here's the code for the bubble sort. you will notice it is simply a loop inside a loop. We only swap the elements if the left element is greater than the right element. In a swap routine it is important to keep a temporary variable or else you would end up with the same element in both swapped locations!

Outer loop - for iterations

inner loop

for swapping

```
/* bubble sort */
#include <iostream.h>
 void bsort(int a[], int n); /* bubble sort */
 void print(int a[],int n); /* print array */
/* main function */
void main()
    {
    int a[] = {2,6,5,4,3,1};  /* initialize an array of 6 elements */
    print(a,6);  /* print array before sort */
    bsort(a,6);  /* sort array using bubble sort */
    print(a,6);  /* print array after sort */
    }
/* bubble sort function */
void bsort(int a[], int n)
    {
    int i,j;
    int t;
    /* outer loop */
    for(i=n-1; i>0; --i)
        {
        /* inner loop */
        for(j=0; j<i; j++)
            {
            /* check if left element greater than right element */
            if(a[j] > a[j+1])
                {
                t = a[j]; /* save left element */
                a[j] = a[j+1]; /* assign right to left element */
                a[j+1] = t; /* assign saved left to right  element */
                }
            }
        print(a,n); /* optional print out each sorted iteration */
        }
    }
```

swapping

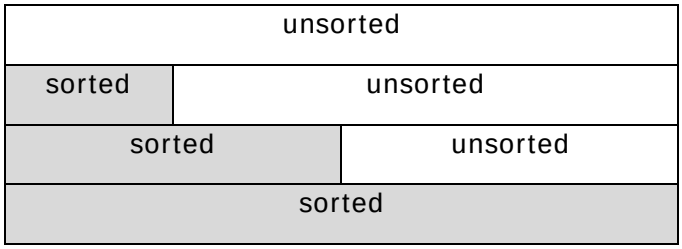
```
/* function to print out an array */
void print(int a[],int n)
        {
        int i;
        cout << "[ ";
        for(i=0;i<n;i++)
        cout << a[i] << " ";
        cout << " ]" << endl;
        }
```

output;

```
[ 2 6 5 4 3 1 ]
[ 2 5 4 3 1 6 ]
[ 2 4 3 1 5 6 ]
[ 2 3 1 4 5 6 ]
[ 2 1 3 4 5 6 ]
[ 1 2 3 4 5 6 ]
[ 1 2 3 4 5 6 ]
```

INSERTION SORT

Insertion sort keeps track of two sub arrays inside an array. A **sorted** sub array and a **unsorted** sub array. The first element of the unsorted sub array is removed an inserted in the correct position of the sorted sub array. The elements of the sorted sub arrays must be shifted right from the insertion point to make room for the key to be inserted. As the sort algorithm is running the unsorted sub array is getting smaller and the sorted sub array is getting larger until the array is sorted. Initially the sorted array is empty and the unsorted sub array is full.



The key to be inserted starts at index 1. The gray shaded elements represent the key to be inserted, the blue shaded elements represent the elements shifted to accommodate where the key will be inserted. The violet shaded elements represent the position where the key is inserted. We always are shifting between where the key is removed to where the key will be inserted. We only have 5 iterations because the key starts at array index 1 rather than zero. Do you know why ? Because you need a place for shifting and to insert the key.

initial array	**2**	**6**	**5**	**4**	**3**	**1**	key (6) start at index 1
iteration 1	**2**	**6**	**5**	**4**	**3**	**1**	key (6) inserted at index 1
iteration 2	**2**	**5**	**6**	**4**	**3**	**1**	key(5) inserted at index 1,
iteration 3	**2**	**4**	**5**	**6**	**3**	**1**	key (4) inserted at index 1
iteration 4	**2**	**3**	**4**	**5**	**6**	**1**	key (3) inserted at index 1
iteration 5	**1**	**2**	**3**	**4**	**5**	**6**	key (1) inserted at index 0

swap first smallest key found from insertion point with insertion point and shift insertion point right

```
/* insertion sort */
#include <iostream.h>
 void insort(int a[], int n); /* insertion sort */
 void print(int a[],int n); /* print array */

/* main function */
void main()
        {
        int a[] = {2,6,5,4,3,1};  /* initialize an array of 6 elements */
        print(a,6);  /* print array before sort */
        insort(a,6);  /* sort array using bubble sort */
        print(a,6);  /* print array after sort */
        }

/* insertion sort */
void insort(int a[],int n)
        {
        int i; /* iteration counter */
        int key,pos; /* key and place to insert key */

        /* loop from array index 1 to end of array */
        for(i=1; i<n; i++)
                {
                pos = i;  /* position is always i */
                key = a[i];  /* get key value */
                /* shift array element right starting from pos and moving left */
                /* until left element is greater than key or left end of array */
                while(pos > 0 && a[pos-1] > key)
                        {
                        a[pos] = a[pos-1]; /* assign left to right element */
                        pos--; /* decrement position */
                        }
                a[pos]=key; /* insert key into array */
                print(a,6); /* optional print out of array per iteration */
                }
        } /* end insertion sort */

        /* function to print out an array */
        void print(int a[],int n)
                {
                int i;
                cout << "[ ";
                for(i=0;i<n;i++)
                cout << a[i] << " ";
                cout << "]" << endl;
                }
```

```
[ 2 6 5 4 3 1 ]
[ 2 6 5 4 3 1 ]
[ 2 5 6 4 3 1 ]
[ 2 4 5 6 3 1 ]
[ 2 3 4 5 6 1 ]
[ 1 2 3 4 5 6 ]
[ 1 2 3 4 5 6 ]
```

SELECTION SORT

For each iteration we select the smallest key in the unsorted array and put at the **start** of the array and shift the element list to where the smallest key was found. The gray shaded elements represent the smallest selected element to be inserted. The blue shaded elements represents where the minimum element is swapped to. When there is no corresponding blue element, this means the element is swapped with itself.

initial array	**2**	**6**	**5**	**4**	**3**	**1**	select element 1
iteration 1	**1**	**6**	**5**	**4**	**3**	**2**	swap with start of array
iteration 2	**1**	**2**	**5**	**4**	**3**	**6**	swap element 2 with element 6
iteration 3	**1**	**2**	**3**	**4**	**5**	**6**	swap element 3 with element 5
iteration 4	**1**	**2**	**3**	**4**	**5**	**6**	select element 5, no swap
iteration 5	**1**	**2**	**3**	**4**	**5**	**6**	select element 6, no swap

select smallest element and put at start of array and shift array right

```
/* selection sort */
#include <iostream.h>
 void selsort(int a[], int n); /* selection sort */
 void print(int a[],int n); /* print array */
/* main function */
void main()
        {
        int a[] = {2,6,5,4,3,1};  /* initialize an array of 6 elements */
        print(a,6);  /* print array before sort */
        selsort(a,6);  /* sort array using bubble sort */
        print(a,6);  /* print array after sort */
        }

/* selection sort */
void selsort(int a[],int n)
        {
        int i, j;
        int min;
        int t;
        /* loop through all elements in array */
        for(i=0; i<n; i++)
                {
                /* find the minimum element after the current one */
                min = i;
                for(j=i+1; j<n; j++)
                        {
                        if(a[j] < a[min])min = j; /* keep tack of minimum index */
                        }
                /*  put it at the front of array by swapping */
                t = a[i];
                a[i] = a[min];
                a[min] = t;
                print(a,6); /* optional print out array after each iteration */
                }
        } /* end selection sort */
```

```
/* function to print out an array */
void print(int a[],int n)
        {
        int i;
        cout << "[ ";
        for(i=0;i<n;i++)
        cout << a[i] << " ";
        cout << "]" << endl;
        }
```

```
[ 2 6 5 4 3 1 ]
[ 1 6 5 4 3 2 ]
[ 1 2 5 4 3 6 ]
[ 1 2 3 4 5 6 ]
[ 1 2 3 4 5 6 ]
[ 1 2 3 4 5 6 ]
[ 1 2 3 4 5 6 ]
[ 1 2 3 4 5 6 ]
```

LESSON 4 EXERCISE 1

Type in the bubble sort, insertion sort , selection sort programs and get them going. Trace through them with the debugger and watch them go. Change each so that they sort in descending order rather than ascending order. Now all the numbers will sort from a high number to a low number.

LESSON 4 EXERCISE 2

Make a recursive bubble sort, insertion sort and selection sort. Call then bsortr.c, insertr,c and selectr.c

LESSON 4 EXERCISE 3
Make a bubble sort, insertion sort and selection sort that sort at both ends at the same time.

SHELL SORT

Shell sort works by dividing the array into many sub sequence. Each subsequence is sorted by using an insertion sort algorithm. Each subsequence consists of keys spaced a distance (delta) apart of each other. The process is repeated but now a smaller distance (delta) is used. This delta is usually 1/2 the previous delta. The process is repeated, decreasing delta until delta is 1. Then the whole array is sorted using the insertion sort. Shell sort worst case timing is O(n**1.5) which is less than O(n**2). The speed is gained by applying insertion sort to small segments. When delta is 1 the final array is almost sorted and most of the work has been done. Shell sort is like making order out of chaos. Rather than tackling chaos all at once just attack it in small chunks. Once you got all the small chunks behaving, it doesn't take much effort to get everything in order. Here's the complete code for shell sort: Our starting delta is 1/3 of the array size.

```
#include <iostream.h>
void shell(int a[], int n,int i,int k);
void print(int a[],int s,int n);
/* main driver */
void main()
        {
        int i;
        int a[] = {2,6,5,4,3,1};
        int k=6;
        print(a,0,6); /* print array before sorting */
```

```
        do
                {
                k = 1 + k/3;    /* calculate delta */
                for(i=0;i<k;i++)shell(a,6,i,k); /* call shell insertion sort algorithm */
                } while(k > 1);
        print(a,0,6);  /* print array after sorting */
        }


/* shell insertion sort algorithm  */
void shell(int a[],int n,int i,int delta)
        {
        int left,right;
        int key;
        right = i + delta;
        /* loop while  right border less than length of array */
        while(right < n)
                {
                key = a[right];  /* get key from right border */
                left = right;  /* start left at right border */
                /* loop while left is greater than border and greater than key */
                while( left != i && a[left-delta] > key)
                        {
                        a[left] = a [left-delta];  /* shift right */
                        left = left - delta; /* go left */
                        }
                        a[left] = key;  /* insert key */
                        print(a,left,right);  /* optional print sub segment */
                        right = right + delta;  /* go right */
                        }
                }
/* print out array sub segment */
void print(int a[],int s,int n)
        {
        int i;
        cout << "[ ";
        for(i=s;i<n;i++)cout << a[i] << " ";
            cout << " ]" << endl;
        }
```

```
[ 2 6 5 4 3 1 ]
[ ]
[ 3 5 4 ]
[ 1 4 6 ]
[ 1 3 ]
[ ]
[ ]
[ ]
[ ]
[ 2 ]
[ ]
[ ]
[ 5 ]
[ 1 2 3 4 5 6 ]
```

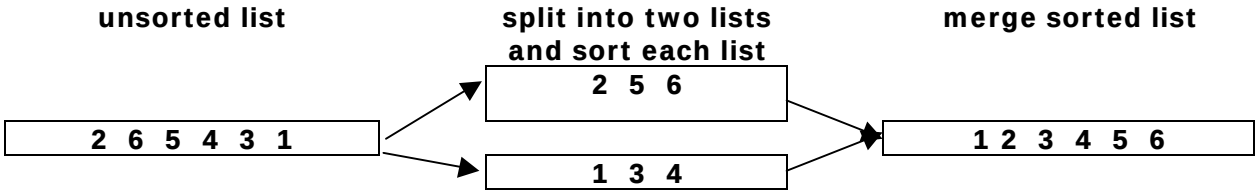
The shell sort is more complicated then the other sorting algorithms so we will trace program execution by using a chart. The keys are in blue. The inserted key is in violet. The shifted inserted elements are in yellow, and the subsection between left and right borders are in gray.

i	delta	left	right	key a[right]	left-delta	a[left-delta]	outer loop	inner loop
0	3	--	3	--	--	--	--	--
		3	3	4	0	2	T	F
		--	6	--	--	--	F	--
1	3	--	4	--	--	--	--	--
		4	4	3	1	6	T	--
		4	4	3	1	6	T	T
		1	4	3	--	--	T	F
		--	7	--	--	--	F	--
2	3	--	5	--	--	--	T	--
		5	5	1	2	5	T	T
		5	5	1	--	--	T	T
		1	5	1	--	--	T	F
			8	--	--	--	F	--
0	2	--	2	--	--	--	T	--
		2	2	1	0	2	T	T
		2	2	1	0	2	T	T
		0	--	1	--	--	T	F
0			4				T	--
	2	4	4	6	2	2	T	F
			6				F	--
1	2	--	3	--	--	--	--	--
		3	3	4	1	3	T	F
		3		4				
		5	5	5	3	4		
			7				F	
0	1		1					
		1	1	3	0	1	F	--
		1	2	2	1	3	T	F
		1	2	2	0	1	T	F
		4	4	6	3	4	T	F
		4	5	6	3	4	T	F
		5	5	5	4	6	T	T
		4	6	5	3	4	T	F

0	1	2	3	4	5
2	6	5	4	3	1
2	6	5	4	3	1
2	6	5	4	3	1
2	6	5	4	3	1
2	6	5	4	3	1
2	6	5	4	6	1
2	3	5	4	6	1
2	3	5	4	6	2
2	3	5	4	6	1
2	3	5	4	6	1
2	3	5	4	6	5
2	3	1	4	6	5
1	3	5	4	5	2
2	3	1	4	6	5
2	3	1	4	6	5
2	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	2	3	4	6	5
1	3	2	4	6	5
1	2	2	4	6	6
1	2	3	4	5	6

MERGE SORT

The idea behind merge sort is to divide a list in half, sort each list then merge the lists together as a final sorted list. To merge the two lists we make a third list. We insert into the third list the smallest element of each of the sorted lists.



The merge sort is done recursively. We first sort call the **msort()** function to get each half of the array. Then we call **merge()** function to sort and combine the two halves. Since this is done using recursion every thing is done in stages. **Merge()** is called many times to sort all the partial stages. The merge sort algorithm sorts the two separate arrays itself and merges them.

```
/* msort.cpp */
#include <iostream.h>

#define N 6
void msort(int a[], int b[],int left, int right);
void merge(int a[], int b[], int lpos, int rpos, int rend);
void print(int a[],int n);
/* driver */
void main()
        {
        int a[] = {2,6,5,4,3,1};
        int *b = new int[N+1]; /* allocate temp array */
        if(b != NULL)
                {
                print(a,N);
                msort(a,b,0,N-1);  /* call merge sort */
                print(a,N);
                delete(b);
                }
        }
/* sort array */
void msort(int a[], int b[],int left, int right)
        {
        int middle;

        if(left<right)
                {
                middle = (left+right)/2; /* calculate center */
                msort(a,b, left, middle);    /* sort left */
                msort(a,b, middle+1, right);   /* sort right */
                merge(a,b, left, middle+1, right);  /* merge sorted arrays */
                }
```

calls it self twice

```
/*
* Merges the two adjacent sub sections
* a       the array of which to merge subsequences
* b       temporary array for merging
* lpos    the left index of the first subsequence
* rpos    the right index of the first subsequence
* rend    the last index of the second subsequence
*/
/* merge two sorted arrays */
void merge(int a[], int b[], int lpos, int rpos, int rend)
        {
        int i;
        int lend,num,tpos;
        lend  = rpos - 1;
        tpos = lpos;
        num = rend - lpos + 1;
        /* loop breaks when one is emptied */
        while(lpos <=lend && rpos <= rend)
                {
                /* choose minimum */
                if(a[lpos] <= a[rpos])b[tpos++] = a[lpos++];
                else b[tpos++] = a[rpos++];
                }
        /* swap what's left into b */
        while(lpos<=lend)b[tpos++] = a[lpos++];
        while(rpos<=rend)b[tpos++] = a[rpos++];
        /* return result in array a */
        for(i=0;i<num;i++,rend--)a[rend]=b[rend];
        }

/* print out array */
void print(int a[],int n)
        {
        int i;
        cout << "[ ";
        for(i=0;i<n;i++)cout << a[i] << " ";
        cout << "]" << endl;
        }
```

merges two sorted arrays

Since the operation is quite complex and we only supply a top level chart.

	stack pairs left, right	function	left lpos	middle rpos	right rend	a	b
0	ra	msort	0	---	5	2 6 5 4 3 1	0 0 0 0 0 0
1	ra 0,5	msort	0	2	5	2 6 5 4 3 1	0 0 0 0 0 0
2	ra 0,5 0,2	msort	0	1	2	2 6 5 4 3 1	0 0 0 0 0 0
3	ra 0,5 0,2 0,1	msort	0	0	1	2 6 5 4 3 1	0 0 0 0 0 0
4	ra 0,5 0,2 0,1	msort	0	0	0	2 6 5 4 3 1	0 0 0 0 0 0
5	ra 0,5 0,2	msort	0	1	1	2 6 5 4 3 1	0 0 0 0 0 0
6	ra 0,5 0,2 0,1	msort	1	--	1	2 6 5 4 3 1	0 0 0 0 0 0
7	ra 0,5 0,2	**merge**	0	1	1	2 6 5 4 3 1	0 0 0 0 0 0
8	ra 0,5	msort	0	0	2	2 6 5 4 3 1	2 6 0 0 0 0
9	ra 0,5 0,2	msort	2	--	2	2 6 5 4 3 1	2 6 0 0 0 0

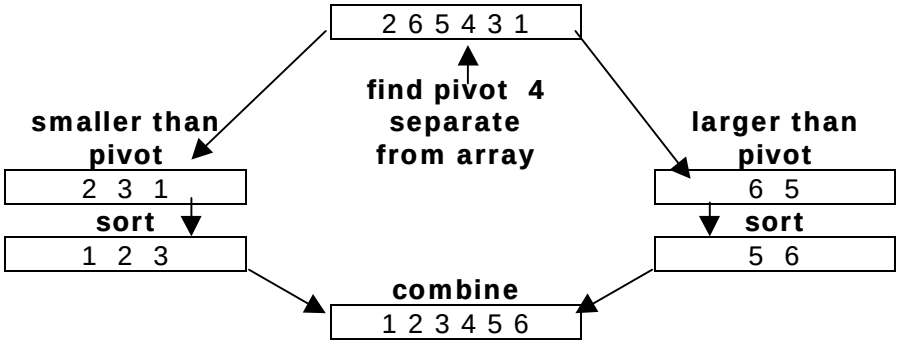
10	ra 0,5	**merge**	0	2	2	2 6 5 4 3 1	2 6 0 0 0 0
11	ra	msort	0	3	5	2 5 6 4 3 1	2 5 6 0 0 0
12	ra 0,5	msort	3	4	5	2 5 6 4 3 1	2 5 6 0 0 0
13	ra 0,5 3,5	msort	3	4	4	2 5 6 4 3 1	2 5 6 0 0 0
14	ra 0,5 3,5 3,4	msort	3	--	3	2 5 6 4 3 1	2 5 6 0 0 0
15	ra 0,5 3,5	msort	3	4	4	2 5 6 4 3 1	2 5 6 0 0 0
16	ra 0,5 3,5 3,4	msort	4	--	4	2 5 6 4 3 1	2 5 6 0 0 0
17	ra 0,5 3,5	**merge**	3	4	4	2 5 6 4 3 1	2 5 6 0 0 0
18	ra 0,5 3,5	msort	3	4	5	2 5 6 3 4 1	2 5 6 3 4 0
19	ra 0,5 3,5	msort	5	--	5	2 5 6 3 4 1	2 5 6 3 4 0
20	ra 0,5	**merge**	3	4	5	2 5 6 3 4 1	2 5 6 3 4 0
21	ra	**merge**	0	2	5	2 5 6 1 3 4	2 5 6 1 3 4
22	ra	msort	0	2	5	1 2 3 4 5 6	1 2 3 4 5 6

LESSON 4 EXERCISE 4

Type in the shell sort, merge sort programs and get them going. Trace through them with the debugger and verify our traces. Change each so that they sort in descending order, high to low.

QUICKSORT

Quicksort is the fastest known sort algorithm. The average running time is O(n log n). Quick sort is implemented with recursion and is easy to understand. The algorithm is as follows: Pick a pivot point. Put the smaller elements less than the pivot point in a left subset, put the larger elements then the pivot in a right subset,. sort each subset. Again the quick sort algorithm is used to sort itself by recursion. The trick is to pick the pivot point. Choosing the correct pivot point will make this routine run fast. Picking the wrong pivot will give poor performance.




```
/* qsort.c */
#include <iostream.h>
void qsort(int a[], int left,int right);
int partition(int a[],int left,int right,int pivot);
void print(int a[],int n);
```

```
/* driver */
void main()
        {
        int a[] = {2,6,5,4,3,1};
        print(a,6);
        qsort(a,0,6-1);
        print(a,6);
        }

/* qsort */
void qsort (int a[],int left,int right)
{
int temp;
/* calculate pivot index */
int pivot = (left+right)/2;
/* swap pivot with right */
temp = a[pivot];
a[pivot]=a[right];
a[right]=temp;
/* partition array. return middle of partition */
pivot = partition(a, left-1,right,a[right]);
/* put pivot at index */
temp = a[pivot];
a[pivot]=a[right];
a[right]=temp;
/* the pivot is at the left index, recurse on both sides of it */
if((pivot - left) > 1) qsort (a,left, pivot-1);
if((right-pivot) > 1) qsort (a,pivot+1, right);
}
/* partition array, move from outer ends of array */
/*  to middle of array swapping values */
int partition(int a[],int left,int right,int pivot)
{
int temp;
do
{
/* scan right until element larger or indices cross */
while (a[++left] < pivot);
/* scan left until an element smaller or indices cross */
while (right > 0 && (a[--right] > pivot));
/* swap right with left element */
temp = a[left];
a[left] = a[right];
a[right] = temp;
} while(left < right);
/* swap right with left element */
temp = a[left];
a[left] = a[right];
a[right] = temp;
return left;
}
```

sort two arrays recursively

choose partition

```
/* print out array */
void print(int a[],int n)
        {
        int i;
        cout << "[ ";
        for(i=0;i<n;i++) cout << a[i] << " ";
        cout << "]" << endl;
        }
```

.

LESSON 4 EXERCISE 5

Type in the quick sort program. Trace through them with the debugger and make a chart like the one in merge sort.

LESSON 4 EXERCISE 6

Generate 10000 ransdom numbers. Use quick sort to sort the numbers. Use quick sort to sort the numbers. Which one is fastest?

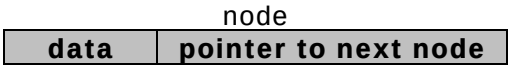
"

C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 5

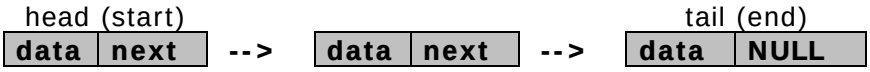
File:	CdsGuideL5.doc
Date Started:	July 24,1998
Last Update:	Dec 22,2001
Status:	proof

LESSON 5 SINGLE LINK LISTS

The link list is a more efficient memory method but a little more difficult to implement. A link list is made up of a connected list of **nodes**. A node is a structure that has a data field and a pointer to the next node.



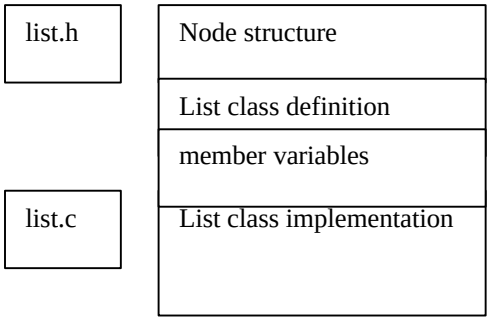
A link list is simply memory blocks called nodes pointing to each other in a chain. Think of a fire brigade. The people are the nodes, the water buckets are the data and the arms are the links joining the people nodes together passing the data (the water). To make our life easier, our link list will have a head pointer and a tail pointer, indicating the start of the link list and the end of the link list.



A link list can be built up from **tail to tail** or from **head to head**. The head to head method is much easier because it just evolves adding nodes to the start of the list. The disadvantage of this method is, that the link list will be in a reverse order. The tail to tail methods is a little more complicated, because more work is needed to add nodes. The tail to tail method is preferred because the list will be in order of insertion. The link list is a nightmare of NULL pointers. A null pointer arises when you access a pointer to a memory location and the pointer address is NULL or full of garbage. In either case the computer program will crash and you will have to reboot your computer.

Implementing a Link List

To implement a Link List you need functions to insert, remove and search nodes. Each node will contain the data and a pointer to the next node. You may also want functions to insert at the start or end of the Link List or to insert in ascending or descending order. We use the modular approach to implement our Link List ADT where a structure is use to hold the start and end nodes and how many nodes we have in the list. We also need a structure to represent a node.



The Node data structure holds the data and link to next node.

The List class definition file contains the member variables function declarations. The List member variables hold the list start and end pointers and number of nodes in list.

The List class implementation file contains the function code definitions.

Here's the code for the List ADTclass. We have a small file defs.h that includes definitions for true, false, error and bool. If your compiler does not supply these for you then you need to include the defs.h file in list.h.

```
/* defs.h */
#ifndef __DEFS
#define __DEFS
#define false 0
#define true 1
#define error -1
typedef int bool;
#endif
```

The link nodes will have there own separate data structure separate from the list data structure. Here's the link list code:

```
/* single link list header file */
/* list.hpp */
//#include "defs.h"
/* node data */
typedef int NodeData;
/* node data structure */
struct Node
        {
        Node* next;
        NodeData data;
        Node(NodeData data)
            {
            this->data=data;
            this->next=NULL;
            }
        };
/* the link list will have head and tail pointers */
class List
        {
        private:
        Node* head;
        Node* tail;
        public:
        /* function prototypes for single link list */
        /* initialize link list structure */
        List();
        /* insert item into list */
        bool insertList(NodeData data);
        /* remove data item from list */
        bool removeList(NodeData data);
        /* find data item in list */
        /* return pointer to node if found, otherwise return NULL */
        Node* findList(NodeData data);
        /* find data item in list */
        /* return index of node if found, otherwise return -1 */
        int findListAt(NodeData data);
```

```
        /* print list */
        friend ostream& operator<<(ostream& out, List& list);
        /* de-allocate memory for list */
        ~List();
        };
```

Implementation code file for Link List:

```
/* list.cpp */

/* single link list code implementation */
#include <iostream..h>
#include "list.hpp"
/* initialize list */
List::List()
        {
        this->head = NULL; /* set head to default value */
        this->tail = NULL; /* set tail to default value */
        }

/* destroy list */
List::~List()
        {
        Node* next;
        Node* curr = this->head; /* point to start of list */
        /* loop till end of list */
        while(curr != NULL)
                {
                next = curr->next; /* get pointer to next node */
                delete(curr); /* free current node */
                curr = next; /* current points to next node */
                }
        this->head = NULL; /* set head to default value */
        this->tail = NULL; /* set tail to default value */
        }

/* insert item into list */
/* 4 conditions to watch out for
* (1) insert int empty list
* (2) insert at start at list
* (3) insert in middle of list
* (4) insert at end of list
*/
```

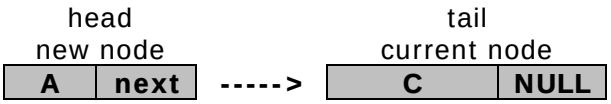
empty list

head	NULL
tail	NULL

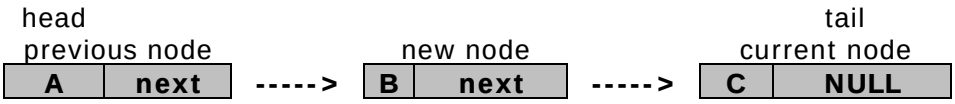
insert "C" into empty list

head	tail
data	next
C	**NULL**

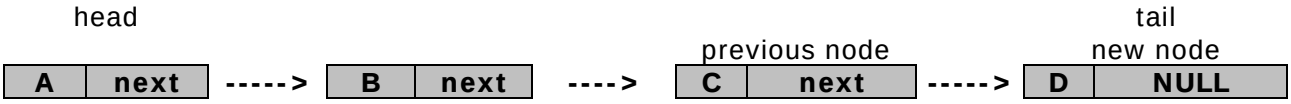
insert "A" at start of list (no previous node)



insert "B" into middle of list

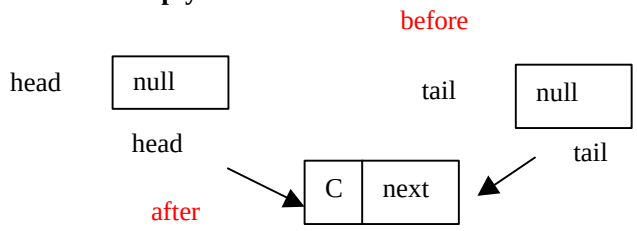


insert "D" at end of list (no current node)




```
bool List::insertList(NodeData data)
        {
        Node* prev = NULL; /* pointer to a previous node */
        Node* curr = NULL; /* pointer to a current node */
        /* allocate memory for new node */
        Node* node = new Node(data);
        /* check for empty list */
        if(this->head == NULL)
                {
                this->head = node; /* set head to point to new node */
                this->tail = node; /* set tail to point to new node */
                return true;
                }
```

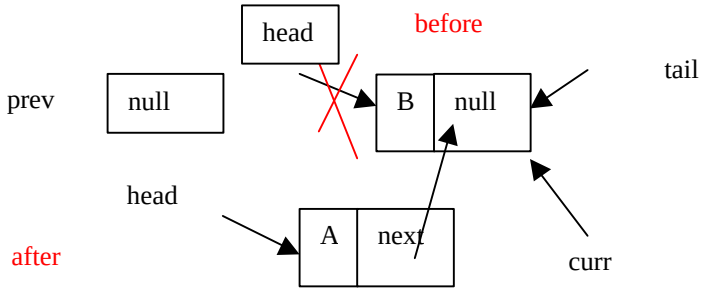
insert C into empty list




```
        /* find out where to insert new node, insert in ascending order */
        curr = this->head; /* point to start of list */
        /* loop till end of list */
        while(curr != NULL)
                {
                if(data < curr->data)break; /* if data to insert value is less than current break */
                prev=curr; /* save previous node */
                curr=curr->next; /* current point to next node */
                }
```

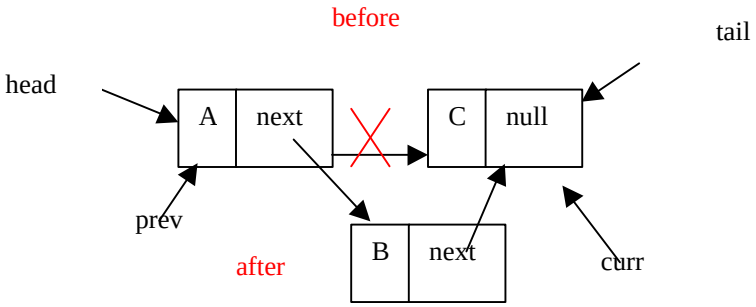
```
/* check for start of list */
/* node points to start of list */
if(prev == NULL)
        {
        node->next = this->head; /* new node points to start of list */
        this->head = node; /* make head point to new node */
        }
```

insert A at beginning of list

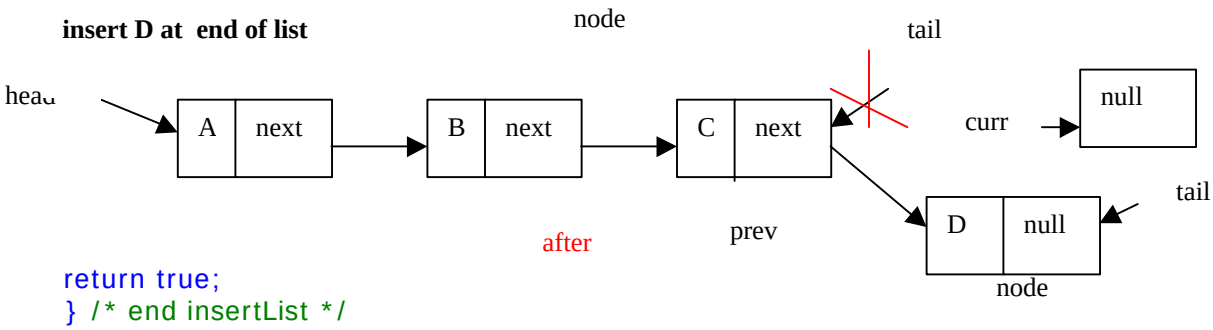



```
/* check for middle of list */
else
        {
        node->next=prev->next; /* new node points to current */
        prev->next=node; /* previous node points to new node */
        if(curr == NULL)this->tail = node; /* IF end of list, assign tail to end */
        }
```

insert B into middle of list



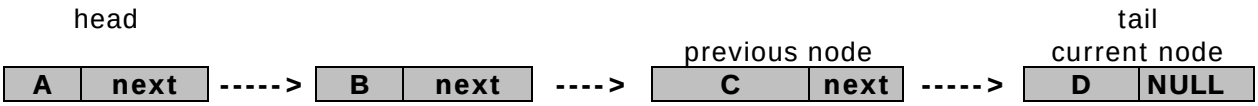
insert D at end of list



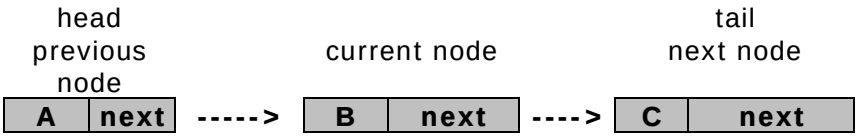

```
return true;
} /* end insertList */
```

```
/* delete node from list */
/* four conditions to watch out for */
/* delete from end of list */
/* delete from middle of list */
/* delete from start of list */
/* delete last item in list */
```

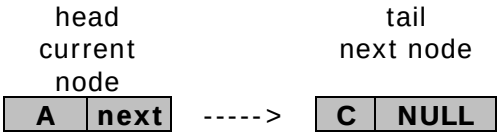
delete "D" at end of list (no next node)



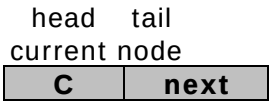
delete "B" at middle of list



delete "A" start of list (no previous node)



delete "C" last item in list (head point equals tail pointer)

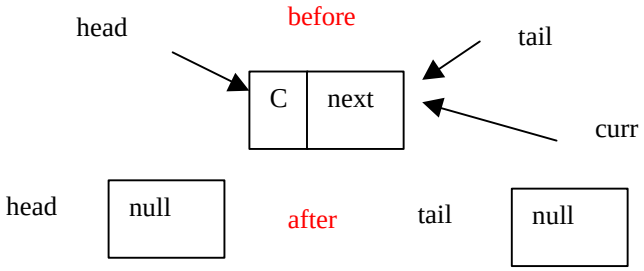


list is empty


```
/* remove node from list */
bool List::removeList(NodeData data)
        {
        NodePtr prev= NULL; /* pointer to a previous node */
        NodePtr curr = this->head; /* point to start of list */
        /* find data to delete */
        /* loop till end of list */
        while(curr != NULL)
                {
                if(curr->data == data)break; /* exit when data found */
                prev = curr; /* save previous node */
                curr=curr->next; /* point to next node */
                }
```

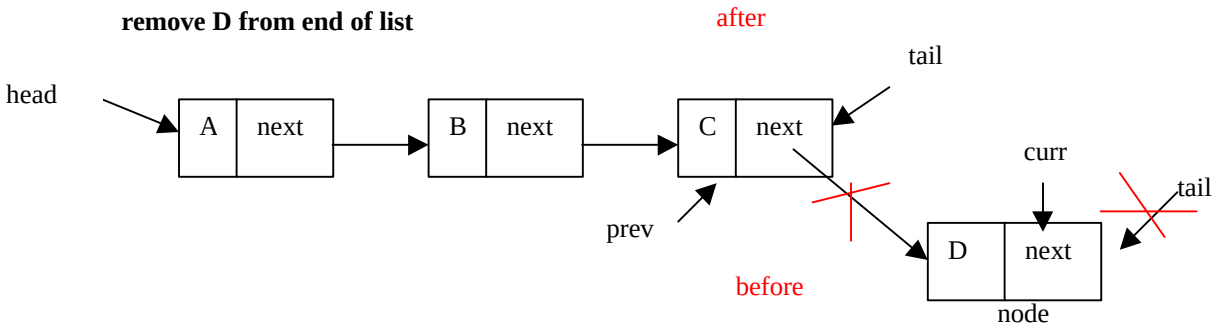
```
/* item not found */
if(curr == NULL)return false;
/* check for only one item in list */
if(this->head==this->tail)
        {
        /* remove last one node in link list */
        this->head=NULL; /* set head to empty value */
        this->tail=NULL; /* set tail to empty value */
        }
```

remove C from list



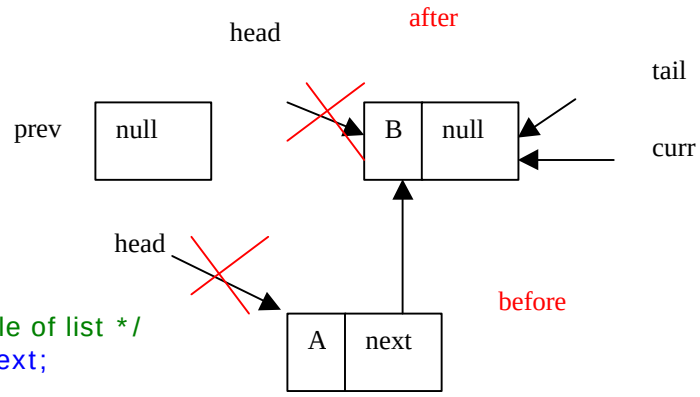

```
/* check for end of list */
else if(curr->next == NULL)
        {
        /* remove node at end of list */
        prev->next=NULL; /* set prev node to point to no node */
        this->tail = prev; /* set tail to point to previous node */
        }
```

remove D from end of list



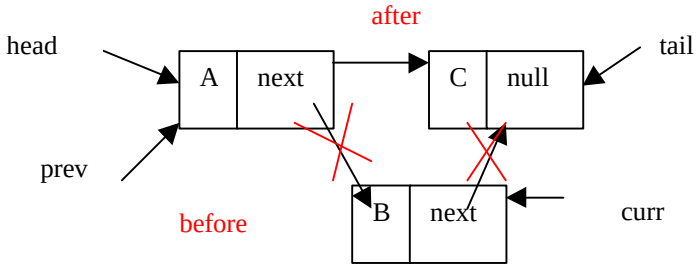
```
        /* check for start of list */
        else if(prev == NULL)
                /* remove node from start of list */
                this->head=curr->next; /* heads point to next node */
```

remove A at beginning of list




```
        /* remove node from middle of list */
        else prev->next = curr->next;
```

remove B from middle of list




```
        delete (curr); /* deallocate current node that was removed */
        return true;
        }

/* find data element in list */
/* if found return node pointer in list */
/* if not found return NULL */
Node* List::findList(NodeData data)
        {
        Node* curr = this->head; /* point to start of list */

        /* loop till end of list */
        while(curr != NULL)
                {
                /* return curr if data found */
                if(data == curr->data)return curr;
                curr=curr->next; /* point to next node */
                }
        return NULL;
```

```
/* find data element in list */
/* if found return position in list */
/* if not found return error */
int List::findListAt(NodeData data)
        {
        int i = 0; /* set index to zero */
        Node* curr = this->head; /* point to start of list */
        /* loop till end of list */
        while(curr != NULL)
                {
                /* return index if data found */
                if(data == curr->data)return i;
                i++; /* increment index */
                }
        return -1;
        }
/* print a list */
/* print data elements in list using recursion */
/* function calls itself until last node reached */
ostream& operator<<(ostream& out, List& list)
        {
        Node* curr = list.head; /* point to start of list */
        out << "list: ";
        /* loop till end of list */
        while(curr != NULL)
                {
                out << curr->data << " "; /* print out item value */
                curr = curr->next; /* point to next data item */
                }
        out << endl;
        return out;
        }


/* test driver */
int main()
        {
        List list(); /* create a list structure */
        list.insertList(10); /* insert 10 into link list */
        cout << list; /* print out link list */
        list.insertList(8); /* insert 8 into link list */
        cout << list; /* print out link list */
        list.insertList(5); /* insert 5 into link list */
        cout << list; /* print out link list */
        list.insertList(20); /* insert 20 into link list */
        cout << list; /* print out link list */
        if(list.findList(8))cout << "item 8 found"<< endl;
        if(list.findList(15))cout << "item 15 found"<<endl;
        list.removeList(15); /* remove 15 from list */
        cout << list; /* print out link list */
        list.removeList(8); /* remove 8 from list */
        cout << list; /* print out link list */
```

```
        list.removeList(20); /* remove 20 from list */
        cout << list; /* print out link list */
        list.removeList(8); /* remove 8 from list */
        cout << list; /* print out link list */
        list.removeList(5); /* remove 5 from list */
        cout << list; /* print out link list */
        list.removeList(10); /* remove 10 from list */
        cout << list; /* print out link list */
        return 0;
        } /* end main */
```

Program output:

```
list: 10
list: 8 10
list: 5 8 10
list: 5 8 10 20
item 8 found
list: 5 8 10 20
list: 5 10 20
list: 5 10
list: 5 10
list: 10
list:
```

LESSON 5 EXERCISE 1

Write functions to insert an item at the start of a list called **insertHead()** and to inset an item at the end of a list called **insertTail()** to the single link list module.

LESSON 5 EXERCISE 2

Write a function that reverses the links in a single link list. Now the head pointer will point to the last element and the tail pointer will point to the first element.

LESSON 5 EXERCISE 4

Write function **iterate()** that initializes a node pointer to the start of the list. Write functions **next()** that returns the current node and sets the node pointer to the next node. You may want to use operator function ++ for functions next().

LESSON 5 EXERCISE 5

Re-write all of the single link list functions as recursive functions. You may need driver functions. Call your modules Listr.hpp and Listr.cpp.

LESSON 5 EXERCISE 6

Write the Link List class as a template class call it TLink, now you can have any data type you want.

C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 6

File:	CppdsGuideL6.doc
Date Started:	July 24,1998
Last Update:	Dec 22, 2001
Status:	proof

LESSON 6 DOUBLE LINK LISTS

DOUBLE LINK LIST

The double link list is similar to the single link list, but it has link pointer to nodes in both directions. There is now a previous pointer and a next pointer.

dnode

pointer to previous node	data	pointer to next node

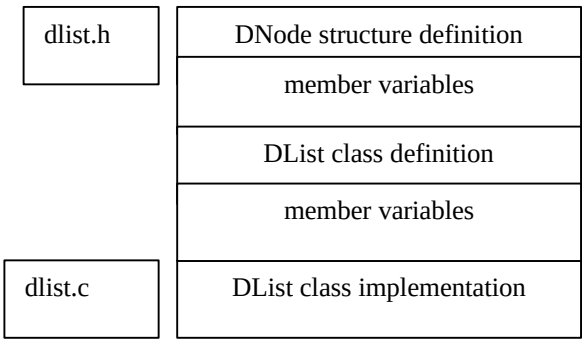
The double link list aids in insertion and deletion operations. The double link has the advantage that if a link gets broken then data can still be retrieved by going in the opposite direction.

A double link list is said to be circular if the ends point to itself can also Only the head and tail pointers indicate where it starts and ends. circular link lists come in very handy in communication applications, transmitting and receiving of data.

The operation and code is identical to the single link list with a little change to accommodate the double links.

Implementing a Double Link List

To implement a Double Link List you need functions to insert, remove and search nodes. Each node will contain the data and pointers to the next and previous nodes. You may also want functions to insert at the start or end of the Double Link List or to insert in ascending or descending order. We use the modular approach to implement our Double Link List ADT where a DList structure is use to hold the start and end nodes and how many nodes we have in the list. We also need a DNode structure to represent a node.



The DNode data structure holds the data and links to next and previous nodes.

The DList class holds the start and end of list pointers and number of nodes in list and class function declarations

The class implementation contains the function definitions.

Here's the code for the DList ADT class. We have a small file defs.h that includes definitions for true, false, error and bool. If your compiler does not supply these for you then you need to include the defs.h file in list.h.

```
/* defs.h */
#definre false 0
#define true 1
#define error -1
typedef  int  bool;
```

The DLink nodes will have there own separate data structure separate from the DList class. Since you are now pro's to link list here's the double link list code:

```
/* dlist.hpp */
/* double link list */
//#include "defs.h"
#include <iostream.h>
/* double link list node data */
typedef int DNodeData;
/* double node stucture */
struct DNode
{
DNode* prev;
DNodeData data;
DNode* next;

/* initialize DNode */
DNode(DNodeData data)
{
this->prev=NULL;
this->data=data;
this->next=NULL;
}
};
/* double link list class */
class DList
{
private:
DNode* head;
DNode* tail;
public:
```

```
        /* function declarations */
        /* initialize double link list pointers */
        DList();
        /* free double link list nodes */
        ~DList();
        /* insert item into double link list */
        bool insertDList(DNodeData data);
        /* remove item from double link list */
        bool removeDList(DNodeData data);
        /* find item in double link list */
        /* return address of double node if found otherwise NULL */
        DNode* findDList(DNodeData data);
        /* find item in double link list */
        /* return index if found otherwise -1 */
        int findDListAt(DNodeData data);
        /* print out double link list items from start of list */
        friend ostream& operator <<(ostream& out, DList& dlist);
        /* print out double link list items from end of list */
        void printEnd();
        };
```

Implementation code file for Double Link List:

```
        /* dlist.cpp */
        /*double link list */
        #include <iostream.h>
        #include "dlist.hpp"
        /* initialize dlist pointers */
        DList::DList()
                {
                this->head = NULL; /* set head to empty value */
                this->tail = NULL; /* set tail to empty value */
                }
        /* free nodes in dlist */
        DList::~DList()
                {
                DNode* next; /* pointer to a next dnode */
                DNode* curr = this->head; /* point to start of list */
                /* loop till end of list */
                while(curr != NULL)
                        {
                        next = curr->next; /* point to next node */
                        delete(curr); /* free current dnode */
                        curr = next; /* set curr to next dnode */
                        }
                this->head = NULL; /* set head to empty value */
                this->tail = NULL; /* set tail to empty value */
                }
```

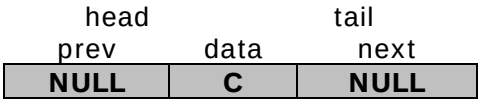
inserting nodes into a list

When insert an item into double link list there are 4 conditions to watch out for :

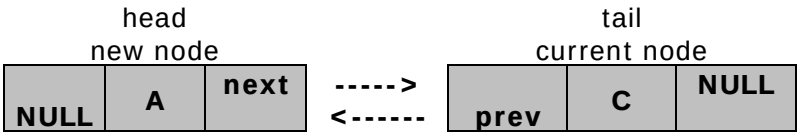
(1) inset into an empty list
(2) insert at start of list
(3) insert in middle of list
(4) insert at end of list

start with empty list

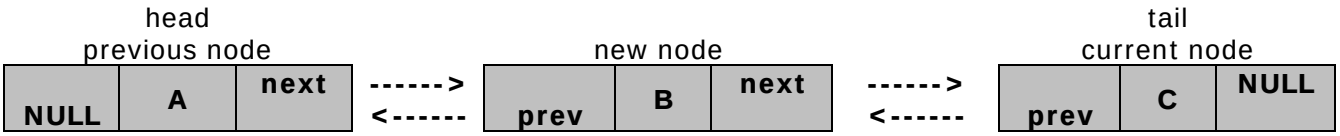
insert "C" into empty list



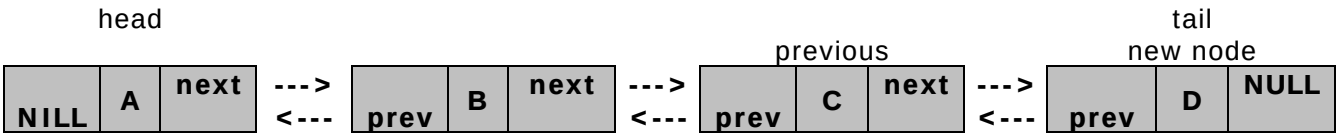
insert at "A" start of list (no previous node)



insert "B" in middle of list



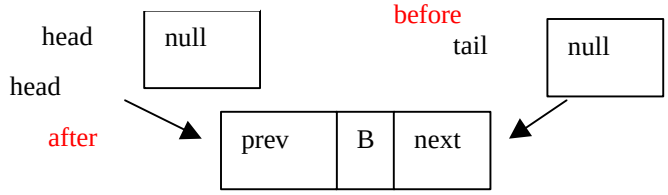
insert "D" at end of list (no current node)




```
/* insert item into dlist
* ----------------------
* 4 conditions to watch out for
* (1) inset into empty list
* (2) insert at start at list
* (3) insert in middle of list
* (4) insert at end of list
*/
```

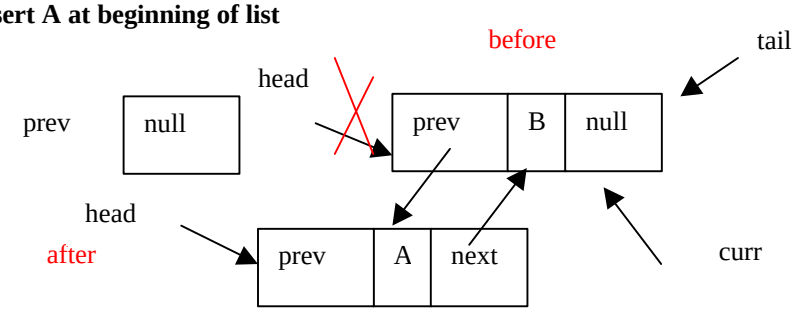
```
bool DList::insertDList(DNodeData data)
        {
        DNode* curr=NULL; /* pointer to a current dnode */
        DNode* prev=NULL; /* pointer to a previous dnode */
        DNode* next=NULL; /* pointer to a next dnode */
        /* allocate memory for new dnode */
        DNode* dnode = new DNode(data);
        if (dnode == NULL)return false;

        /*check for empty list */
        if(this->head == NULL)
                {
                this->head = dnode; /* head points to new node */
                this->tail = dnode; /* tail points to new node */
                return true;
                }
```

insert C into empty list



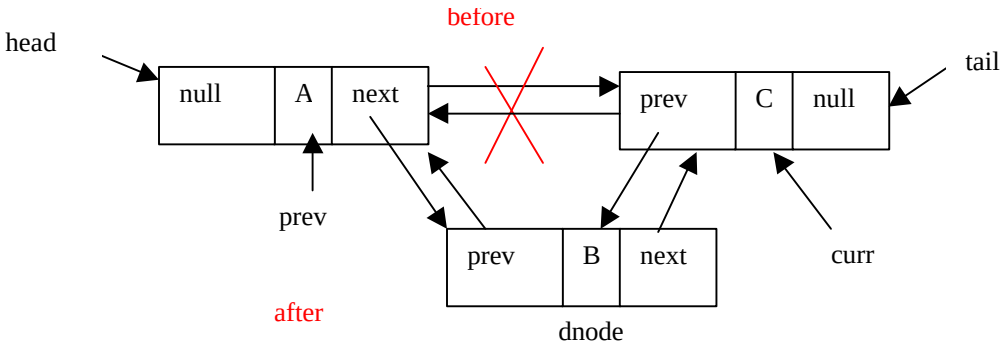

```
/* find out where to insert new dnode */
curr = this->head; /* point to start of dlist */
/* loop till end of dlist */
while(curr != NULL)
        {
        if(data < curr->data)break; /* break if data found */
        prev = curr; /* save previous dnode */
        curr=curr->next; /* point to next dnode */
        }
/* check for start of list */
if(prev == NULL)
        {
        dnode->next = curr; /* new dnode points to start of list */
        curr->prev = dnode; /* start of list points to new dnode */
        this->head = dnode; /* head points to new dnode */
        }
```

insert A at beginning of list

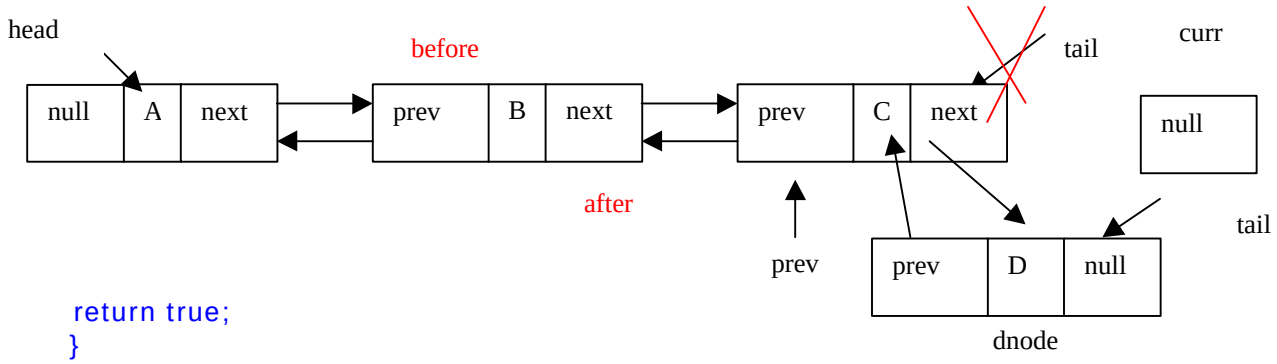


```
/* check for middle of list */
else
        {
        dnode->next=prev->next; /* new dnode next points to current */
        dnode->prev = prev; /* new dnode prev points to previous */
        prev->next = dnode; /* previous next points to new dnode */
        /*check for end of list */
        if(curr == NULL)this->tail = dnode;
        /* current previous points to new dnode */
        else curr->prev = dnode;
        }
```

insert B into middle of list



insert D at end of list



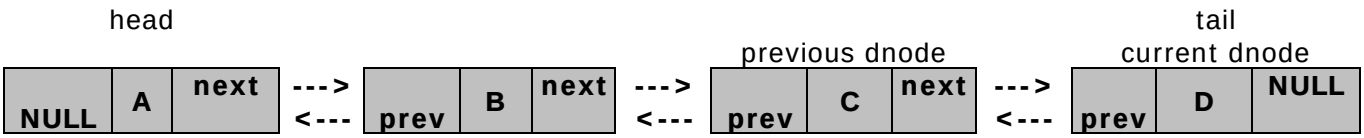

```
return true;
}
```

deleting nodes from a list

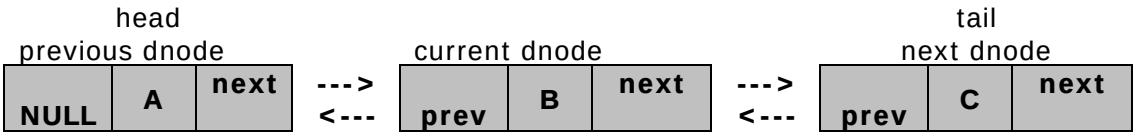
When deleting node item from a double link list there are four conditions to watch out for:

(1)delete item from end of list
(2)delete item from middle of list
(3)delete item from start of list
(4)delete item last item in list

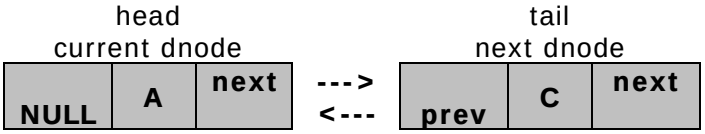
delete "D" at end of list (no next dnode)



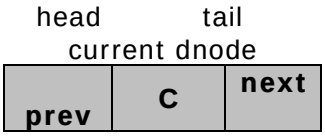
delete "B" at middle of list



delete "A" at start of link list (no previous dnode)



delete "C" last item in list (head is equal to tail)

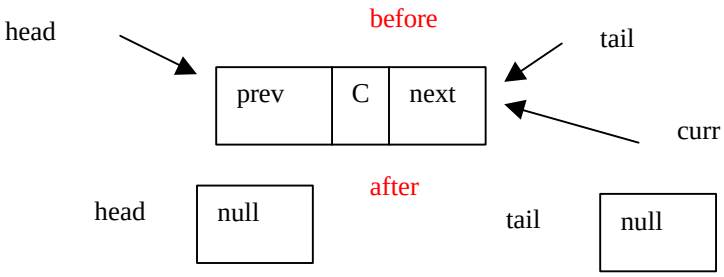


list is empty


```
/* delete node from dlist
* ----------------------
* four conditions to watch out for
* remove from end of list
* remove from middle of list
* remove from start of list
* remove last item in list
*/
bool DList::removeDList(DNodeData data)
        {
        DNode* prev = NULL; /* pointer to a previous dnode */
        DNode* next = NULL; /* pointer to a next dnode */
        DNode* curr = this->head; /* point to start of dlist */
```

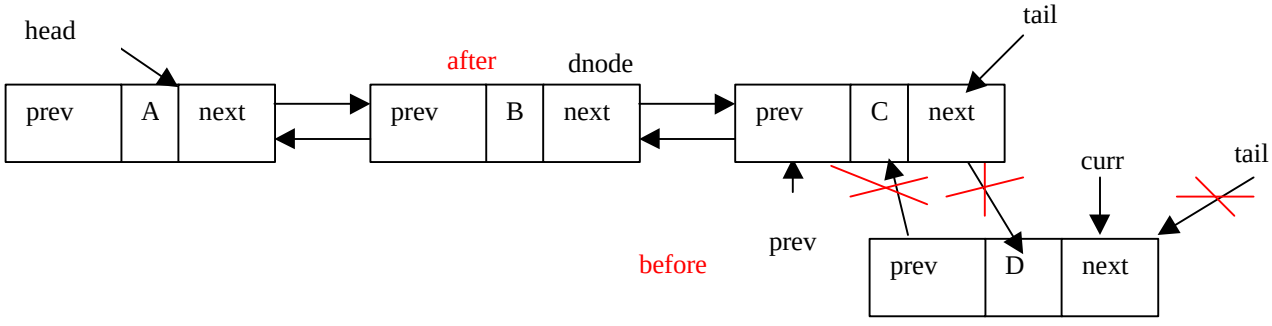
```
        /* find data to remove */
        /* loop till end of dlist */
        while(curr != NULL)
                {
                if(curr->data == data)break; /* stop when data found */
                curr=curr->next; /* point to next dnode */
                }

        /* check for not found or empty list */
        if(curr == NULL)return false;

        prev = curr->prev; /* save pointer to previous dnode */
        next = curr->next; /* save pointer to next dnode */
        /* check for last one item in list */
        if(this->head==this->tail)
                {
                this->head=NULL; /* set head to empty value */
                this->tail=NULL; /* set tail to empty value */
                }
```

remove C from list



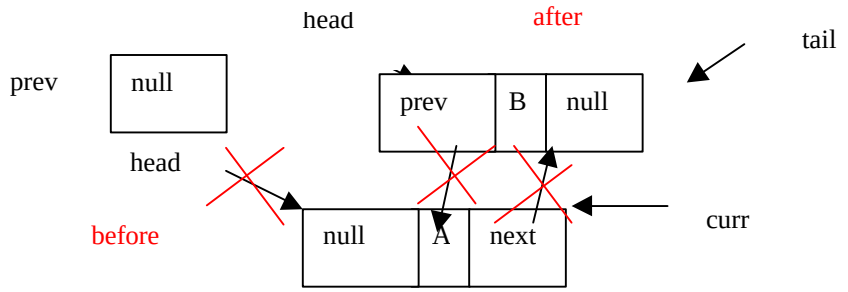

```
/* check for end of list */
else if(curr->next == NULL)
        /* remove dnode from end of list */
        {
        prev->next = NULL; /* set previous dnode next to empty */
        this->tail = prev; /* set tail to previous node */
        }
```

remove D from end of list



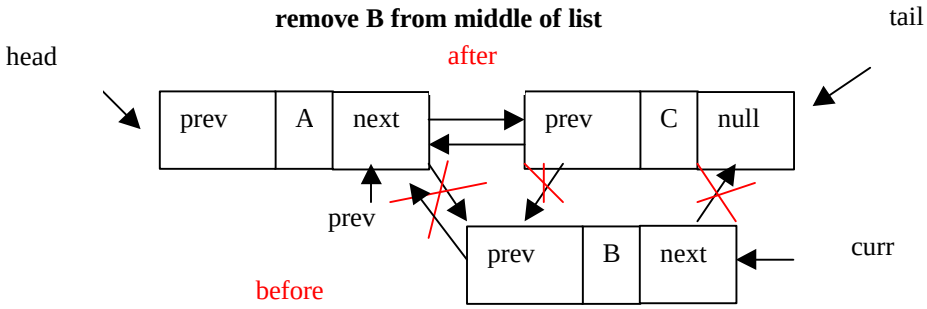
```
/* check for start of list */                                    dnode
else if(prev == NULL)
        /* remove dnode from start of list */
        {
        this->head = next; /* head points to next dnode */
        next->prev = NULL; /* current points to empty node */
        }
```

remove A at beginning of list




```
/* check for middle of list */
else
        /* remove dnode from middle of list */
        {
        prev->next = next; /* previous node points to next */
        next->prev = prev; /* next previous points to dnode */
        }
```

remove B from middle of list




```
delete(curr); /* deallocate memory for current node */
return true;
}
```

```
/* find data element in list */
/* if found return position in list */
/* if not found return error */
int DList::findDListAt(DNodeData data)
        {
        int i=0; /* index counter */
        DNode* curr = this->head; /* point to start of list */
        /* loop till item found */
        while(curr != NULL)
                {
                if(data == curr->data)return i; /* if found return index */
                i++; /* increment index */
                curr=curr->next; /* point to next dnode */
                }
        return -1; /* item not found */
        }

        /* find data element in list */
        /* if found return dnode */
        /* if not found return NULL */
        DNode* DList::findDList(DNodeData data)
        {
        DNode* curr = this->head; /* point to start of list */
        /* loop till item found */
        while(curr != NULL)
                {
                if(data == curr->data)return curr; /* if found return curr */
                curr=curr->next; /* point to next dnode */
                }
        return NULL; /* item not found */
        }
/* print out items at start of dlist */
ostream& operator <<(ostream& out, DList& dlist)
        {
        DNode* curr=dlist.head; /* point to start of dlist */
        /* check if dlist empty */
        if(curr == NULL)
                {
                out << "double list is empty" << endl;
                return out;
                }
        out << "dlist from start: ";
        /* loop till end of list */
        while(curr != NULL)
                {
                out << curr->data << " "; /* print out data item */
                curr = curr->next; /* point to next item */
                }
        out << endl;
        return out;
        }
```

```
/* print out items at end of dlist */
void DList::printEnd()
        {
        DNode* curr=this->tail; /* point to end of dlist */
        /* check if dlist empty */
        if(curr == NULL)
                {
                cout << "double list is empty" << endl;
                return;
                }
        cout << "dlist from end: ";
        /* loop till start of list */
        while(curr != NULL)
                {
                cout << curr->data << " "; /* print out data item */
                curr = curr->prev; /* point to previous dnode */
                }
        cout << endl;
        }

/* main program */
int main()
        {
        DList dlist;
        dlist.insertDList(10);
        cout << dlist; // print out dlist from start
        dlist.printEnd();
        dlist.insertDList(8);
        cout << dlist; // print out dlist from start
        dlist.printEnd();
        dlist.insertDList(5);
        cout << dlist; // print out dlist from start
        dlist.printEnd();
        dlist.insertDList(20);
        cout << dlist; // print out dlist from start
        dlist.printEnd();
        cout << "found 8 at index " << dlist.findDListAt(8) << endl;
        cout << "found 15 at index " <<  dlist.findDListAt(15) << endl;
        dlist.removeDList(10);
        cout << dlist; // print out dlist from start
        dlist.printEnd();
        dlist.removeDList(15);
        cout << dlist; // print out dlist from start
        dlist.printEnd();
        dlist.removeDList(20);
        cout << dlist; // print out dlist from start
        dlist.printEnd();
        dlist.removeDList(8);
        cout << dlist; // print out dlist from start
        dlist.printEnd();
```

```
dlist.removeDList(5);
cout << dlist; // print out dlist from start
dlist.printEnd();

return 0;
}
```

program output:

```
dlist from start: 10
dlist from end: 10
dlist from start: 8 10
dlist from end: 10 8
dlist from start: 5 8 10
dlist from end: 10 8 5
dlist from start: 5 8 10 20
dlist from end: 20 10 8 5
found 8 at index 1
found 15 at index -1
dlist from start: 5 8 20
dlist from end: 20 8 5
dlist from start: 5 8 20
dlist from end: 20 8 5
dlist from start: 5 8
dlist from end: 8 5
dlist from start: 5
dlist from end: 5
double list is empty
double list is empty
```

LESSON 6 EXERCISE 1

Add a length node counter and isEmpty() function to the double link list module. Add the insertDListHead and insertDListTail functions to the double link list module. Add the removeDListHead and removeDListTail functions to the double link list module.

LESSON 6 EXERCISE 2

Make a circular double link list module (cdlist) using the double link list module as a guide. All you need now is a head pointer. Add the insertCDList function and others. Modify any other function that is required.

LESSON 6 EXERCISE 3

Re-write all of the double link list functions as recursive functions. You may need driver functions. Call your modules DListr.h and DListr.c.

LESSON 6 EXERCISE 4

Re-write all of the double link list functions as template class. Now you can have any data type you want.

LESSON 6 EXERCISE 5

Add an iterator() function that will set an iterator DNode pointer to start of dlist. Write functions next() to get next item and prev() to get previous item. You may want to use operator functions ++ and - - for functions next() and prev().

C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 7

File:	CppdsGuideL7.doc
Date Started:	April 15,1999
Last Update:	Dec 22,2001
Status:	draft

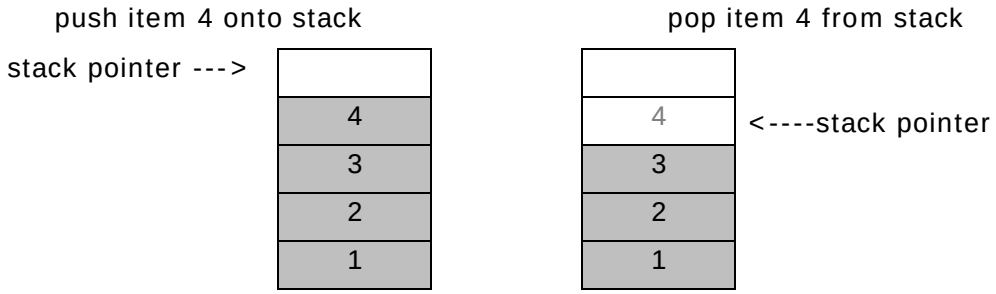
LESSON 7 ABSTRACT DATA TYPES USING LINK LISTS

Implementing ADT's with link lists is ideal. Link lists allow the flexibility for adding new data indefinitely. Link lists have an added benefit is that you can insert data in ascending or descending order quite easily. When you insert or remove elements from an ordered link list you do not have to reshuffle data as with when using an Array. The link list also lets us easily make a priority queue. A priority queue allows some items to be service readily over others For example when a millionaire comes into the bank he can be served first, ahead of all the customers who are waiting in line! The following chart lists the performance of ADT's using Link Lists.

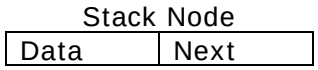
Link List ADT	Implementation Difficulty	Worst Case insertion Time	Worst Case retrieval Time	Memory Requirements
stack	easy	O(1)	O(1)	moderate
queue	easy	O(1)	O(1)	moderate
priority queue	difficult	O(n)	O(1)	moderate
dequeue	easy	O(1)	O(1)	moderate

STACK LIST ADT CLASS

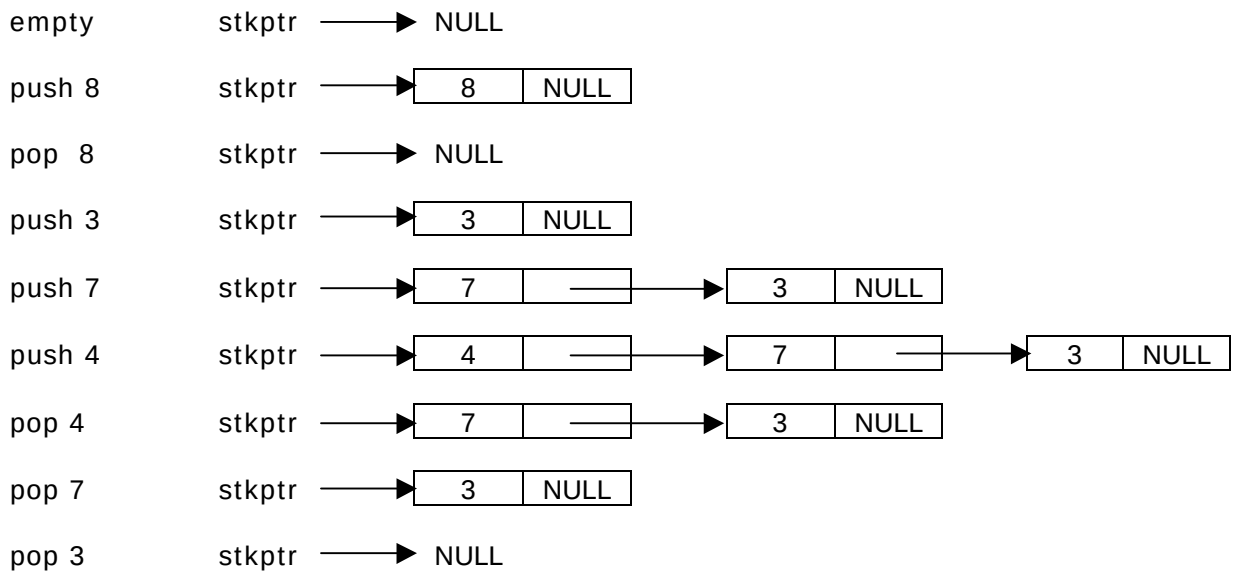
A **Stack** allows you to insert and retrieve items as **last in first out** (LIFO) into a list. This means if you insert the number 1, 2 then 3 you will get back 3, 2 and 1 in the reverse order. Stack operations are known as **push** and **pop**. You use the push operation to insert an item into the stack. You use the pop operation to remove an item from the stack. The location to insert the data item into the stack is pointed to by the stack pointer. The stack pointer is initially set to the beginning of the stack. The beginning of the stack is known as the stack **bottom**. The end of the stack is known as the stack **top.** When an item is pushed onto the stack the stack pointer is incremented after the operation. When an item is popped of the stack the stack pointer is decremented **before** the operation. When implementing a stack using link list the stack pointer can point to the start or end of the link list. You can always think of a stack as a stack of books.



We use a link list to build the stack. A link is a memory block that has a data field and a pointer to another node. We call the nodes for our Stack link list Stack Nodes.

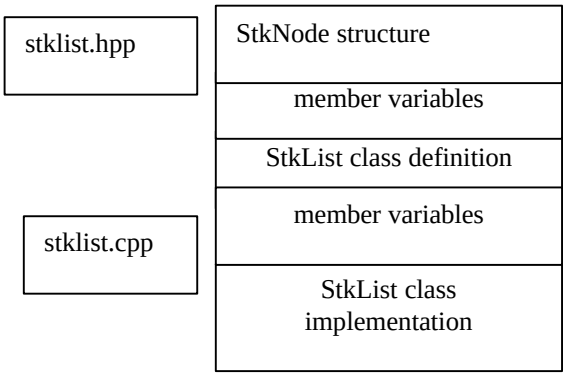


We insert into the front of the list rather than at the end. By inserting at the front of the list we simulate the last in first out stack operation. The link list will be built right into the stack module.



Implementing a Stack Link List

To implement a Stack Link List you need to insert a node for the push operation, and remove a node for the pop operation. Each node will contain the data and pointers to the next node. We use the modular approach to implement our Stack List ADT where the STKLIST structure is used to hold the start of the list and how many nodes we have in the list. We also need a STK_NODE structure to represent a node.



The StkNode data structure holds the data and links to next and previous nodes.

The StkList class member variables holds the start of list and number of nodes in the list.

The implementation code contains the StkList function definitions

Here's the code for the StkList ADT module. We have a small file defs.h that includes definitions for true, false, error and bool. If your compiler does not supply these for you then you need to include the defs.h file in stklist.h.

```
/* defs.h */
#ifndef  __DEFS
#define  __DEFS

#define true  1
#define false 0
#define error -1
typedef int bool;
#endif
```

The link nodes will have there own separate data structure separate from the list data structure. Here's the code for the Stack list module:

```
/* stklist.hpp */
#ifndef __STKLIST
#define __STKLIST
#include <iostream.h>
//#include "defs.h"
/* stack class  data items */
typedef int  StkData;
/* stack node structure */
struct StkNode
{
StkData data;
StkNode*  next;

StkNode(StkData data,StkNode* next)
        {
        this->data=data;
        this->next=next;
        }
};

/* stack list  ADT class  */
class StkList
{
private:
StkNode * stkptr; /* stack pointer */
public:
/* stack module function prototypes */
/* initialize stack */
StkList();
/* put data item onto stack */
bool push(StkData data);
/* get data item from stack */
StkData pop();
/* print stack contents */
friend ostream& operator<<(ostream& out, StkList& list);
/* deallocate memory for stack */
~StkList();
};
#endif
```

Stack List class implementation code file:

```
/* stklist.cpp */
#include <iostream.h>
#include "stklist.hpp"
//#include "defs.h"

/* stack module functions */
/* initialize stack module */
StkList::StkList()
        {
        this->stkptr = NULL; /* set stackptr to top of stack */
        }
/* push item into stack */
bool StkList::push(StkData data)
        {
        /* insert into vector at stpkptr++ location */
        StkNode* node = new StkNode(data,this->stkptr);
        this->stkptr=node;
        return node != NULL;
        }
/* get item from stack */
StkData StkList::pop()
        {
        StkData data;
        StkNode* node;
        if(this->stkptr == NULL)return -1; /* check if stack empty */
        data = (this->stkptr)->data;
        /* remove from item at --stpkptr location */
        node = (this->stkptr)->next;
        delete(this->stkptr);
        this->stkptr = node;
        return data;
        }
/* print stack items */
ostream& operator <<(ostream& out, StkList& stk)
        {
        StkNode* node = stk.stkptr;;
        out << "stack: [ ";

        /* print out stack items */
        while(node != NULL)
                {
                out << node->data << " ";
                node = node->next;
                }
        out << "]" << endl;
        return out;
        }
```

```
/* deallocate memory for stack */
StkList::~StkList()
        {
        StkNode* node=this->stkptr;
        /* loop till end of list */
        while(node != NULL)
                {
                StkNode* next = node->next; /* deallocate memory for stack item */
                delete(node);
                node = next;
                }
       this->stkptr=NULL;
       }
/* main function to test stack module */
int main()
        {
        StkData data; /* create stack data structure in memory */
        StkList stk; /* create a stack object */
        cout << stk; /* print stack contents */
        stk.push(8); /* push "8" into stack */
        cout << stk; /* print stack contents */
        data = stk.pop(); /* get item from stack */
        cout << "data from stack: " << data << endl;
        data = stk.pop(); /* get item from stack */
        cout << "data from stack: " << data << endl;
        stk.push(3); /* push "3" into stack */
        stk.push(7); /* push "7" into stack */
        stk.push(4); /* push "4" into stack */
        cout << stk; /* print stack contents */
        data = stk.pop(); /* get item from stack */
        cout << "data from stack: " << data << endl;
        cout << stk; /* print stack contents */
        data = stk.pop(); /* get item from stack */
        cout << "data from stack: " << data << endl;
        cout << stk; /* print stack contents */
        data = stk.pop(); /* get item from stack */
        cout << "data from stack: " << data << endl;
        cout << stk; /* print stack contents */
        return 0;
        }
```

program output:

```
stack: [ ]
stack: [ 8 ]
data from stack: 8
data from stack: -1
stack: [ 4 7 3 ]
data from stack: 4
stack: [ 7 3 ]
data from stack: 7
stack: [ 3 ]
data from stack: 3
stack: [ ]
```

LESSON 7 EXERCISE 1

Add an **isEmpty()** method to the stack module that returns true if the stack is empty.

LESSON 7 EXERCISE 2

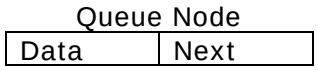
Implement the stack list ADT with a double link list. When printing out the double link list print the stack in reverse order from the top to the bottom. Call your program stkdlist.c and your header file stkdlist.h.

LESSON 7 EXERCISE 3

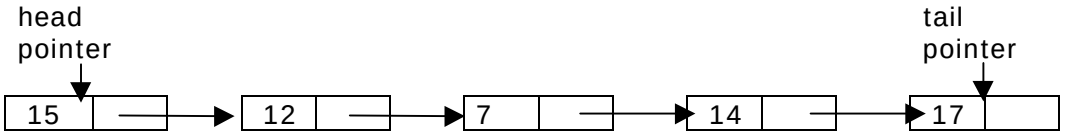
Re-write the stack list ADT as a template class. This way we can have any dada type in our stack.

QUEUE LIST ADT MODULE

A Queue lets you add items to the end of a list and to remove items from the start of a list. A Queue implements **first in first out** (FIFO). This means if you insert 1, 2, and 3 you will get back 1,2 and 3. A Queue has both a **head pointer** and a **tail pointer**. Think as a Queue as a line in a bank. People enter the bank and stand in line. People get served in the bank at the start of the line. As each person is served they are removed from the Queue. Newcomers must start lining up at the end of the line. The first people who enters the bank are the first to be served and the first to leave. When you insert an item on the Queue it is known as **enqueue**. When you remove an item from the Queue it is known as **dequeue**. We build our queue using link lists. A link as you know is a memory block that has a data field and a pointer to another node. We call the nodes for our Queue link list Queue Nodes.

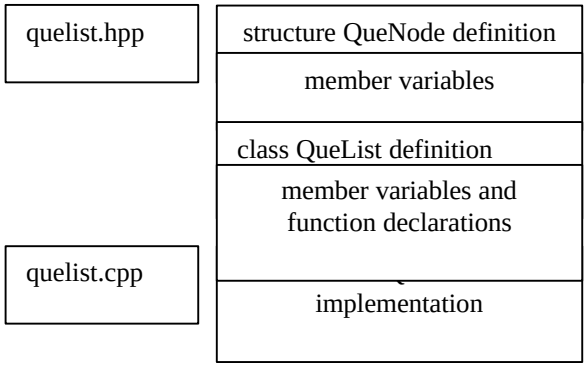


Every element in our queue will be a queue node.



Implementing a Queue Link List

To implement a Queue Link List you need to insert a node for the enqueue operation, and remove a node for the dequeue operation. Each node will contain the data and pointers to the next node. We use the modular approach to implement our Queue Link List ADT where the Queue List structure is use to hold the start and end of the list and how many nodes we have in the list. We also need a QUEUE NODE structure to represent a node.



The QueNode data structure holds the data and links to next and previous nodes.

The QueList class holds the start of list and number of nodes in list.

The QueList implementation contains the function definitions.

Here's the code for the queList ADT module. V
true, false, error and bool. If your compiler dc
the defs.h file in list.h.

```
/* defs.h */
#ifndef __DEFS
#define __DEFS

#define true  1
#define false 0
#define error -1
typedef int bool;
#endif
```

The queue link nodes will have there own separate data structure separate from the queue list class
s. Here's the code for the queue list ADT:

```
/* quelist.hpp */
#ifndef __QUELIST
#define __QUELIST
//#include "defs.h"
typedef int QueData;
/* queue list node */
struct QueNode
        {
        QueData data;
        QueNode* next;

        QueNode(QueData data)
        {
        this->data = data;
        this->next=NULL;
        }
};
```

```
/* queue list module data structure */
class QueList
        {
        private:
        QueNode*  head;
        QueNode*  tail;
        /* queue module function prototypes */
        public:
        /* initialize queue */
        QueList();
        /* put item into queue */
        bool enqueue(QueData data);
        /* remove data item from queue */
        QueData dequeue();
        /* deallocate memory for queue */
        ~QueList();
        /* print out contents of queue */
        friend ostream& operator << (ostream& out, QueList& list);
        };
        #endif
```

Queue List Implementation code file:

```
        /* quelist.cpp */
        #include <iostream.h>
        #include "quelist.hpp"
        /* queue module functions */
        /* initialize queue */
        QueList::QueList()
                {
                this->head = NULL;
                this->tail = NULL;
                }
        /* insert items into the queue list */
        bool QueList::enqueue(QueData data)
                {
                /* allocate memory for queue node */
                QueNode* node = new QueNode(data);

                /* check if memory allocated */
                if(node != NULL)
                        {
                        if(this->tail != NULL)(this->tail)->next = node;
                        else this->head = node;
                        this->tail = node;
                        }
                return node != NULL;
                }
```

```
/* remove items from the queue */
QueData QueList::dequeue()
        {
        QueData data;
        QueNode* node;
        /* check if queue empty */
        if(this->head == NULL)return -1;
        data = (this->head)->data;
        /* remove from item at head of queue */
        node = (this->head)->next;
        free(this->head);
        this->head = node;
        if(node == NULL)this->tail = NULL;  /* queue is empty */
        return data;
        }
/* deallocate memory for queue */
QueList::~QueList()
        {
        QueNode* node = this->head;
        while(node != NULL)
                {
                /* deallocate memory for queue item */
                QueNode* next = node->next;
                delete (node);
                node = next;
                }
this->head = NULL;
this->tail = NULL;
        }
/* print out queue items */
ostream& operator << (ostream& out, QueList& list)
        {
        QueNode* node = list.head;
        out << "queue: [ ";

        while(node != NULL)
                {
                out << node->data << " ";
                node = node->next;
                }
        out << "]"<<endl;
        return out;
        }


/* main function to test queue module */
int main()
        {
        QueList  que; /* create queue object */

        que.enqueue(5); /* put 5 into queue */
        cout << que; /* print out queue */
```

```
        /* get data item from queue and print out value */
        cout << "data value: " << que.dequeue() << endl;
        cout << que; /* print out queue */
        que.enqueue(1); /* put 1 into queue */
        cout << que; /* print out queue */
        que.enqueue(2); /* put 2 into queue */
        cout << que; /* print out queue */
        que.enqueue(3); /* put 3 into queue */
        cout << que; /* print out queue */
        /* get data item from queue and print out value */
        cout << "data value: " << que.dequeue() << endl;
        cout << que; /* print out queue */
        /* get data item from queue and print out value */
        cout << "data value: " << que.dequeue() << endl;
        cout << que; /* print out queue */

        /* get data item from queue and print out value */
        cout << "data value: " << que.dequeue() << endl;
        cout << que; /* print out queue */
        /* get data item from queue and print out value */
        cout << "data value: " << que.dequeue() << endl;
        cout << que; /* print out queue */
        return 0;
        }
```

program output:

```
queue: [ 5 ]
data value: 5
queue: [ ]
queue: [ 1 ]
queue: [ 1 2 ]
queue: [ 1 2 3 ]
data value: 1
queue: [ 2 3 ]
data value: 2
queue: [ 3 ]
data value: 3
queue: [ ]
data value: -1
queue: [ ]
```

LESSON 7 EXERCISE 4

Add an **isEmpty()** method to the queue list module that returns true if the queue is empty.

LESSON 7 EXERCISE 5

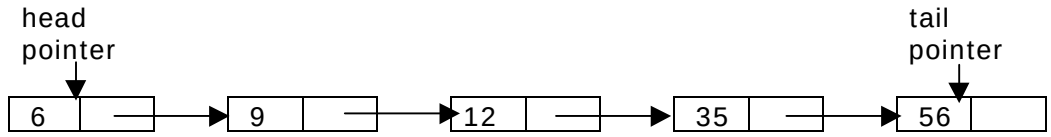
A dequeue lets you add and remove nodes from both ends of the queue. Make a dequeue using a double link list. The double link list will let you make insertions and deletions from both end of the dequeue more easily. Call your dequeue module header file dquelist.h and your module dquelist.c.

LESSON 7 EXERCISE 6

Re-write the queue list ADT as a template class. This way we can have any dada type in our queue.

PRIORITY QUEUE

A priority queue is like a queue but the big difference is that all the queue items are arrange in importance. The high priority items are at the head of the queue and the lowest priority item are at the end of the queue. All of the module functions are the same except the enqueue function must order the items in priority. Every item data element will be prioritized to represent a time a value or a name. No matter what the item is it must be inserted in priority. The most common priority is ascending order or descending order. Ascending order gives small value the greatest priority. 2 6 9 12 35 56, descending order gives high values greater priority 56 35 12 9 6 2. It all depend how you code your enqueue function. Every element in the priority queue will be in ascending or descending order.



We use recursion for inserting the items in a prioritized order into the queue list. it is very easy to implement the enqueue function using recursion. Recursion works by copying the node pointers until it finds the spot where to insert the new item. Recursion is a little slower than non-recursion approach bur uses fewer lines of code. since the queues are not very long then we do not worry about the time requirements. A driver is needed for recursive programs. Here's the enqueue() function for a priority queue.

```
/* insert items into the queue list by priority */
bool PQueList::enqueue(QueData data)
{
QueNode *prev=NULL, *curr; /* pointers to nodes */
/* allocate memory for queue node */
QueNode* node = new QueNode(data);
if (node == NULL)return false;
/* check for empty list */
if(this->head == NULL)
        {
        this->head = node; /* set head to point to new node */
        this->tail = node; /* set tail to point to new node */
        return true;
        }
/* find out where to insert new node */
/* insert in ascending order */
curr = this->head; /* point to start of list */
/* loop till end of list */
while(curr != NULL)
        {
        /* if data to insert value is less than current break */
        if(data < curr->data)break;
        prev=curr; /* save previous node */
        curr=curr->next; /* current point to next node */
        }
```

```
/* check for start of list */
/* node points to start of list */
if(prev == NULL)
        {
        /* new node points to start of list */
        node->next = this->head;
        this->head = node; /* make head point to new node */
        }
/* check for middle of list */
else
        {
        node->next=prev->next; /* new node points to current */
        prev->next=node; /* previous node points to new node */
        /* check for end of list */
        if(curr == NULL)this->tail = node; /* assign tail to end */
        }
return true;
} /* end enqueue */
```

LESSON 7 EXERCISE 5

Make a priority queue class called PQueList using the above **enqueue()** function. Make sure your PQueList class has a **isEmpty()** function.

LESSON 7 EXERCISE 6

Add a parameter to the insert routine to the priority queue module so that is can prioritize in ascending or descending order.

LESSON 7 EXERCISE 7

Rewrite the prioritized enqueue function using recursion. The enqueue function will call an insert function. By using recursion there will be less code. Call your header file rpriquelist.h and your module rpriquelist.c.

LESSON 7 EXERCISE 8

Make a priority queue using a double link list.

C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 8

File:	CdsGuideL8.doc
Date Started:	April 15,1999
Last Update:	Dec 26,2001
Status:	draft

LESSON 8 SORTING LINK LISTS

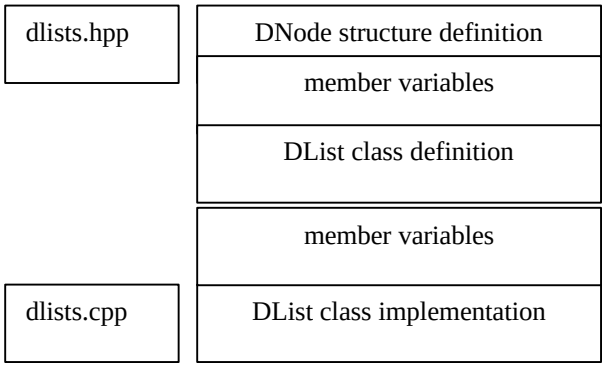
SORTING LISTS

Sorting is arranging a list in ascending or descending order. There are many sorting algorithms. The goal of a sorting algorithm is to sort fast. It is almost a contest to see which algorithm is the fastest. Sort time is measured by **big O** notation. Big O notation states the **worst case** running time. An example if you had an array of n items and you wanted to search for an item, we can use big O notation to estimate the running time to find the item. If the item was the last element, and we start searching at the start of the list. The search time would take n comparisons. This would be the worst case. If the item to be found was at the start of the list then this would be the best case. Unfortunately, we always need the worst case estimate. No matter where the item is, the search time would be the worst case of **n** items. This is what **big O** notation is all about, the **worst case** estimate. Big O states the worst case time estimate. No matter where your item is in the list, it is still O(n). Constants are not to be included in big O notation O(2n) is the same as O(n). Sorting lists is not too much different then sorting arrays. We are still swapping data. The only difference is that we are traversing lists rather than indexing through arrays. When sorting link lists we do not change the links we just swap the data. Most of the list sort routines are O(n**2) O(n squared) this is because we need a loop inside a loop for sorting. The inner loop is O(n) and the outer loop is O(n). When you have loops inside loops then the worst case timing is a multiplication of the two separate loops. Total worst case timing is O(N) * O(N) = O(n ** 2). We use double link lists for sorting. The following table lists the sorting routines for link list and performance.

list sort algorithm	worst case timing	comment
bubble sort	O(n**2)	simple
insertion sort	O(n**2)	simple
selection sort	O(n**2)	simple
merge sort	O(n log n)	recursive
quick sort	O(n log n)	recursive

IMPLEMENTING A DOUBLE LINK LIST

To implement a Double Link List you need functions to insert, remove and search nodes. Each node will contain the data and pointers to the next and previous nodes. You may also want functions to insert at the start or end of the Double Link List or to insert in ascending or descending order. We use the modular approach to implement our Double Link List ADT where a DList structure is use to hold the start and end nodes and how many nodes we have in the list. We also need a DNode structure to represent a node.



The DNode data structure holds the data and links to next and previous nodes.

The DList class holds the start and end of list pointers and number of nodes in list and class function declarations

The class implementation contains the function definitions.

Here's the code for the DList ADT class. We have a small file defs.h that includes definitions for true, false, error and bool. If your compiler does not supply these for you then you need to include the defs.h file in dlists.hpp.

```
/* defs.h */
#define false 0
#define true 1
#define error -1
typedef  int  bool;
```

The DLink nodes will have there own separate data structure separate from the DList class. Since you are now pro's to link list here's the double link list code:

```
/* dlists.hpp */
/* double link list */
//#include "defs.h"
#include <iostream.h>
/* double link list node data */
typedef int DNodeData;
/* double node structure */
struct DNode
{
DNode* prev;
DNodeData data;
DNode* next;
/* initialize DNode */
DNode(DNodeData data)
        {
        this->prev=NULL;
        this->data=data;
        this->next=NULL;
        }
};
```

```
/* double link list class */
class DList
{
public:
DNode* head;
DNode* tail;
int length;
public:
/* function declarations */
/* initialize double link list pointers */
DList();
/* free double link list nodes */
~DList();
/* insert item into double link list */
bool insertDLTail(DNodeData data);
/* remove node at start of list */
int removeDLHead();
/* remove node at end of list */
int removeDLTail();
/* print out double link list items from start of list */
friend ostream& operator <<(ostream& out, DList& dlist);
};
```

Implementation code file for Double Link List:

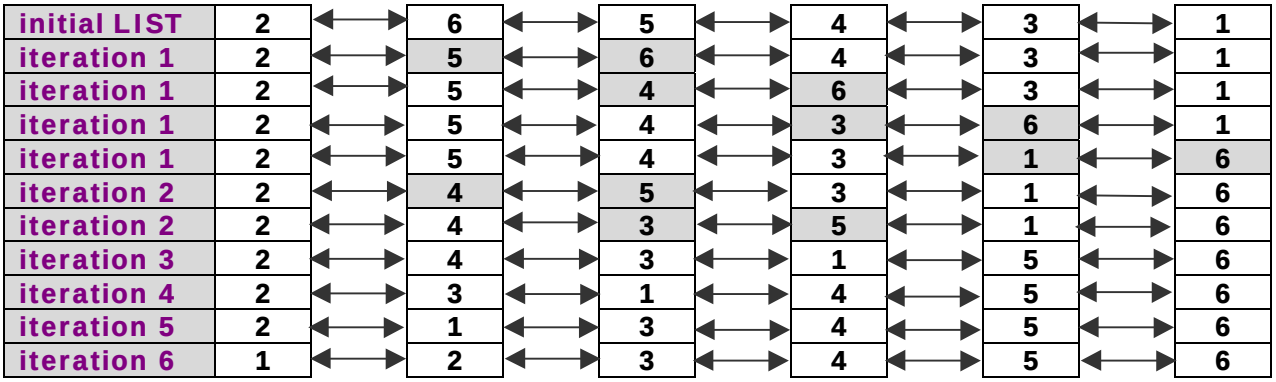
```
/* dlist.cpp */
/*double link list */
#include <iostream.h>
#include "dlist.hpp"


/* initialize dlist pointers */
DList::DList()
        {
        this->head = NULL; /* set head to empty value */
        this->tail = NULL; /* set tail to empty value */
        this->length=0;
        }
/* free nodes in dlist */
DList::~DList()
        {
        DNode* next; /* pointer to a next dnode */
        DNode* curr = this->head; /* point to start of list */
        /* loop till end of list */
        while(curr != NULL)
                {
                next = curr->next; /* point to next node */
                delete(curr); /* free current dnode */
                curr = next; /* set curr to next dnode */
                }
        this->head = NULL; /* set head to empty value */
        this->tail = NULL; /* set tail to empty value */
        }
```

```
/* insert item into dlist at end of list */
 bool DList::insertDLTail(DNodeData data)
        {
        DNode* dnode = new DNode(data); /* allocate memory for new dnode */
        if (dnode == NULL)return false;
        (this->length)++;    /* increment node count */
        /*check for empty list */
        if(this->head == NULL)
                {
                this->head = dnode; /* head points to new node */
                this->tail = dnode; /* tail points to new node */
                return true;
                }
        /* attach to end of list */
        (this->tail)->next = dnode;
        dnode->prev=dlist->tail;
        /* update tail */
        this->tail=dnode;
        return true;
        }
        /* remove node at start of list */
        int DList::removeDLHead(DListPtr dlist)
                {
                DNode* curr = this->head; /* pointer to current dnode */
                DNode* next = this->head->next; /* pointer to a next dnode */
                DNodeData data = curr->data;
                (this->length)--; /* decrement node count */
                /* check for last one item in list */
                if(this->head==this->tail)
                        {
                        this->head=NULL; /* set head to empty value */
                        this->tail=NULL; /* set tail to empty value */
                        }
                /* remove dnode from start of list */
                else
                        {
                        this->head = next; /* head points to next dnode */
                        next->prev = NULL; /* current points to empty node */
                        }
                delete(curr); /* deallocate memory for current node */
                return data;
                }
        /* remove node at start of list */
        int DList::removeDLTail()
                {
                DNode* curr = this->tail; /* pointer to current dnode */
                DNode* prev = this->tail->prev; /* pointer to a previous dnode */
                DNodeData data = curr->data; /* current is tail dnode */
                (this->length)--; /* decrement count */
```

```
        /* check for last one item in list */
        if(this->head==this->tail)
                {
                this->head=NULL; /* set head to empty value */
                this->tail=NULL; /* set tail to empty value */
                }
        /* remove dnode from end of list */
        else
                {
                this->tail->prev = NULL; /* tail points to no node */
                this->tail->next = NULL; /* tail points to no node */
                this->tail=prev; /* tail now points to previous dnode */
                }
        delete(curr); /* deallocate memory for current node */
        return data;
        }
/* test if list is empty */
bool DList::isEmpty()
        {
        return this->length == 0;
        }
/* print out items at start of dlist */
ostream& operator <<(ostream& out, DList& dlist)
        {
        DNode* curr=dlist.head; /* point to start of dlist */
        /* check if dlist empty */
        if(curr == NULL)
                {
                out << "double list is empty" << endl;
                return out;
                }
        out << "dlist from start: ";
        /* loop till end of list */
        while(curr != NULL)
                {
                out << curr->data << " "; /* print out data item */
                curr = curr->next; /* point to next item */
                }
        out << endl;
        return out;
        }
```

BUBBLE SORT

This is the most common sorting algorithm. It works by traversing a list of n nodes n times always exchanging or swapping 2 node data elements, only if the a element is greater than the next element. Since the list will be examined n times we are assured that the list will be sorted. There will be 6 iterations. For each iteration the bubble sort algorithm will check if the one node element is greater than the right element. If they are then they will be swapped. We do not change the link links we just swap the data. Each element pairs are scanned sequentially. This means an element could be swapped twice. We show the swapped pairs shaded in gray for the first iteration. We only show the final results for iteration 2 to 6. At iteration 6 the list is sorted. The double arrow are the links double link list.



initial LIST	2	↔	6	↔	5	↔	4	↔	3	↔	1
iteration 1	2	↔	5	↔	6	↔	4	↔	3	↔	1
iteration 1	2	↔	5	↔	4	↔	6	↔	3	↔	1
iteration 1	2	↔	5	↔	4	↔	3	↔	6	↔	1
iteration 1	2	↔	5	↔	4	↔	3	↔	1	↔	6
iteration 2	2	↔	4	↔	5	↔	3	↔	1	↔	6
iteration 2	2	↔	4	↔	3	↔	5	↔	1	↔	6
iteration 3	2	↔	4	↔	3	↔	1	↔	5	↔	6
iteration 4	2	↔	3	↔	1	↔	4	↔	5	↔	6
iteration 5	2	↔	1	↔	3	↔	4	↔	5	↔	6
iteration 6	1	↔	2	↔	3	↔	4	↔	5	↔	6

Here's the code for the bubble sort. you will notice it is simply a loop inside a loop. We only swap the elements if the left element is greater than the right element. In a swap routine it is important to keep a temporary variable or else you would end up with the same element in both swapped locations!. We use a double link list so that we can traverse the list frontward and backwards. You will need the dlist module from previous lessons.

```
/* bdlsort.cpp */
#include <iostream.h>
#include "dlists.hpp"
void bdlsort(DList& dl);
/* driver to test bubble sort of double link list */
void main()
        {
        DList dlist;
        dlist.insertDLTail(2);
        dlist.insertDLTail(6);
        dlist.insertDLTail(5);
        dlist.insertDLTail(4);
        dlist.insertDLTail(3);
        dlist.insertDLTail(1);
        cout << dlist;
        bdlsort(dlist);
        cout << dlist;
        }
```

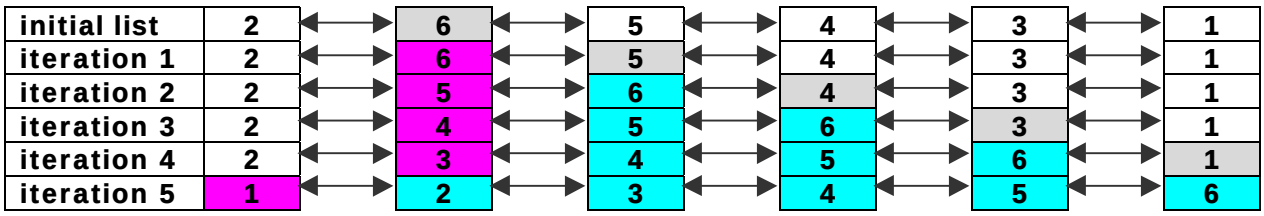
program output:

```
dlist from start: 2 6 5 4 3 1
dlist from start: 1 2 3 4 5 6
```

```
/* bubble-sort on a double link list */
void bdlsort(DList& dl)
        {
        int t;
        DNode *n1,*n2;
        if(dl.head == NULL) return;
        /* traverse backward through list */
        for(n2 = dl.tail; n2 != dl.head; n2=n2->prev)
                {
                /* traverse forward through list */
                for(n1 = dl.head; n1!=n2; n1 = n1->next)
                        {
                        /* check if left greater than right */
                        if(n1->data > (n1->next)->data)
                                {
                                t = n1->data;
                                n1->data = (n1->next)->data; /* swap */
                                (n1->next)->data = t;
                                }
                        }
                }
        }
```

INSERTION SORT

Insertion sort keeps track of two sub lists inside an list. A sorted list and a unsorted sub list. The **first** element of the unsorted sub list is removed and inserted in the correct position of the sorted sub list. The elements of the sorted sub list must be shifted right from the insertion point to make room for the key to be inserted. As the sort algorithm is running the unsorted sub list is getting smaller and the sorted sub array is getting larger until the list is sorted. Initially the sorted list is empty and the unsorted sub list is full. The gray shaded elements represent the key to be inserted, the blue shaded elements represent the elements shifted to accommodate where the key will be inserted. The violet shaded elements represent the position where the key is inserted. We are always shifting between where the key is to be removed to where the key will be inserted. We only have 5 iterations because the key starts at array index 1 rather than zero. Do you know why ? You need a place for shifting and to insert the key.



initial list	2	6	5	4	3	1
iteration 1	2	6	5	4	3	1
iteration 2	2	5	6	4	3	1
iteration 3	2	4	5	6	3	1
iteration 4	2	3	4	5	6	1
iteration 5	1	2	3	4	5	6

```
/* insertion sort on double link list */
void insdlsort(DList& dl)
        {
        int key;
        DNode *n1 = dl.head;   /* point to start of list */
        DNode *n2;
```

```
/* traverse from start to end of double link list */
for(n1=n1->next; n1 != NULL; n1=n1->next)
        {
        n2 = n1;    /* point to list position */
        key = n1->data;    /* get data value at list position */
        /* shift all data elements forward from list position */
        while(n2->prev != NULL && (n2->prev)->data > key)
            {
            n2->data = (n2->prev)->data;  /* assign previous data to current position */
            n2=n2->prev;  /* point to previous node */
            }
        /* insert key at list position */
        n2->data=key;
        cout << dl;  /* optional print */
        }
        }
```

program output:

```
dlist from start: 2 6 5 4 3 1
dlist from start: 2 6 5 4 3 1
dlist from start: 2 5 6 4 3 1
dlist from start: 2 4 5 6 3 1
dlist from start: 2 3 4 5 6 1
dlist from start: 1 2 3 4 5 6
dlist from start: 1 2 3 4 5 6
```

SELECTION SORT

For selection sort for each iteration we select the smallest key in the unsorted list and put the selected key at the start of the list and shift the element right form where the smallest key was found. The gray shaded elements represent the minimum selected element to be inserted, the blue shaded elements represents where the minimum element is swapped When there is no corresponding blue element, this means the element is swapped with itself.

initial array	**2**	↔	**6**	↔	**5**	↔	**4**	↔	**3**	↔	**1**
iteration 1	**1**	↔	**6**	↔	**5**	↔	**4**	↔	**3**	↔	**2**
iteration 2	**1**	↔	**2**	↔	**5**	↔	**4**	↔	**3**	↔	**6**
iteration 3	**1**	↔	**2**	↔	**3**	↔	**4**	↔	**5**	↔	**6**
iteration 4	**1**	↔	**2**	↔	**3**	↔	**4**	↔	**5**	↔	**6**
iteration 5	**1**	↔	**2**	↔	**3**	↔	**4**	↔	**5**	↔	**6**

```
/* selection sort */
void seldlsort(DList& dl)
        {
        int t;
        DNode *cur,*min,*n;
         /* traverse from start of list to end */
        for(cur = dl.head; cur!=dl.tail; cur=cur->next)
                {
                min = cur ;
                n = cur;
                 /* find the smallest element after and including cur */
                do
                        {
                        n = n->next;
                        if(n->data < min->data)min = n;
                        }while(n != dl.tail);
                /* put at front */
                 t = cur->data;
                cur->data = min->data;
                min->data = t;
                }
        }
```

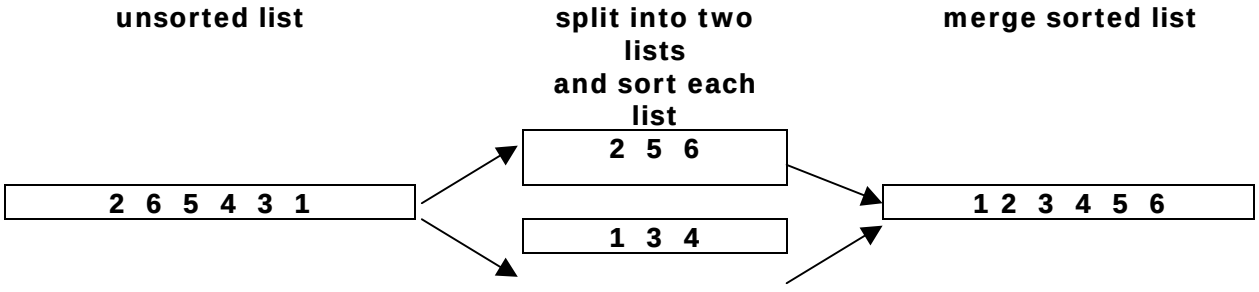
program output:
dlist from start: 2 6 5 4 3 1
dlist from start: 1 2 3 4 5 6

LESSON 8 EXERCISE 1

Type in the bubble sort, insertion sort , selection sort, programs and get them going. Trace through them with the debugger and watch them go. Verify our traces.

MERGE SORT

The idea behind merge sort is to divide a list in half, sort each list then merge the lists together as a final sorted list. To merge the two lists we make a thirds list. We insert into the third list the smallest element of each of the sorted lists as we traverse through the sorted lists.



The merge sort is done recursively. We first sort call the mdlsort() routine to sort each 1/2 of the list, then we call merge() function to combine the two halves. Since this is recursive every thing is done in stages. Merge() is called many times to sort all the partial stages. The merge sort sorts the two separate lists itself and merges them.

```
#include <iostream.h>
#include "dlists.hpp"
void mdlsort(DList& dl);
void merge(DList& dl1, DList& dl2, DList& dl);

/* driver to test  merge  sort */
void main()
        {
        DList dlist;
        dlist.insertDLTail(2);
        dlist.insertDLTail(6);
        dlist.insertDLTail(5);
        dlist.insertDLTail(4);
        dlist.insertDLTail(3);
        dlist.insertDLTail(1);
        cout << dlist;
        mdlsort(dlist);
        cout << dlist;
        }
 /* merge sort on double link list */
void mdlsort(DList& dl)
        {
        int i;
        int d;
        DList dl1;
        DList dl2;
        int n = dl.length;
        if(n < 2)return;
        /* copy for  two lists */
        for (i=1; i <= (n+1)/2; i++)
                {
                d=dl.removeDLHead();
                dl1.insertDLTail(d);
                }
        for (i=1; i <= n/2; i++)
                {
                d=dl.removeDLHead();
                dl2.insertDLTail(d);
                }
        cout << dl1;  /* optional print list dl1 */
        cout << dl2;  /* optional print list dl2 */

        mdlsort(dl1); /* sort both lists by recursion */
        mdlsort(dl2);
        cout << dl1;  /* optional print list dl1 */
        cout << dl2;  /* optional print list dl2 */
        /* merge two lists */
        merge(dl1,dl2,dl);
        cout << dl;  /* optional print list dl */
        }
```

```
/* Merges sorted list dl1 and dl2 into a sorted list dl */
void merge(DList& dl1, DList& dl2, DList& dl)
	{
	int d,d1,d2;
	/* loop till both lists empty */
	while(!dl1.isEmpty() && !dl2.isEmpty())
		{
		d1 = ((DNode*)(dl1.head))->data;
		d2 = ((DNode*)(dl2.head))->data;
		/* check if first element of dl1 is less than dl2 */
		if(d1 < d2)
			{
			d = dl1.removeDLHead();
			dl.insertDLTail(d);
			}
		else
			{
			d = dl2.removeDLHead();
			dl.insertDLTail(d);
			}
		}
	/* empty list dl1 into dl */
	if(dl1.isEmpty())
		{
		while(!dl2.isEmpty())
			{
			d = dl2.removeDLHead();
			dl.insertDLTail(d);
			}
		}
		/* empty list dl2 into dl */
		if(dl2.isEmpty())
		{
		while(!dl1.isEmpty())
			{
			d = dl1.removeDLHead();
			dl.insertDLTail(d);
			}
		}
	 }
```

Since the operation is quite complex and we cannot trace each routine. we only supply a top level chart.

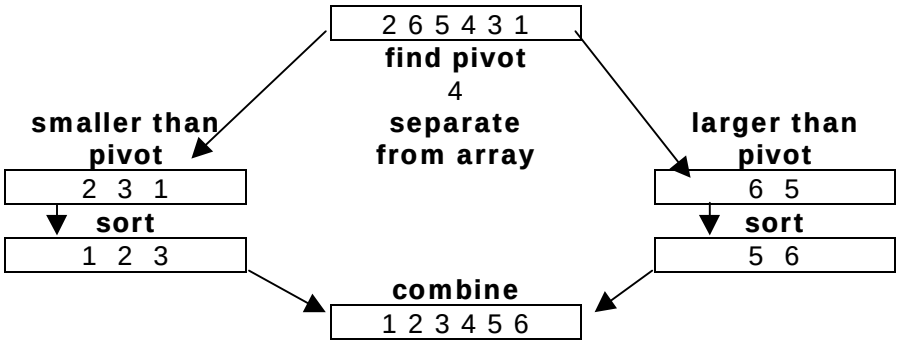
	stack pairs left, right	functio n	left lpos	middle rpos	right rend	a	b
0	ra	msort	0	---	5	2 6 5 4 3 1	0 0 0 0 0 0
1	ra 0,5	msort	0	2	5	2 6 5 4 3 1	0 0 0 0 0 0
2	ra 0,5 0,2	msort	0	1	2	2 6 5 4 3 1	0 0 0 0 0 0
3	ra 0,5 0,2 0,1	msort	0	0	1	2 6 5 4 3 1	0 0 0 0 0 0
4	ra 0,5 0,2 0,1	msort	0	0	0	2 6 5 4 3 1	0 0 0 0 0 0
5	ra 0,5 0,2	msort	0	1	1	2 6 5 4 3 1	0 0 0 0 0 0
6	ra 0,5 0,2 0,1	msort	1	--	1	2 6 5 4 3 1	0 0 0 0 0 0
7	ra 0,5 0,2	**merge**	0	1	1	2 6 5 4 3 1	0 0 0 0 0 0
8	ra 0,5	msort	0	0	2	2 6 5 4 3 1	2 6 0 0 0 0
9	ra 0,5 0,2	msort	2	--	2	2 6 5 4 3 1	2 6 0 0 0 0
10	ra 0,5	**merge**	0	2	2	2 6 5 4 3 1	2 6 0 0 0 0
11	ra	msort	0	3	5	2 5 6 4 3 1	2 5 6 0 0 0
12	ra 0,5	msort	3	4	5	2 5 6 4 3 1	2 5 6 0 0 0
13	ra 0,5 3,5	msort	3	4	4	2 5 6 4 3 1	2 5 6 0 0 0
14	ra 0,5 3,5 3,4	msort	3	--	3	2 5 6 4 3 1	2 5 6 0 0 0
15	ra 0,5 3,5	msort	3	4	4	2 5 6 4 3 1	2 5 6 0 0 0
16	ra 0,5 3,5 3,4	msort	4	--	4	2 5 6 4 3 1	2 5 6 0 0 0
17	ra 0,5 3,5	**merge**	3	4	4	2 5 6 4 3 1	2 5 6 0 0 0
18	ra 0,5 3,5	msort	3	4	5	2 5 6 3 4 1	2 5 6 3 4 0
19	ra 0,5 3,5	msort	5	--	5	2 5 6 3 4 1	2 5 6 3 4 0
20	ra 0,5	**merge**	3	4	5	2 5 6 3 4 1	2 5 6 3 4 0
21	ra	**merge**	0	2	5	2 5 6 1 3 4	2 5 6 1 3 4
22	ra	msort	0	2	5	1 2 3 4 5 6	1 2 3 4 5 6

LESSON 8 EXERCISE 2

Type in the merge sort programs and get them going. Trace through them with the debugger and watch them go. Verify our traces.

QUICKSORT

Quicksort is the fastest known sorting algorithm. The average running time is O(n log n). Quick sort is implemented with recursion and is easy to understand. The algorithm is as follows : pick a pivot point. Put the smaller elements less that the pivot point in a list subset, put the larger elements then the pivot in a right list subset,. sort each list subset. The quick sort algorithm is used to sort itself by recursion. The trick is to pick the pivot point. Choosing the correct pivot point will make this routine run fast. Picking the wrong pivot will give poor performance.




```
 /* qdlsort.cpp*/
#include <iostream.h>
#include "dlists.hpp"
/* prototypes */
void qdlsort(DList& dl);
/* test driver for quick sort */
void main()
        {
        DList dlist;
        dlist.insertDLTail(2);
        dlist.insertDLTail(6);
        dlist.insertDLTail(5);
        dlist.insertDLTail(4);
        dlist.insertDLTail(3);
        dlist.insertDLTail(1);
        cout << dlist;
        qdlsort(dlist);
        cout << dlist;
        }
/* quick sort using double link list */
void qdlsort(DList& dl)
        {
        int pivot;
        int d;
        DList dll;
        DList dlr;
        if (dl.length < 2)return;
        cout << dl;/* optional print list */
        pivot = dl.removeDLTail();
        cout << "pivot: "<< pivot << endl;
        /* fill in the left and right elements */
        while ( !dl.isEmpty() )
                {
                /* if next element in the list is less than pivot */
                /* insert into left otherwise insert into right list. */
                d = dl.removeDLHead();
                if ( d < pivot) dll.insertDLTail(d);
                else dlr.insertDLTail(d);
                }
```

```
        qdlsort(dll); /* sort the left partition */
        qdlsort(dlr); /* sort the right partition */
        cout << dll; /* optional print list */
        cout << dlr; /* optional print list */
/* insert the left elements, smaller than right  */
while ( !dll.isEmpty() )
        {
        d = dll.removeDLHead();
        dl.insertDLTail(d);
        }
/* insert pivot , bigger then left smaller than right */
dl.insertDLTail(pivot);
/* insert the right end, the largest elements */
while (!dlr.isEmpty() )
        {
        d = dlr.removeDLHead();
        dl.insertDLTail(d);
        }
cout << dl; /* optional print list */
}
```

LESSON 8 EXERCISE 3

Type in the quick sort program. Trace through them with the debugger and make a chart.

LESSON 8 EXERCISE 4

Convert bubble sort, insertion sort , selection sort, merge sort and quick sort programs to using a single link list rather than a double link list.

LESSON 8 EXERCISE 5

Make a list of 10000 random elements which is the fastest algorithm, which is the slowest ? List the algorithms from fastest to slowest.

C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 9

File:	CppdsGuideL9.doc
Date Started:	July 24,1998
Last Update:	Dec 26, 2001
Status:	proof

LESSON 9 BINARY SEARCH, HASH TABLES AND BINARY HEAPS

BINARY SEARCH

A Binary search is used when you have many items to search for in an array that has already been sorted. The binary search splits the array in half. If the item to be searched is not found, then the array is split in half again until the item is found. Which array half to search in is determined by the item value. If the item value is higher than the medium value then the upper half is searched. If the item value is lower than the medium value then the lower half is searched. We demonstrate with ten sorted numbers looking for the key 7. The gray shaded area is the left side where the blue shaded area is the right side. The violet shade is when the element is found. We first split the array up and 8 will be located in the upper half. We split this upper half and the value 8 will again be located in the upper half.

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

Here is the binary search code.

```
/* binsrch.cpp */
#include<iostream.h>

int binsrch( int* a, int x, int n );
void print(int a[],int s,int n);
/* test driver for binary search */
void main()
        {
        int i;
        int k = 7;
        int a[] = {0,1,2,3,4,5,6,7,8,9};
        cout << "finding: " << k << endl;
        i = binsrch(a,k,10);
        cout << "found: " << a[i] << endl;
        }
```

Here is the binary search function:

```
/* binary search */
int binsrch( int* a, int x, int n )
        {
        int low = 0;
        int high = n - 1;
        int mid;
        /* search while low is less than high */
        while( low < high )
                {
                print(a,low,high+1);
                mid = ( low + high ) / 2;  /* calculate middle */
                if( a[ mid ] < x ) /* test key */
                low = mid + 1;     /* upper half */
                else
                high = mid;       /* lower half */
                }
        return low;             /* low is key */
        }
/* print array from start to end */
void print(int a[],int s,int n)
        {
        int i;
        cout << "[ ";
        for(i=s; i<n; i++)
        cout << a[i] << " ";
        cout << " ]" << endl;
        }
```

program output:

```
finding: 7
[ 0 1 2 3 4 5 6 7 8 9 ]
[ 5 6 7 8 9 ]
[ 5 6 7 ]
found: 7
```

binary search using recursion

The binary search routine easily converts to a recursive function. If the key is lower than the middle value the function calls itself from the low to the high. If the key is higher then the middle position then the function calls itself from middle to high. Recursion stops when the key is the low value.

```
/* binsrchr.cpp */
#include <iostream.h>
int binsrchr( int* a, int k, int low, int high);
void print(int a[],int s,int n);


/* test driver for binary search recursion */
void main()
        {
        int i;
        int k = 7;
        int a[] = {0,1,2,3,4,5,6,7,8,9};
        cout << "finding:" << k << endl;
        i = binsrchr(a,k,0,9);
        cout << "found: " << a[i] << endl;
        }
```

```
/* binary search  using recursion */
int binsrchr( int* a, int k, int low, int high )
        {
        int mid;
        print(a,low,high+1);
        /* done if low equals middle */
        if(low == high)
                {
                if(a[low] == k) return low;  /* key found */
                else return -1;  /* key not found */
                }
        else
                {
                mid = ( low + high ) / 2;         /* calculate middle */
                if( a[ mid ] < k )              /* test key */
                   return binsrchr(a,k,mid + 1,high); /* use upper half */
                else
                   return binsrchr(a,k,low,mid);  /* use lower half */
                }
        }
/* print array from start to end */
void print(int a[],int s,int n)
        {
        int i;
        cout << "[ ";
        for(i=s;i<n;i++)
        cout << a[i] << " ";
        cout << "]" << endl;
        }
```

program output:

```
finding: 7
[ 0 1 2 3 4 5 6 7 8 9 ]
[ 5 6 7 8 9 ]
[ 5 6 7 ]
[ 7 ]
found: 7
```

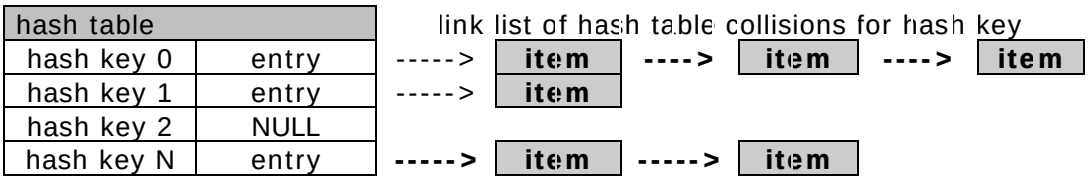
LESSON 9 EXERCISE 1

Type in the non-recursive and recursive binary search routine and trace them and make a chart., Verify that the both work the same. Modify both functions so they are more efficient as soon as it finds the key as the medium exit.

HASH TABLE ADT CLASS

Hash tables allows you to have faster insertion, search and deletion times . A hash table is an array of values represented by a **key**. Each item in the hash table is represents by a unique key called the **hash key**. The hash key is the hash table **index** used for inserting and retrieving items. The hash key is usually the item value to be inserted **mod** the length of the hash table. The mod is the remainder after division. A clock does mod 12 arithmetic. If you are at 11:00 o'clock and you add 2 hours then you are at 1:00 o'clock not 13:00 o'clock. 13 mod 12 is 1. A number mod length will give you a unique key. For string or numeric numbers you can add up each individual characters or digits and then take the mod of the hash table length. It is desirable to get a unique key. Unfortunately this is not possible and **collisions** occur. Collisions happen when two input values have the same hash key. There are many algorithms to handle collisions bur it is easier to build a **link list** of all collisions for a particular key. It is most desirable to avoid collisions and keep the link lists as short as possible.

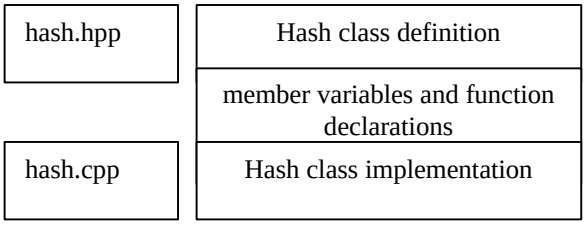
hash key = value * **mod** table length



To construct a hash table we need the **Link List** module ADT. When a hash table is 50 % full it has to be rehashed. Rehash means the hash table size has to be increased by 2. By re-hashing or increasing the size of the hash table the occurrence of collisions is reduced. To rehash a table we must copy all the entries from the old hash table and recalculate the new hash key locations. Here's the code for the hash table module header file:

Implementing a Hash Table

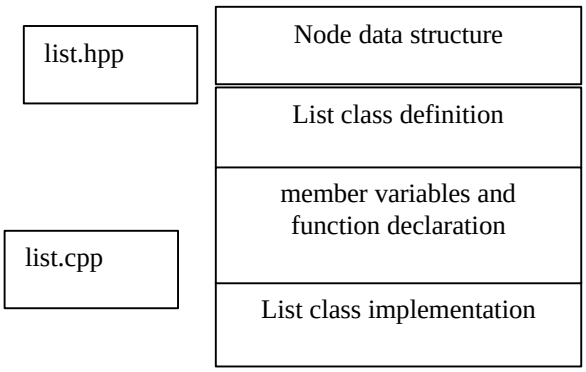
To implement a Hash Table you need functions to insert, remove and search. You will need an array of list pointers and the Link List ADT to store the collisions. We use a class to implement our Hash Table ADT where the class will store the size of the hash table and the array of link list pointers.



The Hash class holds the array of link list pointers and size of the hash table and function declarations

The implementation contains the function definitions.

From the Link List Lesson here is the Link List ADT. You will have to change the NodeData type from int to char*.



The Node data structure holds the data and link to next node.

The List class holds the start and end of list pointers and number of nodes in list.

The List class implementation contains the function definitions.

Here's the code for the Hash Table ADT module. We have a small file defs.h that includes definitions for true, false, error and bool. If your compiler does not supply these for you then you need to include the defs.h file in list.h.

```
/* defs.h */
#ifndef __DEFS
#define __DEFS
#definre false 0
#define true 1
#define error -1
typedef int bool;
#endif
```

Here's the Hash Table class definition header file:

```
/* hash.hpp */
//#include "defs.h"
#include "list.hpp"
/* hash table module header file */
/* data structure definitions */
typedef char* HashData; /* hash table data type */
/* hash ADT module data structure */
class Hash
{
private:
List* h; /* pointer to vector module */
int size; /* size of hash table */

public:
/* initialize hash table */
Hash(int size);
/* calculate hash key from data */
int hashKey(HashData data);
/* insert item into hash table */
bool insertHash(HashData data);
/* free all memory belonging to hash table */
~Hash();
/* remove item from hash table */
bool removeHash(HashData data);
/* search hash table for entry */
Node* searchHash(HashData);
/* print out hash table entries */
ostream& operator++(ostream& out,Hash& hash);
};
```

Here's the hash class implementation file:

```
/* hash.cpp */
#include <iostream.h>
#include <string.h>
#include "hash.hpp"
#include "list.hpp"
```

```
/* initialize hash table */
Hash::Hash(int size)
        {
        /* make an array of list pointers */
        this->h = new List[size];
        this->size=size; /* store hash table size */
        }
/* free all memory belonging to hash table */
Hash::~Hash()
        {
        }
/* calculate hash key from data */
int Hash::hashKey(HashData data)
        {
        int sum = 0;
        /* add up all characters in data */
        for(unsigned int i=0;i<strlen(data);i++) sum = sum + data[i];
        return (sum % this->size); /* calculate and return key */
        }
/* insert item into hash table */
bool Hash::insertHash(HashData data)
        {
        int key = hashKey(data); /* calculate hash key */
        List* list = &this->h[key]; /* get list */
        return list->insertList(data); /* insert item into list */
        }
/* remove item from hash table */
bool Hash::removeHash(HashData data)
        {
        int key = hashKey(data); /* calculate hash key */
        List* list = &this->h[key]; /* get list */
        return list->removeList(data); /* remove item from list */
        }
/* search hash table for entry */
Node* Hash::searchHash(HashData data)
        {
        Node* node; /* node of item in list */
        int key = hashKey(data); /* calculate hash key */
        List* list = &this->h[key]; /* get list */
        node = list->findList(data); /* find node in list */
        return node;
        }
```

```
/* print out hash table entries */
ostream operator <<(ostream& out,Hash& hash)
        {
        out << "hash table: " << endl;
        /* loop through hash table */
        for(int i=0;i<hash.size;i++)
                {
                List* list = &hash.h[i]; /* get list */
                out << "key "<<  i << " ";
                out << *list; /* print put list */
                }
        return out;
        }
/* main test program */
int main()
{
Hash hash(5); /* create hash object size 5*/
hash.insertHash("today is cloudy"); /* insert item */
cout << hash; /* print hash table */
hash.removeHash("how are you?"); /* remove item */
cout << hash; /* print hash table */
hash.insertHash("it will rain today"); /* insert item */
cout << hash; /* print hash table */
hash.insertHash("do you have an umbrella?"); /* insert item */
cout << hash; /* print hash table */
cout <<"searching for: today is cloudy"; /* search for an item */
Node* node = hash.searchHash("today is cloudy");
if(node==NULL) cout << " : cannot find" << endl;
else cout << " : found " << endl;
hash.removeHash("today is cloudy"); /* remove item */
cout << hash; /* print hash table */
hash.removeHash("today is windy"); /* remove item */
cout << hash; /* print hash table */
hash.insertHash("is it warm today ?"); /* insert item */
cout << hash; /* print hash table */
return 0;
}
```

hash program output:

```
hash table:
key 0: list: today is cloudy
key 1:
key 2:
key 3:
key 4:

hash table:
key 0: list: today is cloudy
key 1:
key 2:
key 3:
key 4:
```

```
hash table:
key 0: list : today is cloudy
key 1:
key 2:
key 3: list: it will rain today
key 4:
hash table:
key 0: list: do you have an umbrella? today is cloudy
key 1:
key 2:
key 3: list: it will rain today
key 4:
searching for: today is cloudy : found
hash table:
key 0: list: do you have an umbrella?
key 1:
key 2:
key 3: list: it will rain today
key 4:
hash table:
key 0: list: do you have an umbrella?
key 1:
key 2:
key 3: list: it will rain today
key 4: list: today is windy
hash table:
key 0: list: do you have an umbrella?
key 1: list: is it warm today ?
key 2:
key 3: list: it will rain today
key 4: list: today is windy
```

LESSON 9 EXERCISE 2

Write the routine to resize the hash table when it gets half full. Add a counter to keep track of how many entries the hash table has. You will have to re-key all items in the old hash table. Call your function reHash.

LESSON 9 EXERCISE 3

Write the hash table using a vector adt and a double link list. The Vector ADT will let you dynamically resize the hash table when it gets empty. A double link list will decrease search time of collisions by able to search from the start or end of the list.

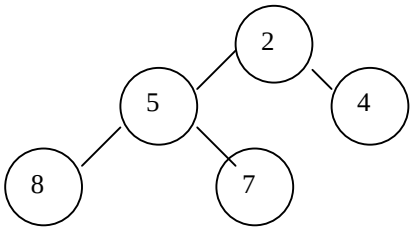
BINARY HEAPS

A binary heap is a binary tree stored in an array. A heap has two properties:

(1) structure property
(2) heap order property

structure property

A heap is a binary tree that is completely filled except the bottom level can be the only exception.



Because of the structure property a heap can be easily represented by an array.

0	1	2	3	4	5	6	7
	2	5	4	8	7		

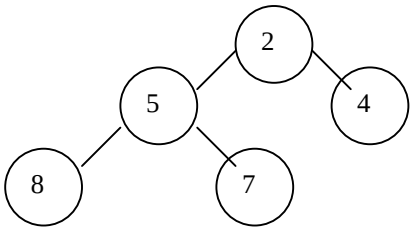
Notice the first element is left blank. To make operations to add and delete elements the first position in the array is left blank. For any element in the array at position i:

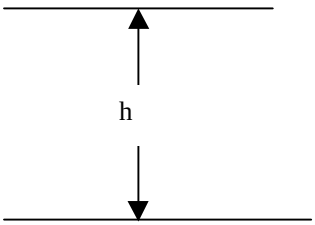
left child	2i
right child	2i + 1
parent of i	i/2

The right child follows the left child.

number of elements

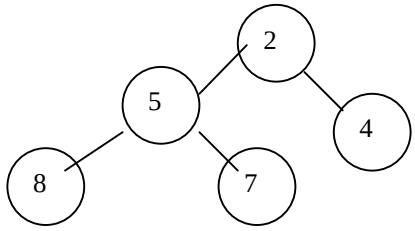
A heap of height h has between 2^{h} to $2^{h+1} - 1$ elements
for out tree of height 2 this is 2^{2} to $2^{3} - 1$ which is 4 to 7





heap order property
The smallest element is at the root in position 1. Any node should be smaller than it descendents. This means all children are greater or equal to their parents. This does not necessary mean the tree is sorted it just means a heap order is enforced.

top item 2 is the smallest
5 and 4 are larger than parent 2
8 and 7 are larger than parent 5



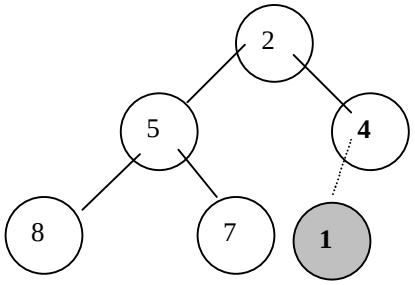
Since a heap is ordered they are used to build priority queues.

HEAP OPERATIONS

inserting an item

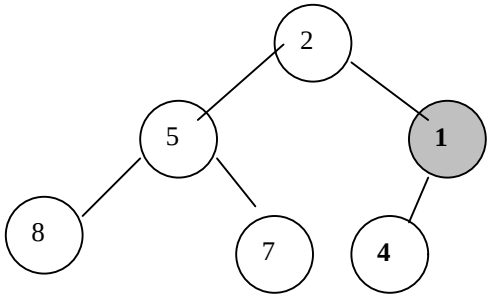
When we insert an item the heap order must be preserved. This is easy to do !. We always insert a new item at the end of the array. Since we have started our heap at **position** 1 the parent of the inserted item will always be **i / 2.** To preserve the heap property we must make sure the parent is less than the item to be inserted. If not we need to swap the parent with the new item. When we swap with the parent we need again to check if its parent is less. We continue this process. This is known as **percolating up**. As an example lets insert item 1 in our tree.

insert new item 1



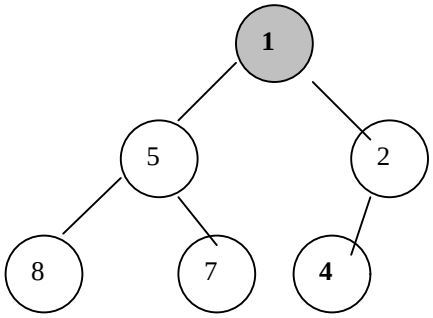
The parent of new item 1 is item 4. Item 4 is larger than the new item 1 so we must swap 1 and 4,

items 1 and 4 swapped



The parent of item 1 which is item 2 is larger so we must swap again.

items 1 and 2 swapped



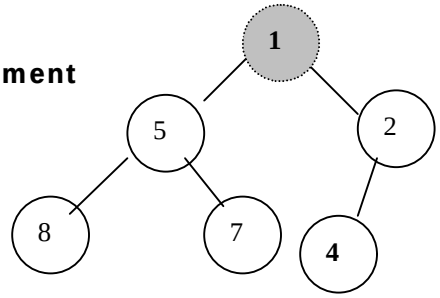
Now item 1 is at the top of the tree. Notice the heap is not necessary ordered but the heap property is preserved. Here's the code to insert an item into a heap.

```
void insert(int n, int items[], int* count))
    {
    int i = ++(*count);
   while(items[i/2] > n)
        {
        items[i] = items[i/2];
        i=i/2;
        }
    items[i] = n;
   }
```

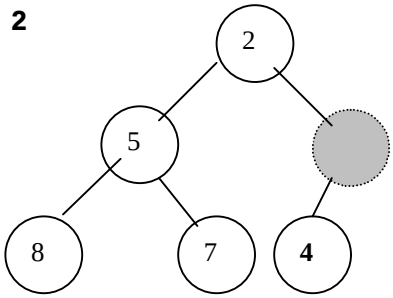
removing item from heap

We know where the smallest item is! It's at the top of the tree at array element position 1. We then must choose one of its children to be placed into the removed position. We continually move up all the children until the end of the heap. Here's the sequence of events.

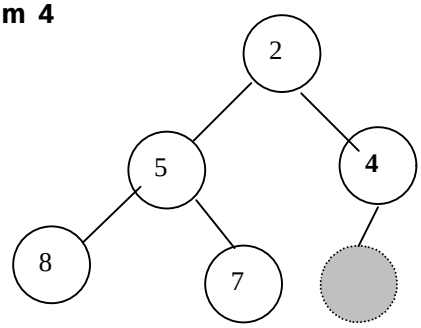
remove first element



percolate up item 2



percolate up item 4



We are done !

Here's the code for remove:

```
// remove with heap property
int remove(int items[], int* count))
{
// check for empty tree
if(count == 0)
{
printf("empty");
return items[0];
}
int min = items[1]; // minimum item
int last = items[(*count)--]; // last item
int child = 0;
int i ;
for(i=1;i*2 <= *count;i=child)
        {
        // find smaller child
        child=i * 2;
        if((child != (*count)) && (items[child+i] < items[child]))
        child++;
        // percolate up one level
        if(last > items[child])
        items[i] = items[child];
        else
        break;
        }
items[i] = last;
return min;
}
```

Here's our complete test program

```
// Heap.cpp
#include <iostream.h>

class Heap
{
private:
int* items;
int count;

public:
Heap(int size);
~Heap();
void insert(int n);
int remove();
friend ostream& operator<<(ostream& out, Heap& h);
};


Heap::Heap(int size)
{
count=0;
items = new int[100];

for(int i=0;i<100;i++)
items[i]=0;
}

Heap::~Heap()
{
delete items;
items=NULL;
count=0;
}

// insert an item
void Heap::insert(int n)
{
int i = ++(count);
while(items[i/2] > n)
        {
        items[i] = items[i/2];
        i=i/2;
        }
items[i] = n;
}
```

```
// remove an item
int Heap::remove()
{
int min = items[1];
int last = items[(count)--];
int child = 0;
int i = 0;

if(count == 0)
        {
        cout << "heap empty" << endl;
        return items[0];
        }

for(i=1; i*2 <= count; i=child)
        {
        // find smaller child
        child=i * 2;
        if((child != count) && (items[child+i] < items[child]))
        child++;
        // percolate one level
        if(last > items[child])
                items[i] = items[child];
        else break;
        }

items[i] = last;
return min;
}

// print out tree
ostream& operator++(ostream& out, Heap& h)
{
int i = 1;
int k = 1;
int j = 0;

while(i <= h.count)
        {
        for(j=0; j< (40 - i*2);j++)
                out << " ";

        for(j=i;(j<i+k) && (j<=h.count);j++)
                out << h.items[j];

        out << endl;
        i = i + k;
        k = k << 1;
        }
out << endl;
return out;
}
```

```
// test driver
void main()
{
Heap heap(100);
heap.insert(2);
heap.insert(5);
heap.insert(4);
heap.insert(8);
heap.insert(7);
cout << heap;
heap.insert(1);
cout << heap;
heap.remove();
cout << heap;
heap.remove();
cout << heap;
heap.remove();
cout << heap;
heap.remove();
cout << heap;
}
```

program output:

```
      2
    5  4
  8  7

      1
    5  2
  8  7  4

      2
    5  4
  8  7

      4
    5  7
  8

      5
    8  7

      7
    8
```

LESSON 9 EXERCISE 4

Make a search routine to search for items in a heap.

LESSON 9 EXERCISE 5

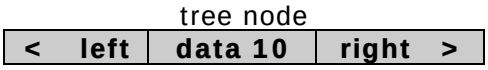
Use a heap to make a Priority Queue.

C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 10

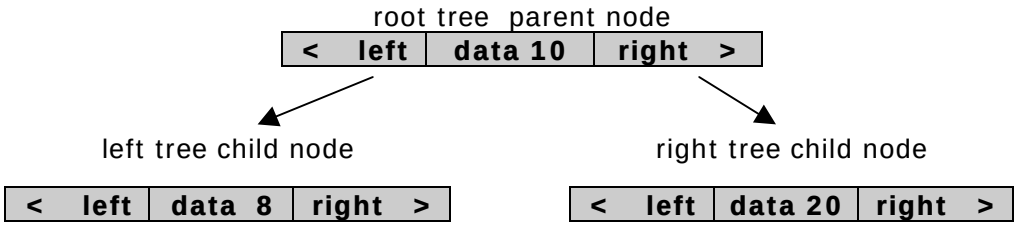
File:	CppdsGuideL10.doc
Date Started:	July 24,1998
Last Update:	Dec 22, 2001
Status:	proof

BINARY SEARCH TREES

A binary tree is a collection of binary tree nodes linked together to enable the user to choose between a smaller data value and a larger data value. A binary tree node is made up of a **left link**, **data element** and a **right link**:



Each left or right link points to another tree node. The left link is used to point to items that are **smaller** the tree node data you are at. The right link is used to point to items that are **larger** then the tree node data you are at. Trees do not have duplicate data.

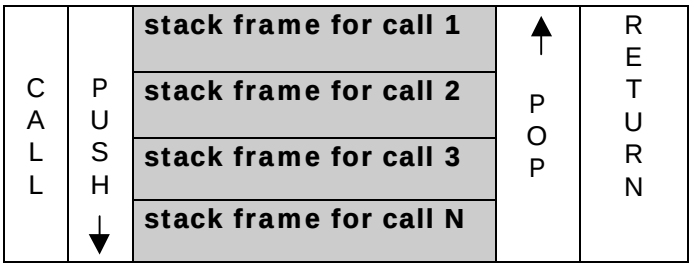


The node that the link points to is known as a **child**. The node that is pointing is known as the **parent**. Nodes that do not have links to other nodes are known as **leaves**. The first node in a tree is known as the **root**. With binary search trees are you can search much faster for items than by using link lists or arrays. The search or insertion time is O(log n). This is because you just have to search each level of the tree not every item. For example: if there are 16 items in a tree it will only take 4 tries to find the item.

$2^4 = 16$ $\log_{16} = 4$

Implement Trees with Recursion

Recursion is used to insert, delete, find and print nodes in a tree. Recursion is the easiest approach for binary trees. Recursion must be used because you need to keep track of all the parent nodes that you visit as you go through the tree. Recursion behaves like an automatic stack. As you travel through the tree nodes. the parent nodes are automatically pushed unto the stack, as it goes back it automatically pops the preceding parent nodes. Recursion is a term used when the function calls itself. Every time the function calls itself it save all its variables and return address on a stack frame. When the function returns it goes back to an upper stack frame. It now uses these variables on the stack frame. When the function returns and there us no more stack frames the function will return back to the calling function. Recursion can solve very many complicated programming problems with a few lines of code. The only draw back of Recursion is that it needs a large stack and can be very slow.



Each stack frame has a copy of all the variables of the function. Every time the function is called a new stack frame is created. Every time a function returns the function returns to a previous stack frame.

Tree Terminology

tree node	data structure having a left and right links and a data value
edge	connection between nodes (links)
binary tree	collection of tree nodes linked together in a predefined order
root	start of tree, first node in tree
parent	the node that points to a left or right child
children	node of left or right parent link
left child	smaller value than parent
right child	larger value than parent
interior node	node with children
leaves	nodes with no children at end of tree
siblings	nodes with same parent
path	sequence of nodes
traverse	follow a path
length	number of edges in path
depth	length of unique path from root
height	longest path from root to leaf
level	all nodes that have same depth
complete tree	all nodes on all levels are filled
balanced tree	all left sub trees are same level as all right sub trees A balanced tree does not have to be complete

Inserting items into binary tree example:
Insert the following numbers 10, 8, 5, 20, 15, 30. Insertion order is unpredictable, you will not get the same tree for every time if the order of numbers is changed.

Initially the tree pointer is NULL:

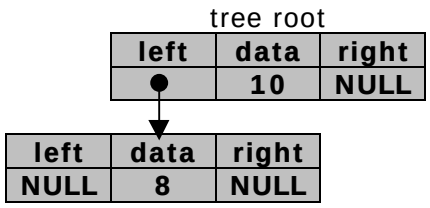
tree	NULL

insert 10 into tree:

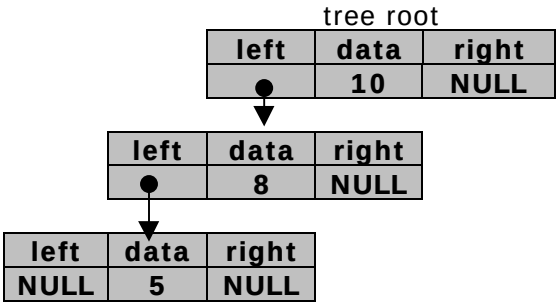
tree root

left	data	right
NULL	10	NULL

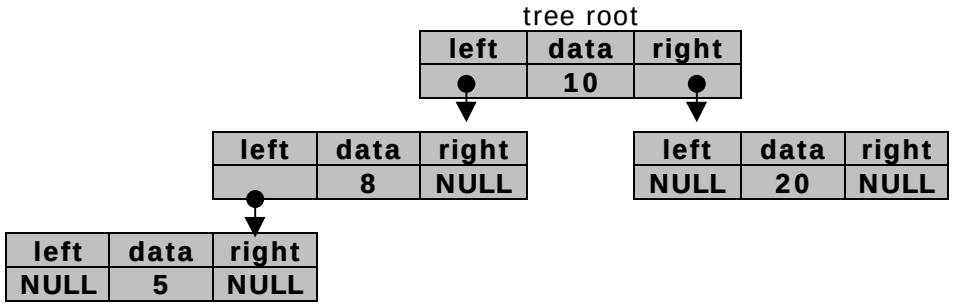
insert 8 into tree



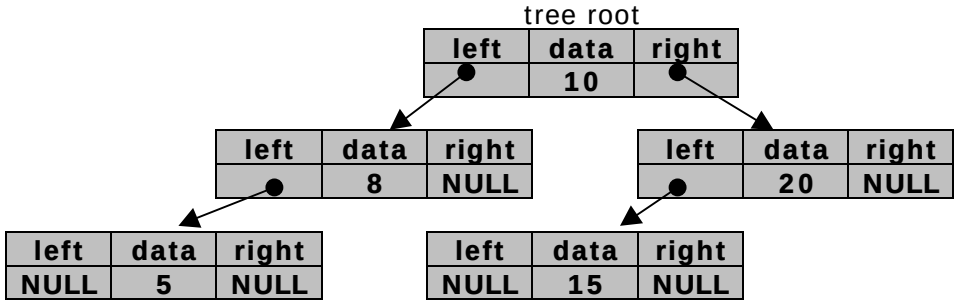
insert 5 into tree



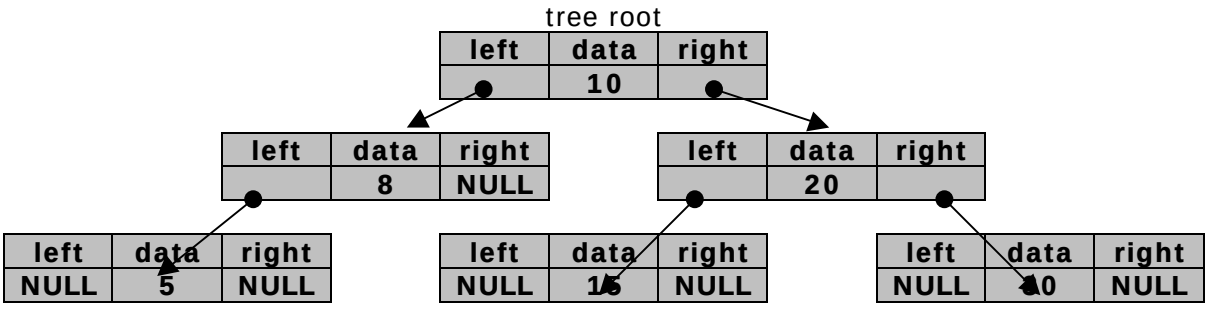
insert 20 into tree



insert 15 into tree



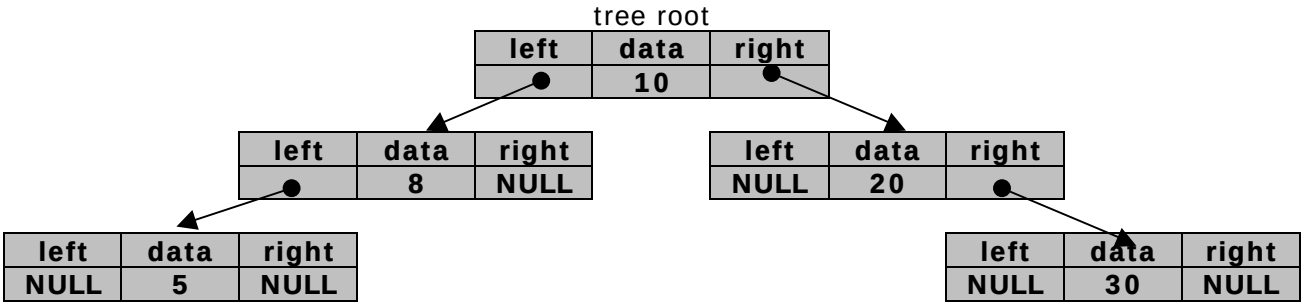
insert 30 into tree



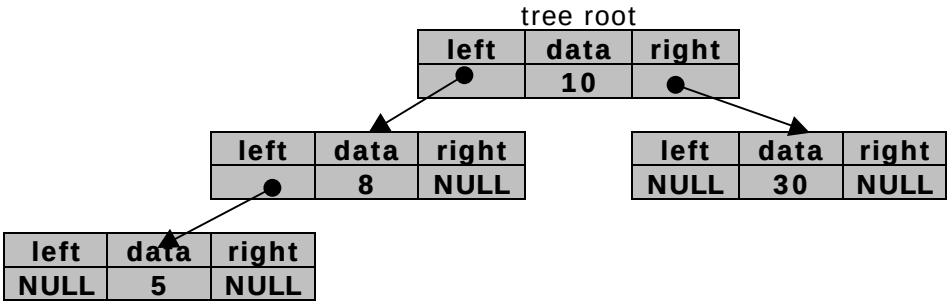
Removing items from binary tree example:

When an item is removed then the links to that item are set to Null. If the root node is deleted a new root node is assigned.

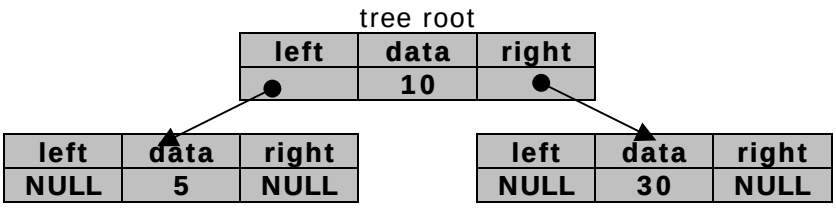
remove 15



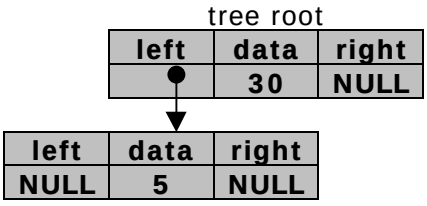
remove 20



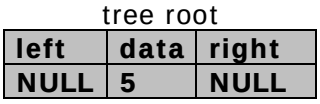
remove 8



remove 10

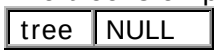


remove 30



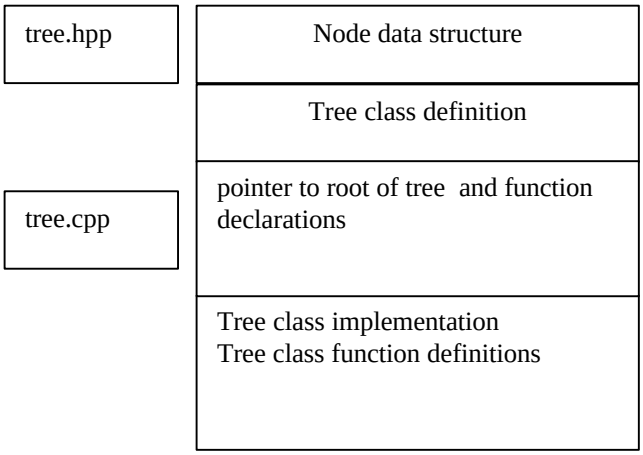
remove 5

The tree is empty



Implementing a Binary Tree

To implement a Binary Tree you need functions to insert, remove and search tree nodes. Each tree node will contain the data and a pointer to the next left and right tree nodes. To implement a Binary tree we only need a structure to represent a binary tree node. We will have A Tree class that has a pointer to a binary tree node to represent the root of the tree.



The Tree Node data structure holds the data and link to next node.

```
struct TNode
{
TNode* left; /* left child */
TNodeData data; /* data value */
TNode* right; /* right child */
};
```

The Tree class contains the member variables and function definitions and declarations.

BINARY TREE CODE

We have a TNode structure that is used to construct a TNode for the Tree class. This way the Tree class can use the TNode's variables. Recursion functions presents some problems when defined in a class. We have two choices we can put all recursive in a wrapper class or we can use the functions directly. Using functions directly is okay if they are called within another class. Since we call all our Tree class functions from the main() function we use wrapper functions for all our recursive functions. All the recursive functions are private and the wrapper classes public. Using wrapper functions is a good approach to encapsulation. The user of the Tree class should not need tom know what a tree root or TNode is, they just want to insert data and remove data transparently. We also have a wrapper function for printing out a binary tree we overload the **<<** operator. We put the Tree and TNode class definitions in the header file called "tree.hpp" and the Tree class implementation functions in the file "tree.cpp". Here's the binary tree code:

```
// tree.hpp
// tree node class definition
//#include "defs.h"
#ifndef __TREE
#define __TREE
#include <iostream.h
typedef int TNodeData;

        /* Tree Node structure */
        struct TNode
                {
                public:
                TNode* Left; // pointer to left child
                TNodeData Data; // tnode data value
                TNode* Right; // pointer to right child
                // constructor
                TNode(TNode* left,TNodeData data,TNode* right)
                        {
                        Left = left; // set left child
                        Data = data; // set tnode data value
                        Right = right; // set right child
                        }
                };
        // binary search tree class definition
        class Tree
                {
                private:
                TNode* Root;

                public:

                // constructor
                Tree()
                {
                Root=NULL;
                }
```

```
        // destructor
        ~Tree()
        {
        destroyTree(Root);
        Root=NULL;
        }

        private:
        // insert item into tree
        TNode* insert(TNode* tnode,TNodeData data);
        // remove data item from tree
        TNode* remove(TNode* tnode,TNodeData data);
        // find smallest element in tree
        TNode* findMin(TNode* tnode);
        // find data item in tree
        TNode* find(TNode* tnode,TNodeData data);
        // find data item in tree
        void print(TNode* tnode);
        // destroy tree
        void destroyTree(TNode* tnode);
        // operator wrapper functions
        public:
        // add item to tree
        void  insert(TNodeData data)
                {
                root = insert(root,data); // insert item into binary tree
                }
        // remove item from tree
        void remove(TNodeData data)
                {
                root = remove(root,data); // remove item from binary tree
                }
        // find item in tree
        TNode* find(TNodeData data)
                {
                return find(root,data); // find item in binary tree
                }
        // inline print tree wrapper
        friend ostream& operator<<(ostream& out ,Tree& tree)
                {
                out << "tree: ";
                tree.print(tree.Root); // print out binary tree
                return out;
                }
        };
#endif
```

```
// binary search tree code
// tree.cpp
// tree class code implementation
#include <iostream.h>
#include "tree.hpp"
```

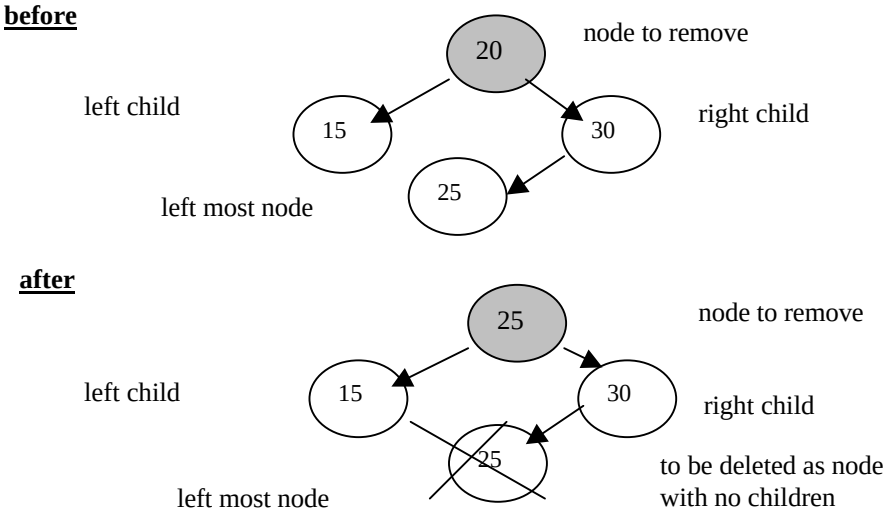
Inserting items into binary tree

We use recursion to insert an item into a binary tree. Recursion copies every link as it traverses through the tree. it traverses by **comparing** the data to be inserted with the data value at each node it encounters. If the value is **smaller** it traverse **left**, if the data value is **larger** it traverse **right**. When it comes to a leaf a new node is allocated and attached to the left or right of the leaf node.

```
// insert item into binary tree
// recursion will find the point to insert the new node
// it traverse the tree till it finds where to inset the new node

TNode* Tree::
insert(TNode* tnode,TNodeData data)
        {
        // check if tnode empty
        if(tnode == NULL)
                // make new tree node
                tnode = new TNode(NULL,data,NULL);
        // data smaller than tnode ?
        else if(data < tnode->Data)
                // traverse and copy left
                tnode->Left = insert(tnode->Left,data);
        // data larger than tnode ?
        else if(data > tnode->Data)
                // traverse and copy right
                tnode->Right = insert(tnode->Right,data);
        return tnode; // return current tnode
        }
```

Removing items from binary tree

We use recursion to remove an item from a binary tree, The function traverses throughout the tree looking for the node to delete. It traverses by **comparing** the data to be removed with the data value at each node it encounters. If the value is **smaller** it traverse **left**, if the data value is **larger** it traverse **right**. When the data to be removed is found two conditions must be considered: (1`) a node with two children (2) a node with one child. We will consider a node with two children first. We must find the smallest value from **right child** of the item to be deleted. The **findMin**() function is used to do this. It will find the smallest value by continuing the traversal left only. In a binary tree the smallest element is the most left node. Once the smallest node is found then the traversal is continued from this data value. We then copy these values from the smallest node into the node to be deleted. We continue traversing from the right child to delete the smallest value node located on the left most leaf. The item to be removed because e replace its data value with the smallest data value then continue to remove the left most node. We find the smallest node because we want to preserve the tree ordered property. This left most node will be deleted as a node with no children quite easily.



If a node has only one child then we keep a pointer to the opposite child link and delete the current node through a temporary pointer and return the child link. The child link may become the new root if the node that was removed was the tree root.

```
// removing item from binary tree
// case 1 : two children
// case 2 : one child

TNode* Tree::remove(TNode* tnode,TNodeData data)
	{
	TNode* temp; // a temp tnode pointer
	TNode* cnode; // a child tnode pointer

	// check if tnode empty
	if(tnode != NULL)
		{
		// find where to remove item
		// data smaller than current tnode ?
		if(data < tnode->Data)
			// traverse and copy left
			tnode->Left=remove(tnode->Left,data);
		// data larger than current tnode ?
		else if(data > tnode->Data)
			// traverses and copy right
			tnode->Right=remove(tnode->Right,data);
		// found data item
		else
			{
			// case two children
			if(tnode->Left && tnode->Right)
				{
				temp = findMin(tnode->Right); // find smallest item
				tnode->Data = temp->Data; // substitute for tnode
```

```
                        // traverse right
                        tnode->Right = remove(tnode->Right,tnode->Data);
                        }
                // case one child
                else
                        {
                        temp = tnode; // save current tnode
                        // store right child
                        if(tnode->Left == NULL) cnode = tnode->Right;

                        // store left child
                        if(tnode->Right == NULL) cnode = tnode->Left;

                        delete tnode; // delete current tnode
                        return cnode; // return child tnode
                        }
                }
        }
return tnode; // return current tnode
} // end removeTree
// findMin
// find smallest data item in binary tree
// the smallest element must be the most left item
TNode* Tree::findMin(TNode* tnode)
        {
        // if tnode not empty
        if(tnode != NULL)
                {
                // check if end of tree
                if(tnode->Left==NULL)return tnode;

                // search for min tnode
                else return (findMin(tnode->Left));
                }
        return tnode;
        }
```

Finding elements in binary tree

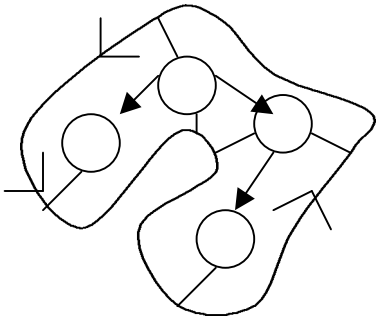
To find an item in a binary tree we just traverse through the tree. If the data item we are looking for is smaller than the current mode then we go left. If the data item is larger than the current node then we go right. we stop traversing when we find the item we are looking for.

```
// find data element in tree
// if found return position in tree
// if not found return error
TNode* Tree::
find(TNode* tnode, TNodeData data)
        {
        // check if tnode empty
        if(tnode != NULL)
                {
                // item smaller than current node ?
                if(data < tnode->Data)
```

```
        return (find(tnode->Left,data)); // search left
        // item larger than current node ?
        else if(data > tnode->Data)
        return (find(tnode->Right,data)); // search right
        }
return tnode; // return current tnode
}
```

Printing out a binary tree

\Recursion is used to print out a binary tree. A binary tree may be printed out **pre-order**, **in order** or **post order.** To traverse a tree we start at the root of the tree. Moving counterclockwise we walk around the tree passing every node. As you traverse the tree pretend you are touching or hugging the tree.



As you traverse a tree a node can be touched more than once

order	explanation	example						comment
pre-order	a node is printed the **first time** it is passed	10	8	5	20	15	30	prints left side then right side of tree
in-order	we print a **leaf** the **first time** we pass it we print an **interior node** the **second time** we pass it	8	5	10	15	20	30	all nodes in ascending order
post-order	a node is printed the **last time** we pass it	5	8	15	30	20	10	prints all tree nodes bottom up

The print routine can be changed for pre-order, inorder or post-order just by the position of the cout statement.

order	code	position of print statement

pre-order	cout << tnode->Data;
	printTree(tnode->Left);
	printTree(tnode->Right);
		top

in-order	printTree(tnode->Left);
	cout << tnode->Data;
	printTree(tnode->Right);
		middle

post-order	printTree(tnode->Left);
	printTree(tnode->Right);
	cout << tnode->Data;
		bottom

```
// print a tree
// print tree data elements in order using recursion
// function calls itself until last node reached
void Tree::print(TNode* tnode)
        {
        // check if tnode empty
        if(tnode != NULL)
                {
                print(tnode->Left); // traverse left
                cout << tnode->Data << " "; // print tnode data value
                print(tnode->Right); // traverse right
                }
        }

// called by destructor
/* using postfix we deallocate all memory for tree  recursively */
void Tree::destroyTree(TNode* tnode)
{
if(tnode!=NULL)
     {
     destroyTree(tnode->Left);
     destroyTree(tnode->Right);
     delete(tnode);
      }
}
// main function
int main()
        {
        Tree tree; // make a tree object
        tree.insert(10); // add item 10 to binary tree and print tree
        cout << endl << "tree: ";
        cout << tree;
        tree.insert(8);// add item 8 to binary tree and print tree
        cout << endl << "tree: ";
        cout << tree;
        tree.insert(5); // add item 5 and 20 to binary tree and print tree
        tree.insert(20);
        cout << endl << "tree: ";
        cout << tree;
        if(tree.find(8))
        cout << "\n found tree item 8";
        if(tree.find(15))
        cout << "\n found tree item 10";
        tree.insert(15); // add item 55 and 30 to binary tree and print tree
        tree.insert(30);
        cout << endl << "tree: ";
        cout << tree;
        tree.remove(15); // remove item 15 from binary tree and print tree
        cout << endl << "tree: ";
        cout << tree;
        tree.remove(20); // remove item 20 and 8 from binary tree and print treetree-8;
        cout << endl << "tree: ";
```

```
cout << tree;
tree.remove(10); // remove item 10 from binary tree and print tree
cout << endl << "tree: ";
cout << tree;
tree.remove(30); // remove item 30 from binary tree and print tree
cout << endl << "tree: ";
cout << tree;
tree.remove(5); // remove item 5 from binary tree and print tree
cout << endl << "tree: ";
cout << tree;
return 0;
}
```

tree program output:

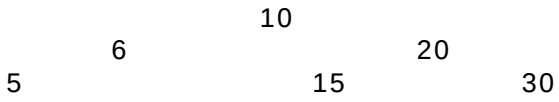
```
tree: tree: 10
tree: tree: 8 10
tree: tree: 5 8 10 20
found tree item 8
tree: tree: 5 8 10 15 20 30
tree: tree: 5 8 10 20 30
tree: tree: 5 8 10 30
tree: tree: 5 8 30
tree: tree: 5 8
tree: tree: 8
```

Lesson 10 Exercise 1

Trace through the insertTree, removeTree and printTree routines, write down on paper the recursion sequence, Do this many times till you understand how every thing works.

Lesson 10 Exercise 2

Write a print tree routine that will print out the tree as it appears level by level . For example the output for our example tree will be: Hint use a Queue.



Lesson 10 Exercise 3

Write a function that will print out all the paths found in a tree and its length. For example for Exercise 4 the tree paths would be

10->6->4 length 3	10->20->15 length 3	10->20->30 length 3

Lesson 10 Exercise 4

Write a function that will print out all the number of nodes in a tree

Lesson 10 Exercise 5

Write a function that will print out all the number of leafs in a tree

Lesson 10 Exercise 6

Write a function that will print out all the number of interior nodes in a tree

Lesson 10 Exercise 7

Write a function that will print out all the number of full nodes in a tree, nodes that have both left and right children

Lesson 10 Exercise 8

Write a function that will print out all the number of left child nodes in a tree

Lesson 10 Exercise 9

Write a function that will print out all the number of right child nodes in a tree

Lesson 10 Exercise 10

Write a function that will print out all the balance between the left and right nodes

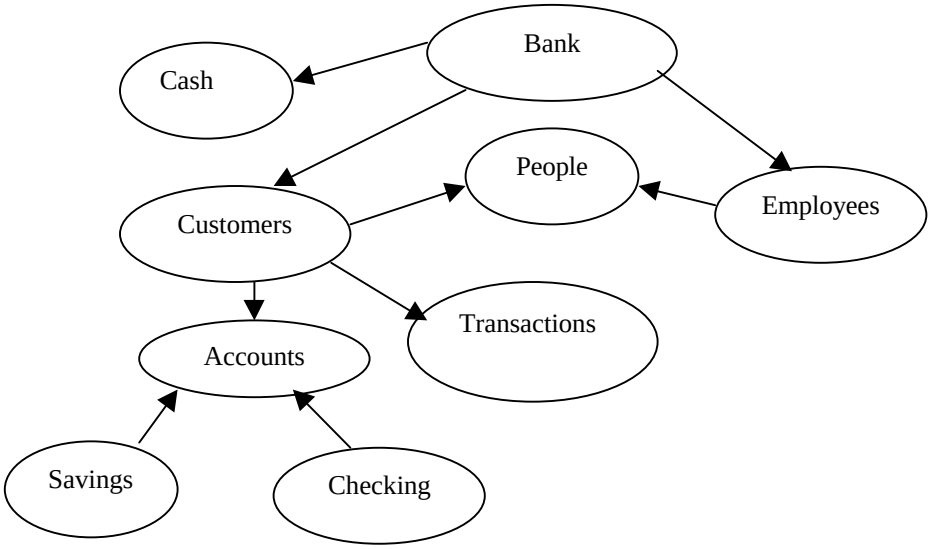
C DATA STRUCTURES PROGRAMMERS GUIDE PROJECT

File:	CdsGuideL11.doc
Date Started:	July 24,1998
Last Update:	Dec 21,2001
Status:	proof

PROJECTS = WORK = LEARNING = FUN

INTRODUCTION

We will implement a Bank using many classes. A Bank is a very good example of object oriented programming. A bank has customers, employees, accounts, transactions and cash. Customers and Employees are people. Customers have accounts and transactions. Accounts may be Checking or Savings. Transactions identify the customer account operations as deposit, withdraw or transferring amounts between accounts. A bank makes money from charging customers fees for banking services. Banks must also pay interest to customers and salaries to employees. Each component of a bank will be represented by a class. Classes may be sub class of other class. Class may also use other class. An up ↑ arrows means a class is a sub class. A down ↓ arrow means a class will use another class as a variable. The following is our bank programming model.



SPECIFICATIONS

Before you can write any program you need to have **specifications**. A specification describes what the program is supposed to do and what components, a program is suppose to have. Specifications usually describe the top level of a program first. A program can be broken down into many sub components. When using object oriented programming each sub component can be a class. Object oriented programming is ideal to break down large complicated programming problem into smaller manageable tasks. We can now write the specification from the real-life components of a real bank. We write our specification top down which means we describe all the top levels first and then proceed down to describe the inner levels.

Bank class

A bank must have Customers, Employees. Accounts and Cash. Customers and Employees are People, Accounts represent the customers deposits cash represents what money the bank is making. Banks make money from charging customers fees for banking services. The bank must pay money to Employees and pay interest to Customers. Both customers and employees will be represented by separate binary search trees. The search key will be the customer's last name. In case of identical last name the sort process will include middle initial and first name. The accounts will be a hash table using the account number has the search key. The hash table must be implemented as an expandable hash table.

member	data type	base class	description
customers	Tree of Customer*	Person	represents an Binary Search Tree of pointers to Customer objects
maxCustomers	int		maximum number of customers
numCustomers	int		number of customers in array
employees	Tree of Employee*	Person	represents an Binary Search Tree of pointers to Employee objects
maxEmployees	int		maximum number of employees
numEmployees	int		number of employees in array
cash	float		how much money the bank has
accounts	Hash table of Account*		represents an Hash Table of pointers to Account objects

Customers and Employees will inherit the Person class. We use Binary Tree to represent the Customers and Employees for fast lookup. We use a hash table for Accounts because we also need to locate an account quickly. You can use any algorithm for the hash key function. We need an expandable hash table because as we need more customers the hash table will double in size frequently or even vice-vera. Collisions are store in a link list sorted by account number.

Bank class Operations

A bank needs to perform operations, each implemented by the following functions: Functions used by other classes should be **public**. Functions just used by this class should be **private**.

function prototype	description
Customer();	default constructor
bool addCustomer(Customer *);	add new customer
bool deleteCustomer(Customer *);	remove customer
Customer* searchCustomer (String *lastName);	search for customer by last name
Customer* searchCustomer (long accountNumber);	search for customer by account number
bool makeTransaction();	customer deposits, withdraws or transfers money between accounts.
Customer* update();	pay monthly interest to all accounts, collect all service fees
bool addEmployee(Employee *);	add new employees
bool deleteEmployee(Employee *);	remove employees
Customer* searchEmployee(String *lastName);	search for employee by last name
bool payEmployees();	pay employees
bool addCash(float amount);	add amount to cash

Person class

A Person represents a Customer or Employees name, address, phone and SIN, all represented by String objects. You can use the String class from this course or the one supplied by your compiler. . All member variables in the class should be **private**, and only if necessary **protected**.

member	data type	description
firstName	String	First name
middle Initial	String	middle initial
lastName	String	last name
street	String	street number and street name
apt	String	apt or suite number
city	String	city
state	String	state or province
postalCode	String	postal code
phone	Sting	home phone
SIN	String	social insurance number

Person class operations

For this class you just need a default constructor, a **getName()** function and overloaded operated friend function >> and << to get and print out the person information. The **getName()** function must combine the first, initial and last name into one string. The operator >> function will be used to get customer information from the keyboard or file. The operator << function will be used to print out customer information to the screen or file. All functions are **public**.

function prototype	description
Person();	default constructor
String& getName();	add new customer
friend istream& operator>>(istream& in, Person& p);	get person info
friend ostream& operator<<(ostream& in, Person& p);	print person info

Customer class

The Customer class inherits the Person class. You just need a few more things for the customer class. like a pointer to the bank object, an Link List of pointers to Accounts objects and an Double Link List of pointers to Transactions objects. We use a single link list for the accounts because we do not have too many. We use a Double Link list of transactions because we want to display resent first or last first. A double link list will let us view transactions forward or backward. All member variables in the class should be **private**.

member	data type	description
bank	Bank*	pointer to bank object
business phone	String	work phone number
accounts	LinkList of Account* objects	an link list of pointers to account objects
maxAccounts	int	max number of Accounts
numAccounts	int	number of accounts in account array
transactions	Double Link List of Transaction* Objects	double link list of pointers to Transaction object
maxTransactions	int	maximum number of transactions
numTransactions	int	number of transactions if Transaction array

Customer class operations

You need a default constructor , the constructor must also call the Persons base class default constructor. You need an operator == function to compare two customer objects by search key. You will also need functions to add, delete and view accounts, add and view transactions. You will need an overloaded operator friend function's >> and << to get and print out the Customer information. The Customer class constructor must also call the Persons base class default constructor. Functions to be used by other classes should be **public**. Functions only used by this class should be **private**.

function prototype	description
Customer();	default constructor
bool operator = =(Customer& customer);	compare two customer objects by full name
bool addAccount(Account* account)	add new customer account
bool viewAccount(long number);	view customer account
bool deleteAccount(Account* account)	delete customer account
bool addTransaction (Account* account)	add new customer account
bool viewTransactions(long number);	search for customer account
friend istream& operator>>(istream& in, Customer& p);	get customer info
friend ostream& operator<<(ostream& in, Customer& p);	print customer info

Employee class

The Employee class inherits the Person class. You just need a few more things for the Employee class like a pointer to the bank object, salary and total salary paid so far. All member variables in the class should be **private**.

member	data type	description
bank	Bank*	pointer to bank object
salary	float	employees yearly salary amount
totalPaid	float	how much salary paid so far

Employee class operations

You need a default constructor, the constructor must also call the Persons base class default constructor. You need an operator == function to compare two employee objects by search key. You need a function called **pay()** to pay the employees. You will need an overloaded operator friend function's >> and << to get and print out the Employee information.

function prototype	description
Employee();	default constructor
bool operator = =(Employee& customer);	compare two employee objects by full name
void pay();	pay this employee
friend istream& operator>>(istream& in, Employee& p);	get employee info
friend ostream& operator<<(ostream& in, Employee& p);	print employee info

Account class

The account class contains all the common information about an account like Account number, pointer to Customer, balance and service fees. All member variables in the class should be **private**, and only if necessary **protected**.

member	data type	description
number	long	account number
customer	Customer*	pointer to customer object
balance	float	money in account
serviceFee	float	monthly service fee

Account class operations

You will need an initializing constructor that accepts a pointer to a Customer object, service fee and an initial balance. The account number will be internally generated by the derived classes. You need an operator == function to compare two Account objects by search key. You will also need **pure virtual** functions **update(). withdraw()**, **deposit()** and **transfer().** The **update()** function will apply all interest to account balance and pay the bank for service fees. You will need an overloaded operator friend function's >> and << to get and print out the Account information. Functions to be used by other classes should be **public.** Functions only used by this class should be **private**.

function prototype	description
Account(Customer* customer, float serviceFee, float balance);	initializing constructor
bool operator = =(Account& customer);	compare two account objects by account number
virtual void deposit()=0;	deposit funds into account
virtual void withdraw()=0;	withdraw funds from this account
virtual void transfer()=0;	funds from this account to another
virtual void update()=0;	update this account, pay all interest, collect all service fees
friend istream& operator>>(istream& in, Account acc);	get account info
friend ostream& operator<<(ostream& in, Account& acc);	print account info

Checking class

The Checking class inherits the Account class. A checking account has an overdraft but do not give interest. There is a base monthly service fee but they also charge 1 dollar for every bank transaction. There is also a static variable to assign new checking account numbers. All member variables in the class should be **private**.

member	data type	description
overdraft	float	Overdraft amount. if no over draft set to 0
transactionFee	float	charge for any transaction
nextNumber	static long	next account number starting from 1000000

Checking class operation

The initializing constructor must also receive the overdraft and transaction fee and call the account base class initializing constructor. It is inside the Checking constructor where the account number is assigned and incremented. The Checking class must also implement the **update(), withdraw()** and **deposit()** and **transfer()** functions. The **update()** function will pay the bank for the monthly service fees. For every transaction you must add the transaction fee to the banks cash. You will need an overloaded operator friend function's >> and << to get and print out the Checking information. Functions to be used by other classes should be **public.** Functions only used by this class should be **private.**

function prototype	description
Checking(Customer* customer, float serviceFee, float balance);	initializing constructor
void deposit();	deposit funds into account
void withdraw();	withdraw funds from this account
void transfer();	funds from this account to another
void update();	update this account, pay all interest, collect all service fees

Savings class

The Savings class inherits the Account class. The savings account gives interest but no overdraft protection. There is a base monthly service fee. There is also a static variable to assign new checking account numbers. All member variables in this class should be made **private**.

member	data type	description
interest	float	paid monthly interest
nextNumber	static long	next account number starting from 2000000

A Savings class operation

The initializing constructor must also receive the overdraft and interest rate and call the Account base class initializing constructor. It is Inside the Savings constructor where the account number is assigned and incremented. The savings class must also implement the **update()**, **withdraw()**, deposit() and **transfer()** functions. The **update()** function will pay the bank for the monthly service fees and pay the customer interest on the balance. You could use a monthly service fee of $15 and a monthly interest rate of .005 % (6% per annum). You will need an overloaded operator friend function's >> and << to get and print out the Savings information. Functions to be used by other classes should be made **public.** Functions only used by this class should be **private.**

function prototype	description
Savings(Customer* customer, float serviceFee, float balance, float transactionFee);	initializing constructor
void deposit();	deposit funds into account
void withdraw();	withdraw funds from this account
void transfer();	funds from this account to another
void update();	update this account, pay all interest, collect all service fees

Transaction class

We must keep track of the transactions that each customer makes. A transaction class will have the following members. All member variables in this class should be made **private**,

member	data type	description
type	enum	Withdraw , Deposit, Transfer, ServiceFee, InterestPayment, TransactionFee
fromAccount	String	from account number
toAccount	String	to account number
amount	float	amount to deposit or withdraw

Transaction class operation

You will need an initializing Constructor to initialize transaction type, accounts and amount. All Transaction objects are added to the Customer object the account resides in. In cases of deposit and withdraw the account from/to parameters can be passed codes to identify checkin, cashin, cashout etc. Use an **enum** for the from/to values. You will need an overloaded operator friend function's >> and << to get and print out the Transaction information. . Functions to be used by other classes should be made **public**. Functions only used by this class should be **public**.

function prototype	description
Transaction(TransactionType type, long from, long to, float balance, float serviceFee, float balance, float transactionFee);	initializing constructor
friend istream& operator>>(istream& in, Transaction& t);	get accountinfo
friend ostream& operator<<(ostream& in, Transaction& t);	print account info

USER INTERFACE

You will need a menu for the bank employee for bank operations like this:

```
Welcome to Super Bank
==================
(1) Add new Customer
(2) View Customer Accounts
(3) Make a Transaction
(4) Pay Interest*
(5) Add Employee
(6) View Employees
(7) Pay Employee
(8) View Bank Info
(9) Go home for the day
```

Each menu selection is described as follows:

(1) Add /Delete/View new Customer	add or delete customer
(2) View Customer Accounts	view all customer accounts
(3) Make a Transaction	customers make a withdraw, deposit or transfer
(4) Update Accounts	Pay monthly interest to all account. apply service charge to all accounts
(5)Add/delete/View Employee	add a new employee or delete an existing employee
(6) View Employees	view all employees
(7) Pay Employee	pay employees there weekly salary
(8) View Bank Info	view all bank information

Each menu selection will call sun menu operations as follows:

Add/Delete/View customer

(1)	ask for customer account number or **N** for new and **D** for delete
(2)	if new customer get all customer information and the account they want to open
(3)	if existing customer show information, accounts and transactions

Make transaction

(1)	Get customer account by account number or by name. If no account tell this person they are at the wrong bank
(2)	Ask if Deposit Withdraw or Transfer. Inform customer if they cannot withdraw amount
(3)	ask for amount, this must be a positive number

Update customers

(1)	Apply interest and monthly service charge to all customer accounts. The service charges will be added to the bank's cash, the interest payments will come from the bank cash.

Add/ Delete/View employee

(1)	ask for employee name or **N** for new and **D** for delete
(2)	if new employee get all employee information and salary
(3)	if existing employee show information

Pay employees

(1)	Pay employees their weekly wage. The money must come from the banks cash

class summary

You will have the following classes. Put the **main()** function in the Bank class.

header files:	**implementation files:**
Bank.hpp	Bank.cpp
Person.hpp	Person.cpp
Customer.hpp	Customer.cpp
Employee.hpp	Employee.cpp
Account.hpp	Account.cpp
Savings.hpp	Savings.cpp
Checking.hpp	Checking.cpp
Transaction.hpp	Transaction.cpp

main function

The main function can include the menu and call functions from the bank class to process the requested bank operation. You may want to put a **run()** function in the Bank class that contains the menu selection and calls the Bank class functions to do the requested operations.

Hints on completing Project:

(1) Sketch an operational model of all classes showing how all the classes interact.
(2) Write all the class definitions first top down. Start at the top of the class model and proceed to the bottom class Write the Base class definitions first.
(3) Write each class implementation one at a time bottom up. Start at the bottom class of the class model and then finish up at the top. Always write the base classes before writing the derived classes.

(4) Write your comments so that if someone just reads the comments they know how the program works.
(5) Use the **new** operator to allocate memory, and the **delete** operator to delete any memory allocated with new.
(6) Avoid **circular references** when using the **#include** directive. Circular #include references arise when one hpp file includes another hpp file and this hpp file includes the one that called it. The only way to avoid tins situation is to use a **forward reference.** For example the **bank.hpp** file needs to include **customer.hpp** file. The **customer.hpp** file needs to include **bank.hpp** file. The result is a circular #include reference. The only way out of this dilemma is to use a **forward reference.**

In the account class just say:

```
class Bank* bank;
```

This will solve all your problems. Now the compiler knows that Bank is a pointer to a Bank object.
(7) calling overloaded operator functions
Each class will have the same operator functions so it will be difficult to select the one you want. A solution is to type cast.

```
cin >> customer;
```

If you just want to print out the person part you need to type cast.

```
cin >> (Person&)customer;
```

(Don't forget to pass by reference).
When you have a pointer you need to type cast to the target pass type.

```
cout << *customer;
cout << (Person&)(*customer)
```

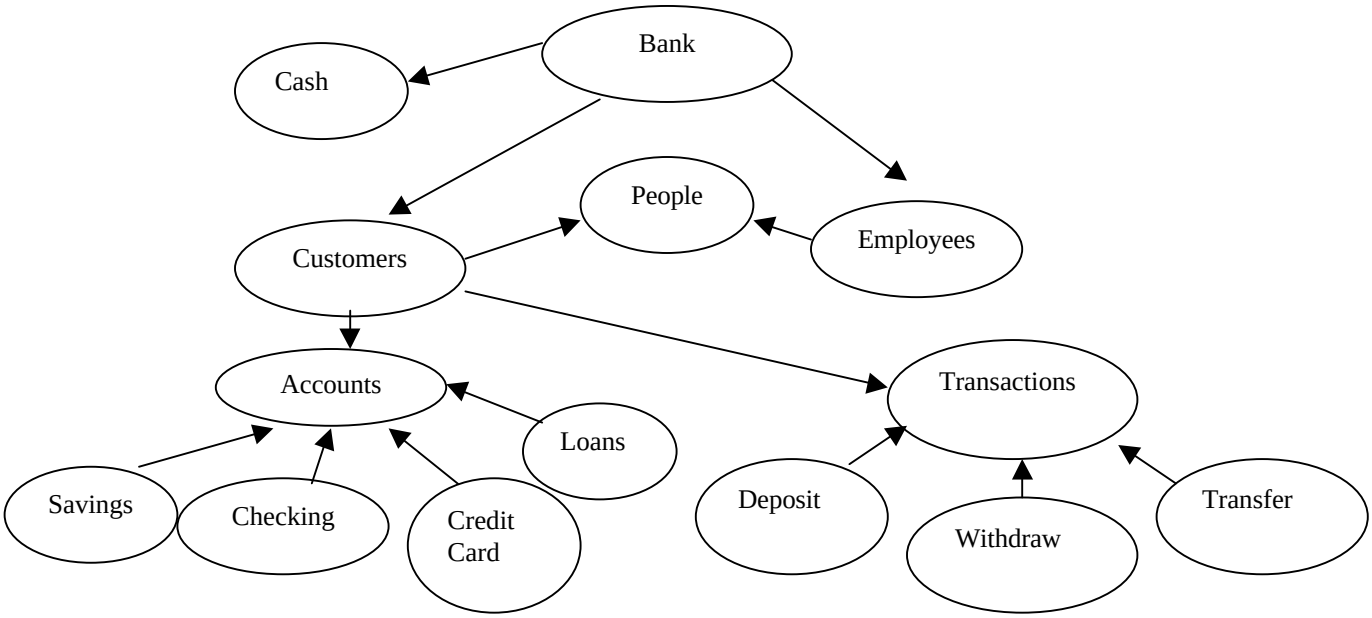
We type cast as a reference because we want to pass a reference to the **memory address** the pointer is pointing to.

Marking Scheme:

CONDITION	RESULT	GRADE
Program crashes	Retry	**R**
Program works	Pass	**P**
Program is impressive	Good	**G**
program is ingenious	Excellent	**E**

ENHANCEMENTS

(1) Write all customer accounts, employees and transaction objects to a file. When your program starts up it will read the file and update the arrays used to store our objects. Call your file bank.dat.
(2) Add a credit card account.
(3) And a bank loan account. Have different types of bank loans: Personnel, RRSP and mortgage.
(4) Print out monthly bank statements for each customer account.
(5) Add Transaction derived classes. Deposit, Withdraw, and Transfer.

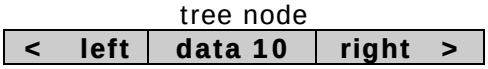


C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 12

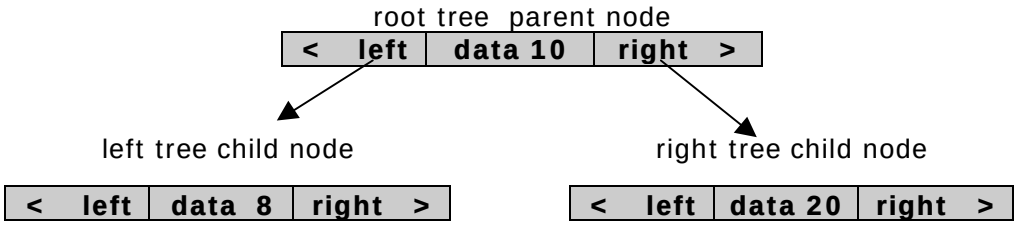
File:	CppdsGuideL12.doc
Date Started:	July 24,1998
Last Update:	Dec 27, 2001
Status:	proof

LESSON 12 BINARY SEARCH TREE RECURSION ROUTINES

Recursion is used to solve a lot of operations on binary trees. From our Binary tree lesson you were told a binary tree consists of a binary tree node that has a left link, data element and a right link:



Each left or right link points to another tree node. The left link is used to point to items that are **smaller** the tree node data you are at. The right link is used to point to items that are **larger** then the tree node data you are at. Trees do not have duplicate data.



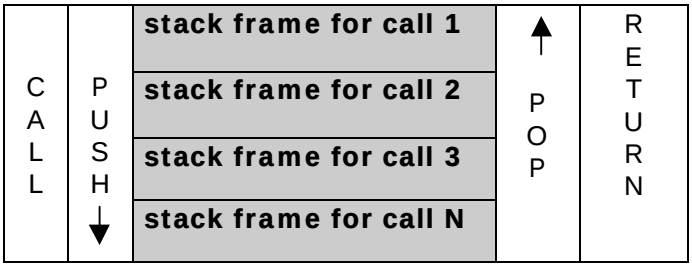
The node that the link points to is known as a **child**. The node that is pointing is known as the **parent**. Nodes that do not have links to other nodes are known as **leaves**. The first node in a tree is known as the **root**. We have collected additional binary tree functions for your Tree class:

function	purpose
TNode* insert(TNodeData data);	insert item into tree
TNode* remove(TNodeData data);	remove data item from tree
TNode* findMin(TNodePtr tnode);	find smallest element in tree
TNode* find(TNodeData data);	find node in tree
int numNodes(TNode* tree);	calculate number of nodes in a tree
int numLeaves(TNode* tree);	calculate number of leaves in a tree
int numFullNodes(TNode* tree);	calculate number of full nodes in a tree
bool hasNeg(TNode* tree);	return true if tree has a negative value node
bool hasPos(TNode* tree);	return true if tree has a positive value node
bool hasValue(TNode* tree, TNodeData data);	return true if data value in tree
int maxValue(TNode* tree);	return maximum value of tree
int minValue(TNode* tree);	return minimum value of tree
int sumTree(TNode* tree);	return sum of all nodes in tree
int maxDepth(TNode* tree);	return max depth of tree
bool isBalanced(TNode* tree);	return true if tree balanced
bool isComplete(TNode* tree);	return true if tree complete
void printPreorder(TNode tree);	print tree in pre order
void printInorder(TNode tree);	print tree in order

void printPostorder(TNode tree);	print tree in post order
void printTree(TNode* tree);	print tree by indentation
void printLevel(TNode* tree);	print tree by level
void destroyTree(TNode* tree);	de-allocate memory for tree

Implement Trees with Recursion

Recursion is used to insert, delete, find and print nodes in a tree. Recursion is the easiest approach for binary trees. Recursion must be used because you need to keep track of all the parent nodes that you visit as you go through the tree. Recursion behaves like an automatic stack. As you travel through the tree nodes. the parent nodes are automatically pushed unto the stack, as it goes back it automatically pops the preceding parent nodes. Recursion is a term used when the function calls itself. Every time the function calls itself it save all its variables and return address on a stack frame. When the function returns it goes back to an upper stack frame. It now uses these variables on the stack frame. When the function returns and there us no more stack frames the function will return back to the calling function. Recursion can solve very many complicated programming problems with a few lines of code. The only draw back of Recursion is that it needs a large stack and very slow.



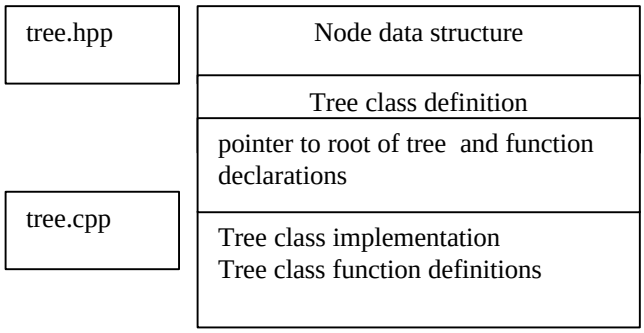
Each stack frame has a copy of all the variables of the function. Every time the function is called a new stack frame is created. Every time a function returns the function returns to a previous stack frame.

Tree Terminology

tree node	data structure having a left and right links and a data value
edge	connection between nodes (links)
binary tree	collection of tree nodes linked together in a predefined order
root	start of tree, first node in tree
parent	the node that points to a left or right child
children	node of left or right parent link
left child	smaller value than parent
right child	larger value than parent
interior node	node with children
leaves	nodes with no children at end of tree
siblings	nodes with same parent
path	sequence of nodes
traverse	follow a path
length	number of edges in path
depth	length of unique path from root
height	longest path from root to leaf
level	all nodes that have same depth
complete tree	all nodes on all levels are filled
balanced tree	all left sub trees are same level as all right sub trees A balanced tree does not have to be complete

Implementing a Binary Tree

To implement a Binary Tree you need functions to insert, remove and search tree nodes. Each tree node will contain the data and a pointer to the next left and right tree nodes. To implement a Binary tree we only need a structure to represent a binary tree node. We will have A Tree class that has a pointer to a binary tree node to represent the root of the tree.



The Tree Node data structure holds the data and link to next node.

```
struct TNode
{
TNode* Left; /* left child */
TNodeData Data; /* data value */
TNode* Right; /* right child */
};
```

The Tree class contains the member variables and function definitions and declarations.

BINARY TREE CODE

We have a TNode structure that is used to construct a TNode for the Tree class. This way the Tree class can use the TNode's variables. Recursion functions presents some problems when defined in a class. We have two choices we can put all recursive in a wrapper class or we can use the functions directly. Using functions directly is okay if they are called within another class. Since we call all our Tree class functions from the main() function we use wrapper functions for all our recursive functions. All the recursive functions are private and the wrapper classes public. Using wrapper functions is a good approach to encapsulation. The user of the Tree class should not need tom know what a tree root or TNode is, they just want to insert data and remove data transparently. We also have a wrapper function for printing out a binary tree we overload the **<<** operator. We put the Tree and TNode class definitions in the header file called "tree.hpp" and the Tree class implementation functions in the file "tree.cpp".Here's the binary tree code:

```
// tree.hpp
// tree node class definition
//#include "defs.h"
#ifndef __TREE
#define __TREE
#include <iostream.h>
typedef int TNodeData;

        /* Tree Node structure */
        struct TNode
                {
                public:
                TNode* Left; // pointer to left child
                TNodeData Data; // tnode data value
                TNode* Right; // pointer to right child
```

```
        / constructor
        TNode(TNode* left,TNodeData data,TNode* right)
                {
                Left = left; // set left child
                Data = data; // set tnode data value
                Right = right; // set right child
                }
        };
// binary search tree class definition
class Tree
        {
        private:
        TNode* Root;

        public:

        // constructor
        Tree()
        {
        Root=NULL;
        }

        // destructor
        ~Tree()
        {
        destroyTree(Root);
        Root=NULL;
        }

        private:

        TNode* insert(TNode* tnode,TNodeData data); // insert item into tree
        TNode* remove(TNode* tnode,TNodeData data); // remove data item from tree
        TNode* findMin(TNode* tnode); // find smallest element in tree
        TNode* find(TNode* tnode,TNodeData data); // find data item in tree
        void print(TNode* tnode); // print out tree
        void destroyTree(TNode* tnode); // destroy tree
        int numNodes(TNode* tree);        //calculate number of nodes in a  tree
        int numLeaves(TNode* tree);       //calculate number of leaves in a  tree
        int numFullNodes(TNode* tree);    //calculate number of full nodes in a  tree
        bool hasNeg(TNode* tree);  //return true if tree has  a negative value node
        bool hasPos(TNode* tree);  //return true  if tree has a positive value node
        bool hasValue(TNode* tree, TNodeData data);     //return true if data value in tree
        int maxValue(TNode* tree); //return maximum value of tree
        int minValue(TNode* tree);  //return minimum value of tree
        int sumTree(TNode* tree);  //return sum of all nodes in tree
        int maxDepth(TNode* tree);   //return max depth of tree
        bool isBalanced(TNode* tree);       //return true if tree balanced
        bool isComplete(TNode* tree);       //return true if tree complete
        void printPreorder(TNode* tree);    //print tree in pre order
        void printInorder(TNode* tree);     //print tree in order
        void printPostorder(TNode* tree);  //print tree in post order
```

```
void printTree(TNode* tree, int level);      //print tree by indentation
void printLevel(TNode* tree);        //print tree by level
// operator wrapper functions
public:
// add item to tree
void  insert(TNodeData data)
{
Root = insert(Root,data); // insert item into binary tree
}
// remove item from tree
void remove(TNodeData data)
{
Root = remove(Root,data); // remove item from binary tree
}
// find item in tree
TNode* find(TNodeData data)
{
return find(Root,data); // find item in binary tree
}
// inline print tree wrapper
friend ostream& operator<<(ostream& out ,Tree& tree)
{
out << "tree: ";
tree.print(tree.Root); // print out binary tree
return out;
}
int numNodes()       // calculate number of nodes in a  tree
{
return numNodes(Root);
}
int numLeaves()      // calculate number of leaves in a  tree
{
return numLeaves(Root);
}
int numFullNodes()   // calculate number of full nodes in a  tree
{
return numFullNodes(Root);
}
bool hasNeg()// return true if tree has  a negative value node
{
return hasNeg(Root);
}
bool hasPos() // return true  if tree has a positive value node
{
return hasPos(Root);
}
bool hasValue(TNodeData data)     // return true if data value in tree
{
return hasValue(Root,data);
}
int maxValue()         // return maximum value of tree
{
```

```
        return maxValue(Root);
        }
        int minValue()        // return minimum value of tree
        {
        return minValue(Root);
        }
        int sumTree() // return sum of all nodes in tree
        {
        return sumTree(Root);
        }
        int maxDepth()         // return max depth of tree
        {
        return maxDepth(Root);
        }
        bool isBalanced()     // return true if tree balanced
        {
        return isBalanced(Root);
        }
        bool isComplete()     // return true if tree complete
        {
        return isComplete(Root);
        }
        void printPreorder()  // print tree in pre order
        {
        printPreorder(Root);
        }
        void printInorder()   // print tree in order
        {
        printInorder(Root);
        }
        void printPostorder() // print tree in post order
        {
        printPostorder(Root);
        }
        void printTree()          // print tree by indentation
        {
        printTree(Root,10);
        }
        void printLevel()     // print tree by level
        {
        printLevel(Root);
        }
        };
#endif
// binary search tree code
// tree.cpp
// tree class code implementation
#include <iostream.h>
#include "tree.hpp"
#include "quelist.hpp"
```

Inserting items into binary tree

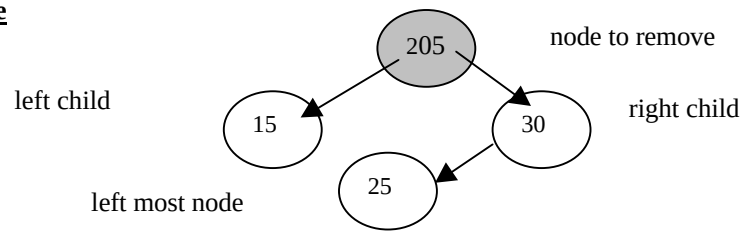
We use recursion to insert an item into a binary tree. Recursion copies every link as it traverses through the tree. it traverses by **comparing** the data to be inserted with the data value at each node it encounters. If the value is **smaller** it traverse **left**, if the data value is **larger** it traverse **right**. When it comes to a leaf a new node is allocated and attached to the left or right of the leaf node.

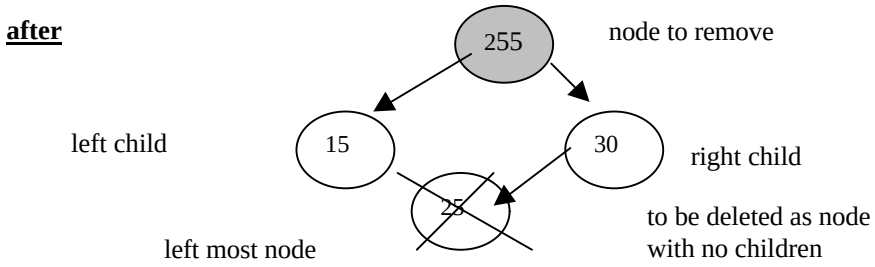
```
// insert item into binary tree
// recursion will find the point to insert the new node
// it traverse the tree till it finds where to inset the new node

TNode* Tree::insert(TNode* tnode,TNodeData data)
        {
        // check if tnode empty
        if(tnode == NULL)
                // make new tree node
                tnode = new TNode(NULL,data,NULL);

        // data smaller than tnode ?
        else if(data < tnode->Data)
                // traverse and copy left
                tnode->Left = insert(tnode->Left,data);
        // data larger than tnode ?
        else if(data > tnode->Data)
                // traverse and copy right
                tnode->Right = insert(tnode->Right,data);
        return tnode; // return current tnode
        }
```

Removing items from binary tree

We use recursion to remove an item from a binary tree, The function traverses throughout the tree looking for the node to delete. It traverses by **comparing** the data to be removed with the data value at each node it encounters. If the value is **smaller** it traverse **left**, if the data value is **larger** it traverse **right**. When the data to be removed is found two conditions must be considered: (1`) a node with two children (2) a node with one child. We will consider a node with two children first. We must find the smallest value from **right child** of the item to be deleted. The **findMin**() function is used to do this. It will find the smallest value by continuing the traversal left only. In a binary tree the smallest element is the most left node. Once the smallest node is found then the traversal is continued from this data value. We then copy these values from the smallest node into the node to be deleted. We continue traversing from the right child to delete the smallest value node located on the left most leaf. The item to be removed because e replace its data value with the smallest data value then continue to remove the left most node. We find the smallest node because we want to preserve the tree ordered property. This left most node will be deleted as a node with no children quite easily.

before





If a node has only one child then we keep a pointer to the opposite child link and delete the current node through a temporary pointer and return the child link. The child link may become the new root if the node that was removed was the tree root.

```
// removing item from binary tree
// case 1 : two children
// case 2 : one child
TNode* Tree::remove(TNode* tnode,TNodeData data)
	{
	TNode* temp; // a temp tnode pointer
	TNode* cnode; // a child tnode pointer

	// check if tnode empty
	if(tnode != NULL)
		{
		// find where to remove item
		// data smaller than current tnode ?
		if(data < tnode->Data)
			// traverse and copy left
			tnode->Left=remove(tnode->Left,data);
		// data larger than current tnode ?
		else if(data > tnode->Data)
			// traverses and copy right
			tnode->Right=remove(tnode->Right,data);
		// found data item
		else
			{
			// case two children
			if(tnode->Left && tnode->Right)

				{
				temp = findMin(tnode->Right); // find smallest item
				tnode->Data = temp->Data; // substitute for tnode
				// traverse right
				tnode->Right = remove(tnode->Right,tnode->Data);
				}
			// case one child
			else
				{
				temp = tnode; // save current tnode
				// store right child
				if(tnode->Left == NULL) cnode = tnode->Right;
```

```
                        // store left child
                        if(tnode->Right == NULL) cnode = tnode->Left;

                        delete tnode; // delete current tnode
                        return cnode; // return child tnode
                        }
                }
        }
return tnode; // return current tnode
} // end removeTree
// findMin
// find smallest data item in binary tree
// the smallest element must be the most left item
TNode* Tree::findMin(TNode* tnode)
        {
        // if tnode not empty
        if(tnode != NULL)
                {
                // check if end of tree
                if(tnode->Left==NULL)return tnode;

                // search for min tnode
                else return (findMin(tnode->Left));
                }
        return tnode;
        }
```

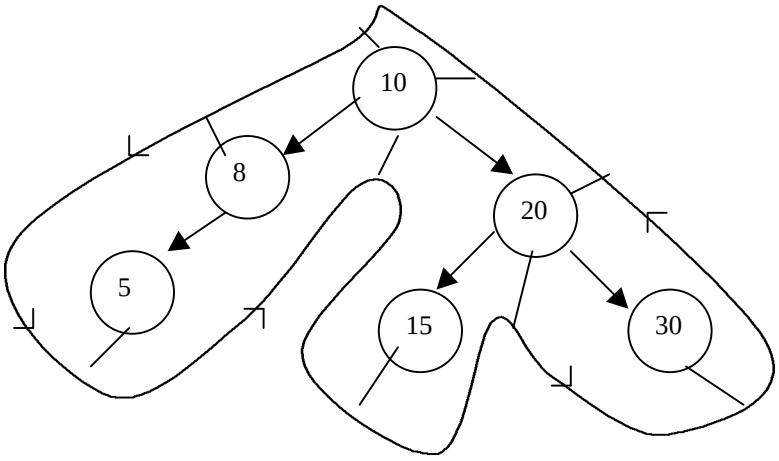
Finding elements in binary tree

To find an item in a binary tree we just traverse through the tree. If the data item we are looking for is smaller than the current mode then we go left. If the data item is larger than the current node then we go right. we stop traversing when we find the item we are looking for.

```
// find data element in tree
// if found return position in tree
// if not found return error
TNode* Tree::find(TNode* tnode, TNodeData data)
        {
        // check if tnode empty
        if(tnode != NULL)
                {
                // item smaller than current node ?
                if(data < tnode->Data)
                return (find(tnode->Left,data)); // search left
                // item larger than current node ?
                else if(data > tnode->Data)
                return (find(tnode->Right,data)); // search right
                }
        return tnode; // return current tnode
        }
```

Printing out a binary tree

Recursion is used to print out a binary tree. A binary tree may be printed out **pre-order**, **in order** or **post order.** To traverse a tree we start at the root of the tree. Moving counterclockwise we walk around the tree passing every node. As you traverse the tree pretend you are touching or hugging the tree.



As you traverse a tree a node can be touched more than once

order	explanation	example					
pre-order	a node is printed the **first time** it is passed	10	8	5	20	15	30
in-order	we print a **leaf** the **first time** we pass it we print an **interior node** the **second time** we pass it	5	8	10	15	20	30
post-order	a node is printed the **last time** we pass it	5	8	15	30	20	10

The print routine can be changed for pre-order, inorder or post-order just by the position of the cout statement.

order	code	position of print statement

pre-order	cout << tnode->Data;
	printTree(tnode->Left);
	printTree(tnode->Right);
		top

in-order	printTree(tnode->Left);
	cout << tnode->Data;
	printTree(tnode->Right);
		middle

post-order	printTree(tnode->Left);
	printTree(tnode->Right);
	cout << tnode->Data;
		bottom

```
/* print tree data elements in pre-order using recursion */
void Tree::printPreorder(TNode* tnode)
        {
        /* check for empty tree */
        if(tnode != NULL)
                {
                cout<<tnode->Data<<" "; /* first  pre-order */
                printPreorder(tnode->Left); /* print out left data item */
                printPreorder(tnode->Right); /* print out right data item */
                }
        }
```

```
/* print tree data elements in order using recursion */
void Tree::printInorder(TNode* tnode)
        {
        /* check for empty tree */
        if(tnode != NULL)
                {
                printInorder(tnode->Left); /* print out left data item */
                cout<<tnode->Data<<" "; /* middle inorder */
                printInorder(tnode->Right); /* print out right data item */
                }
        }
/* print tree data elements in post order using recursion */
void Tree::printPostorder(TNode* tnode)
        {
        /* check for empty tree */
        if(tnode != NULL)
                {
                printPostorder(tnode->Left); /* print out left data item */
                printPostorder(tnode->Right); /* print out right data item */
                cout<<tnode->Data<<" "; /* last  post order */
                }
        }
```

destroying a binary tree

Using postfix we can safely delete a node the last time we pass it. (called by destructor)

```
/* using postfix we deallocate all memory for tree  recursively */
void Tree::destroyTree(TNode* tnode)
{
if(tnode!=NULL)
     {
     destroyTree(tnode->Left);
     destroyTree(tnode->Right);
     delete(tnode);
      }
}
```

Calculate Number of Nodes in a Tree

This function traverses through every node in a tree and always adding 1 to the return value.

```
/* calculate number of nodes of tree */
int Tree::numNodes(TNode* tree)
        {
        if(tree != NULL)
                {
                /* add 1 to count as traverse to tree */
                return (numNodes(tree->Left) + numNodes(tree->Right) + 1);
                }
        else return 0;
        }
```

Calculate Number of Leaves in a Tree

The nodes at the end of a tree are known as leaves, they have no children. This function traverses through every node in a tree and only adding 1 to the return value if a node has no children.

```
/* calculate number of leaves of tree */
int Tree::numLeaves(TNode* tree)
        {
        if(tree != NULL)
                {
                /* add 1 to count if no children as traverse to tree */
                if((tree->Left == NULL) && (tree->Right==NULL))
                return (numLeaves(tree->Left) + numLeaves(tree->Right) + 1);
                /* traverse without adding 1 */
                else return (numLeaves(tree->Left) + numLeaves(tree->Right));
                }
        else return 0;*
        }
```

Calculate Number of Full Nodes in a Tree

Full nodes have both a left and right child. This function traverses through every node in a tree and only adding 1 to the return value if a node has both children.

```
/* calculate number of full nodes of tree */
int Tree::numFullNodes(TNode* tree)
        {
        if(tree != NULL)
                {
                /* add 1 to count if two children as traverse to tree */
                if((tree->Left != NULL) && (tree->Right!=NULL))
                return (numFullNodes(tree->Left) + numFullNodes(tree->Right) + 1);
                /* traverse without adding 1 */
                else return (numFullNodes(tree->Left) + numFullNodes(tree->Right));
                }
        else return 0;
        }
```

Return true if tree has a Negative value Node

This function traverses through every node in a tree and returns true if a node has a negative data value. The | operator is used to return true if a positive value was found.

```
/* has a negative value node */
bool Tree::hasNeg(TNode* tree)
        {
        if(tree != NULL)
            return ((tree->Data < 0) | hasNeg(tree->Left) | hasNeg(tree->Right));
        else return false;
        }
```

Return true if tree has a Positive value Node

This function traverses through every node in a tree and returns true if a node has a positive data value. The | operator is used to return true if a positive value was found.

```
/* has a positive value node */
bool Tree::hasPos(TNode* tree)
	{
	if(tree != NULL)
		{
		return ((tree->Data >= 0) | hasPos(tree->Left) | hasPos(tree->Right));
		}
	else return false;
	}
```

Return true if tree has a specified value Node

This function traverses through every node in a tree and returns if a node has the specified data value. The | operator is used to return true if a the specified value was found.

```
/* has a specified value node */
bool Tree::hasValue(TNode* tree,TNodeData data)
	{
	if(tree != NULL)
	{
	return ((tree->Data == data) | hasValue(tree->Left,data) | hasValue(tree->Right,data));
	}
	else return false;
	}
```

Return maximum value in tree

This function traverses a tree and always return the maximum value of a nodes left ad right nodes. Using this approach the last comparison will return the largest value in the tree. This function is most useful for binary trees that are not ordered meaning the tree node does not necessary contains a smaller value and the right node does not necessary contain a larger value.

```
	/* return maximum value */
	int Tree::maxValue(TNode* tree)
		{
		if(tree != NULL)
			{
			int maxLeft = maxValue(tree->Left);
			int maxRight = maxValue(tree->Right);
			if((tree->Data > maxLeft) && (tree->Data > maxRight))
				return tree->Data;
			else if (maxLeft > maxRight)
			    return maxLeft;
			else return
			    maxRight;
			}
		else return -1;
		}
```

Return minimum value in tree

This function traverses a tree and always return the maximum value of a nodes left ad right nodes. Using this approach the last comparison will return the largest value in the tree. This function is most useful for binary trees that are not ordered meaning the tree node does not necessary contain a smaller value and the right node does not contain a larger value.

```
/* return maximum value */
int Tree::minValue(TNode* tree)
	{
	if(tree != NULL)
	{
	int maxLeft = maxValue(tree->Left);
	int maxRight = maxValue(tree->Right);
	if((tree->Data < maxLeft) && (tree->Data < maxRight))
		return tree->Data;
	else if (maxLeft < maxRight)
		return maxLeft;
	else
		return maxRight;
	}
	else return 10000;
	}
```

Sum up all node values in a tree

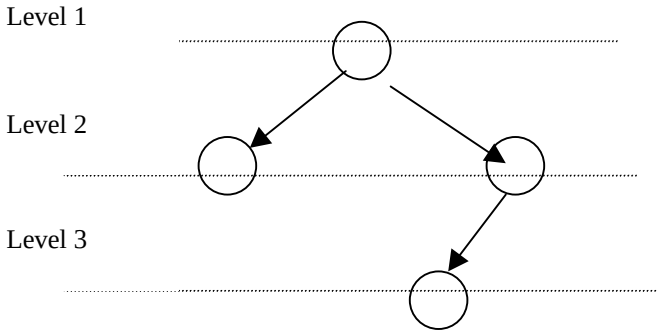
This function traverses every node in a tree adding all the node values.

```
	/* sum of tree */
	int Tree::sumTree(TNode* tree)
		{
		if(tree != NULL)
		{
		return tree->Data + sumTree(tree->Left) + sumTree(tree->Right);
		}
		else return 0;
		}
```

Return maximum depth of a tree

If a tree is balanced the left level is equal to the right level. This function traverses a tree and return the maximum level of a tree.

```
/* max depth of tree */
int Tree::maxDepth(TNode* tree)
	{
	if(tree != NULL)
	{
	int depthLeft = maxDepth(tree->left);
	int depthRight = maxDepth(tree->right);
	if(depthLeft > depthRight)
	    return depthLeft + 1;
```

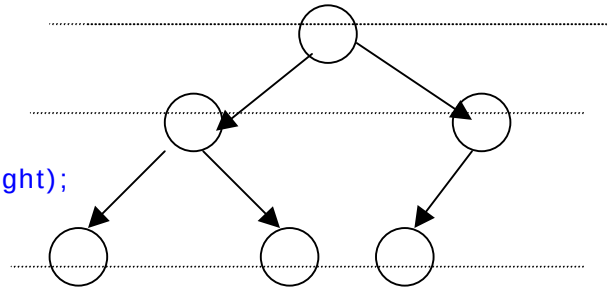


```
        else
            return depthRight + 1;
        }
        else return 0;
        }
```

check if a tree is balanced

A binary tree is balanced if all nodes in the left sub tree have the same level as node in the right sub tree. A balance tree is not necessary complete.

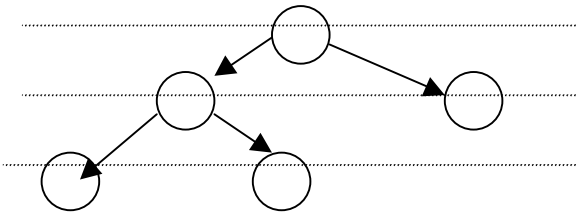
```
/* return true if tree balanced */
bool Tree::isBalanced(TNode* tree)
        {
        if(tree != NULL)
        {
        return maxDepth(tree->Left) == maxDepth(tree->Right);
        }
        else return false;
        }
```



check if a tree is complete

A binary tree is complete if all interior nodes have two children. A complete tree is not necessarily balanced.

```
/* return true if tree complete */
bool Tree::isComplete(TNode* tree)
        {
        if(tree != NULL)
        {
        /* add 1 to count if no children as traverse to tree */
        return (!(isComplete(tree->left) == NULL) ^ (isComplete(tree->right)==NULL));
        }
        else return false;
        }
```



print out tree using indentation

By using inorder and indentation we can print out a tree has it actually suppose to appear. Every time the function is called we increment the level. Every time a the function returns we decrease the level. We use the level to print out indentation. The only problem here is that you need to rotate the image.

```
/* print tree data elements as they appear */
/* using indentation by level and pre order */
void Tree::printTree(TNode* tnode, int level)
{
int i;
level++; /* increment level */
if(tnode != NULL)
{
printTree(tnode->Left,level);
for(i=0;i<level-10;i++)cout<<"   ";
cout<<tnode->Data<<endl;
printTree(tnode->Right,level);
}
level--;  /* decrement level */
}
```

```
5
    8
9
        10
15
    20
30
```

Print tree as it actually appears

Print out the tree as it appears level by level . For example the output for our example tree will be:

```
                10
        8               20
5       9       15      30
```

To do this you need a Queue to store the nodes by level. We traverse the tree level by level. We then print out the tree using the nodes stored in the queue. Print out the tree by level is difficult to do since we need to keep track if the node is a right most node (end of the level). Once we get the end node we must start a new line. We also have to calculate the distance between nodes to get a pleasing output. We use our link list queue.

```
/* print tree by level */
void Tree::printLevel(TNode* tree)
{
int i=1,j,k; /* counters */
int space = 6; /* space distance */
TNodePtr tnode;
QueList que; /* queue */
que.enqueue(tree); /* push tree root into queue */
/* print leading space */
for(j=0;j<space/2;j++)cout<<"  ";
/* loop till queue is empty */
while(!que.isEmpty())
{
tnode = que.dequeue(); /* get tree node */
/* print tree node data value */
if(tnode != NULL)
cout<<tnode->Data<<" ";
else
cout<<"  ";
```

```
/* print space between data */
for(j = 0;j<space;j++)cout<<"  ");
/* push all children of this node into queue */
if(tnode->Left!=NULL)que.enqueue(tnode->Left);
if(tnode->Right!=NULL)que.enqueue(tnode->Right);
i++; /* increment node counter */
/* check if node counter power of 2 */
k=0;
for(j=i;j!=0;j=j>>1)
{
if(j&1)k++;
}
if(j&1)k++;

/* if power of two start new line */
if(k==1)
{
cout<<endl;
space = space/2; /* new space distance */
/* print leading space */
for(j=0;j<space/2;j++) cout<<"  ");
}
} /* end while */
}
```

Here is the main test function:

```
/* main program */
int main()
{
Tree tree;
TNode* tnode;
/* insert */
tree.insert(10);
tree.insert(8);
tree.insert(5);
tree.insert(20);
tree.insert(15);
tree.insert(30);
/* print tree by order */
cout <<"preorder: ";
tree.printPreorder();
cout<<"inorder: ";
tree.printPreorder();
cout <<"postorder: ";
tree.printPostorder();
```

```
/* find nodes */
tnode = tree.find(8);
if(tnode->Data==8)
cout << "found tree item 8" << endl;
tnode=tree.find(15;
if(tnode->Data==15)
cout << "found tree item 15"<<endl;
/* remove */
tree.remove(15);
tree.remove(20);
tree.remove(8);
tree.remove(10);
tree.remove(30);
tree.remove(5);
tree.~Tree();
/* insert tree again */
tree.insertTree(10);
tree.insertTree(8);
tree.insertTree(5);
tree.insert(20);
tree.insert(15);
tree.insert(30);
/* counting */
cout << "number nodes: " << tree.numNodes()<<endl;
cout << "number leaves: " << tree.numLeaves()<<endl;
cout << "number full nodes: " << tree.numFullNodes()<<endl;
/* has value */
cout<<"has negative nodes: "<<tree.hasNeg()<<endl;
cout<<"has positive nodes: "<<tree.hasPos()<<endl;
cout<<"has value node: "<<tree.hasValue(8)<<endl;
/* max/min value */
cout<<"max value: "<<tree.maxValue()<<endl;
cout<<"min value: "<<tree.minValue()<<endl;
/* sum of tree */
cout<<"sum of tree: "<<tree.sumTree()<<endl;
/* max depth */
cout<<"max depth of tree: "<<tree.maxDepth()<<endl;
/* tree balanced */
cout<<"tree balanced: "<<tree.isBalanced()<<endl;
 /* tree complete */
cout<<"tree complete: "<<tree.isComplete()<<endl;
/* make tree complete */
tree.insert(9);
cout<<"tree complete: "<<tree.isComplete()<<endl;
/* print tree using indentation */
cout<<"\n print tree by identation:" << endl;
tree.printTree();
/* print tree by level */
cout<<"\n print tree by level:" << endl;
tree.printLevel();
return 0;
}
```

tree program output:

```
preorder: 10 8 5 20 15 30
inorder: 10 8 5 20 15 30
postorder: 5 8 15 30 20 10
found tree item 8
found tree item 10
number nodes: 6
number leaves: 3
number full nodes: 2
has negative nodes: 0
has positive nodes: 1
has value node: 1
max value: 30
min value: 8
sum of tree: 88
max depth of tree: 3
tree balanced: 1
tree complete: 0
tree complete: 1
print tree by indentation:
                5
                    8
                9
                            10
                15
                        20
                30
print tree by level:
          10
    8           20
5     9     15    30
```

Lesson 12 Exercise 1

Copy edit paste or type in the above programs and run and trace through them.

Lesson 12 Exercise 2

Write a tree function that returns the number of positive node elements in a tree

Lesson 12 Exercise 3

Write a tree function that returns the number of negative node elements in a tree

Lesson 12 Exercise 4

Write a tree function that returns the number of left nodes

Lesson 12 Exercise 5

Write a tree function that returns the number of right nodes

Lesson 12 Exercise 6

Write a tree function that returns true if the number of left nodes are equal to the number of right modes.

Lesson 12 Exercise 7

Write a tree function that returns true if a tree is complete and balanced.

Lesson 12 Exercise 8

Write a tree function that returns true if a the number of nodes in the left sub tree are equal to the number of nodes in the right sub tree.

Lesson 12 Exercise 9

Write a program that will print out all the paths found in a tree and its length. For example for Exercise 4 the tree paths would be
10->6->4 length 3
10->20->15 length 3
10->20->30 length 3

Lesson 12 Exercise 10

Write a program that will print out all the number of interior nodes in a tree

Lesson 12 Exercise 11

Write a program that will print out all the left child nodes in a tree

Lesson 12 Exercise 12

Write a program that will print out all the right child nodes in a tree

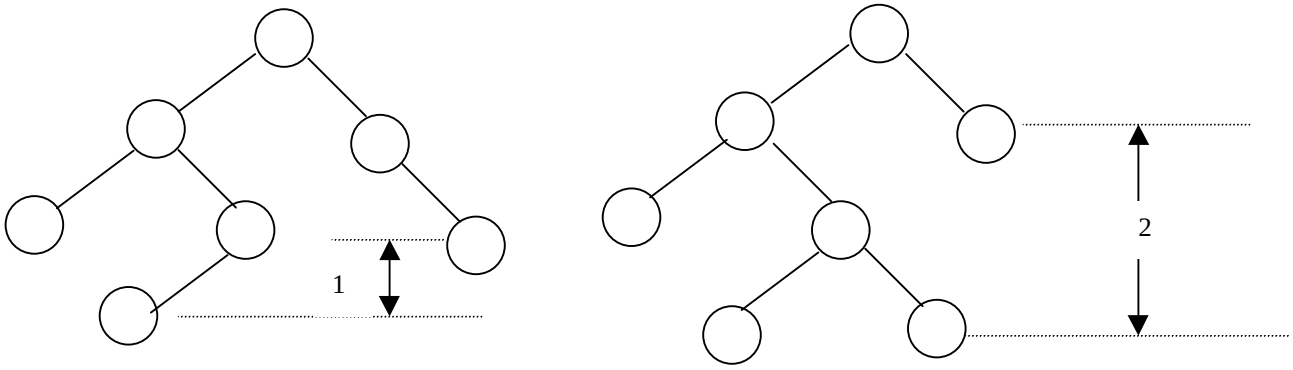
C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 13

File:	CppdsGuideL13.doc
Date Started:	Oct 24,1999
Last Update:	Dec 26, 2001
Status:	draft

LESSON 13 AVL TREES

AVL TREE PROPERTIES

AVL Trees were invented buy Adelson-Velskii and Landis and are **binary search trees** that are balanced. A balanced tree ensures that the depth of the tree is log n and that each tree node has left and right sub trees of the same height or differ by 1 Which following binary search tree is an AVL tree ?



The tree on the left is a AVL tree because the height between of the left and right subtrees only differ by 1. Where the height betweEn subtres on the right binary tree differ by 2.

AVL Tree nodes

Each AVL Tree node will have a data field, a height, a left and right pointer to another AVL tree node that will represent a sub tree.

left	data	height	right

The data filed is the tree key and the nodes are insetered in a order where nodes having a data field less than the parent node are inserted on the left and nodes greater than the parent node are inseted on the right. To keep track of the heigth of each subtree, the height information is kept in the node for each tree node. Every time a tree node is inserterd into the tree the height of the node must be calculated. The AVL tree node is as follows:

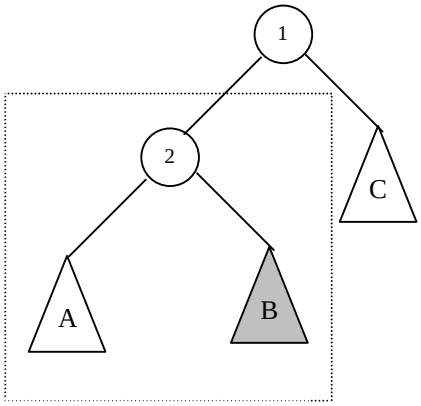
```
/* avl tree node */
struct  AVLNode
{
AVLNode *left;
int data;
int height;
AVLNode *right;
};
```

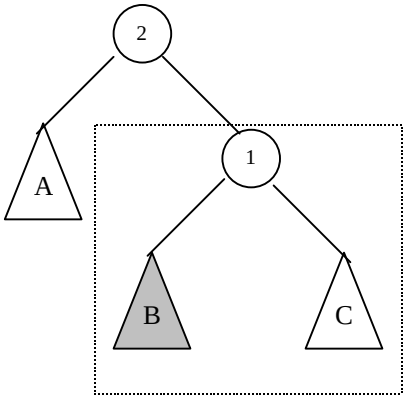
INSERTING A NODE INTO A AVL TREE

When we insert a Node into a AVL tree the tree must be re-balanced if the insertion routine causes the properties of the AVL tree to be violated. Balancing a tree is quite simple all we have to do is rotate sub-trees. We will have left rotation and right rotation. When we only rotate once this is called single rotation. Some times we have to rotate twice before a tree is balance this is known as double rotation.

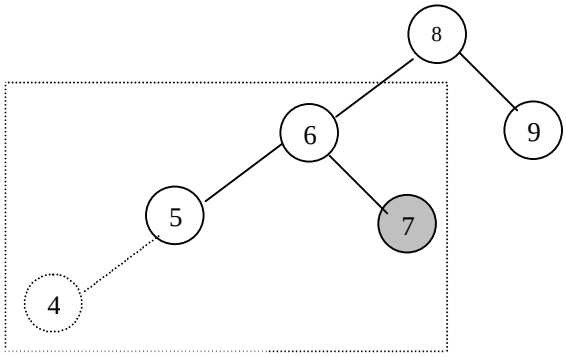
single left rotation.

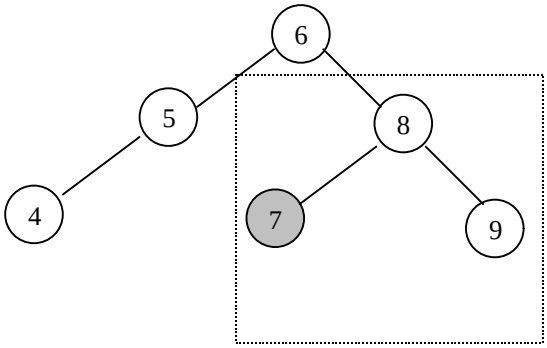
Rotate left child of 1 (node 2) enclosed by the square with right child of 2 sub tree B colored gray.





We use single left rotation to insert the node inserted on the left

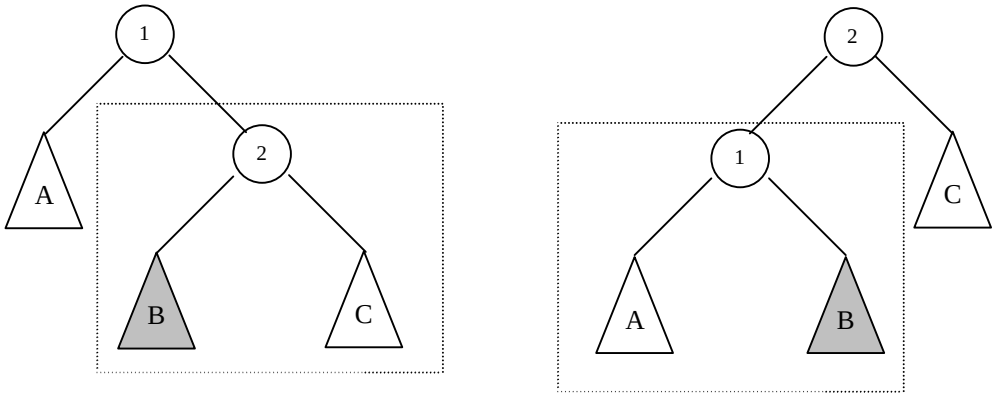




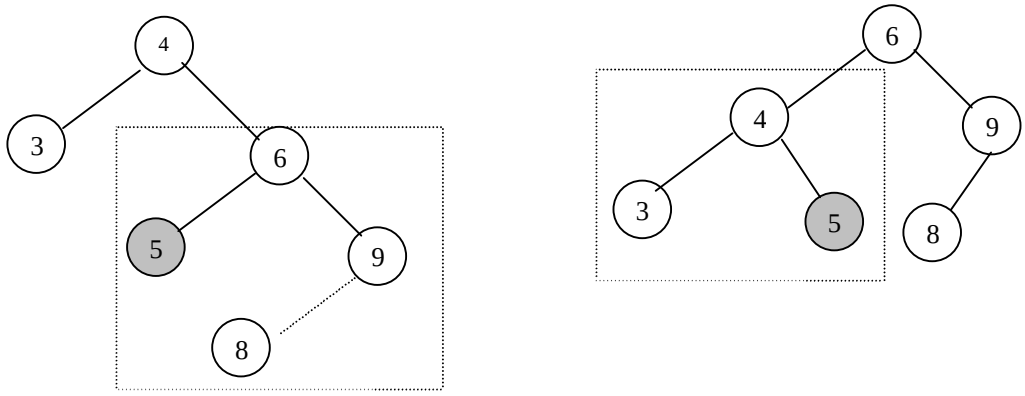
```
/* single rotate left */
AVLNode* rleft_single(AVLNode* T2)
      {
      AVLNode* T1;
      T1 = T2->left;
      T2->left = T1->right;
      T1->right = T2;

      T2->height = max(height(T2->left),height(T2->right))+1;
      T1->height = max(height(T1->left),T2->height)+1;
      return T1;
      }
```

single right rotation.

Rotate right child of 1 (node 2) enclosed by the square with left child of 2 sub tree B colored gray.



We use single left rotation to inset the node inserted on the left

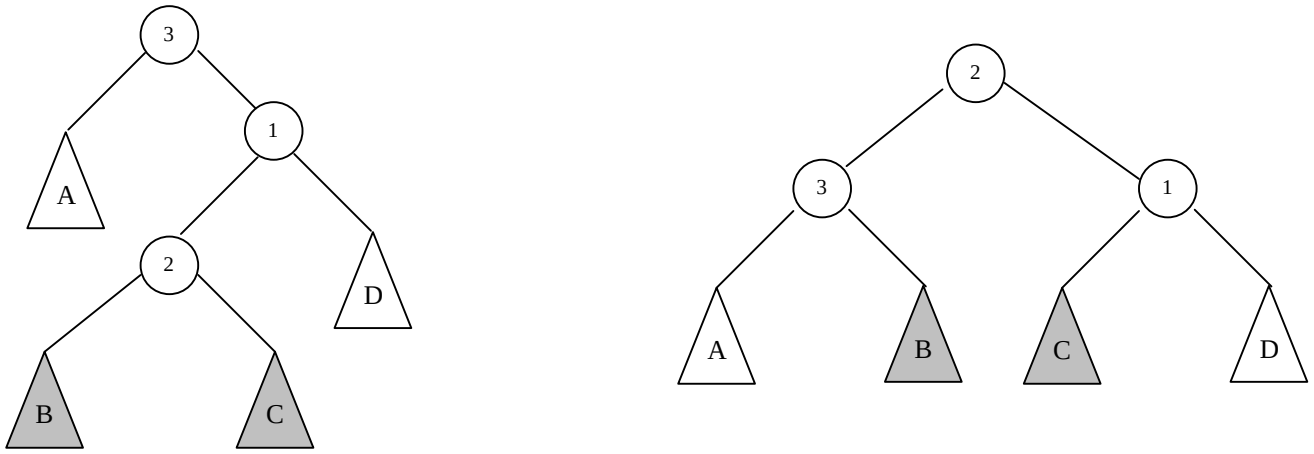


```
/* single rotate right */
AVLNode* rright_single(AVLNode* T2)
      {
      AVLNode* T1;
      T1 = T2->right;
      T2->right = T1->left;
      T1->left = T2;

      T2->height = max(height(T2->right),height(T2->left))+1;
      T1->height = max(height(T1->right),T2->height)+1;
      return T1;
}
```

DOUBLE ROTATION

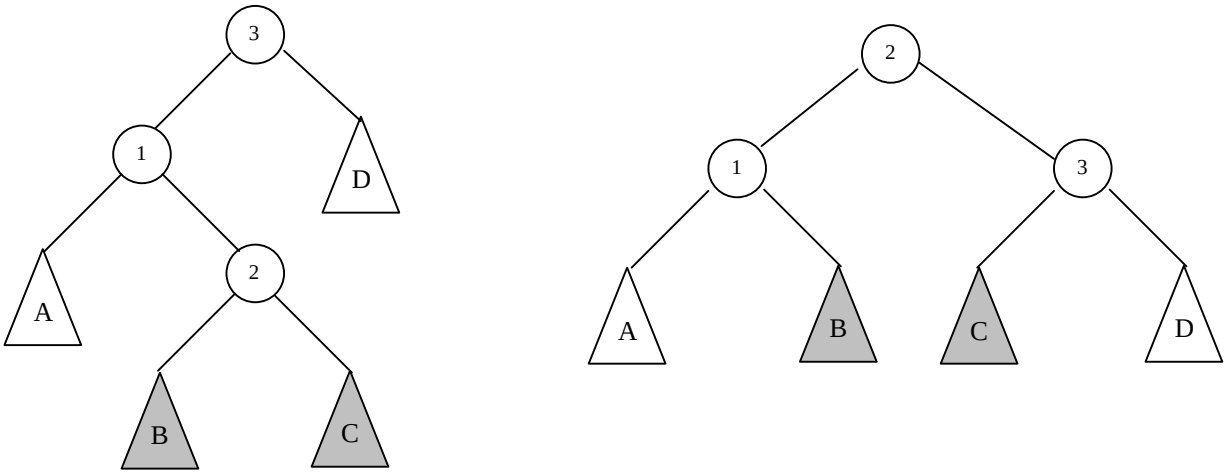
There are cases when single rotation cannot fix a tree imbalance. In this case we have to rotate twice. Double rotation involves 4 sub trees. You do not rotate the two sub trees in the same direction. You either rotate right then left or left then right.

right-left double rotation




```
/* double rotate right-left */
AVLNode* rleft_double(AVLNode* T3)
      {
      T3->left = rright_single(T3->left);

      return(rleft_single(T3));
      }
```

left-right double rotation




```
/* double rotate left-right */
AVLNode* rright_double(AVLNode* T3)
      {
      T3->right = rleft_single(T3->right);

      return(rright_single(T3));
}
```

Here is the complete AVL Code:

```
// avl.cpp
#include <iostream.h>
#include <stdlib.h>

/* avl tree node */
struct AVLNode
{
int data;
AVLNode *left;
AVLNode *right;
int height;
};

// prototypes
AVLNode* insert(AVLNode* T,int data);
AVLNode* rleft_single(AVLNode* T2);
AVLNode* rright_single(AVLNode* T2);
AVLNode* rleft_double(AVLNode* T2);
AVLNode* rright_double(AVLNode* T2);
int height(AVLNode* N);
void print_tree(AVLNode* T);
void destroyTree(AVLNode* T);
```

```
/* avl functions */

/* insert data into avl tree */
AVLNode* insert(AVLNode* T,int data)
      {
      if(T == NULL)
            {
            /* allocate tree node */
            T = new AVLNode;
            if(T == NULL)
              {
              cout<<"out of memory\n";
              return NULL;
              }
        /* fill node */
        else
              {
              T->data =data;
              T->height = 0;
              T->left = NULL;
              T->right = NULL;
              }
        }
  else
        {
        /* insert left */
        if( data < T->data)
              {
              T->left = insert(T->left,data);
              if((height(T->left)-height(T->right))==2)
                    {
                    if(data < T->left->data)
                              T = rleft_single(T);
                    else
                              T = rleft_double(T);
                    }
              else
                    T->height = max(height(T->left),height(T->right))+1;
              }
        /* insert right */
        else if( data > T->data)
              {
              T->right = insert(T->right,data);
              if((height(T->left)-height(T->right))==-2)
                    {
                    if(data > T->right->data)
                          T = rright_single(T);
                    else
                          T = rright_double(T);
                    }
              else
                    T->height = max(height(T->left),height(T->right))+1;
              }
        }
        return T;
      }
```

```
/* single rotate left */
AVLNode* rleft_single(AVLNode* T2)
      {
      AVLNode* T1;
      T1 = T2->left;
      T2->left = T1->right;
      T1->right = T2;
      T2->height = max(height(T2->left),height(T2->right))+1;
      T1->height = max(height(T1->left),T2->height)+1;
      return T1;
      }

/* double rotate left */
AVLNode* rleft_double(AVLNode* T3)
      {
      T3->left = rright_single(T3->left);
      return(rleft_single(T3));
      }

/* single rotate right */
AVLNode* rright_single(AVLNode* T2)
      {
      AVLNode* T1;
      T1 = T2->right;
      T2->right = T1->left;
      T1->left = T2;
      T2->height = max(height(T2->right),height(T2->left))+1;
      T1->height = max(height(T1->right),T2->height)+1;
      return T1;
      }

/* double rotate right */
AVLNode* rright_double(AVLNode* T3)
      {
      T3->right = rleft_single(T3->right);
      return(rright_single(T3));
      }

/* return height of node */
int height(AVLNode* N)
      {
      if(N == NULL) return -1;
      else return N->height;
      }

  /* print tree data elements in order using recursion */
  /* function calls itself until last node reached */
  void print_tree(AVLNode* T)
        {
        if(T != NULL)
              {
              print_tree(T->left);
              cout<<T->data<<" height "<<T->height<<" ";
              print_tree(T->right);
              }
        }
```

```
        /* removes all nodes in tree */
        void destroyTree(AVLNode* T)
              {
              if( T != NULL )
                    {
                    destroyTree( T->left );
                    destroyTree( T->right );
                    free (T);
                    }
              }

/* avl test */
void main()
{
        AVLNode* T=NULL;
        cout<<"\n tree: ";
        print_tree(T);
        T = insert(T,10);
        cout<<"\n tree: ";
        print_tree(T);
        T = insert(T,8);
        cout<<"\n tree: ";
        print_tree(T);
        T = insert(T,5);
        T = insert(T,20);
        T = insert(T,15);
        T = insert(T,30);
        cout<<"\n tree: ";
        print_tree(T);
        destroyTree(T);
        }
```

program output:

```
tree: 10 height 0
tree: 8 height 0 10 height 1
tree: 5 height 0 8 height 1 10 height 0 15 height 2 20 height 1 30 height 0
```

LESSON 13 EXERCISE 1

Make an AVL Tree class using the example code.

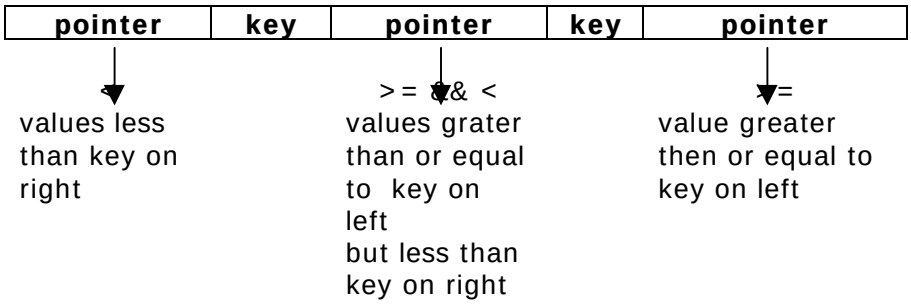
C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 14

File:	CppdsGuideL14.doc
Date Started:	Apr 20,1999
Last Update:	Dec 26, 2001
Status:	proof

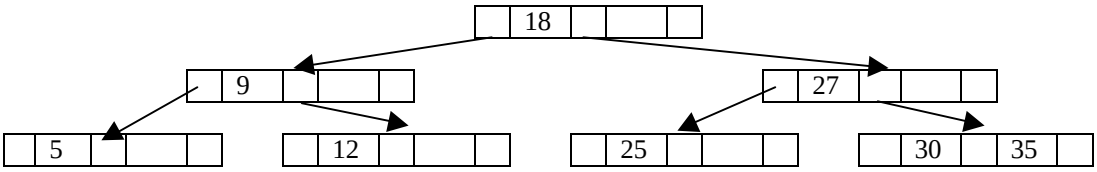
LESSON 14 B-TREES

B-Trees are like binary search trees except each node has more one key and more than two pointers to other nodes. Pointers to others nodes are located between the keys. If a node has three keys then the node will have 4 pointers to other nodes. The node on the left of a key contains keys smaller than this key. Nodes on the right of a key contains keys greater than or equal to this key. All the keys in a node are arranged in order of smallest to largest. The data is contained in the leaves of the B-Tree. The order m of a B-Tree is known as the maximum number of children each node can have. Each node may have between 1 and m-1 keys.

B-Tree node



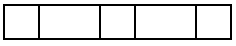
complete B -Tree



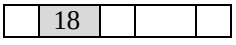
INSERTING KEYS IN A B-TREE

Inserting keys into B-Trees can get quite complicated. Before a key can be inserted the tree must be searched to determine where to insert the key. Keys are inserted one by one into a node until it is filled up. When all the keys are filled in a node then the node needs to be split in two and the two nodes attach to the parent. If the parent is filled then the parent most be split and attach to its parent this is known as pushing up.

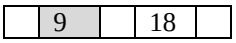
Initially the B-Tree is empty



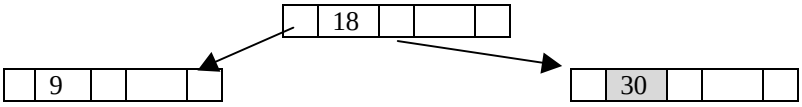
insert 18



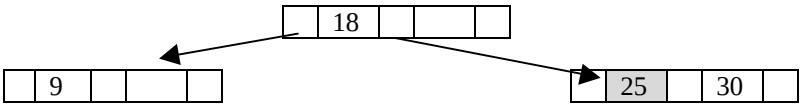
insert 9



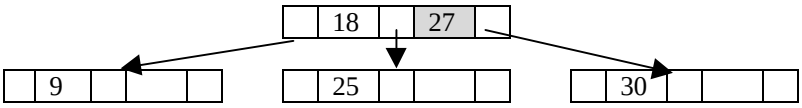
insert 30



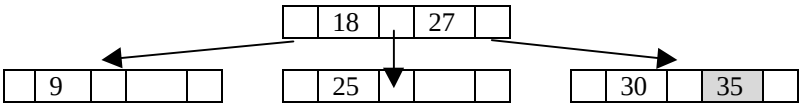
insert 25



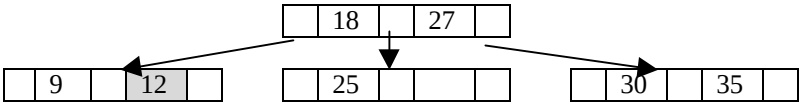
insert 27



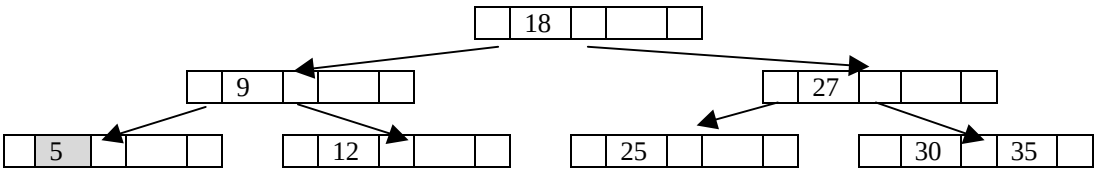
insert 35



insert 12



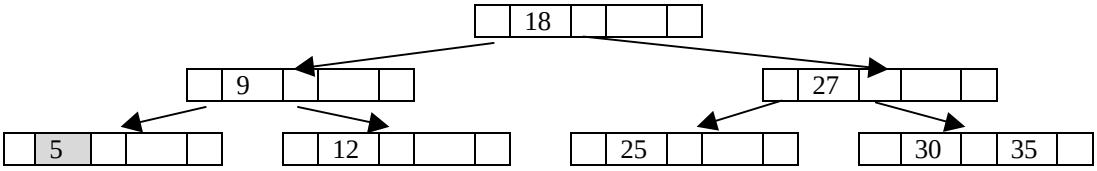
insert 5



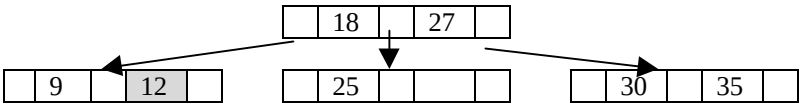
DELETING NODES IN A B-TREE

Deleting keys from B-Trees can get quite complicated. The tree must be searched to determine where to remove the key. Keys are removed one by one until a node becomes empty. When all the keys are removed from a node the lower level nodes need to be attached to the parent. When we delete nodes then we may have to recombine nodes.

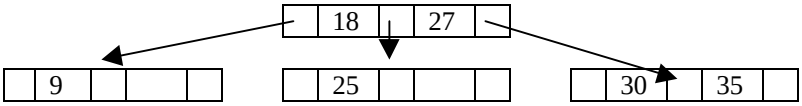
delete 5



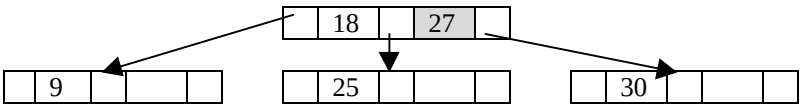
delete 12



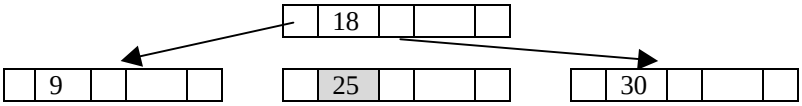
delete 35



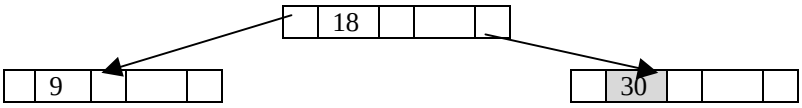
delete 27



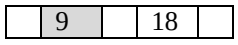
delete 25



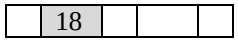
delete 30



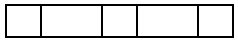
delete 9



delete 18



The b-tree is empty



IMPLEMENTING B-TREES

B-trees are a little difficult to implement. You will find very few books with complete code to implement b-trees. it will take you along tome to write code to implement b-trees. even if you do it may not work for all key combinations. B-trees are mores easily implemented using recursion. We need to write code that will handle any order B-Tree. Our B-Tree class will have the following functions. The searchTree and printTree functions need to be wrapped by the search and print functions. They need to be wrapped because they are recursive and have many parameters. Wrapping means the search and print functions call the searchNode or printNode functions and supply the arguments needed for the recursion. The user of the class does not want to be bothered about supplying many parameters. We have provided B-Tree code for you. we have examined many b-tree algorithms this is the best one we have found.

function	description
searchTree	search a B-Tree the find node to insert key into
searchNode	search a node to insert key in the correct position
insertNode	insert key into node
splitNode	when a node is filled it needs to be split in two and a new

	node formed from 1/2 of the original node.
deleteNode	delete key from node
printTree	print out nodes in B-tree

wrapper functions

function	description
insert	insert key into Btree calls insertBTree
print	prints out Btree call printBTree

There are two data structures a BTNode and a Entry included inside the BTree class:

structure	field	description
Entry	key	key value
	data	optional data
BTNode	numEntries	number of entries
	entries	an array of entries containing key and data
	links	array of pointers to other nodes

```
/* btree.hpp */
#include <iostream.h>
//#define true 1
//#define false 0
// btree class
class BTree
        {
        public:
        /* Entry includes key and optional data */
        struct Entry
                {
                int key;
                Entry(){key = 0;}
                Entry(int key) {this->key = key;}
                };
        private:
        /* btree node */
        struct BTNode
                {
                int numentries; /* number of data elements in node */
                Entry *entry; /* keys */
                BTNode **link; /* links to other nodes */
                // default constructor
                BTNode()
                        {
                        numentries=0;
                        entry=NULL;
                        link=NULL;
                        }
                };
```

```
BTNode* root; // start of btree
public:
BTree(){root = NULL;} // default constructor
~BTree(){}
/* wrapper to insert key into tree */
void insert(Entry& newEntry, int order)
{root = insertBTree(newEntry,root, order);}
/* wrapper to print b-tree */
void print(){printBTree(root,0);}
private:
/* search node for key */
int searchNode(int key, BTNode *root, int *pos);
/* insert Entry into node */
void insertNode
(Entry& splitEntry, BTNode *splitRight,BTNode *current, int pos);
/* insert Entry into b-tree */
BTNode* insertBTree(Entry& newEntry, BTNode *current, int order);
/* search b-tree for Entry */
searchBTree
(Entry& entry,BTNode *current,Entry *splitEntry,BTNode **splitRight,int order);
/* split node */
void splitNode(BTNode *current, Entry& splitEntry,BTNode *splitRight, int
pos,BTNode**newnode,Entry *newmid, int order);
/* print b-tree */
void printBTree(BTNode *, int level);
};
/* B-Tree class implementation */
/* btree.cpp */
#include<iostream.h>
#include "btree.hpp"
typedef int BTreeData;
/* insert Entry into B-tree */
BTree::BTNode* BTree::insertBTree
(Entry& newEntry, BTNode *root,int order)
        {
        Entry   splitEntry;
        BTNode  *splitRight,*newnode;
        /* search tree to place Entry */
        if(searchBTree(newEntry, root, &splitEntry, &splitRight,order))
                {
                /* allocate memory for node and node entries and links */
                newnode = new BTNode();
                newnode->entry = new Entry[order+1];
                newnode->link = new BTNode*[order+1];
                newnode->entry[1] = splitEntry;
                newnode->link[0] = root;
                newnode->link[1] = splitRight;
                newnode->numentries = 1;
                return newnode;
                }
        return root;
        }
```

```
/* Search tree to find where to insert new Entry */
/* returns true if found otherwise false */
int BTree::searchBTree(Entry& newEntry, BTNode *current,Entry *splitEntry,BTNode
**splitRight, int order)
        {
        int pos, pos_value;
        /* insertion point found */
        if (!current)
             {
             *splitEntry = newEntry;
             *splitRight = NULL;
             return TRUE;
             }
        else
             {
             /* do not insert duplicate keys */
             if (searchNode(newEntry.key, current,&pos))
                       {
                       cout << "cannot insert duplicate key";
                       return FALSE;
                       }
                pos_value = searchBTree
                (newEntry, current->link[pos],splitEntry,splitRight,order);



                /* insert info into this node */
                if(pos_value)
                       {
                       /* check if node is filled */
                       if(current->numentries < order)
                              {
                              insertNode(*splitEntry,*splitRight,current, pos);
                              return FALSE;
                              }
                       /* node is filled split node */
                       else
                              {
                              splitNode(current,*splitEntry,*splitRight,pos,
                              splitRight,splitEntry,order);
                              return TRUE;
                              }
                       }
                       return FALSE;
                }
        }
```

```
/* Search node for key  return true if found */
int BTree::searchNode(int key, BTNode *root, int *pos)
        {
        int i;
        /* key less than first Entry key */
        if(key < root->entry[1].key)
                {
                *pos = 0;
                return FALSE;
                }
        /* key greater than or equal to first Entry key */
        else
                {
                /* search for key */
                for(i = root->numentries; i>1; i--)
                {
                if(key >= root->entry[i].key)break;
                }


        *pos = i;
        /* return true if key found */
        return (key == root->entry[*pos].key);
        }
}
/* insert Entry into node */
void BTree::insertNode
(Entry& splitEntry, BTNode *splitRight,BTNode *current, int pos)
        {
        int i;
        /* shift entries right to make room */
        for(i=current->numentries; i>pos; i--)
                {
                current->entry[i+1] = current->entry[i];
                current->link[i+1] = current->link[i];
                }
        /* insert Entry */
        current->entry[pos+1] = splitEntry;
        current->link[pos+1] = splitRight;
        current->numentries++;
        return;
        }
/* split current node into two nodes*/
void BTree::splitNode
(BTNode *current, Entry& splitEntry,BTNode *splitRight,
int pos, BTNode**newnode,Entry *newmid,int order)
        {
        int i;
        int midpoint;
        if(pos<=order/2)midpoint = order/2;
        else midpoint = order/2+1;
```

```
        /* allocate memory for new node */
        *newnode = new BTNode();
        (*newnode)->entry = new Entry[order+1];
        (*newnode)->link = new BTNode*[order+1];

        /* copy upper nodes into new node */
        for(i=midpoint+1; i <= order; i++)
                {
                (*newnode)->entry[i-midpoint] = current->entry[i];
                (*newnode)->link[i-midpoint] = current->link[i];
                }

        /* set newnode number of entries */
        (*newnode)->numentries = order - midpoint;
        /* update  current number of entries */
        current->numentries = midpoint;
        /* insert nodes */
        if(pos<=order/2) insertNode(splitEntry, splitRight,current, pos);
        else insertNode(splitEntry,splitRight, *newnode, pos-midpoint);
        /* pointer to  middle node */
        *newmid = current->entry[current->numentries];
        /* set new node link 0 */
        (*newnode)->link[0] = current->link[current->numentries];
        current->numentries--; /* decrement current number of entries */
        }
/* recursively print out b-tree  in-order */
void BTree::printBTree(BTNode *root,int level)
        {
        int i,j;
        if (!root)return;
        else
                {
                level++;
                for(i=0; i<=root->numentries; i++)printBTree(root->link[i],level);
                cout << "level: " <<  level << ": ";
                for(j=1; j<=root->numentries; j++)cout << root->entry[j].key << " ";
                cout << endl;
                }
        }
/* btree.cpp */
#include<iostream.h>
#include "btree.hpp"
typedef int BTreeData;
/* test driver for btree */
void main()
        {
        BTree btree; // declare btree
        int order = 2; // set order
        /*insert Entry into btree*/
        btree.insert(BTree::Entry(18),order);
        cout << "\nb tree: \n";   /*print b_tree*/
        btree.print();
```

```
        btree.insert(BTree::Entry(9),order);
        cout << "\nb tree: \n";   /*print b_tree*/
        btree.print();
        btree.insert(BTree::Entry(30),order);
        cout << "\nb tree: \n";   /*print b_tree*/
        btree.print();
        btree.insert(BTree::Entry(25),order);
        cout << "\nb tree: \n";   /*print b_tree*/
        btree.print();
        btree.insert(BTree::Entry(27),order);
        cout << "\nb tree: \n";   /*print b_tree*/
        btree.print();
        btree.insert(BTree::Entry(35), order);
        cout << "\nb tree: \n";   /*print b_tree*/
        btree.print();
        btree.insert(BTree::Entry(12),order);
        cout << "\nb tree: \n";   /*print b_tree*/
        btree.print();
        btree.insert(BTree::Entry(5),order);
        cout << "\nb tree: \n";   /*print b_tree*/
        btree.print();
        }
```

program output:

```
b tree:
level 1: 18
b tree:
level 1: 9 18
b tree:
level 2: 9
level 2: 30
level 1: 18
b tree:
level 2: 9
level 2: 25 30
level 1: 18
b tree:
level 2: 9
level 2: 25
level 2: 30
level 1: 18 27
b tree:
level 2: 9
level 2: 25
level 2: 30 35
level 1: 18 27
b tree:
level 2: 9 12
level 2: 25
level 2: 30 35
level 1: 18 27
b tree:
level 3: 5
```

```
level 3: 12
level 2: 9
level 3: 25
level 3: 30 35
level 2: 27
level 1: 18
```

LESSON 14 EXERCISE 1

Type in the B-Tree program. Trace though it with the debugger and figure out how it works. Draw a programming module how all the functions work.

LESSON 14 EXERCISE 2

Write a print routine that will print out the B-Tree as it actually appears. Your output could look like this:

```
                    18

          9                   27

    5          12       25          30    35
```

LESSON 14 EXERCISE 3

Write the destructer to delete the BTree.

LESSON 14 EXERCISE 4

Now that you know how the insert routines work. write the function to delete keys in the B-Tree.

LESSON 14 EXERCISE 5

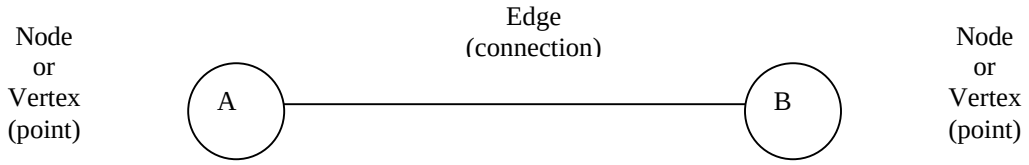
Write a pure abstract template class that defines the Entry structure. Remove the Entry structure from the Btree class. Derive a class from your Entry template abstract class to handle any data type keys and any data types.

C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 15

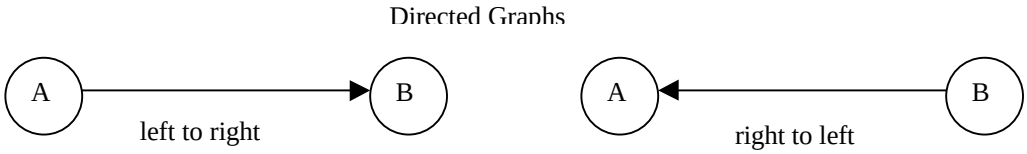
File:	CppdsGuideL15.doc
Date Started:	April 15,1999
Last Update:	Dec 26, 2001
Status:	draft

LESSON 15 CONSTRUCTING GRAPHS

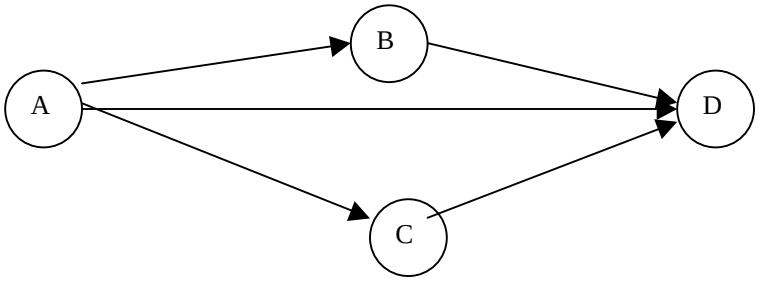
Graphs are used when you need to connect many things together. The "things" are called **nodes** or **vertices** and the **connection** between the nodes are called **edges**.



The above graph is said to be **undirected** meaning the direction of travel can be left to right or right to left. Graphs can also be **directed** either forcing direction from left to right or from right to left.



Graph's are made using many edges and vertices. The following is a directed graph of 4 vertices. Why directed ?



Graph Terminology

Before you can start doing work with graphs you need to know what all the terms mean.

term	description
graph	connection of points
vertex or node	points on a graph, each vertex has a name
edge	connection between vertices
adjacent vertices	two vertices connected together by an edge vertex A connected by an edge to vertex B
indegree	number of edges pointing **toward** this vertex
outdegree	number of edges pointing **away** from this vertex

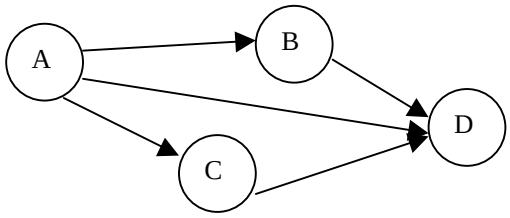
path	sequence of connecting edges from one vertex to another
visit vertices in a graph	follow edge connections from vertex to vertex
cycle	when travelling a path you return to a vertex that we previously visited
circuit	visit every vertex in the graph
connected graph	a path exists between every vertex in the graph
complete graph	every vertex is adjacent to every another vertex
directed graph (digraph)	a edge only allows direction one way as stated by the connection arrow
undirected graph	edge allows travel in both directions
directed acylic graph (DAG)	a directed graph with no cycles
weighted graph	each edge has a value greater or less than 1

DATA STRUCTURES REPRESENTING GRAPHS

A graph may be represents by using a matrix called an **adjacency matrix** or using a link list of nodes called an **adjacency list**. We will start describing how to construct a graph using a adjacency matrix then proceed to describe constructing a graph using an adjacency list. Adjacency means a list of all connecting vertices.

REPRESENTING A GRAPH USING AN ADJANCENCY MATRIX

A matrix can be used to represent directed and undirected graphs. Each **row** and **column** index of the matrix are **vertices** and the **value** inside the matrix represents an **edge**. A 0 means no connection where a value of 1 means a connection. In case of **weighted** graphs the value would represent the weight of the connecting **edge**. The direction of the edge can be stated as row to column or as column to row. Since rows and columns represent vertices the row is the left vertex where the column is the right vertex. An adjacency matrix is used to represent a graph with many connections.



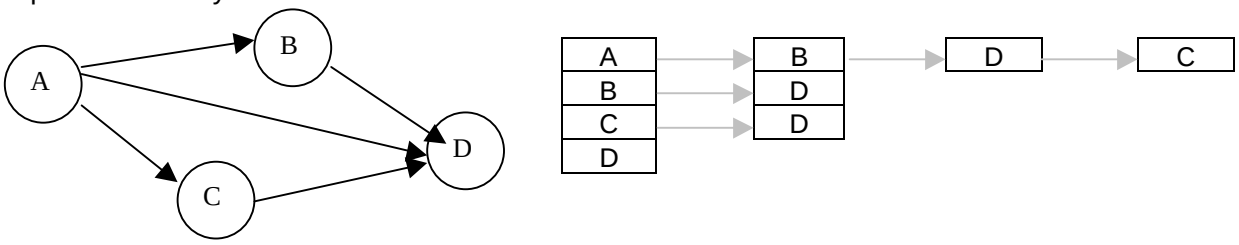
	A	B	C	D
A	0	1	1	1
B	0	0	0	1
C	0	0	0	1
D	0	0	0	0

Trace the graph and verify the adjacency matrix represents the graph. Draw the graph from the adjacency matrix.

REPRESENTING A GRAPH USING AN ADJANCEY LIST

An **adjacency list** may also be used to represent directed and undirected graphs. The vertices are listed in a header array or link list. The vertices listed in the header are the vertices on the left hand side. "the from vertex". The connections from each from vertex is listed in an additional adjacency list. Each node in the adjacency represents a connection **from** the **left** vertex to the **right** vertex "the to vertex". The link node stores the name of the right vertex and additional information like the **weight** of the connection edge. For directed graphs there will be only one node for each connection.

For undirected graphs then the entry must be made twice for each vertices.. One entry in each adjacency list for each direction. Sometimes a node may contain a pointer to another vertex to represent a two way connection pointing back to the left vertex. An adjacent list is used for large graphs with many vertices but few connections.



Trace the graph and verify the adjacency list represents the graph. Draw the graph from the adjacency list.

ABSTATRACT DATA TYPE FOR GRAPHS GRAPH ADT

We need to define an Abstract Data Type ADT for graphs. It is very important that we use good object oriented programming techniques. By using good object oriented programming technique we can use the same functions for graphs made from an adjacency matrix or from graphs using an adjacency list. This means we do not need to re-write the code to uses either graph representation method**.** We first need to define all the methods needed for constructing a graph. We would need to add a vertex, add an edge, test if a vertex is in the graph, test if an edge connects to vertices. etc. The method names would be the same for graph implemented by a adjacency matrix or adjacency list. The only difference would be the code inside. Vertex are identified by string name for easy identification rather than a numeric value. We would need the following methods:

operation	method	description
initialization	GraphX(int size);	initialize graph object to size
add a vertex	bool addVertex(Vertex v);	add vertex to graph
add an edge	bool addEdge (Vertex v1, Vertex v2, int w);	connect to vertices v1 and v2 together by weight w
find a vertex	int findVertex(Vertex v);	search for vertex in graph
remove a vertex	boolean removeVertex(Vertex v);	remove vertex from graph
remove an edge	bool removeEdge(Vertex v1, Vertex v2)	remove edge from graph
get edge	int getEdge(Vertex v1,Vertex v2);	get edge weight for specified vertices
print a graph	operator <<(ostream& out, GraphM& g);	print graph

GRAPHM CLASS GRAPH ADT USING ADJANCEY MATRIX

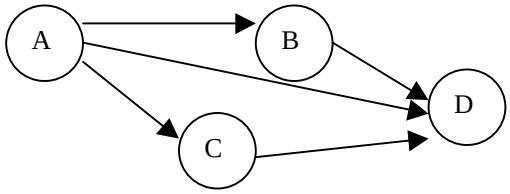
We first need a class called GraphM to hold all the information about the graph . The most obvious thing we need is a two dimensional array to keep track of all the edge weights. We also need an array that holds the vertex name. Finally we need variables for the max size and number of vertices stored in the graph itself. We use a char to represent vertices and int to represent weights. We use the vertex name for graph operations for simplicity. Here's the code to implement a graph using an adjacency matrix.

```
// graphm.hpp
#include <iostream.h>
//typedef int bool;
#define sp " "
//enum {false,true};
/* set data types for vertices and edges */
typedef char Vertex;
typedef int Edge;

// class to  implement a graph using an adjacency matrix
class GraphM
        {
        private:
        int** m; /* pointer to array rows of columns */
        int size; /* max size of  graph */
        Vertex* v; /* array of vertices names */
        int n; /* number of vertices in graph */
        public:
        /* initialize graph */
        GraphM (int size );
         /* add vertex to graph */
        bool addVertex(Vertex v1);
        /* add edge to a graph */
        bool addEdge(Vertex v1, Vertex v2, Edge w);
        /* find a vertex in a graph */
        int findVertex(Vertex v1);
        /* get an edge weight from a graph */
        Edge getEdge(Vertex v1, Vertex v2);
        /* print out a graph */
        friend ostream& operator>>(ostream& out, GraphM& g);
        };  /* end graphm */
        /* Graphm implementation */
        // graphm.cpp
        #include <iostream.h>
        #include "graphm.hpp"
        /* initialize graph */
        GraphM::GraphM (int size )
                {
                /* allocated memory for a 2 dimensional array */
                m = new Edge*[size];
                for(int i=0;i<size;i++)
                        {
                        m[i]=new Edge[size]; // allocate for each row
                        for(int j=0;j<size;j++)
                        m[i][j]=0; // set elements to zero
                        }
                v = new Vertex[size]; /* names of vertices */
                this->size = size;   /* size of matrices */
                this->n = 0;  /* number of vertices in graph */
                }
```

```
/* add vertex to graph */
bool GraphM ::addVertex(Vertex v1)
        {
        if(findVertex(v1)>=0)return false;
        v[n] = v1;
        n++;
        return true;
        }
/* add edge to a graph */
bool GraphM ::addEdge(Vertex v1, Vertex v2, Edge w)
        {
        int i = findVertex(v1);
        int j = findVertex(v2);
        if(i > n || j > n)return false;
        if(i < 0 || j < 0)return false;
        m[i][j] = w;
        return true;
        }
/* find a vertex in a graph */
int GraphM ::findVertex(Vertex v1)
        {
        for(int i=0;i<n;i++)if(v1 == v[i])return i;
        return -1;
        }
/* get an edge weight from a graph */
Edge GraphM ::getEdge(Vertex v1, Vertex v2)
        {
        int i = findVertex(v1);
        int j = findVertex(v2);
        if(i < 0 || j < 0)return 0;
        return m[i][j];
        }
/* print out a graph */
osteam& operator>>(ostream& out, GraphM& g)
        {
        int i;
        out << "  ";
        // loop through all vertices
        for(i=0;i<g.n;i++)out << g.v[i] << sp; /* print out vertices names */
        out << endl;
        // loop through all vertices
        for(i=0;i<g.n;i++)
                {
                out << g.v[i] << sp;
                for(int j=0;j<g.n;j++)
                out << g.m[i][j] <<  sp; // print out edge weights
                out << endl;
                }
        out << endl;
        return out;
        }
```

```
/*  test driver for adjacency matrix graph */
void main()
{
GraphM g(4);
g.addVertex('a');  // add vertex by name
g.addVertex('b');
g.addVertex('c');
g.addVertex('d');
g.addEdge('a','b',1); // add edge by vertex name and weight
g.addEdge('a','c',1);
g.addEdge('a','d',1);
g.addEdge('b','d',1);
g.addEdge('c','d',1);
g.print();
}
```

Program Output:




```
   a b c d
a  0 1 1 1
b  0 0 0 1
c  0 0 0 1
d  0 0 0 0
```

LESSON 15 EXERCISE 1

Type in the code and get it working. Draw some of your own undirected and directed graphs and enter them in the main method. Use the print graph method to print them out.

LESSON 15 EXERCISE 2

Write the remove vertex methods. If you remove a vertex in one list it needs to be removed in all lists. Call your project L15ex1.

LESSON 15 EXERCISE 3

Include an automatic iterator mechanism in the GraphM class. an iterator will let you advance to the next adjacent vertex The iterator() method will set an array of counters for each vertex called itr to the first adjacent vertex of a particular specified vertex. The getVertex() method will get the vertex at the iterator. The nextVertex() method will advance the iterator and return the next vertex.

LESSON 15 EXERCISE 4

When an edge is added keep track of the number of indegrees and out degrees for each vertex. The vertex at the left gets it outdegree counter incremented. The vertex on the right gets it indegrees counter incremented. Supply functions indegrees() and outdegrees() that return the number of indegrees or outdegrees a vertex has.

GRAPHL CLASS GRAPH ADT USING ADJANCY LIST

We need to use the link list class with the graph class. Each node in the link list module will have a vertex name and a weight and a reference to the next node.

vertex name	edge weight	next adjacent vertex

We have to modify the link list class. We will rename our node GNode and the link list class GList. The basic functions of the list module do not change. Only the insert method because we now the list nodes have names and weights. The GList class needs an additional member the vertex name. Here's the GNode and GList class and the methods needed for the Graph class operations.

```
// graphl.hpp
#include <iostream.h>
//typedef int bool;
#define sp " "
//enum {false,true};
/* set data types for vertices and edges */
typedef char Vertex;
typedef int Edge;
// link list class for graph
class GList
        {
        public:
        // class to hold nodes for adjacency list
        struct GNode
                {
                public:
                GNode* next; // next node
                Vertex v; // vertex name
                Edge w; // weight
                // initialize vertex name and weight
                GNode (Vertex v, Edge w )
                        {
                        this->v = v; // set vertex name
                        this->w = w; // set weight
                        next=NULL; // set next node to NULL
                        }
                };
        public:
        GNode* head; // start of list
        GNode* tail; // end of list
        Vertex v; // vertex name
// default constructor
GList ();
// set vertex name
GList (Vertex v );
// insert node at end of list
void insertGTail(Vertex v,int w);
/* find v element in list */
/* if found return position in list */
```

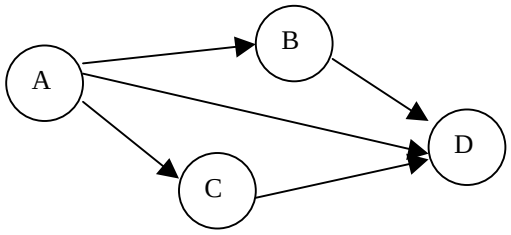
```
/* if not found return error */
GNode* findGList(Vertex v1);
}; // end glistl
/* GList class implementation */
/* GList.cpp */
#include "glist.hpp"
// default constructor
GList::GList ()
        {
        head = NULL;
        tail = NULL;
        v = 0;
        }
// set vertex name
GList::GList (Vertex v )
        {
        head = NULL;
        tail = NULL;
        this->v = v;
        }

// insert node at end of list
void GList::insertGTail(Vertex v,int w)
        {
        /* make new node */
        GNode* node = new GNode(v,w);
        /* check for empty list */
        if(head == NULL)
                {
                head = node;
                tail = head;
                }
        else
                {
                tail->next=node; // insert node at end of list
                tail=node; // adjust end
                }
        }
/* find v element in list */
/* if found return position in list */
/* if not found return error */
GList::GNode* GList::findGList(Vertex v1)
        {
        GNode* curr = head;
        while(curr != NULL)
                {
                if(v1 == curr->v)return curr;
                curr=curr->next;
                }
return NULL;
}
```

Once we have our glist class, we need a class called GraphL to hold all the information about the graph list class. The graph module will have an array of graph lists and variables for holding the max number of vertices and number of vertices in the graph. Here's the code for the graph class using an adjacency list:

```
/* glist.hpp */
#include "glist.hpp"
// graph list class
class GraphL
        {
        private:
        GList* h; /* array of GList objects */
        int size; /* max size of  graph */
        int n; /* number of vertices in graph */
        Vertex* v; /* vertices */
        public:
        /* allocate memory for a graph */
        GraphL(int size);
        /* add vertex to graph */
        bool addVertex(Vertex v1);
        /* add edge to a graph */
        bool addEdge(Vertex v1, Vertex v2, Edge w);
        /* find a vertex in a graph */
        int findVertex(V v1);
        /* print out a graph */
        friend ostream& operator<<(ostream& out, GraphL& g);
        /* get an edge weight from a graph */
        Edge getEdge(V v1, V v2);
        }; /* end graphl */
        /* graphl class implementation */
        // graphl.cpp
        #include <iostream.h>
        #include "graphl.hpp"
        /* allocate memory for a graph */
        GraphL::GraphL(int size)
                {
                /* allocated memory for an array of GList's */
                h = new GList[size];
                this->size = size;   /* size of graph list */
                n = 0;  /* number of vertices in graph */
                v = new Vertex[size]; /* vertex names */
                }
        /* add vertex to graph */
        bool GraphL::addVertex(Vertex v1)
                {
                int i = findVertex(v1);
                if(i >= 0)return false;
                if(n>=size)return false;
                v[n] = v1;
                h[n++].v = v1;
                return true;
                }
```

```
/* add edge to a graph */
bool GraphL::addEdge(Vertex v1, Vertex v2, Edge w)
        {
        int i = findVertex(v1);
        int j = findVertex(v2);
        if(i > n || j > n)return false;
        h[i].insertGTail(v2,w);
        return true;
        }
/* find a vertex in a graph */
int GraphL::findVertex(Vertex v1)
        {
        for(int i=0;i<n;i++)if(v1 == h[i].v)return i;
        return -1;
        }
/* print out a graph */
ostream& operator<<(ostream& out, GraphL& g)
        {
        out << endl;
        // loop for each list
        for(int i=0;i<g.n;i++)
                {
                out << g.h[i].v << sp;
                GList::GNode* curr = g.h[i].head;
                // loop for each node in list
                while(curr != NULL)
                        {
                        out << curr->v << sp;
                        curr = curr->next;
                        }
                out << endl;
                }
                out << endl;
                return out;
                }

/* get an edge weight from a graph */
Edge GraphL::getEdge(Vertex v1, Vertex v2)
        {
        int i = findVertex(v1);
        if(i < 0)return 0;
        GList::GNode* n  = h[i].findGList(v2);
        if(n!= NULL)
        return n->w;
        else return 0;
        }
```

```
/* driver to test graph using adjacency list */
void main()
        {
        GraphL g(4); // create a graph object
        g.addVertex('a'); // add vertices
        g.addVertex('b');
        g.addVertex('c');
        g.addVertex('d');
        // add edges between vertices with weight 1
        g.addEdge('a','b',1);
        g.addEdge('a','c',1);
        g.addEdge('a','d',1);
        g.addEdge('b','d',1);
        g.addEdge('c','d',1);
        cout << g;
        }
```



Program Output:

```
a: b c d
b: d
c: d
d:
```

LESSON 15 EXERCISE 5

Type in the graph list class and get it going. Write the remove edge and remove vertex methods. If you remove a vertex in one list it needs to be removed in all lists. Call your project L15ex5.

LESSON 15 EXERCISE 6

Include an automatic iterator mechanism in the GraphL class. An iterator will let you advance to the next adjacent vertex The **iterator()** method will set an iterator pointer each vertex called itr to the first adjacent vertex of a particular specified vertex. The **getVertex()** method will get the vertex at the iterator. The **nextVertex()** method will advance the iterator to the next vertex in the list and return the vertex.

LESSON 15 EXERCISE 7

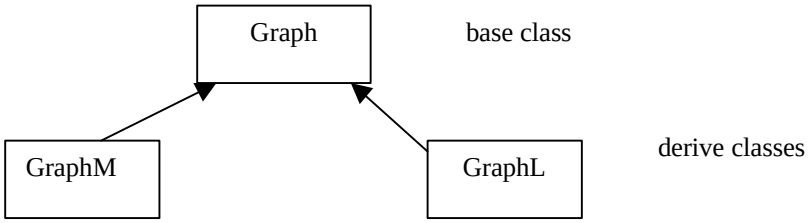
When an edge is added keep track of the number of indegrees and out degrees for each vertex. The vertex at the left gets it outdegree counter incremented. The vertex on the right gets it indegrees counter incremented. Supply functions **indegrees()** and **outdegrees()** that return the number of indegrees or outdegrees a vertex has.

C++ PROGRAMMERS GUIDE LESSON 16

File:	CppdsGuideL16.doc
Date Started:	April 15,1999
Last Update:	Dec 27, 2001
Status:	draft

LESSON 16 GRAPH CLASS

From the previous lesson we have the GraphM class implementing a graph by an adjacency matrix, and the GraphL class implementing a graph by an adjacency list. Our next step is to create a Graph base class that will have all the common operations for both the GraphM and GraphL classes. Both the GraphM class and the GraphL class will be derived from the base class Graph.



The Graph class will define and declare all the common functions but only declare the functions to be implemented by the derived classes. There functions will be pure virtual. Each derived class will define functions for use with the adjacency matrix or for use with the adjacency list

Implementing the Graph Class

To implement the Graph algorithms we need additional functions. The graph class needs additional member variables and functions for keeping track of the vertex nodes, the paths, visited nodes, weights of edges, indegrees and out degrees etc.

member	description
int maxDist;	max distance between any vertex
int size;	max number of vertices this graph can handle
int n;	number of vertices in graph
Vertex* v;	array of vertices names
int* d;	array of distances for each vertex
Vertex* vs;	array of visited vertices
Vertex* p;	array of paths of vertices
int* indeg;	array of in degrees for each vertex
int* outdeg;	array of out degrees for each vertex

To avoid duplication we put all the common functions in the Graph base class. Use the following defs.h file for common definitions used for all classes:

```
// defs.h
#ifndef __DEFS
#define __DEFS
#define sp " "
//typedef int bool;
//enum {false,true};
typedef char Vertex;
typedef int Edge;
#endif
```

Here's the graph class.

```
/* graph class */
/*graph.hpp */
#ifndef __GRAPH
#define __GRAPH
#include "list.hpp"
class Graph
{
protected:
int maxDist; /* max distance */
int size; /* max size of  graph */
int n; /* number of vertices in graph */
Vertex* v; /* array of vertices names */
int* d; /* array of distances */
Vertex* vs; /* array of visited vertices */
Vertex* p; /* array of paths vertices */
int* indeg;/* array of in degrees */
int* outdeg;/* array of out degrees */
public:
/* initializing constructor */
Graph (int size, int maxDist );
/* return number vertices */
int Graph::numVertices();

/* add edge to a graph */
virtual bool addEdge(Vertex v1, Vertex v2, int w)=0;
/* add vertex to graph */
virtual bool addVertex(Vertex v)=0;
/* find a vertex in a graph */
virtual int findVertex(Vertex v)=0;
/* get an edge weight from a graph */
virtual Edge getEdge(Vertex v1, Vertex v2)=0;
// get vertex in list at iterator
virtual Vertex getVertex(Vertex v)=0;
/* get weight between current adjacent vertices */
virtual Edge getWeight(Vertex v)=0;
/* iterate through adjacency vertices */
virtual bool iterate (Vertex v )=0;
/* get next adjacent vertex */
virtual Vertex nextVertex(Vertex v)=0;
/* get next adjacent vertex from v not in list */
```

```
virtual Vertex nextVertexList(Vertex v,List& list)=0;
/* remove edge between to vertices */
virtual bool removeEdge(Vertex v1, Vertex v2)=0;
/* set edge to a graph */
virtual bool setEdge(Vertex v1, Vertex v2, Edge w)=0;
/* clear all visited vertices */
void clearVisited();
// find vertex in graph
int findVertex(Vertex v);
/* return in degrees for this vertex */
int inDegrees(Vertex v1 ) ;
/* return out degrees for this vertex */
int outDegrees(Vertex v1 ) ;
/* get distance for vertex */
int getDist(Vertex v);
/* return max distance */
int getMaxDist ( );
/* initialize distance and path arrays */
bool initDist(Vertex v);
/* check if vertex is visited */
bool isVisited(Vertex v);
/* find minimum unknown distance */
Vertex minXdist();
/* get path at graph index */
Vertex pathAt(int i);
/* set distance for vertex */
bool setDist(Vertex v,int d);
/* set path */
bool setPath(Vertex w,Vertex v);
/* set vertex as being visited */
bool setVisited(Vertex v);

/* get vertex at graph index */
Vertex vertexAt(int i);
/* print graph */
void print();
}; /* end class graph */
```

Here is Graph class implementation code:

```
/* graph.cpp */
#include "graph.hpp"
/* initializing constructor */
Graph::Graph (int size, int maxDist )
{
this->size = size;   /* size of graph list */
this->n = 0;  /* number of vertices in graph */
this->maxDist = maxDist; /* maximum distance */
this->d = new int[size]; /* array of distances */
this->vs = new Vertex[size]; /* array of visited vertices */
this->p = new Vertex[size]; /* array of paths */
this->indeg = new int[size]; /* array of in degrees */
```

```
this->outdeg = new int[size]; /* array of out degrees */
for(int i=0;i<size;i++) /* clear */
        {
        indeg[i]=0;
        outdeg[i]=0;
        }
}
/* return number vertices */
int Graph::numVertices()
{
return n;
}
/* clear all visited vertices */
void Graph::clearVisited()
{
int i;
for(i=0;i<n;i++)vs[i] = 0;   /* set to not visited */
}
/* get distance for vertex */
int Graph::getDist(Vertex v)
{
int i = findVertex(v); /* check if exists */
if(i < 0)return  0; /* no distance */
return d[i];  /* return distance */
}
/* return max distance */
int Graph::getMaxDist ( )
{
return maxDist;
}
/* initialize distance and path arrays */
bool Graph::initDist(Vertex v)
{
int i;
for(i=0;i<n;i++)
{
d[i] = maxDist;/* set all distances to maximum */
p[i] = ' ';  /* clear all paths */
}
/* check if vertex exists */
i = findVertex(v);
if(i < 0)return false;
d[i]= 0; /* set vertex distance to zero */
return true;  /* success */
}
/* check if vertex is visited */
bool Graph::isVisited(Vertex v)
{
int i = findVertex(v);  // find vertex position
if(i >= 0) return vs[i] != 0;  // check if vertex visited
else return false;  /* vertex not in graph */
}
```

```
/* find minimum unknown distance */
Vertex Graph::minXdist()
{
int i;
int mind = 0;  /* set to minimum distance  */
int mini = -1;  /* set to not found inde4x */
/* set min to maximum distance */
for(i=0;i<n;i++)if(d[i] > mind)mind = d[i];
/* loop through table */
for(i=0;i<n;i++)
        {
        /* check if visited */
        if(vs[i] == 0)
                {
                /* keep track of minimum distance and exit */
                if(d[i] < mind)
                {
                mini = i;  /* store minimum index */
                mind = d[i];  /* store minimum value */
                }
        }
}

if(mini < 0)return ' ';  /* no minimum distance found */
return v[mini]; /* return vertex with minimum distance */
}
/* get path at graph index */
Vertex Graph::pathAt(int i)
{
return p[i];
}
/* set distance for vertex */
bool Graph::setDist(Vertex v,int d)
{
int i = findVertex(v); /* check if exits */
if(i < 0)return false;
this->d[i]= d; /* set distance */
return true;  /* success */
}
/* set path */
bool Graph::setPath(Vertex w,Vertex v)
{
int i,j;
i = findVertex(w);  /* check if exists */
if(i < 0)return false;
p[i] = v;  /* put vertex in path */
return true;  /* success */
}
```

```
/* set vertex as being visited */
bool Graph::setVisited(Vertex v)
{
int i = findVertex(v); /* find vertex position */
if(i >= 0)
                {
                vs[i] = 1;  /* set vertex visited */
                return true;  /* success */
        }
else return false;  /* vertex not in graph */
}
/* get vertex at graph index */
Vertex Graph::vertexAt(int i)
{
return v[i];
}
/* return in degrees for this vertex */
int Graph::inDegrees(Vertex v1 )
{
int i = findVertex(v1); /* check if exists */
if(i < 0)return  0; /* no distance */
return indeg[i];  /* return distance */
}
/* return out degrees for this vertex */
int Graph::outDegrees(Vertex v1 )
{
int i = findVertex(v1); /* check if exists */
if(i < 0)return  0; /* no distance */
return outdeg[i];  /* return distance */
}
```

IMPLEMENTING DERIVED CLASSES

The GraphM class has all the functions to implement a graph using a adjacency matrix. The GraphL class has all the functions to implement a graph using an adjacency list. Each derived class will have an iterator. An iterator points to a start of a List. There are functions to retrieve the node at an iterator or to increment the iterator and return the next node. **Iterate()** sets the iterator to the first vertex of an adjacency list, **getVertex()** will get the vertex at that node and **nextVertex()** will increment the iterator and return the vertex pointed to by the iterator. We need the **getVertex()** function because we may have to get additional data from the iterator node like the weight of a edge.

GRAPHM DERIVED CLASS

The Graphm class implements an adjacency matrix. Here is the header and implementation files.

```
// graphm.hpp
#include <iostream.h>
#include "graph.hpp"
#include "defs.h"
```

```
// class to  implement a graph using an adjacency matrix
class GraphM:public Graph
{
private:
int** m; /* pointer to array rows of columns */
int* itr; /* list of iterators */
public:
/* initialize graph */
GraphM (int size, int maxDist );
/* add vertex to graph */
bool addVertex(Vertex v1);
/* add edge to a graph */
bool addEdge(Vertex v1, Vertex v2, Edge w);
/* find a vertex in a graph */
int findVertex(Vertex v1);
/* get an edge weight from a graph */
Edge getEdge(Vertex v1, Vertex v2);
/* get vertex at iteration point */
Vertex getVertex(Vertex v);
/* get weight between current adjacent vertices */
Edge getWeight(Vertex v);
/* iterate through adjacency vertices */
bool iterate (Vertex v );
/* get next adjacent vertex */
Vertex nextVertex(Vertex v);
/* get next adjacent vertex from v not in list */
Vertex nextVertexList(Vertex v,List& list);
/* remove edge between to vertices */
bool removeEdge(Vertex v1, Vertex v2);
/* set edge to a graph */
bool setEdge(Vertex v1, Vertex v2, Edge w);
/* print graph */
void print();
};  /* end graphm */
/* Graphm implementation */
// graphm.cpp
#include <iostream.h>
#include "graphm.hpp"
/* initialize graph */
GraphM::GraphM (int size, int maxDist ): Graph(size,maxDist)
{
/* allocated memory for a 2 dimensional array */
m = new Edge*[size];
itr = new int[size];
for(int i=0;i<size;i++)
        {
        m[i]=new Edge[size]; // allocate for each row
        itr[i]=0;
        for(int j=0;j<size;j++)
        m[i][j]=0; // set elements to zero
        }
        }
```

```
 /* add vertex to graph */
bool GraphM ::addVertex(Vertex v1)
{
if(findVertex(v1)>=0)return false;
v[n] = v1;
n++;
return true;
}
/* add edge to a graph */
bool GraphM ::addEdge(Vertex v1, Vertex v2, Edge w)
{
int i = findVertex(v1);
int j = findVertex(v2);
if(i > n || j > n)return false;
if(i < 0 || j < 0)return false;
indeg[j]++;
outdeg[i]++;
m[i][j] = w;
return true;
}
/* find a vertex in a graph */
int GraphM ::findVertex(Vertex v1)
{
for(int i=0;i<n;i++)if(v1 == v[i])return i;
return -1;
}
/* get an edge weight from a graph */
Edge GraphM::getEdge(Vertex v1, Vertex v2)
{
int i = findVertex(v1);
int j = findVertex(v2);
if(i < 0 || j < 0)return 0;
return m[i][j];
}
/* get weight at current iteration */
Edge GraphM::getWeight(Vertex v)
{
int i = findVertex(v);
if(i < 0)return 0;
return m[i][itr[i]];
}
/* remove edge between to vertices */
bool GraphM::removeEdge(Vertex v1, Vertex v2)
{
int i = findVertex(v1);
int j = findVertex(v2);
indeg[i]--;
outdeg[j]--;
if(i < 0 || j < 0)return false;
m[i][j] = 0;
return true;
}
```

```
/* set iterator to start */
bool GraphM::iterate (Vertex v )
{
int i = findVertex(v);  /* get vertex position */
if(i < 0)return false;   /* vertex not found */
itr[i] = 0;  /* set pointer to first vertex */
return true;
}
/* get next adjacent vertex */
Vertex GraphM::getVertex(Vertex v1)
{
int i=findVertex(v1);
if(i >= 0 && itr[i] < n)
        {
        int k = 0;
        for(int j=0;j<n;j++)
                {
                if(m[i][j] != 0)
                        {
                        if(itr[i] == k)return v[j];
                        k++;
                        }
                }
        }
return ' ';
}
/* get next adjacent vertex */
Vertex GraphM::nextVertex(Vertex v1)
{
int i=findVertex(v1);
itr[i]++;
return getVertex(v1);
}

/* get next adjacent vertex not in list */
Vertex GraphM::nextVertexList(Vertex v1,List& list)
{
int j;
int i=findVertex(v1);  /* get vertex index */
if(i < 0)return ' ';  /* exit if vertex not in graph */
/* loop through all adjacent vertices */
for(j=0;j<n;j++)
        {
        if(m[i][j] != 0)   /* check if edge */
        /* return vertex if not in list */
        if(list.find(v[j])==NULL)return v[j];
        }
return ' ';  /* no more vertices */
}
```

```
/* set edge to a graph */
bool GraphM::setEdge(Vertex v1, Vertex v2, Edge w)
{
int i = findVertex(v1);
int j = findVertex(v2);
if(i < 0 || j < 0)return false;
m[i][j] = w;
return true;
}
/* print out a graph */
void GraphM::print()
{
int i;
cout << "  ";
// loop through all vertices
for(i=0;i<n;i++)cout << v[i] << sp; /* print out vertices names */
cout << endl;
// loop through all vertices
or(i=0;i<n;i++)
        {
        cout << v[i] << ": ";
        for(int j=0;j<n;j++)
        cout << m[i][j] <<  sp; // print out edge weights
        cout << endl;
        }
cout << endl;
}
```

GRAPHL DERIVED CLASS

The GraphL class implements an adjacency list. Here is the header and implementation files.

```
// graph list class
/* graphl.hpp */
#include "graph.hpp"
#include "glist.hpp"
#include <iostream.h>
class GraphL:public Graph
{
private:
GList* h; /* array of GList objects */
public:
/* allocate memory for a graph */
GraphL(int size, int maxDist);
/* add vertex to graph */
bool addVertex(Vertex v1);
/* add edge to a graph */
bool addEdge(Vertex v1, Vertex v2, Edge w);
/* find a vertex in a graph */
int findVertex(Vertex v1);
/* print out a graph */
friend ostream& operator<<(ostream& out, GraphL& g);
```

```
/* get an edge weight from a graph */
Edge getEdge(Vertex v1, Vertex v2);
/* get vertex at iteration point */
Vertex getVertex(Vertex v);
/* get weight between current adjacent vertices */
Edge getWeight(Vertex v);
/* iterate through adjacency vertices */
bool iterate (Vertex v );
/* get next adjacent vertex */
Vertex nextVertex(Vertex v);
/* get next adjacent vertex from v not in list */
Vertex nextVertexList(Vertex v,List& list);
/* remove edge between to vertices */
bool removeEdge(Vertex v1, Vertex v2);
/* set edge to a graph */
bool setEdge(Vertex v1, Vertex v2, Edge w);
/* print graph */
void print();
}; /* end graphl */


/* graphl class implementation */
//graphl.cpp
#include <iostream.h>
#include "graphl.hpp"
/* allocate memory for a graph */
GraphL::GraphL(int size, int maxDist):Graph(size,maxDist)
{
/* allocated memory for an array of GList's */
h = new GList[size];
}
/* add vertex to graph */
bool GraphL::addVertex(Vertex v1)
{
int i = findVertex(v1);
if(i >= 0)return false;
if(n>=size)return false;
v[n] = v1;
h[n++].v = v1;
return true;
}
/* add edge to a graph */
bool GraphL::addEdge(Vertex v1, Vertex v2, Edge w)
{
int i = findVertex(v1);
int j = findVertex(v2);
if(i > n || j > n)return false;
indeg[j]++;
outdeg[i]++;
h[i].insertTail(new GList::GNode(v2,w));
return true;
}
```

```
/* find a vertex in a graph */
int GraphL::findVertex(Vertex v1)
{
for(int i=0;i<n;i++)if(v1 == h[i].v)return i;
return -1;
}
/* get an edge weight from a graph */
Edge GraphL::getEdge(Vertex v1, Vertex v2)
{
int i = findVertex(v1);
if(i < 0)return 0;
Node* n =  h[i].find(v2);
if(n!= NULL)
return ((GList::GNode*)n)->w;
else return 0;
}
/* get weight between current adjacent vertices */
int GraphL::getWeight(Vertex v)
{
int i = findVertex(v);
if(i < 0)return 0;
return h[i].itr->w;
}
/* set iterator to start of this list */
bool GraphL::iterate(Vertex v)
{
int i = findVertex(v);
if(i < 0)return false;
h[i].itr = (GList::GNode*)h[i].head;
return true;
}
/* get vertex at iterator */
Vertex GraphL::getVertex(Vertex v)
{
int i = findVertex(v);
if(i < 0)return ' ';
if (h[i].itr == NULL)return ' ';
v = h[i].itr->data;
return v;
}
/* increment iterator, get vertex at iterator */
Vertex GraphL::nextVertex(Vertex v)
{
int i = findVertex(v);
if(i < 0)return ' ';
if (h[i].itr == NULL)return ' ';
h[i].itr = (GList::GNode*)h[i].itr->next;
if (h[i].itr == NULL)return ' ';
return h[i].itr->data;
}
```

```
/* get next adjacent vertex from v not in list */
Vertex GraphL::nextVertexList(Vertex v,List& list)
{
Node* n;
int i=findVertex(v);  /* find vertex in graph */
if(i < 0)return ' ';  /* exit if vertex not in graph */
n = h[i].head;  /* point to start of list */
/* loop through all adjacent vertices */
while(n != NULL)
        {
        if(list.find(n->data)==NULL)return n->data; /* check if vertex in list */
        n = n->next;   /* get next vertex in list */
        }
return ' ';  /* no more vertices in list */
}
 /* remove edge between to vertices */
bool GraphL::removeEdge(Vertex v1, Vertex v2)
{
int i = findVertex(v1);
int j = findVertex(v2);
if(i < 0 || j < 0)return false;
h[i].remove(v2);
return true;
}
/* set edge to a graph */
bool GraphL::setEdge(Vertex v1, Vertex v2, int w)
{
int i = findVertex(v1);
int j = findVertex(v2);
if(i < 0 || j < 0)return false;
Node* n = h[i].find(v2);
if (n == NULL)return false;
((GList::GNode*)n)->w = w;
return true;
}
/* print out a graph */
void GraphL::print()
{
cout << endl;
// loop for each list
for(int i=0;i<n;i++)
        {
        cout << h[i].v << ": ";
        cout << (GList)h[i];
        }
cout << endl;
}
```

LINK LIST CLASS

The List class is now the base class and the Glist class is derived from the List class. The List class will use the inner Node class to store names of a vertex, and the Glist class will use the GNode derived from Node to store the value of an edge weight.

```
// list.hpp
#ifndef __LIST
#define __LIST
#include "defs.h"
#include <iostream.h>
typedef Vertex NodeData;
// link list class
class List
        {
        public:
        // Node class
        struct Node
        {
        Node* next; // pointer to next node
        NodeData data;    // data in node
        // initialize constructor
        Node(NodeData data, Node* next)
        {
        this->next = next;
        this->data = data;
        }
        // initialize constructor
        Node(NodeData data)
        {
        this->next = NULL;
        this->data = data;
        }
        bool operator == (NodeData data)
        {
        return this->data==data;
        }
        };

        public:
        Node* head;    // pointer to start of list
        Node* tail;    // pointer to end of list

        // default constructor
        List();
        // find data item in list
        // if found return node in list
        // if not found return null
        Node* find(NodeData data);
```

```
/* insert in front of list */
void insertHead(Node* node);
// insert at end of list
void insertTail(Node* node);
// remove item from list
bool remove(NodeData data);
}; // end class list
```

List class implementation

```
/* list.cpp */
#include "list.hpp"

// default constructor
List::List()
{
head = NULL;
tail = NULL;
}

// find data item in list
// if found return node in list
// if not found return null
Node* List::find(NodeData data)
{
List::Node* curr = head; // point to start of list
// loop through list
while(curr != NULL)
        {
        // stop when data item found
        if(curr==data)return curr;
        curr=curr->next; // point to next node
        }
return NULL; // data item not found
}

/* insert in front of list */
void List::insertHead(Node* node)
        {
        /* check for empty list */
        if(head == NULL)
        {
        head = node;
        tail = node;
        }
        else
        {
        node->next = head;
        head=node;
        }
        }
```

```
// insert at end of list
void List::insertTail(Node* node)
{
/* check for empty list */
if(head == NULL)
                {
                head = node;
                tail = head;
                }
        else
                {
                tail->next=node;
                tail=node;
                }
        }


// remove item from list
bool List::remove(NodeData data)
{
List::Node* prev = NULL;   // pointer to previous node
List::Node* curr = head;   // pointer to current node
// find data to delete, loop till end of list
while(curr != NULL)
        {
        if(curr==data)break; // exit loop if data item found
        prev = curr;     // save previous node
        curr=curr->next; // current is now next node
        }
// item not found
if(curr == NULL)return false;
// curr is the node to delete, prev is the node before curr
// check for only one item in list
if(head==tail)
        {
        head=NULL; // set start of list pointer to NULL
        tail=NULL; // set end of list pointer to NULL
        }
 // check for end of list
else if(curr->next == NULL)
        {
        prev->next=NULL; // set previous node to point to NULL
        tail = prev;     // set end of list to previous node
        }
// check for start of list and  set start of list to point to next node
else if(prev == NULL) head=curr->next;
// check for middle of list
else prev->next = curr->next; // set previous node to point to current next node
return true;
}
```

```
// print list
ostream& operator << (ostream& out, List& list)
{
List::Node* curr = list.head;   // pointer to current node
out << "[ ";
while(curr != NULL)
        {
        cout << curr->data << " ";
        curr=curr->next; // current is now next node
        }
out << "]" << endl;
return out;
}
```

GList class

The Glist class is used in the GraphL class to store all the adjacent vertexes. The Glist is derived from List class and uses the GNode class for its nodes. This GNode class is needed to store names of a vertex. and the weight of the edge between this vertex and the another vertex. Since it is a template class the uses has the option of specifying what data types ate used for the vertices name and edge weight.

```
// glist.hpp
#include <iostream.h>
#include "list.hpp"
struct GNode:public Node
{
public:
Edge w; // weight
// initialize vertex name and weight
GNode((Vertex v, Edge w):Node(v)
{
this->w = w; // set weight
}
bool operator ==(GNode& n2){ return data == n2.data; }
};
// link list class for graph
class GList: public List
{
public:
GNode* itr;
Vertex v; // vertex name
        // initializing constructor
        GList ();
        };
```

```
// glist.cpp
#include <iostream.h>
#include "glist.hpp"
// initializing constructor
GList ():List()
{
v = 0;
itr=NULL;
}
/* print list */
ostream& operator << (ostream& out, GList& list)
{
Node* curr = list.head;   // pointer to current node
out << "[ ";
while(curr != NULL)
        {
        cout << curr->data << "(" << ((GList::GNode*)curr)->w <<") ";
        curr=curr->next; // current is now next node
        }
out << "]" << endl;
return out;
}
```

LESSON 16 EXERCISE 1

Type in or edit copy paste the Graph class. You need to update the GraphM and GraphL, GList and List classes from last lesson. The GraphM and GraphL classes are derived from the Graph class. Use your main program from previous Lesson and make sure everything still runs. The big difference now is your make a graph object instantiating a GraphM class or a GraphL class.

```
// create a graph as a adjacency matrix
Graph* g = (Graph*) new GraphM(5,1000);

// create a graph as a adjacency list
//Graph* g = (Graph*) new GraphL(5,1000);
```

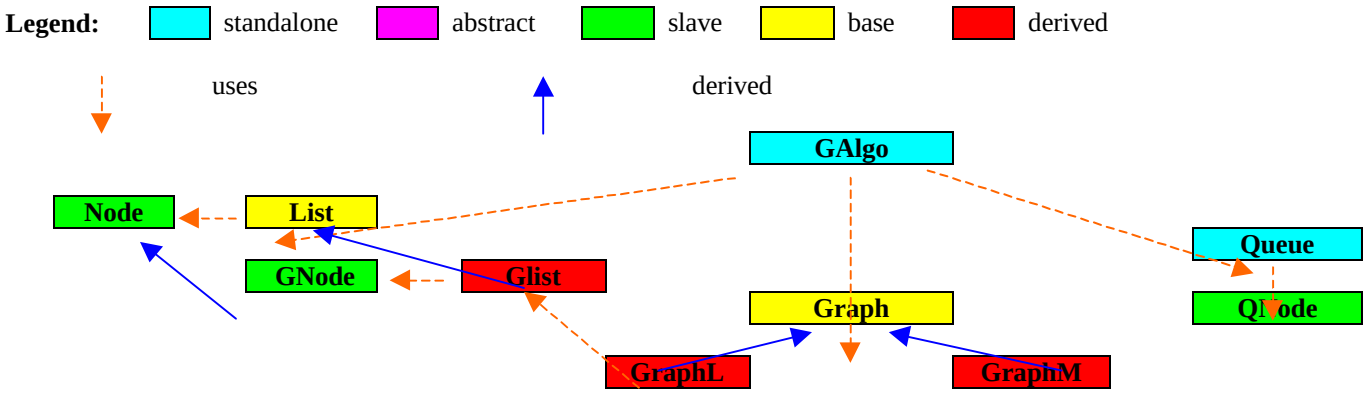
Make sure you can create both types of graphs, graphs that implement adjacency lists with matrix or link lists. Put your main function in L16ex1.cpp.

GRAPH CLASSES HIERACHY

In the next lesson we will create the **GAlgo** class that will have all the graph algorithms for traversing graphs, finding the shortest and longest paths and sorting graph vertices in a certain order. The GraphAlgo class uses the services of the List, Queue and Stack classes. The GraphL class also uses the services of the List class. This application is a good example demonstrating the power of abstract classes. We now have a Graph class that can handle graphs using adjacency matrix or adjacency lists and a List class that will handle any kind of Node to represent any kind of data. We now have all the following classes.

class	type	description
GAlgo	standalone	all graph algorithms
Graph	base class	all common graph operations
GraphM	derived class	graph operations using adjacency matrix
GraphL	derived class	graph operations using adjacency list
List	base class	link list uses Node class
Node	slave class	nodes for list class
GList	derived class	link list for GraphL class inherits List class
GNode	slave class	node for List class
Queue	standalone	implements a Queue using link lists
QueueNode	slave class	store data nodes for Queue

Graph class programming system model:



QUEUE CLASS

The graph algorithms need a queue class. In case you lost or misplace your queue class here is a simple one for you,

```
// queue.hpp
/* queue list class */
#include "defs.h"
#include <iostream.h>
typedef Vertex QueData;
class Queue
{
private:
/* queue node structure */
struct QNode
{
QueData  data;
QNode* next;
```

```
        // initializing constructor
        QNode(QueData data,QNode* next)
        {
        this->data = data;
        this->next = next;
        }
        };
        QNode* head; // point to start of list
        QNode* tail; // point to end of list
        public:
        // initializing constructor
        Queue();
        /* insert items into end of queue list */
        void enqueue(QueData data);
        /* remove items from start of queue */
        QueData dequeue();
        /* test if Queue is empty */
        bool isEmpty();
        /* print out queue items */
        friend ostream& operator<<(ostream& out,Queue& que );
        };  // end queue class
```

Queue class Implementation

```
/* queue.cpp */
#include "queue.hpp"

        // initializing constructor
        *Queue::Queue()
        {
        head = NULL;
        tail = NULL;
        }
        /* insert items into end of queue list */
        void Queue::enqueue(QueData data)
        {
        QNode* node = new QNode(data,NULL); /* allocate memory for queue node */
        // insert at end of queue
        if(tail != NULL)tail->next = node;
        else head = node; // new list
        tail = node;
        }
        /* remove items from start of queue */
        QueData Queue::dequeue()
        {
        /* check if queue empty */
        if(head == NULL)return 0;
        QueData data = head->data;
```

```
/* remove from item at head of queue */
QNode* node = head->next; // refer to next node
head->next=NULL; // remove link
head = node; // head point to next node
if(node == NULL)tail = NULL;  /* queue is empty */
return data;
}

/* test if queue is empty */
bool Queue::isEmpty()
{
return head==NULL;
}

/* print out queue items */
ostream& operator<<(ostream& out,Queue& que )
{
out <<  "Queue: [ ";
Queue::QNode* qnode = que.head;
while(qnode != NULL)
      {
      out << qnode->data <<  " ";
      qnode=qnode->next;
      }
out << "]" << endl;
return out;
}
```

STACK CLASS

The graph algorithms need a stack class. In case you lost or misplace your stack class here is a simple one for you:

```
// stack.hpp
#include "defs.h"
#include <iostream.h>
typedef Vertex StackData;
/* stack class */
class Stack
{
private:

// StackNode structure
struct StackNode
{
public:
StackData data;
StackNode* next;
```

```
// initialize stack
StackNode(StackData data, StackNode*  next)
{
this->data = data;
this->next = next;
}
}; /* end stack node class */
StackNode* stkptr;  // point to last item in stack
public:
// initializing constructor
Stack();
/* push item into stack */
void push(StackData data);
/* get item from stack */
StackData pop();
/* return true if stack empty */
bool isEmpty();
/* print stack items */
friend ostream& operator<<(ostream&,Stack& stk );
};
```

Stack class implementation

```
/* stack.cpp */
#include "stack.hpp"

// initializing constructor
Stack::Stack(){stkptr = NULL;}

/* push item into stack */
void Stack::push(StackData data)
{
/* insert new node at stkptr location */
StackNode* node = new StackNode(data,stkptr);
stkptr = node; // set stkptr to new node
}

/* get item from stack */
StackData Stack::pop()
{
if(stkptr == NULL)return 0; /* check if stack empty */
StackData data = stkptr->data; // get data
/* remove from item at next stkptr location */
StackNode* node = stkptr->next; // refer to next node
stkptr->next = NULL; // remove link
stkptr = node; // stkptr refers t0o next node
return data; // return data
}
```

```
/* return true if stack empty */
bool Stack::isEmpty()
{
return stkptr==NULL;
}

/* friend functions */
ostream& operator<<(ostream& out,Stack& stk )
{
out << "[ ";
Stack::StackNode* node = stk.stkptr;
// loop through all nodes in stack
while(node != NULL)
{
out << node->data << " ";
node = node->next;
}
out << "]" << endl;
return out;
}
```

LESSON 16 EXERCISE 2

Add and test the Stack and Queue classes to your project. Call your project L16ex2. Make sure everything compiles with no mistakes

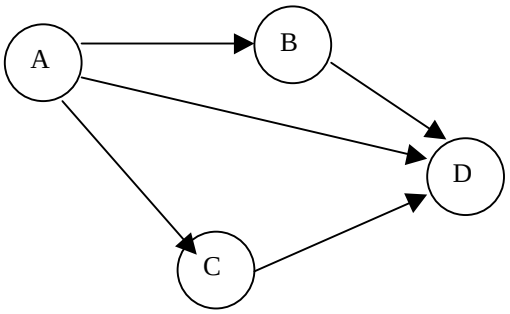
C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 17

File:	CppdsGuideL17.doc
Date Started:	April 15,1999
Last Update:	Dec 26, 2001
Status:	draft

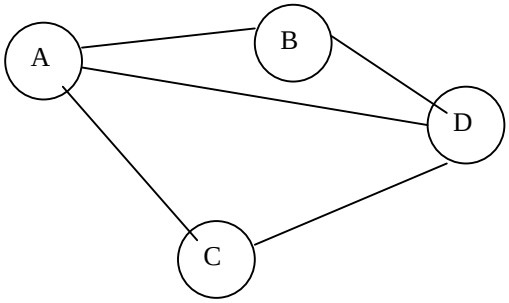
LESSON 17 GRAPH ALGORITHMS

To do operations on graphs we need an **algorithm.** An algorithm is a set of step by step instructions stating how to do what we want to accomplish. What sort of things do we need to do ? We may want to traverse a graph, find the shortest or longest path of a graph from a start vertex to an end vertex and traverse vertices in a particular order There are many operations to do on graphs, for each operation you need a specific algorithm. For which algorithm to use for a specific graph can be sort of confusing. We have two basic types of graphs directed graphs and undirected graphs Each edge in a directed graph has a forced direction ordering between a adjacent vertex, either from left to right or from right to left. Undirected graphs assume edges between adjacent vertices can travel in either direction.

directed graph



undirected graph



The best thing to do for us is to make a chart stated which kind of graph and what we want to do. A chart will help us select the appropriate algorithm to use for a specific graph. The following chart, lists the graph algorithms we will cover in this lesson. Most of the algorithms expect directed graphs. There are two graph traversal algorithms **depth first search dfs** and **breath first search bfs**. **Dfs** traverses by depth, it follows a path as far as it can go. **Bfs** traverses by breadth, meaning it follows all paths vertex by vertex. There are two **shortest path** algorithms one for **un-weighted** graphs called **shortest** path and one for and **weighted** graphs called **dikstras**. Dikstras algorithm tries to find the shortest path having total minimum weights. Our next algorithm is to find the **longest path** in a graph. Why would you need to know the longest path ? There may be instances if you have a graph representing cites and you may need to know the maximum number of cites that can be visited starting from a particular city. . Finally we have **topological sort** which gives a ordering of which vertex needs to be visited first. The most famous example of a topological sort is which courses a student has to take before and you can take other courses. Each course depends on pre-resiquites courses. You need to take the pre-resequite courses first.

algorithm	description	directed	undirected	handle cycles	weighted
depth first search	depth first traversal	yes	?	?	no
breath first search	breath first traversal	yes	?	?	no
shortest path	find shortest path	yes	?	?	no
weighted shortest path (dikstras)	find shortest path that has different weights	yes	?	?	yes
longest path	find longest path in a graph	yes	?	?	no
topological sort	find which vertex must be visited first to preserve a ordering	yes	yes	?	no

We will put all the graph algorithms in class called **GAlgo**. Each algorithm's will be a function of the **GAlgo** class. By using a separate class for the algorithms we can constructs graphs using adjacency matrix or adjacency list. It is very important the algorithms are independent of data structures. The algorithms will call functions from the graph or graph list classes. The graph algorithms are not concerned if a graph is represented an adjacency matrix or adjacency list. We have listed all the graph algorithm functions in the following chart. You will notice some additional supporting methods. We have two supporting methods **printPaths()** and **printPath(). PrintPaths**()will print all paths results for each vertex in a graph, where as **printPath**() will just print a path result associated from a start vertex. Each vertex is represented by a char name for easy identification.

function	description
depth first search	find shortest path by searching by depth
breath first search	find shortest path by searching by breadth
shortest path	
unweighted shortest path (dikstra's)	find shortest path that has different weights
longest path	find longest path in a graph
topological sort	find which vertex must be visited first to preserve a ordering
print paths	print all paths in graph
print path	print path from start vertex

```
// galgo.hpp
#include "graph.hpp"
#include "list.hpp"
#include "queue.hpp"
#include "stack.hpp"
// graph algorithm class
class GAlgo
{
private:
Graph* g;
public:
/* initialize graph */
GAlgo(Graph* g);
// depth first search;
void dfs(Vertex v, List& list);
```

```
/* breath first search */
void bfs(Vertex v, List& list);
/* shortest path */
bool shortestPath(Vertex v);
/* print out path */
void printPaths();
/* print out path */
void printPath(Vertex v);
/* dijkstra's shortest path algorithm for weighted graph's */
bool dijkstra(Vertex v1);
/* find longest path */
int longestPath(Vertex v1);
/* topological sort */
void topsort(Vertex v1, List& list);
// topological sort using a stack
void topSortStack (List& list);
 // topological sort sing a queue
void topSortQueue (List& list);
}; // end graph algorithms class
```

GAlgo implementation:

```
/* galgo.cpp */
#include <iostream.h>
#include "galgo.hpp"
#include "graph.hpp"
#include "graphm.hpp"
#include "graphl.hpp"
/* initialize graph */
GAlgo::GAlgo(Graph* g)
{
this->g=g;
}
```

We will now discus the operation of each graph algorithm separately.

GRAPH TRAVERSALS

A graph traversal visits every vertex from a start vertex to an end vertex. If the graph is connected all vertex will be visited. If the graph is not connected then the vertices only in the path will be visited,

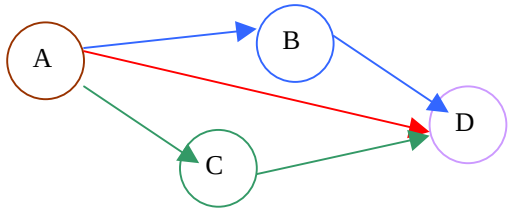
Threes are two types of traversals:

- depth first search
- breath first search

DEPTH FIRST SEARCH DFS

In depth first search we follow the path as far as possible, we then backtrack to search other path choices. The depth first search uses recursion. Recursion automatically supplies the back tracking mechanisms by using the built in execution stack. Here's the depth first search algorithm:

1. start at first vertex
2. mark each vertex wen visited
3. if at end of path then back track to preceding vertices
4. take next path
5. exit when all vertices have been visited



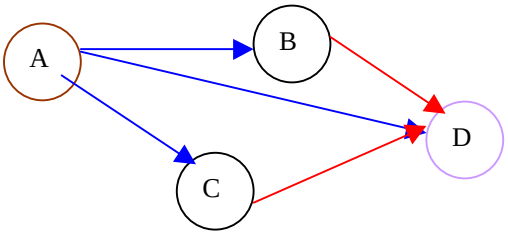
All nodes visited are kept in a list. The list is also used to print out the path.
The **dfs** algorithms use the **nextVertexList ()** method belonging to the Graph class. This function gets the next adjacent vertex not in the list. .All methods search by vertex name rather than by vertex index. An index is the position of the vertex in the graph vertex list. Using a vertex name is more convenient but may take up extra processing time to keep on finding the vertex index for a vertex by name. Here's the code for depth first algorithm:

```
void GAlgo::dfs(Vertex v, List& list)
{
list.insertTail(new Node(v));  // insert vertex in list
Vertex v2 = g->nextVertexList(v,list); //get next adjacent vertex not in list
/* loop till no more available adjacent vertices */
while(v2!=' ')
        {
        dfs(v2,list);  // call dfs to search path by depth
        v2 = g->nextVertexList(v,list); // get next adjacent vertex
        }
}
```

BREADTH FIRST SEARCH BFS

Breath first search visits all adjacency vertices first before visiting other vertices. We keep track of which vertices to visit in a queue and which vertices that have been visited in a list. The list is also used to print out the path.

The dfs algorithms use the nextVertexList() method belonging to the Graph class. This function gets the next adjacent vertex not in the list. .All methods search by vertex name rather than by vertex index. A index is the position of the vertex in the graph vertex list. Using a vertex name is more convenient but may take up extra processing time to keep on finding the vertex index for a vertex by name.



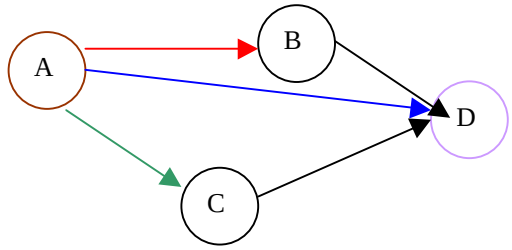
Here's the code for the breadth first algorithm:

```
/* breath first search */
void GAlgo::bfs(Vertex v, List& list)
{
Queue q;  /* make queue */
q.enqueue(v);   /* insert vertex into queue */
list.insertTail(new Node(v)); /* insert vertex into list */
/* loop while queue not empty */
while(!q.isEmpty())
        {
        v = q.dequeue();   /* get vertex from queue */
        /* get next adjacent vertex not in list */
        /* loop till no more available adjacent vertices */
        Vertex v2 = g->nextVertexList(v,list); while(v2!=' ')
                {
                q.enqueue(v2);   /* get vertex from queue */
                list.insertTail(new Node<V>(v2)); // insert vertex in list
                /* get next adjacent vertex not in list */
                v2 = g->nextVertexList(v,list);
                }
        }
}
```

SHORTEST PATH

The shortest path keeps tack of distances in a table. The distances in the table are pre-initialized to a maximum distance., meaning the distance have not been calculated yet. the vertices heaving the shortest paths are kept in another table. We use a queue to keep track of which vertices have calculated distances. The code is similar to the breath first search.

The shortest path algorithm also needs methods to initialize the distance table, set and get distances. These methods are found int the Graph class. The iterate() function sets a pointer to the start of an adjacency list and getVertex() gets the vertex at the iteration pointer, nextVertex() increments the pointer and gets the next vertex in the adjacency list. by using these supporting functions the graph algorithm class is data structure independent. This means theses methods can be used for graphs implemented adjacency matrix or for graph's implement adjacency lists.



```
bool GAlgo::shortestPath(Vertex v)
{
Queue q;  /* create queue */
/* initialize distance table with max distance */
if(!g.initDist(v))return false;
q.enqueue(v);   /* put vertex in queue */
/* loop while queue not empty */
while(!q.isEmpty())
        {
        v = q.dequeue();  /* get vertex from queue */
        g.iterate(v); /* get adjacent vertices from this vertex */
        V w = g.getVertex(v);  /* get fist adjacent vertex */
        // loop till no more vertices found in list
        while(w!= ' ')
                {
                /* get distance for this vertex */
                if(g.getDist(w)==g->getMaxDist() )
                        {
                        /* set w distance to v vertex + 1 */
                        g->setDist(w,g.getDist(v) + 1);
                        g->setPath(w,v);  /* set path w to v */
                        q->enqueue(w);  /* put w vertex in queue */
                        }
                w = g->nextVertex(v); // get next adjacency vertex
                }
        } /* end while */
return true;
}
```

Printing out paths

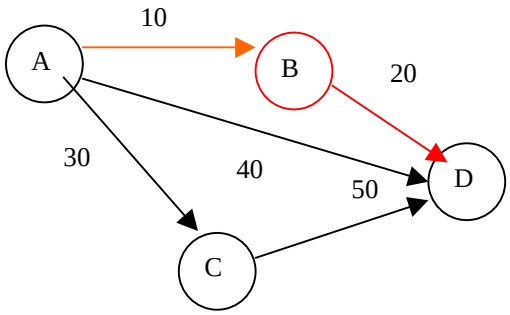
Here's are the functions to printing out the paths. The print paths functions will print out the paths all the paths for a start vertex. The printPath() function will print out the shortest paths from the start vertex to the specified end vertex. Both methods belong to the GraphAlgo class.

```
/* print out path */
void GAlgo::printPaths()
{
for(int i=0;i<g->numVertices();i++)
        {
        cout <<(g->vertexAt(i));
        printPath(g.pathAt(i));
        cout << endl;
        }
}
/* print out path */
void GraphAlgo::printPath(Vertex v)
        {
        int i = g->findVertex(v);
        if(i >= 0) printPath(g->pathAt(i));
        cout << v <<  sp;
        }
```

SHORTEST P
ATH WEIGHTED (Dikstra's algorithm)

When each edge has a weight then we need an algorithm to fin the path with the minimum weight.

Dijkstra's algorithm is similar to the un-weighted shortest path algorithm is that it uses a distance table to keep track of minimum accumulative distances and keeps track of visited node's instead of a queue. The dijkstra's algorithm uses additional functions from the graph and graph list modules. We an array of visited vertices and functions to clear and set the visited vertices. The dijkstras function also needs a function to find the minimum unknown distance in the distance table.

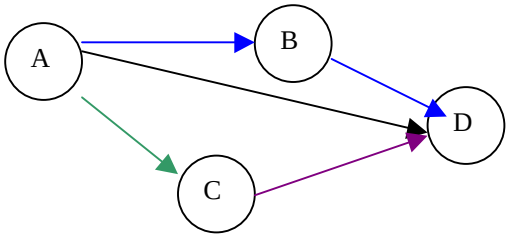


Here's the code for dikstras algorithm:

```
/* dijkstra's shortest path algorithm for weighted graph's */
bool GraphAlgo::dijkstra(Vertex v1)
{
/* set distance table to maximum distance */
if(!g.initDist(v1))return false;
g.clearVisited();    /* clear visited array */
/* loop till all vertices visited */
while(true)
        {
        v1 = g->minXdist();  /* get minimum unknown vertex distance */
        if(v1 == ' ')return true;  /* no vertices available */
        g->setVisited(v1);  /* set vertex visited */
        g->iterate(v1);  /* iterate on this vertex */
        Vertex w = g->getVertex(v1); // get first vertex in adjacent list
        /* loop till end of adjacent vertex  */
        while(w != ' ')
                {
                /* if w not visited */
                if(!g->isVisited(w))
                {
                /* check costs */
                if(g->getDist(v1) + g->getWeight(v1) < g->getDist(w))
                        {
                        /* update distance table */
                        /* to minimum accumulative distance */
                        g->setDist(w,g.getDist(v1) + g->getWeight(v1));
                        g->setPath(w,v1);  /* store path for print out */
                        }
                }
                w = g->nextVertex(v1); /* get next vertex */
                } /* end while w */
        } /* end while true */
} /* end dikstras */
```

LONGEST PATH

The longest path algorithm will search a graph and keep track of the longest possible path. This is a recursive routine that compares path choices from each vertex and chooses the longest and increments a counter. Using recursion the algorithm counts from the end of the paths to the beginning. The blue colored path is the choice for the longest path length.

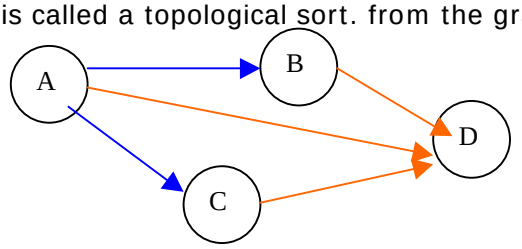


Here's is the code for the longest path algorithm:

```
/* find longest path */
int GAlgo::longestPath(Vertex v1)
        {
        int k = 0; /* check if a path found */
        int d; /* distance */
        int s = 0; /* path counter  */
        g->setVisited(v1); /* set all vertices unvisited */
        g->iterate(v1); /* set iteration to start of v */
        Vertex w = g->getVertex(v1); /* get first adjacent vertex */
        /* loop for all vertices w adjacent to v */
        while(w!=' ')
                {
                /* check if visited  */
                if(!g->isVisited(w))
                        {
                        k++;
                        d = longestPath(w); /* return longest path  */
                        if(d > s)s = d; /* store longest path */
                        }
                        w = g->nextVertex(v1); /* get next vertex w */
                        }
                g->clearVisited();
                if(k > 0)s++; /* increment if a longest path found */
                return s; /* return longest path length */
                }
```

TOPLOLOGICAL SORT

A directed graph with no cycles has an ordering so that no edged connects a vertex that proceed the edges travelling from. the ordering of vertex in sequence is called a topological sort. from the graph you must take C , B or A and then take D.
ALGORITHM:



1. find a vertex with no successor remove from list
2. place vertex inn a list

The topological sort solution is easy for us because we just use the depth first search algorithm and print out the findings in reverse. Here's the topological sort code using a dfs using recursion:

```
/* topological sort */
void GAlgo::topsort(Vertex v1, List& list)
{
/* get next vertex not in list */
Vertex v2 = g->nextVertexList(v1,list);
/* loop till all vertices visited */
while(v2!=' ')
        {
        topsort(v2,list);
        v2 = g->nextVertexList(v1,list); /* get next vertex not in list */
        }
list.insertHead(new Node(v1)); /* insert vertex in list */
}
```

Here's the topological sort code using a stack:

```
// topological sort using a stack
void GAlgo::topSortStack (List& list)
        {
        int* indeg=new int[g.numVertices()];
        Stack stk;
        Vertex v1;
        for(int i=0;i<g->numVertices();i++)
                {
                v1 = g->vertexAt(i);
                indeg[i] = g->inDegrees(v1);
                if(indeg[i]==0)stk.push(v1);
                }
        // loop till stack empty
        while(!stk.isEmpty())
                {
                v1 = stk.pop();
                list.insertTail(new Node(v1));
                g.iterate(v1);
                Vertex w = g->getVertex(v1);
```

```
            // loop till no more adjacent vertices
            while(w!=' ')
                    {
                    int i = g->findVertex(w);
                    indeg[i]--;
                    if(indeg[i]==0)stk.push(w);
                    w = g->nextVertex(v1);
                    }
        }
    delete[] indeg;
    }
```

Here's the topological sort code using a queue:

```
     // topological sort sing a queue
    void GAlgo::topSortQueue (List& list)
            {
            int* indeg=new int[g.numVertices()];
            Queue<V> q;
            Vertex v1;
            for(int i=0;i<g.numVertices();i++)
                    {
                    v1 = g.vertexAt(i);
                    indeg[i] = g->inDegrees(v1);
                    if(indeg[i]==0)q.enqueue(v1);
                    }
            // loop till stack empty
            while(!q.isEmpty())
                    {
                    v1 = q.dequeue();
                    list.insertTail(new Node<V>(v1));
                    g->iterate(v1);
                    V w = g.getVertex(v1);

                    /* loop till no more */
                    while(w!=' ')
                            {
                            int i = g->findVertex(w);
                            indeg[i]--;
                            if(indeg[i]==0)q.enqueue(w);
                            w = g->nextVertex(v1);
                            }
                }
            delete[] indeg;
            }
```

MAIN TEST PROGRAM:

Here's the main test program that tests all the graph algorithms. it works with either the graph as a adjacency matrix or the graph as an adjacency list.

```
// test graph algorithms
void main()
{
// create a graph as a adjacency matrix
Graph* g = (Graph*) new GraphM(5,1000);
// create a graph as a adjacency list
//Graph* g = (Graph*) new GraphL(5,1000);
GraphAlgo ga(g); // make graph algorithm object
// add vertices to graph
g->addVertex('a');
g->addVertex('b');
g->addVertex('c');
g->addVertex('d');
// add edges to graph
g->addEdge('a','b',1);
g->addEdge('a','c',1);
g->addEdge('a','d',1);
g->addEdge('b','d',1);
g->addEdge('c','d',1);

g.print(); // print out graph
cout << "testing graph algorithms" << endl;
/* test dfs */
List list1; // make list to store results
ga.dfs('a', list1);
cout << "depth first search: " << endl;
cout << list1;// print out results

/* test bfs */
List list2; // make list to store results
ga.bfs('a',list2);
cout << "breath first search: " << endl;
cout << list2; // print out results
/* test shortest path */
g->clearVisited();
ga.shortestPath('a');
cout << "shortest path:" << endl;
ga.printPaths();// print out results
/* test weighted shortest path (dijkstra's) */
cout << " adding weights to edges" << endl;
g->setEdge('a','b',10);
g->setEdge('a','c',30);
g->setEdge('a','d',40);
g->setEdge('b','d',20);
g->setEdge('c','d',50);

g.print();// print out graph
```

```
ga.dijkstra('a');
cout << "dijkstra's path:" << endl;
ga.printPaths();// print out results
/* test weighted longest path */
g->clearVisited();
int dist = ga.longestPath('a');
cout << "longest path: " << dist << endl;
/* test topological sort */
List list3; // make list to store results
ga.topsort('a',list3);
cout << "topological sort using recursion: " << endl;
cout << list3;
/* test topological sort */
List list4; // make list to store results
ga.topSortStack(list4);
cout << "topological sort using stack: " << endl;
cout << list4;
/* test topological sort */
List list5; // make list to store results
ga.topSortQueue(list5);
cout << "topological sort using queue: " << endl;
cout << list5;
 }
```

program output:

```
a b c d
a 0 1 1 1
b 0 0 0 1
c 0 0 0 1
d 0 0 0 0
e 0 0 0 0
testing graph algorithms :
depth first search: list: a b d c
breath first search: list: a b c d
shortest path:
a:
b:  a
c:  a
d:  a
e:
adding weights to edges
  a  b  c  d
a 0 10 30 40
b 0  0  0 20
c 0  0  0 50
d 0  0  0  0
e 0  0  0  0
dijkstra's path:
a:
b:  a
c:  a
d:  a b
e:
longest path: 2
topological sort: list: a c b d
```

LESSON 17 EXERCISE 1

Type in the GraphAlgo class. Test for both using the GraphM class and the GraphL class. make sure you get the same results.

LESSON 17 EXERCISE 2

Once you got each module working step through your program and trace the operation of each algorithm using a chart. The chart will be handy for tracing the operation of the algorithms using recursion.

LESSON 17 EXERCISE 3

Usr either dfs or bfs to test if a graph is completely connected. If not determine how many sub graphs there are.

LESSON 17 EXERCISE 4
Add a print table to the longest path algorithm so that we can print out the longest paths.

LESSON 17 EXERCISE 5
Test your algorithms from known graphs you find in text books or design yourself . See if you get the same answer as them.

C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 19

File:	CppdsGuideL19.doc
Date Started:	April 15,1999
Last Update:	Dec 27, 2001
Status:	draft

LESSON 19 SIMULATION TECHNIQUES I

We use simulation to simulate the behavior of a real world model. We need to do this so that we can get a basic idea how a real world system will behave. Typical things that are simulated are an elevator system, a bus route, a restaurant and of course a corporation that manufacture's vehicles. In this Lesson we will simulate the operation of an elevator system. When we simulate the behavior of an elevator system we will be interested in how long people have to wait to get on an elevator. A simulation program will indicate to us how many elevators we need to service the people waiting to use the elevator. The company who is building the office building and the company that is installing and manufacturing the elevators are willing to pay you lots of money if you can write a program that will simulate the actual operations of the elevator for this building. They will be most interested in elevator operation at rush hour and how many people have to wait too get a elevator. For every modeling **system** we need **inputs**, a **calculation** and an **output**. For the elevator the **inputs** will be the people who want to use the elevator, what time they arrive and what floor they want to go to. When they enter the office building then will go to many different floors. It is very difficult to predict which floor they want to go to. We can predict the floor the want to go to by using a random number generator. We also need to predict what time these people are going to arrive. We do know at start of the morning, lots of people will be travelling up the elevators. Are there enough elevators to move people ? What is the maximum time people need to wait to get an elevator? What is the maximum number of people that have to wait ? At lunch time people are coming down the elevators and after they have their lunch they come back up. At the end of the day these people need to go home. Again how long do theses people need to wait to get an elevator? How many people have to wait ?. The **calculations** are the statistics, the maximum average waiting time. The **output** is the statistics report. By predicting the flow of people we can measure if we have enough elevators.

RANDOM NUMBER GENERATOR FUNCTIONS

The following functions are used to generate random numbers. You must include <stdlib.h> when using the random number generator functions and <time.h> if using the **srand()** function.

prototype	description	example using
void srand(unsigned seed);	used to set the starting point for a sequence of random numbers. (seed with time of day)	long t = time(NULL); srand((unsigned) t);
void randomize();	initializes random number generator to some random number, based on the current time obtained from the computer.	randomize();
int rand();	generates a sequence of random numbers from 0 to RAND_MAX (65536)	x = rand();
int random(int num);	generates a random number between 0 and num	x = random(100);

using the number random generator

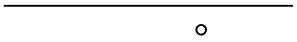
This program **seeds** the random number with the current time. "**seed**" means the random number generator will generator different numbers every time you run the program. If you do not **seed** the random number generator then you will get the same sequence of random numbers every time you run the program. We generate tree random numbers three different ways. The first random number generated is between 0 and 99. Why ? The second random number generated is between 1 and 100. Why ? The last number random number generated is between 0 to 1.0. and 0 to .9999999..... Why ?

```
/* lesson 19 program 1 */
#include <iostream.h>
#include <stdlib.h>
/* generate random numbers between 0 and 99 and 1 and 100 */
void main()
        {
        int num;
        float fnum;
        srand(time(NULL));  /* seed random number generator */
        num = rand() % 100; /* random number between 0 and 99 */
        cout << num << endl;   /* print out random number */
        num = (rand() % 100) + 1; /* random number between 1 and 100 */
        cout << num << endl;   /* print out random number */
        fnum = rand()/MAX_RAND; /* random number between 0 and 1.0 */
        cout << fnum << endl;   /* print out random number */
        fnum = rand()/(MAX_RAND-1); /* random number between 0 and 0.9999  */
        cout << fnum << endl;   /* print out random number */
        }
```

MEAN VARIANCE AND STANDARD DEVIATION

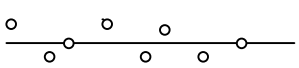
Before you can use random numbers you need to know a little bit about statistics. We will study terms like **mean**. **variance** and **standard deviation**. The **mean** (M) is the easy one it is simply the average of all the random numbers you have. v

$$\text{mean} = \frac{x1 + x2 + x3 + ... + xn}{n}$$



The next important statistical term is **variance** (V) . Variance is a measure of the **dispersion** or scattering of the random numbers about the mean. If all the values of the random numbers you have generated are close to the **mean** then the **variance** is small. If your collection of numbers **deviate** far from the **mean** then the variance is large. Variance is a measure that indicates how far the numbers deviate from the average. To calculate the variance we square each number subtracted from the mean and sum up the results.

$$\text{variance} = \sum_{i=0}^{n} \left((x_i - \text{mean}) \right)^2$$



The **standard deviation** (SD)is simply the square root of the variance.

$$sd = \sqrt{variance}$$

LESSON 19 EXERCISE 1

Write a program to generate 100 random numbers. Calculate the mean, the variance and the standard deviation of the random numbers you have generated.

DISTRIBUTIONS

Now you know what **mean**, **variance** and **standard deviation** are you will now be able to understand what **distributions** are. The numbers that we generate for our simulations will be very important and will depend on a distribution. Do not let the word distribution scare you. All distribution means is how the generated random numbers are going to be randomly generated. Are they going be centered around the middle of a certain range, at the end of a certain range or will be evenly uniform. Since numbers are randomly distributed we want to assign a probability that the random generated number will fall between a certain range.

distribution	description
Exponential	generated random numbers distributed exponentially to a end of a range
Poisson	represents the number of times that a random event occurs over a time interval .
Uniform	generated random numbers evenly distributed over a certain range
Gaussian	generated random numbers distributed over a range centered around middle of range
Random	generated random numbers with no specific distribution

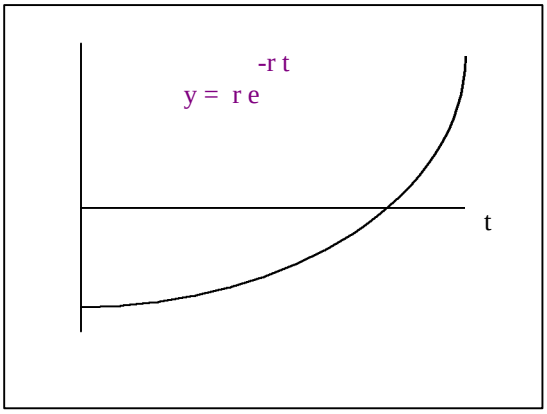
Exponential

Exponential means the random numbers are going to be exponentially distributed accordingly to a specified rate.

y = r exp(-r t)

where **r** is the rate between 0 and 1.0 and **t** is time > 0
we integrate from x to 0 to get
y = 1 - exp(-rx)
take ln of both sides:
ln y = ln 1 + rx ln(y-1) = -rx
solve for x
x = - ln(1.0 - y)/r

where y is a random generated number between 0 and 1.0 and x is a exponentially distributed number between 0 and 1.0



Poisson distribution

Poisson distribution represents the number of times that a random event occurs over a time interval .

$$p = \frac{(r\,t)^{k}\ exp^{-rt}}{k!}$$

r is a uniform rate
t is a time interval length
k is the number of occurrences in time t

The probability of having 2 events in a 6 minute period where events are happening at 5 events per hour would be:

$$p = \frac{(5)(.1)^{2}\ exp^{(-5)(.1)}}{2!} = .0758$$

Uniform

Uniform means the random generated numbers will be evenly distributed between a minimum value to a maximum value.

The formula is:

```
d = (a + (b - a)  *  (random() / (RANDOM_MAX - 1.0))
```

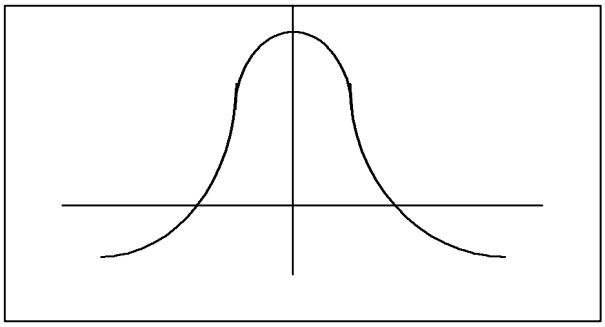
y = 1 / (b - a) where a <= x <= b

Gaussian

Gaussian is also called the normal distribution. This means the randomly generated numbers will concentrate in the center and gradually fall in both ends.

The equation for the gaussian distribution us quite complex:

$$y = \frac{1}{SD\ \sqrt{2 * PI}}\ e^{-(x - mean)^{2} / 2 * SD^{2}}$$



We have to approximate Gaussian distributiojn with the following algorithm. Can you figure out how it works ?

```
// Gaussian lesson 19 program 2
double gaussian()
{
double d1,d2,d,m;
/* loop till median found */
do
        {
        d1 = (double)rand()/(double)RAND_MAX;
        d2 = (double)rand()/(double)RAND_MAX;
        d1 = 2 * d1 - 1;
        d2 = 2 * d2 - 1;
        d = d1 * d1 + d2 * d2;
        }while(d >= 1);
m = sqrt(-2 * log(d)/d);    /* apply gaussian equation */
d = d2 * m;
return d;
}
```

Random

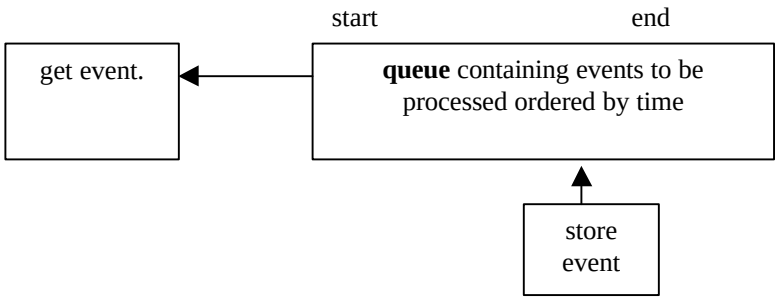
A random distribution is just random numbers with no concentration.

LESSON 19 EXERCISE 2

Make a random number generator module, that has the above distribution functions. Generate 100 random numbers for each distribution and write a function to print out the results for comparisons. See if you get the required distributions.

EVENT QUEUE

We need a mechanism to store events and then execute them at a later time. We will use a queue. A queue is ideal. Events that arrive first are the first to get processed. Events are executed from the head of the queue and events are added to the tail of the queue. We need an event queue because the program cannot execute events simultaneously. This means if we got three events at the same time the program cannot execute each event at the same time. Also events may be scheduled to happen in the future. The program need a place to store these events to be executed later. Events are placed into the queue in time order. Events to be serviced first are place at the beginning of the queue, where events to be executed later are placed at the end of the queue.



EVENTS

An event will be tagged with an arrival time, and the action identification event to perform.

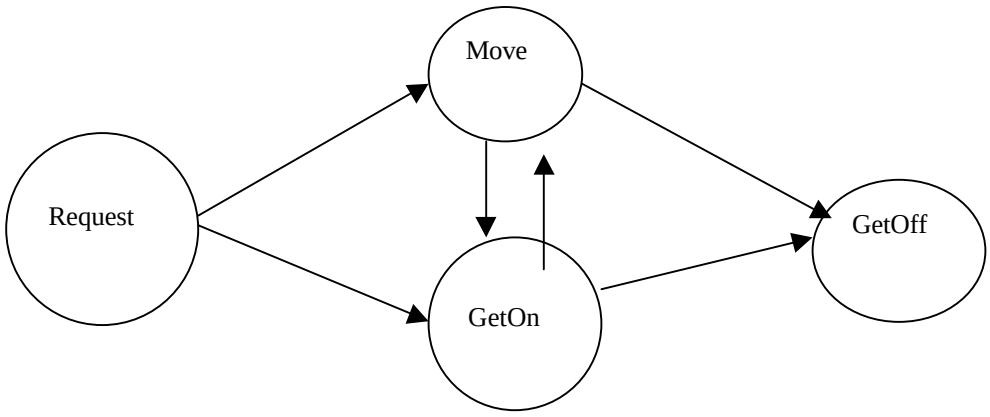
event	
arrival time	ID action to perform

What events do our elevators have ?
Every event will trigger another event. When someone requests an elevator a event must be generated, that will request an elevator to be moved to that floor or generate an event that will allow someone to get on. When someone gets on an event will be generated to move the elevator to another floor or generate an event let the person off after they realized they got on the wrong floor. As the elevator is moving events must be generated to let people off or let people on. When a elevator breaks down an event must be generated that will tell the estimated rime when the elevator will be fixed. When an elevator goes on service we must generate an event and when it goes off service. All events will be put into the queue . We will need a priority queue because each event must be sequenced by time. All Events will be derived from a base Event class . We can list all required events in a table with time, floors and next possible generated event.

event	time	floor from	floor to	generate event
Request	current	generated	generated	get on, move
Get on	current	current	requested	get off, move
Move	current	current	direction up or down	get off, get on, move
Get off	current	current	requested	request, move

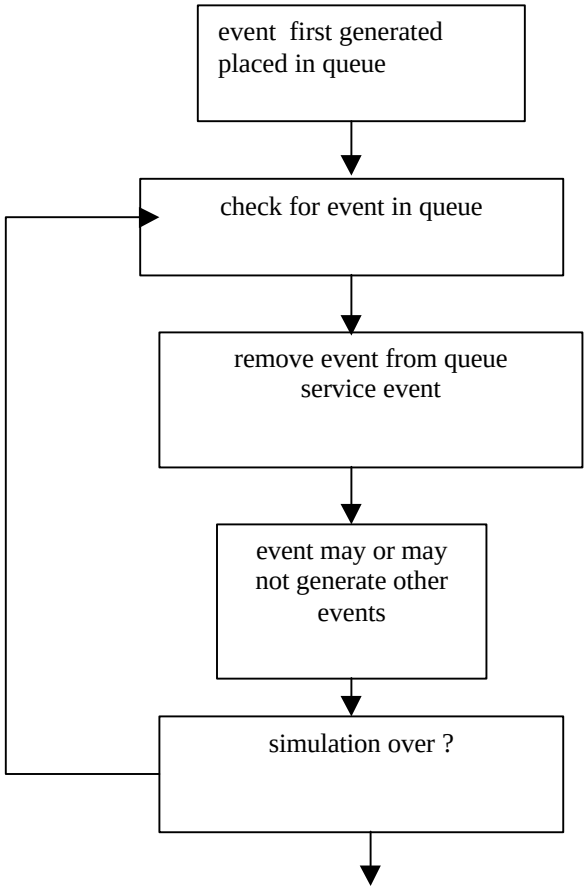
STATE DIAGRAM

Event sequences are usually displayed as a state diagram. Each event is enclosed in a circle where actions are displayed as arrows that lead to other events.



Event loop

The first event is generated and placed in the queue. In the event loop the queue is checked for events. The simulation model depended on events generating other events Each generated event is placed in the queue in time sequence. When the simulation time is up the loop exits.



Time driven loop

in time driven event loop a variable is kept to represent the simulation time in seconds. minutes or hours or just simply some time unit. Every time around the loop the time counter is incremented. every tome the queue is checked for an event it only services events that are less than or equal to the event time. Time driven events are very slow because the loop is always looping waiting for the time counter to equal the event time. In our elevator model we use time driven event and take the time from the computer's real time clock rather than using a variable time counter. We use real time because we want to simulate in real time.

Event driven loop

In event driven loop the events are serviced when they are encountered. The time variable will become the event time. event driven loops execute very fast because there is no waiting for the simulation time to be equal to the event time When using a event driven loop you must have a end event with the end simulation time so that the loop will exit.

TIME FUNCTIONS

The time functions are used to get the current time from the computer. Time functions are needed if you need to get the current time from your computer or you need to determine how many seconds some part of code takes to run. A **time_t** data type represents tome as a long integer.

```
typedef  long time_t;
```

A time_t data structure represents the current time, in seconds, elapsed since 00:00:00 GMT, January 1, 1970.

The structure tm holds the date and time in a structure This is the tm structure declaration from the <time.h> header file:

```
struct tm
        {
        int tm_sec;
        int tm_min;
        int tm_hour;
        int tm_mday;
        int tm_mon;
        int tm_year;
        int tm_wday;
        int tm_yday;
        int tm_isdst;
        };
```

prototype	description	example using
time_t **time** (time__t *time);	returns current calendar time of day in seconds, elapsed since 00:00:00 GMT, January 1, 1970.	time__t t; time(t) long t = time(NULL);
struct tm ***localtime** (time__t * time);	returns a pointer to a **tm** structure	struct tm local; local = Localtime(&t);
char ***asctime** (struct tm *ptr);	returns a pointer to a string that can be used to print out the current time	cout << (asctime(local));
char ***ctime** (const time_t *time);	Converts the date and time to a string.	time_t tim; ctime(tim);
void **gettime** (struct time *timep);	fills in the time structure pointed to by timep with the system's current time.	TIME T; gettime(&t);
double **difftime** (time_t t1, time_t2);	returns the difference in seconds between time t1 and time t2	double diff; diff = difftime(end,start);

using the time functions

The example program gets the current time using the time() function and prints the time out using asctime(). Then a calculation that will take a long time is done. We then get the current time again and printed it out. The difference in time is calculated using the difftime() function. The time difference in seconds is then printed out.

```
/* lesson 19 program 3 */
#include <iostream.h>
#include <stdlib.h>
/* generate random numbers between 0 and 99 and 1* and 100 */
void main()
        {
        time_t start,end;
        unsigned long i;
        time tm;
```

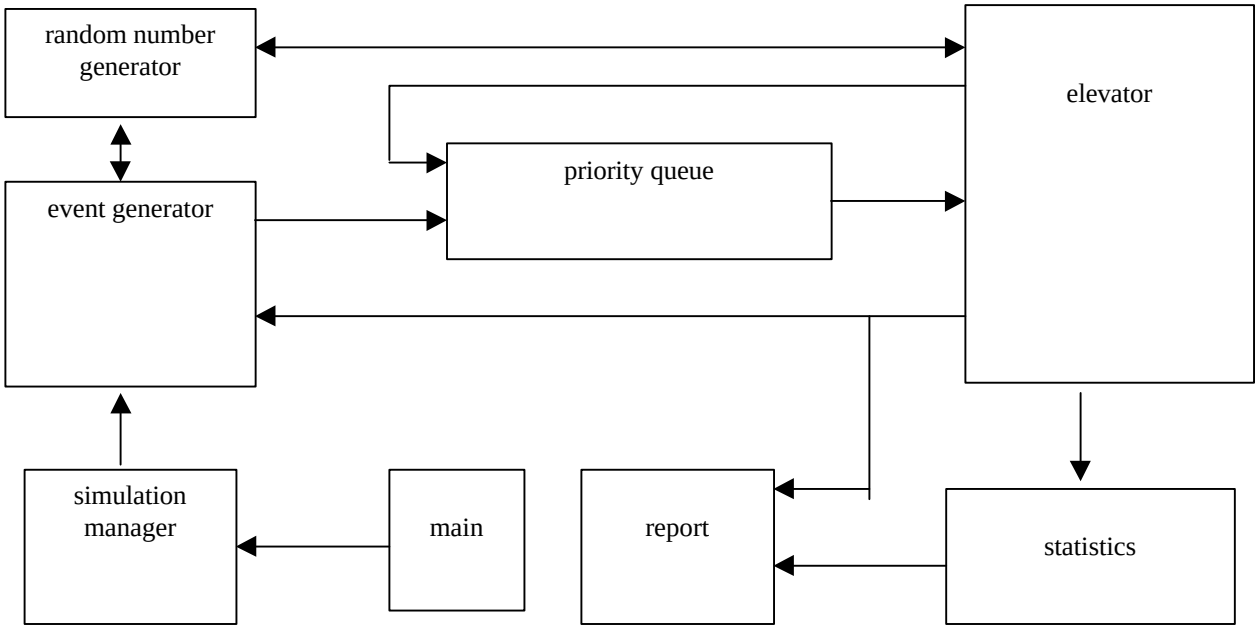
```
start = time(NULL); /* get start time */
cout << " the starting time is: "  << ctime(start << endl;
for(i=0;i<100000;i++);  /* long delay */
end = time(NULL); /* get end time */
cout << "the ending time is: " << ctime(end) << endl;
diff = difftime(end,start) /* calculate difference in time */
cout << "The time for the calculation is: %d seconds" << diff endl;
 tm = localtime(&time(NULL));
cout << "The time is: " << asctime(tm) << endl;
gettime(&tm);
cout << "The time is:" << asctime(tm) << endl;
}
```

CLASSES

We need a **simulation manager** class to take care of executing events placed in the event queue and run the operations. . All Events will be derived from a base Event class. We need an **elevator** class to simulate the elevator actions. We need a **request event generator** class to generate requests. We need a **random generator class** to generate random distributions. We will also need a **statistics class** and **report class**.

class	purpose	header file	implementation
random number generator	generate a random number to a specified distribution	rgen.hpp	rgen.cpp
event generator	generates random events based on data and time	eventgen.hpp	eventgen.cpp
event and derived events	event class will be base class foe elevator events	event.hpp	event.cpp
priority queue	stores events sorted in ascending order by time	pqueue.hpp	pqueue.cpp
elevator	elevator operations	elevator.hpp	elevator.cpp
statistics	keep track of day to day operation statistics	stats.hpp	stats.cpp
reports	give a detailed report from statistic data	reports.hpp	reports.cpp
simulation manager	control system operation	simmgr.hpp	simmgr.cpp
main	get things rolling		L19ex1.cpp

elevator simulation system model



MAIN FUNCTION

The main function will create the simulation manager, call the initialize simulation function, then call the run function with the user defined maximum simulation time, max floors, max patrons and exponential generator rate.

```
/* lesson simulation */
#include "simmgr.hpp"
void main()
        {
        /*sim, maxTime,maxFloors,maxPatrons,double rate */
        SimulMgr sim(10,5,10,0.5);  /* create simulator manager */
        sim.run();   /* run simulation */
        cout << simulation over << endl;
        }
```

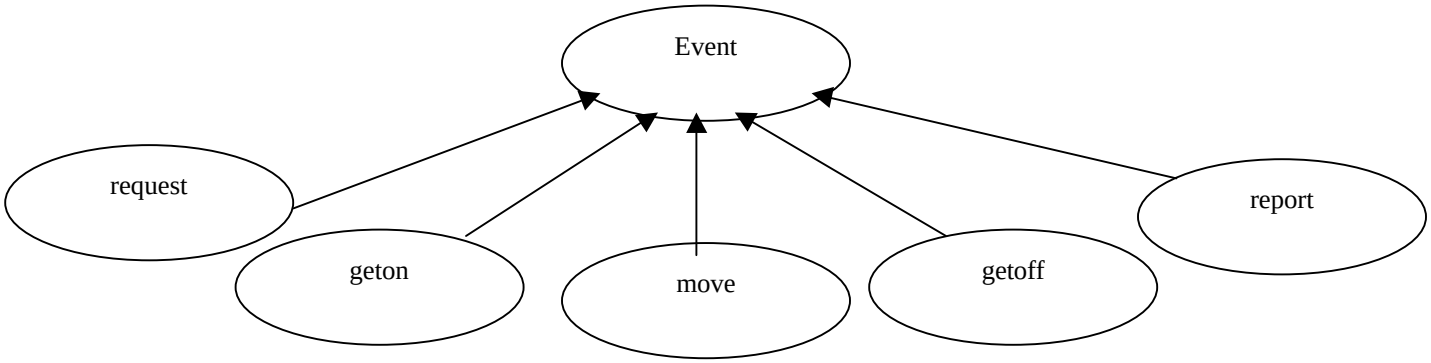
DEFINTIONS

Use the definition file if your compiler does not automatically include these

```
// defs.h
#ifndef __DEFS
#define __DEFS
typedef int bool;
#define false 0
#define true 1
#endif
```

EVENT CLASS

The Event class is the base class for all derived events. You will have a derive class for every event.



Here is a preliminary Event base class that you will need for the following exercise:

```
// event.hpp
#ifndef __EVENT
#define __EVENT

#include <iostream.h>
#include "defs.h"
class Event
{
private:
long time;
public:
Event(long time)
        {
        this->time=time;
        }
void print()
        {
        cout << " time: " << time << endl;
        }
 bool operator <= (Event& evt){return time <= evt.time;}
};
#endif
```

PRIORITY QUEUE CLASS

The priority queue module will store the time in ascending order. This means the earliest times will be at the start of the queue and the latest times will be at the end of the queue. The priority queue keeps a count of how many items it has. When you add an item to a priority queue is known a **enqueue**. When you remove an item from a priority queue it is known as **dequeue**.

```
/* pqueue.hpp */
/* priority queue list module header file */
#ifndef __PQUEUE
#define __PQUEUE
/* priority queue module data structures */
#include "event.hpp"
#include "defs.h"
class PQueue
{
/* priority queue list node */
struct PQNode
        {
        Event* event;          // store event
        PQNode* next;    // next node

        // initializing constructor
        PQNode(Event* evt,PQNode* node){this-evt=evt;this->next=next;}

        // print out contents
        print() {event->print());}
        };
/* priority queue list variables */
PQNode* head; // point to start of list
PQNode* tail;  // point to end of list
int count; // number of nodes in list
/* priority queue class function declarations */

/* default constructor initialize priority queue */
PQueue ();
/* delicate memory for priority queue */
~PQueue();
/* put item into priority queue */
bool enqueue(Event* evt);
/* check if queue is empty */
bool isEmpty();
/* look at item at start of queue */
Event* peekTail();
/* look at item at end of queue */
Event* peekHead();
/* remove data item from priority queue */
Event* dequeue();
/* print out contents of priority queue */
void print(char *name);
#endif
/* pqueue.cpp */
#include <iostream.h>
#include "pqueue.hpp"
#include "event.hpp"
#include "defs.h"
```

```
/* initialize priority queue */
PQueue::PQueue()
        {
        head = NULL;
        tail = NULL;
        count = 0;
        }

/* insert items at end of priority queue */
bool PQueue::enqueue(Event* evt)
{
PQNode* prev = NULL; // point to previous node
PQNode* node = head; // point to start of queue
/* insert node into list recursively in ascending order */
/* find where to insert item */
while((node != NULL) && !(*evt <= (Event)(*(node->event))))
        {
        prev = node;
        node = node->next;
        }

        /* allocate memory for priority queue node */
        PQNode* tnode = new PQNode(evt,node);
        /* check to insert at start of list */
        if(prev!=NULL)prev->next=tnode;
        /* adjust head and tail pointers */
        if((head == NULL) || (node != NULL && node == head)) head=tnode;
        if(tnode->next == NULL || tail == NULL)tail=tnode; /* last */
        count++;
        return true;
        }

/* remove items from the head of queue */
Event* PQueue::dequeue()
        {
        Event* data;
        PQNode* next;
        /* check if priority queue empty */
        if(head == NULL)return NULL;
        data = head->data;
        /* remove from item at head of priority queue */
        next = head->next;
        if(next!=NULL)
                {
                head->next = NULL;
                delete head;  /* delete link node */
                }
        head = next;
        if(next == NULL)tail = NULL;  /* priority queue is empty */
        count--;
        return data; /* return data element */
        }
```

```
/* look at item at start of queue */
Event* PQueue::peekTail()
        {
        if(tail==NULL)return NULL;
        return tail->event;
        }
/* look at item at start of queue */
Event* PQueue::peekHead()
        {
        if(head==NULL)return NULL;
        return head->event;
        }
/* check if queue is empty */
bool PQueue::isEmpty(){return (count == 0);}
/* delicate memory for priority queue driver */
PQueue::~PQueue()
        {
        PQNode* node = head; // point to start of queue
        /* loop till all nodes deleted */
        while(node != NULL)
                {
                /* delicate memory for priority queue item */
                PQNode* next = node->next;
                delete node;
                node = next;
                }
        head = NULL;
        tail = NULL;
        }
/* print out priority queue items */
void PQueue::print(char* name)
        {
        cout << "[ " << name;
        PQNode* node = head;

        while(node != NULL)
                {
                node->print() ;
                node =node->next;
                }
        cout << " ] " << endl;
        }
```

LESSON 19 EXERCISE 3

Write a small test program. Type in or copy and paste the priority queue module. Make an event class just to handle **a time entry.** In a loop generate random numbers to represents time as random events and put these times into the priority queue. Try using the different time distributions. In a loop remove events from the queue and print out on the screen.

C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 20

File:	CppdsGuideL20.doc
Date Started:	April 15,1999
Last Update:	Dec 27, 2001
Status:	draft

LESSON 20 SIMULATION TECHNIQUES II
SIMULATION MANAGER CLASS

The simulation manager class will be responsible for running the elevator simulation operation. The **Simulation Manager** constructor will create the elevator, random number generator, request event generator class objects. The **run()** function contains the time driven event loop that checks if there are any events in the priority queue. If there is then it checks if the event time is less than or equal to the current simulation time. If it is the event is removed from the priority queue and executed. We are using real time for our simulation. We use the functions from time.h to get the real time from the computer clock. The time driven event loop calls the execute() function from each event. Each event R**equest, GetOn, Move** and **GetOff** is a derived class from the base class **Event.**

```
/* simmgr.hpp */
#ifndef __SIMMGR
#define __SIMMGR

#include "pqueue.hpp"
#include "eventgen.hpp"
#include "elevator.hpp"
#include "randgen.hpp"
#include "event.hpp"

class SimMgr
{
/* simulation manager variables */
public:
PQueue *pq; /* pointer to priority queue */
EventGen *evtgen; /* pointer to event generator */
Elevator *elev; /* pointer to elevator */
RandGen *rgen; /* pointer to random number generator */
time_t simTime; /* current time */
private:
double maxTime; /* maximum simulation time in seconds */
time_t startTime; /* simulation start time */
time_t endTime; /* simulation end time */
public:
/* initialize simulation manager */
SimMgr(double maxTime,int maxFloors,int maxPatrons,double rate);
void run(); /* run simulation */
};
#endif
```

```
/* simmgr.cpp */
#include <iostream.h>
#include <time.h>
#include "simmgr.hpp"

/* initialize simulation manager */
SimMgr(double maxTime,int maxFloors,int maxPatrons,double rate)
        {
        /* store start and end simulation times */
        this->maxTime = maxTime;
        startTime = time(NULL);
        endTime = startTime + maxTime;
        /* allocate memory for and initialize class objects */
        pq = new PQueue);
        evtgen = new EventGen(this,Elevator::GROUND);
        elev = new Elevator(maxFloors,maxPatrons);
        rgen  = new RandGen(0,maxFloors,rate);
        }

/* run simulation */
void SimMgr::run()
{
/* generate request event depending on time of day  */
evtgen->genEvent();
time = time(NULL);  /* get time */
/* loop for simulation time */
while(simTime < endTime)
        {
        Event* event = pq.peekHead();  /* check event in queue */
        /* get event if queue not empty */
        if(!pq.isEmpty())
                {
                 /* check if time to service this event */
                if(difftime(event->t,simTime)<=0)
                        {
                        pq->dequeue(); /* remove event from queue */
                        event->execute(); // execute event
                        } /* end if time */
                } /* end if empty */
        /* generate request event depending on time of day  */
        evtgen->genEvent();
        simTime = time(NULL);  /* get time */
        } /* end while */
} /* end run */
```

EVENT CLASS

The Event Class is responsible for executing the individual events. The Event base class holds all the common information. All events have the execute function. Each derived class will have its own execute function.

```
// event.hpp
#ifndef __EVENT
#define __EVENT

// a class to manage events
class Event
{
public:
enum Events(REQUEST,GETON,GETOFF,MOVE);
enum Dir(STOPPED,REQUESTED,UP,DOWN);
protected:
Simmgr* sim;
public:
time_t time;  /* event time */
int to; /* to floor */
int from; /* from floor */
Dir dir; /* direction */
int num;  /* event number for tracking */
public:
//initializing constructor
Event(SimMgr* sim,time_t t)
        {
        this->sim=sim;
        this->t=t;
        }
/* copy an existing event */
Event *copy(Event  *event);
// check if one event is less than equal to another
bool operator <= (Event& evt){return time <= evt.time;}
virtual void execute();  // execute event
};

/* request an elevator */
class Request : public Event
   {
   private:
   Events id; /* event id */
   public:
   Request(SimMgr* sim, time_t t):Event(sim,t){id=REQUEST;}
   Request(Event& event);
   void execute();
   }
```

```
/* get on an elevator */
class GetOn : public Event
  {
  private:
  Events id; /* event id */
  public:
  GetOn(Event& event);
  void execute();
  }
/* get off an elevator */
class GetOff : public Event
  {
  private:
  Events id; /* event id */
  public:
  getOff(Event& event);
  void execute();
  }
/* move an elevator */
class Move : public Event
  {
  private:
  Events id; /* event id */
  public:
  move(Event& event);
  void execute();
  }
#endif

/* event.cpp */
/* events */
#include <iostream.h>
#include <time.h>
#include "event.hpp"

// copy constructor
Event::Event(Event& event)
  {
  sim=event.sim;
  t = event.t; /* event time */
  to = event.to; /* to floor */
  from = event.from; /* from floor */
  dir = event.dir; /* direction */
  num=0;  /* event number for tracking */
  }

/* request an elevator */
Request::Request(Event& event):Event(event)
  {
  id = REQUEST; /* event id */
  }
```

```
// execute request event
void Request::execute()
        {
        cout << "request floor  " << to;
        cout << " from " << from << "  at " << ctime(&sim->simTime);
        sim->elev->setElevOn(this);/* put event in elevator on queue */
        bool on =  sim->elevcheckOn(); // check if can get on floor

        /* only generate 1 Geton event per floor  */
        if(on && event->from == sim->elev->floor)
                {
                sim->pq->enqueue(new GetOn(*this));
                }
        /* move elevator to this floor if not moving */
        /* only generate 1 move event */
        /* only generate move event if no GET ON events generated */
        else if(sim->elev->dir == Elevator::STOPPED && !on )
                {
                if(sim->elev->dir == Elevator::STOPPED)sim->elev->dir =
                Elevator::REQUESTED;
                sim->pq->enqueue(new Move(*this));
                delete this;
                }
        }

        /* check to move this elevator */
        Move::Move(Event& event):Event(event)
          {
          id = MOVE; /* event id */
          }

        // execute move
        void Move::execute()
        {
        cout << "move: at floor " << sim->elev->floor;
        cout << " at " << ctime(&t) << endl;
        t = time(NULL);
        bool off = sim->elev->checkOff(); // check to get off floor
        bool on = sim->elev->checkOn();   // check to get on floor
        /* any body wants to get on or off this elevator ? */
        if(on | off)
                {
                if(off)
                        {
                        /* generate get off event */
                        sim->pq->enqueue(new GetOff(*this));
                        }
                if(on)
                        {
                        /* generate get on event */
                        sim->pq->enqueue(GetOn(*this));
                        }
```

```
                delete this;
                }
        /* move elevator to next floor */
         else
                {
                sim-elev->moveTo(this);
                sim->pq->enqueue(this);
                }
        }

/* get on elevator */
GetOn::GetOn(Event& event):Event(event)
  {
  id = GETON; /* event id */
  }

// execute get on elevator
void getOn::execute()
        {
        sim->elev->patrons++; /* count people getting on elevator */
        cout << "person get on floor " << sim->elev->floor;
        cout << " at " << ctime(&sim->simTime) << endl;
        sim->elev->removeOn(this);  /* take out of elevator on queue */
        /* check if all passengers get on */
        if(checkOn(sim->elev)==0)
                {
                if(sim->elev->dir == Elevator::STOPPED)sim->elev->dir =
                Elevator::REQUESTED;
                /* generate move event */
                time = time(NULL) + (long)gaussian(2,5);
                sim->pq-enqueue(*this);
                }
        }

/* get off an elevator */
GetOff::GetOff(Event& event):Event(event)
  {
  id = GETOFF; /* event id */
  }

// execute get off elevator
void getOff::execute()
        {
        cout << sim->elev->checkOff()<< " people get off floor ";
        cout << sim->elev->floor << " at " << ctime(&sim->simTime);
        sim->elev->removeOff(); /* take out of elevator off queue */
```

```
        /* check if any more requests */
        if(sim->elev->moreUp(sim->elev)||sim->elev->moreDown())
                {
                /* generate move event */
                t = time(NULL);
                sim->pq->enqueue(new Move(*this));
                }
/* no more requests */
else
        {
        /* delete this even if ground or parking */
        if(sim->elev->floor == Elevator::GROUND
        || sim->elev->floor == Elevator::PARKING)
                {
                delete this;
                return;
                }
        }
/* more floors to service */
if(!(sim->elev->floor == Elevator::GROUND
|| sim->elev->floor == Elevator::PARKING))
        {
        /* generate an request event when they leave */
        from = sim->elev->floor;
        to = sim->rgen->uniform();
        if(to > from)sim->event->dir = UP;
        else dir = DOWN;
        t= time(NULL) + exponential(sim->rgen);
        sim->pq->enqueue(new Request(*this));
        }
}
```

RANDOM NUMBER GENERATOR CLASS

The random number generator has all the random distribution functions. Exponential distributions are used to generate the time when a patron who enters the building leaves because the probability they will stay a long time is more if they are must going to leave quickly. Uniform probability is used to select a floor they will be going to. Gaussian probability is used to select if they will go to the ground floor or to the garage.

```
/* random number generator class */
#ifndef __RANDGEN
#define __RANDGEN

#include <time.h>
// class to generate random numbers using different distributions
class RandGen
{
private:
int a;   /* uniform a */
int b;   /* uniform b */
double rate; /* exponential rate */
```

```
/* initialize random number generator */
RandGen(int a,int b,double rate);

/* generate gaussian distribution random number */
int gaussian(int min,int max);
/* generate exponential distribution random number */
long exponential();
/* generate uniform distribution random number */
int uniform();
#endif
/* randgen.cpp */
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "randgen.hpp"
/* initialize random number distribution generator */
RandGen::RandGen(int a, int b, double rate)
        {
        rand((unsigned) time(NULL));  /* seed random number generator */
        this->a = a;   /* store a min uniform */
        this->b = b;   /* store b max uniform */
        this->rate = rate;  /* store exponential rate */
        }

/* gaussian distribution */
int RandGen::gaussian(int min,int max)
        {
        double d1,d2,d,m;
        int diff,r;
        /* loop till median found */
        do
        {
        d1 = (double)rand()/(double)RAND_MAX;
        d2 = (double)rand()/(double)RAND_MAX;
        d1 = 2 * d1 - 1;
        d2 = 2 * d2 - 1;
        d = d1 * d1 + d2 * d2;
        }while(d >= 1);
        m = sqrt(-2 * log(d)/d);    /* apply gaussian equation */
        d = d2 * m;
        /* get gaussian distribution between min and max */
        diff = (max-min)*100;
        r = (((int)d * RAND_MAX) % diff) + min*100;
        r = (r+50)/100;    /* round up */
        return r;
        }
```

```
/* exponential distribution */
/* - log((1.0 - random()/RAND_MAX)/rate) */
long RandGen::exponential()
        {
        double d = (double)rand()/(double)RAND_MAX;
        d = (1.0 - d);
        d = d/rate;
        d = -log(d);
        d * 100;
        return (long)d;
        }

/* uniform distribution */
/* (a + (b - a)  *  (random() / (RANDOM_MAX - 1.0)) */
int RandGen::uniform()
        {
        double d =  a + (b - a) * (rand() / (RAND_MAX - 1.0));
        return (int)d;
        }
```

REQUEST EVENT GENERATOR CLASS

The request event generator will generate events at certain intervals. In the mornings it will generate request event going up. At noon we will generate request events going down and going up. At the end of the day the request event generator will generate request events going down. At the night the event generator will generate request events for the cleaning crew and work alcoholics. On the weekend every body relaxes and the event generator will just generate events for the cleaning crew and security guards and a few corporate managers. The event generate will generate events according to the time and week day.

```
/* eventgen.hpp */
#ifndef __EVENTGEN
#define __EVENTGEN
#include "randgen.hpp"
#include "pqueue.hpp"
#include "simmgr.hpp"
// a class to generate Request Events depending on the time of day
class EventGen
{
private:
/* event generator */
int num;  /* current event number */
int startFloor;  /* starting floor */
Simmgr *sim; /* pointer to simulation manager */
public:
/* initialize event generator */
EventGen(Simmgr *sim, int startFloor);
/* generate an request event */
int genEvent();
#endif
```

```
/* eventgen.cpp */
#include <time.h>
#include "eventgen.hpp"

/* initialize event generator */
EventGen::EventGen(Simmgr* sim, int startFloor)
        {
        this->sim=sim;
        this->num = 0; /* current event number */
        this->startFloor=startFloor;
        }
/*generate event considering time of day */
int EventGen::genEvent()
{
tm t;  /* time in hours/min/seconds */
time_t tim; /* time in seconds */
Event *event; /* pointer to event */
Elevator::Dir dir; /* elevator request direction */
int interval = 0; /* time between requests */
int max=0;  /* max number of people per request */
int min=0;  /* min number of people per request */
int patrons=0; /* number of people requesting elevator */
int i;

time(&tim); /* get time */

t = localtime(&tim); /* get time in 24 hr clock */
/* morning */
if(t.tm_hour >= 7 && t.tm_hour <= 9)
  {
  max = 10;
  min = 2;
  interval = 10;
  dir = Elevator::UP;
  }
/* lunch */
else if(t.tm_hour >= 11 && t.tm_hour <= 13)
  {
  max = 5;
  min = 1;
  interval = 50;
  dir = Elevator::DOWN;
  }
/* evening */
else if(t.tm_hour >= 16 && t.tm_hour <= 18)
  {
  max = 10;
  min = 5;
  interval = 10;
  dir = Elevator::DOWN;
  }
```

```
/* all other times */
else
  {
  max = 2;
  min = 1;
  interval = 100;
  dir = Elevator::STOPPED;
  }
tim = time(NULL); /* get current time */
/* generate first event (num means event number) */
if(!(evtgen->num == 0)||((tim % interval)== 0))return 0;
patrons = sim->rgen->gaussian(min,max); /* get number of patrons */

/* loop for number of patrons  requesting elevator */
for(i=0;i<patrons;i++)
        {
        // create request event
        event = new Request(sim,time(NULL));
        /* morning or after lunch go up from ground or parking */
        if(evtgen->num == 0 || dir == Elevator::UP)
          {
          event->from = gaussian(Elevator::PARKING,Elevator::GROUND);
          event->to = sim->rgen->uniform();
          }
        /* lunch time or evening go down */
        else if(dir == Elevator::DOWN)
          {
          event->from = sim->rgen-uniform();
          event->to=rgen>gaussian (Elevator::PARKING,Elevator::GROUND);
          }
        /* all other times completely random */
        else
          {
          event->to = sim->rgen->uniform();
          event->from = sim->rgen->uniform();
          }
        /* set event direction from floors */
        if(event->to >=  event->from) event->dir = Elevator::UP;
        else event->dir = Elevator::DOWN;
        event->num = ++(num); /* set event number */
        sim->pq->enqueue(event); /* put event in queue */
        } // end for
return patrons;
}
```

ELEVATOR CLASS

The elevator class must keep track of what floor the elevator is on, what direction it is going, or if it is stopped or being requested. The elevator class has functions to move the elevator, check if there are requests for getting on or getting off. The elevator has 4 priority queues for each floor: getting on up, getting off up, getting on down and getting off down. We need queues because we need to keep track for each floor how many people are getting on and how many people are getting off. By using 4 separate queues makes are job more easier. It is now easier to determine which direction the elevator must travel. Example if the elevator is going up, we don't want to pickup people who have requested to go down.

```
/* elevator.hpp */
#ifndef __ELEVATOR
#define __ELEVATOR
#include "defs.h"
#include "event.hpp"
#include "pqueue.hpp"
// a class to simulate an elevator
class Elevator
{
public:
/* elevator constants */
enum  ElevStatus{OPERATING,OUT_OF_SERVICE,ON_SERVICE};
enum Dir{STOPPED,REQUESTED,UP,DOWN};
enum ExitFloors{PARKING,GROUND};
int patrons;      /* number of people in elevator */
int floor;       /* floor elevator is at */
Dir dir;         /* direction of elevator */
private:
PQueue *upon;     /* people requesting to going up */
PQueue *downon;   /* people requesting to going down */
PQueue *upoff;    /* people requesting getting off going up */
PQueue *downoff;  /* people requesting getting off going down */
int status;       /* status of elevator */

int capacity;     /* number of people allowed in elevator */
int maxFloors;    /* number of floors in elevator */
public:
/* initialize elevator */
Elevator(int maxFloors, int capacity);
/* move to next floor */
void moveTo();
/* set on requests */
void setElevOn(Event* event);
/* set off requests */
void setElevOff(Event* event);
/* check if people can get on from this floor */
bool checkOn();
/* check if people can get off to this floor */
bool checkOff();
/* remove on requests from elevator */
int removeOn(Event* event);
```

```
/* remove off requests from elevator */
int removeOff();
/* check if elevator needs to move up */
bool moreUp();
/* check if elevator needs to move down */
bool moreDown();
};
#endif
/* elevator.cpp */
#include <iostream.h>
#include <time.h>
#include "elevator.hpp"

/* initialize elevator */
Elevator::Elevator(int maxFloors,int capacity)
        {
        floor = GROUND;
        status = OPERATING;
        dir = STOPPED;
        this->maxFloors = maxFloors;
        this->capacity=capacity;
        patrons = 0;
        upon = new PQueue[maxFloors];
        downon = new PQueue[maxFloors];
        upoff = new PQueue[maxFloors];
        downoff = new PQueue[maxFloors];
        }
/* move elevator to required floor */
void Elevator::moveTo()
        {
        /* check if elevator is operating */
        if(status == OPERATING)
                {
                /* elevator moving up */
                if(dir == UP && floor < elev->(maxFloors-1))floor++;
                /* elevator moving down */
                else if(dir == DOWN && floor > 0)elev->floor--;
                /* elevator not moving */
                else
                        {
                        /* check to go up */
                        if(moreUp())
                          {
                          dir = UP;
                          floor++;
                          }
                        /* check to go down */
                        else if(moreDown())
                          {
                          dir = DOWN;
                          floor--;
                          }
```

```
                /* we are here */
                else cout << "elevator already at this floor "<< endl;
                }
        cout << "move to  " << floor << "  with  " << patrons << " people" << endl;
        }

/* add floor requests to elevator */
void Elevator::setElevOn(Event *event)
        {
        /* elevator moving up */
        if(dir == UP)upon[event->from].enqueue(new Event(*event));
        /* elevator moving down */
        else if(dir == DOWN)
        elev->downon[event->from].enqueue(new Event(*event));
        }
/* check if elevator should stop at this floor */
bool Elevator::checkOn()
        {
        int up = upon[floor].count;
        int down = downon[floor].count;
        if(dir == UP && up>0)return true; /* elevator moving up */
        /* elevator moving down or stationary */
        else if(dir == DOWN && down>0)return true;  /* no direction */
        else
                {
                if(up>0)return true;
                if(down>0)return true;
                }
        return false;
        }
/* check if elevator should stop at this floor */
bool Elevator::checkOff()
        {
        int up = upoff[floor].count;
        int down =downoff[floor].count;
        /* elevator moving up or stationary */
        if(dir == UP && up>0)return  true;
        /* elevator moving down or stationary */
        else if(dir == DOWN && down>0)return true;
        /* no direction */
        else
                {
                if(up>0)return true;
                if(down>0)return true;
                }
        return false;
        }
```

```
/* remove requests to get on from this floor, add to off list */
int Elevator::removeOn(Event* event)
        {
        int numon = 0;
        /* elevator moving up */
        if(event->dir == UP)
        {
        event = upon[floor].dequeue();
        /* loop till all up on events requests removed *./
        while(event != NULL)
          {
          numon++;
          /*  put in off list */
          upoff[event->to].enqueue(new Event(event));
          event = upon[floor]).dequeue();
          }
        }
/* elevator moving down or stationary */
else
        {
        event = downon[floor].dequeue();
        /* loop till all down on events requests removed *./
        while(event != NULL)
                {
                numon++;
                /* put in off list */
                downoff[event->to].enqueue(new Event(*event));
                event = downon[floor].dequeue();
                }
        }
        return numon;
}
/* remove requests to get off from this floor */
void Elevator::removeOff()
        {
        int numoff=0;
        Event *event;
        /* elevator moving up or stationary */
        if(dir == UP || dir == STOPPED)
                {
                event = upoff[floor]).dequeue();
                /* loop till all down up off  events requests removed *./
                while(event != NULL)
                {
                numoff++;
                elev->patrons--;
                delete event;
                /* remove from  off list */
                event = upoff[floor].dequeue();
                }
                }  // end if
```

```
        /* elevator moving down  */
        else
                {
                event = elev->downoff[elev->floor].dequeue();
                /* loop till all down off events requests removed *./
                while(event != NULL)
                        {
                        numoff++;
                        patrons--;
                        delete event;
                        // remove from off list
                        event = downoff[floor]).dequeue();
                        }
        /* set elevator to no direction if everyone got off */
        if(patrons == 0)dir = STOPPED;
        return numoff;
        }
/* check if more elevator requests from this floor going up */
bool Elevator::moreUp()
        {
        /* check from this floor going up */
        for(int i=floor;i<maxFloors;i++)
          {
          if(upoff[i].count || upon[i].count)return true;
          if(downoff[i].count || downon[i].count)return true;
          }
        return false;
        }
/* check if any requests from this floor going down
bool Elevator::moreDown()
        {
        /* check from this floor going down */
        for(int i=elev->floor;i>=0;i--)
          {
          if(downoff[i].count ||downon[i].count)return true;
          if(upoff[i].count || upon[i].count)return true;
          }
        return false;
        }
```

STATISTICS CLASS

The statistics class must keep track the awaiting and average waiting times of the people requesting an elevator. We know the time when they requested, when they get on and when they get off the elevator . The Statistics class must have variables to keep track of all these things.

REPORT CLASS

The Report class derived from the Event class will just report the results of the Statistics class at convenient time intervals. A Report event will cause another report event to be generated at the next report time interval.

LESSON 20 EXERCISE 1

Type in or paste and copy all the elevator simulation classes. Use separate header *.hpp files and implementation files *.cpp. Run a few simulations and convince your self everything is working okay.

LESSON 20 EXERCISE 2

Write the Statistics and Report class and integrate them into the elevator simulation program. Change or add a interval constant so that more people will request an elevator to the EventGen class. Now people will have too wait a long time for an elevator.

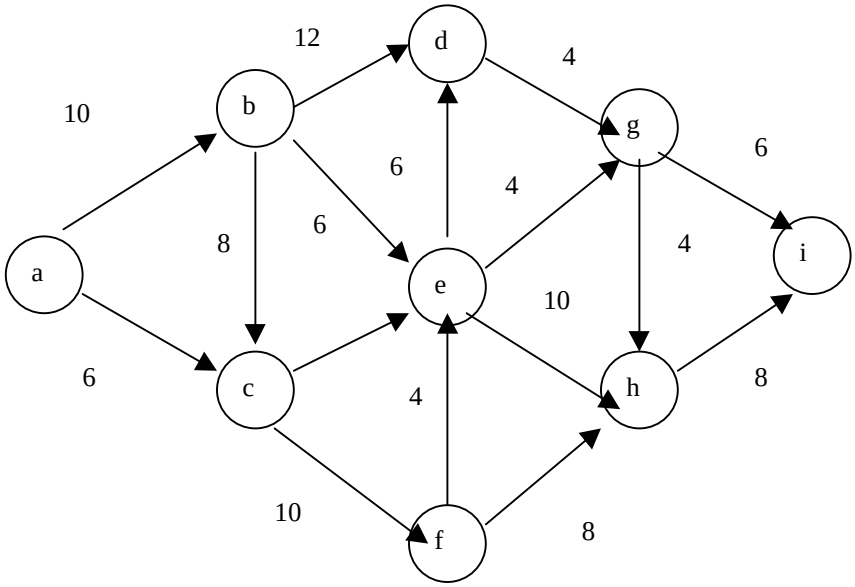
LESSON 20 EXERCISE 3

Add two more elevators, and see if people have too wait less time to get a elevator.

C++ DATA STRUCTURES PROGRAMMERS GUIDE PROJECT II

File:	CppdsGuideL20.doc
Date Started:	July 24,1998
Last Update:	Dec 27, 2001
Status:	proof

An internet E-mail message has to be sent across the internet. The E-mail message must choose the fastest route between hubs. Each hub connected to each other somehow. The distance between hubs is much different. The E-mail message can only go one direction between hubs, and use a hub exactly only once.



	a	b	c	d	e	f	g	h	i
a		19	6						
b			8	12	6				
c					4	10			
d							4		
e				6			4	10	
f					8			6	
g								4	6
h									8

This is what you have to do:

(1) Make a file called **hubs.dat** to represent the distances
The file starts with then number of hubs and each line represents hub pairs with a distance

```
16
a b 50
a c 6
```

(2) Your program should read in the file then print out a adjacency matrix.

(3) print out all possible paths to reach each hub **i** from hub **a**

(4) print out the longest path and the shortest path from hub **a** to hub **i**

(5) print out the optimal route the e-mail must travel to reach hub **i** from hub **a**. For the total shortest distance that each hub is visited exactly only once.

(6) print a list of hubs that must be visited first before you can go to the next hub

(7) Finally print out the optimal path starting from hub **a** where all hubs are visited only once to reach hub **i**. Call your project header file proj2.h and the implementation file propj2.c.

CONDITION	RESULT	GRADE
Program crashes	retry	R
Program works	pass	P
Program is impressive	good	G
program is ingenious	excellent	E

If your program crashes then you must fix it and resubmit your assignment
YOU MUST HAVE A WORKING PROGRAM TO COMPLETE PROJECT 2

IMPORTANT

You should use all the material in all the lessons to do the questions and exercises. If you do not know how to do something or have to use additional books or references to do the questions or exercises, please let us know immediately. We want to have all the required information in our lessons. By letting us know we can add the required information to the lessons. The lessons are updated on a daily bases. We call our lessons the "living lessons". Please let us keep our lessons alive.

E-Mail all typos, unclear test, and additional information required to:

courses@cstutoring.com

E-Mail all attached files of your completed project to: ($25 charge for marking)

students@cstutoring.com