THE COMPUTER SCIENCE TUTORING CENTER

C DATA STRUCTURES PROGRAMMING COURSE E-BOOK OUTLINE

C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 1

File:	cdsGuideL1.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 $1 + 2 + 3 + 4 + \ldots + n = \dfrac{n(n+1)}{2}$ is true using induction for n >= 1

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

step1: **base case** prove for n = 1 the sum must be 1

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

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

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

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) $\boxed{n = k + 1}$

$$\boxed{\sum^{k+1}} = \boxed{\sum^{k} i} + (k+1) = \frac{k\,(k+1)}{2} + (k+1) = \frac{k\,(k+1) + 2\,(k+1)}{2}$$

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

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**

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

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** $1 = \frac{n^2 + n}{2} + \frac{1^2 + 1}{20} = 1$

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

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

$$\boxed{\sum^{n} i^2} = 1^2 + 2^2 + 3^2 + 4^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{2}$$ 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:
3 + 2 + 1 = 6

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**.

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

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

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 <u>linear</u> means the **for** loop increments or decrements by 1

3. non linear loops: **O(log n)**

```
k=0;                                  k=0;

for(i=n; i>0; i=i/2)                  for(i=0; i<n; i=i*2)

         k++;                                  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...

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

O(1) + O(2) + O(3) + O(4) + ...

which is the binomial series:

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

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++;
     }
```

choose branch that has largest

```
/* O(n^2) */
else

   {

   k=0;

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

       for(j=i; j>0; j=j/2)

           k++;
}
```

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 when the program calls itself.

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)**

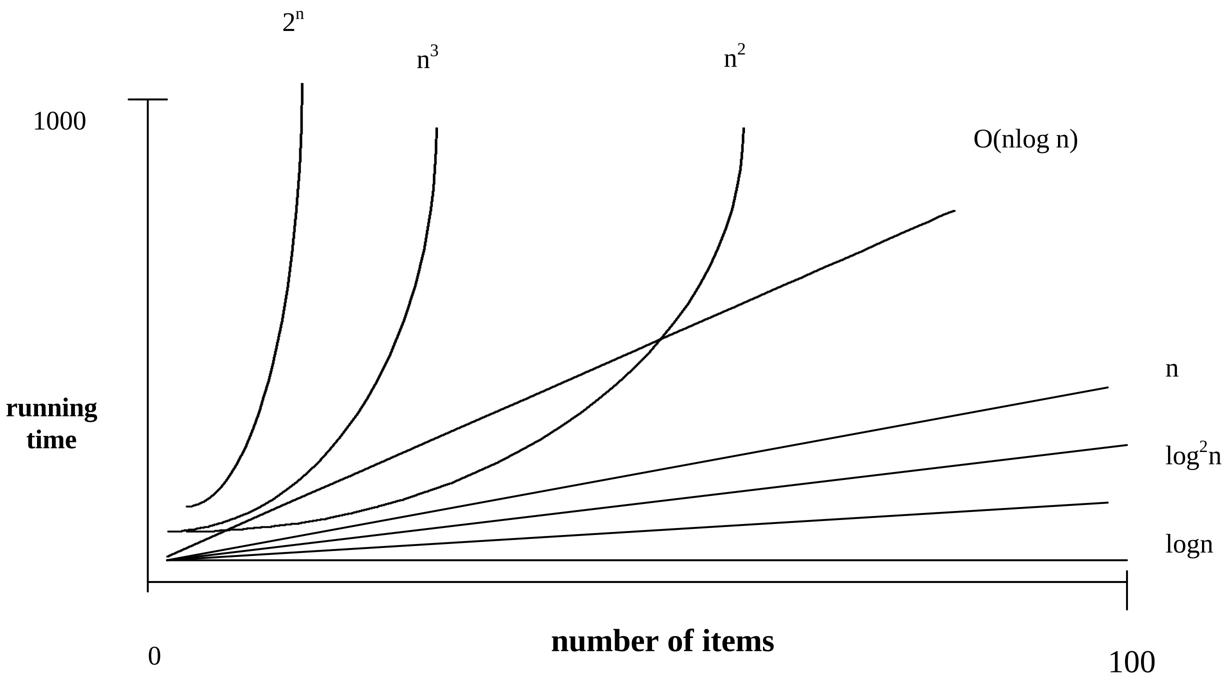
growth rates summary

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:



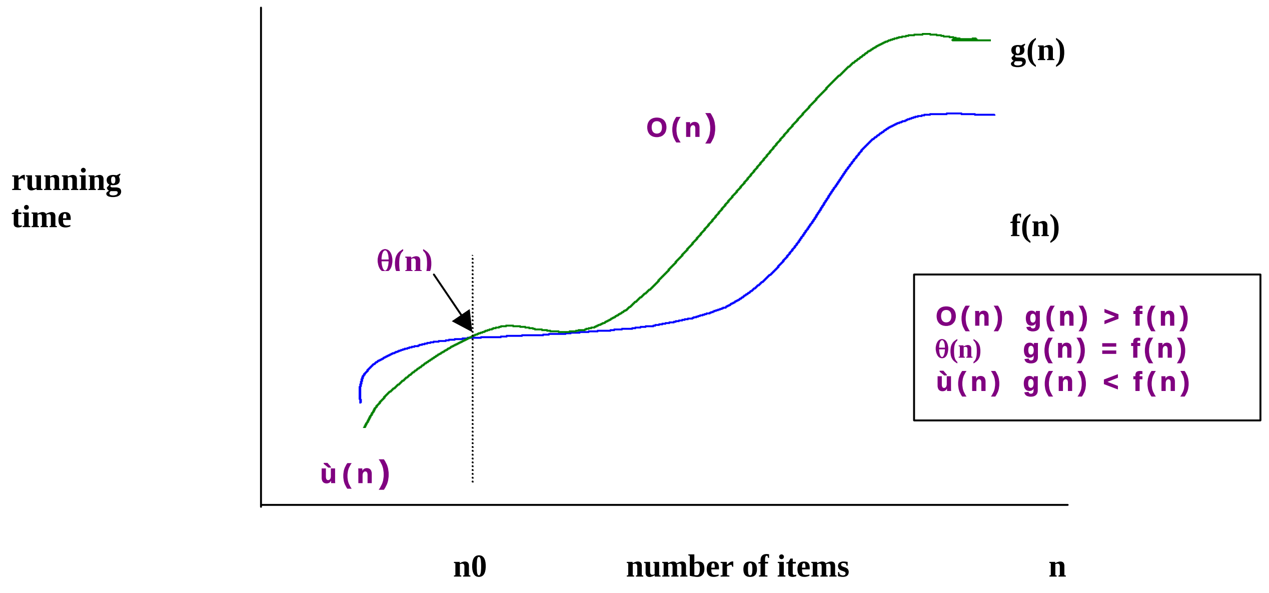
As **n** increases the running depends on the growth rates. For any **n** constant is the fastest and 2^n is the slowest. Worst case timing is also dependent on n For small values of n we can see n^2 is mush faster than n log n. For larger values of **n** (**n** log **n**) is much faster than **n**^{2}.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

$$\boxed{n^2 \sim n \log n}$$

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

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

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

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

LESSON 1 EXERCISE 4

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

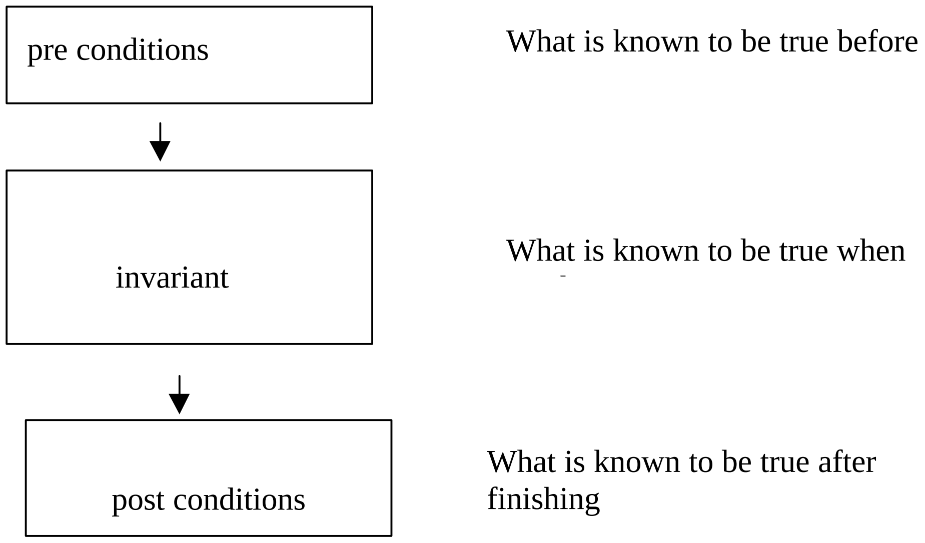
PRECONDITIONS, INVARIANTS AND POSTCONDITIONS

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.

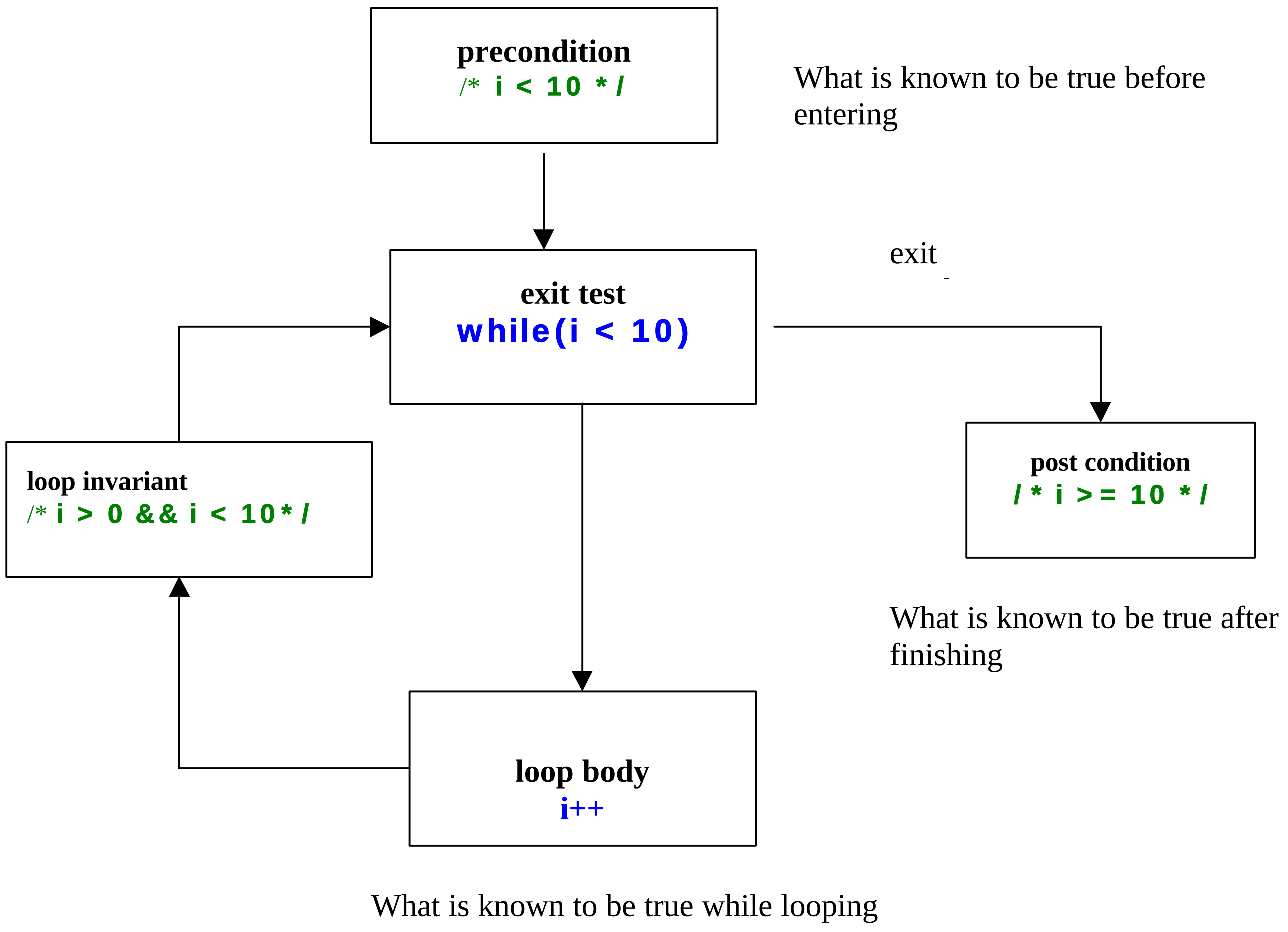

pre conditions
What is known to be true before
invariant
What is known to be true when
post conditions
What is known to be true after finishing

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

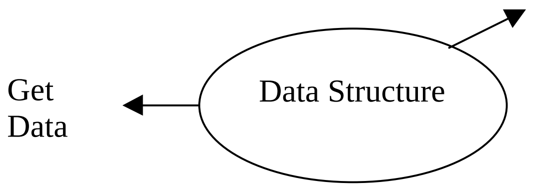
If invariant is true at the beginning of the loop then at the beginning of the next loop it will be true again as long as the loop does not exit

proving loop termination

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

C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 2

File:	CdsGuideL2.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 ?

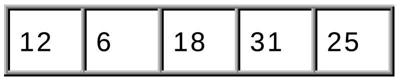
You choose which ADT to use by using the specifications of the project to determine which ADT is suitable for your application. Worst case expected running time can be broken down into search insertion and deletion time. Link lists have a good search time if sorted but longer search time if unsorted. 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 MODULE

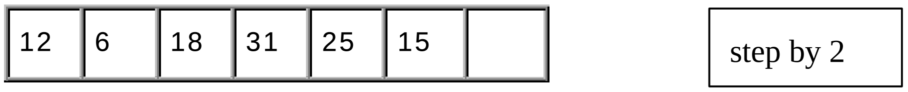
A vector is a dynamically resizing array. The major problem when you declare an array, it is of fixed size. If you need to add more items, and your array is filled you are stuck. If you need to delete a lot of items in your array permanently then you are left with a large amount of computer memory unused. 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:

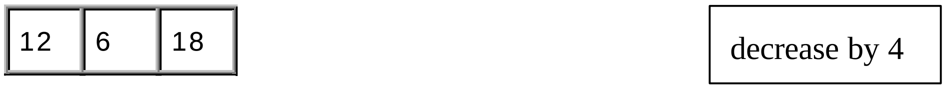
Create a vector of size 5 and step 2 and fill with 5 data items



Add 1 more data item, the vector grows to 7



Delete 3 data items the vector shrinks to 3



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 the modular approach to implement our Vector ADT where a structure is use to hold the size and vector array elements.

vector.h	header file	The header file contains the function prototypes.
Vector	data structure	The data structure holds the allocated array and variables shared between all Vector functions.
vector.c	code	The code contains the function definitions.

Here's the code for the vector 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 vector.h.

```
/* defs.h */

#define TRUE 1
#define FALSE 0
#define ERROR -1
typedef int bool;
```

Vector Module header file:

```
/* vector.h */

/* vector ADT module */

#include "defs.h"

#ifndef __VECTOR
#define __VECTOR

/* vector data type */
typedef int VectorData;

/* vector data type ptr */
typedef int* VectorDataPtr;

/* vector module data structure */
typedef struct

        {
        VectorDataPtr items; /* pointer to array element */
        int size; /* vector size */
        int step; /* vector step */
        int last; /* last element location in vector */
        }Vector,*VectorPtr;

/* function prototypes for vector module */
/* initialize vector module variables and allocate memory */
bool initVec(VectorPtr vec,int size,int step);

/* get number of items in vector module */
int numItem(VectorPtr vec);

/* insert data item into last location of vector */
bool insertVecLast(VectorPtr vec,VectorData data);

/* insert data item into first location vector */
bool insertVecFirst(VectorPtr vec,VectorData data);

/* insert data item at index */
bool insertVecAt(VectorPtr vec,VectorData data,int index);

/* return item value at a specified index */
VectorData getVecAt (VectorPtr vec,int index);
```

```
/* find item in vector, return index location in vector */
int findVec(VectorPtr vec,VectorData data);

/* search and replace a vector item with new data */
bool replaceVec(VectorPtr vec,VectorData newdata,VectorData olddata);

/* replace a vector item at specified index */
bool replaceVecAt(VectorPtr vec,VectorData newdata,int index);

/* search for and delete specified data item, return index */
int removeVec (VectorPtr vec,VectorData data);

/* delete data item at specified index, return data value */
VectorData removeVecAt (VectorPtr vec,int index);

/* print out vector items */
void printVec(VectorPtr vec,char* name);

/* free vector items */
void destroyVec(VectorPtr vec);

#endif
```

Vector module code implementation file:

```
/* vector.c */

#include <stdio.h>
#include <stdlib.h>
#include "vector.h"

/* code implementation functions for vector ADT */

/* initialize vector data structure and allocate memory for vector */
bool initVec(VectorPtr vec,int size,int step)

        {
        int i;

        /* allocate memory for vector at specified size */
        vec->items = (VectorDataPtr)malloc(sizeof(VectorData)* size);
        if(vec->items == NULL)return FALSE;
        vec->size = size; /* set size */
        vec->step = step; /* set step */
        vec->last = 0; /* set last item to start of vector */

        for(i=0;i<vec->size;i++)vec->items[i]=0; /* clear vector */

        return TRUE;
        }
```

```
/* get number of items in vector module */
int numItem(VectorPtr vec)

        {
        return vec->last; /* return last used location in vector */
        }

/* insert a data item into the vector at last index */
bool insertVecLast(VectorPtr vec,VectorData data)

        {
        /* call to insert at last location */
        return insertVecAt(vec,data,vec->last);
        }

/* insert a data item into the vector at last index */
bool insertVecFirst(VectorPtr vec,VectorData data)

        {
        /* call to insert at first location */
        return insertVecAt(vec,data,0);
        }

/* insert a data item into the vector at last index */
bool insertVecAt(VectorPtr vec,VectorData data,int index)

        {
        int i;
        VectorDataPtr newItem;
        VectorDataPtr temp;

        /* check if enough room in vector */
        if(index < vec->size)

                {
                vec->items[vec->last]= data; /* insert data */
                vec->last++; /* increment last */
                return TRUE;
                }

        else

                {
                /* create a new item */
                newItem = (VectorDataPtr)
                malloc(sizeof(VectorData) * (vec->size + vec->step));
                if(newItem == NULL)return FALSE; /* item not created */


                /* clear new vector */
                for(i=0;i<vec->size+vec->step;i++)newItem[i]=0;
                /* copy existing item */
                for(i=0;i<vec->size;i++) newItem[i]= vec->items[i];
                newItem[i++]=data;
                vec->size = vec->size + vec->step; /* increase size */
                vec->last = i; /* store last location */
                temp = vec->items; /* temp point to new items */
                vec->items = newItem; /* vector points to new items */
                free(temp); /* free memory pointed to by temp */
                return TRUE;
                }

        }

/* return item value at a specified index */
VectorData getVecAt (VectorPtr vec,int index)

        {
        /* return value at vector location */
        if(index < vec->last)return vec->items[index];
        else return NULL;
        }

/* find data item in vector */
int findVec(VectorPtr vec,VectorData data)

        {
        int i;
        /* loop through items looking for data item */
        for(i=0;i<vec->last;i++)
             if(vec->items[i]==data)return i; /* exit when found */
        return ERROR;
        }

/* replace a vector item */
int replaceVec(VectorPtr vec,VectorData newdata,VectorData olddata)

        {
        int index = findVec(vec,olddata); /* get location of data item */
        if(index < 0) return FALSE;
        /* if old data item found replace item with new data */
        vec->items[index] = newdata;
        return TRUE;
        }

/* replace a vector item with a new data value at a specified index */
/* return old value */
VectorData replaceVecAt(VectorPtr vec,VectorData newdata,int index)

        {
        VectorData data;
        /* index is less than last location and greater then zero */
        if((index < vec->last) && (index >= 0))

                {
                data = vec->items[index];
                vec->items[index] = newdata; /* replace with new data */
                return data;
                }

        else return NULL;
        }

/* remove data item in vector */
int removeVec(VectorPtr vec,VectorData data)

        {
        /* get location of data item */
        int index = findVec(vec,data);
        if(index >= 0)removeVecAt(vec,index); /* remove item */
        return index;
        }

/* remove a data item from the vector */
VectorData removeVecAt (VectorPtr vec,int index)

        {
        int i;
        VectorDataPtr newItem;
        VectorDataPtr temp;
        VectorData data;

        /* check for out of bounds index */
        if((index > vec->last) || (index < 0))return NULL;

        /* store value to remove */
        data = vec->items[index];

        /* shift up items in vector */
        for(i=index;i<vec->last;i++)vec->items[i] = vec->items[i+1];

        vec->last--;

        /* check to deallocate memory */
        if(vec->last < vec->size-vec->step)

                {
                /* allocate memory for new vector items size - step */
                newItem = (VectorDataPtr)
                malloc(sizeof(VectorData) * (vec->size) - (vec->step));

                /* copy vectors from old items to new items */
                for (i=0;i<vec->last;i++)newItem[i]=vec->items[i];
                temp = vec->items; /* temp points to old vector */
                vec->items = newItem; /* point to new vector memory */
                free (temp); /* deallocate old vector memory */
                }

        return data;
        }

/* free items */
void destroyVec(VectorPtr vec)

        {
        free(vec->items); /* deallocate memory */
        vec->items = NULL; /* set all to default values */
        vec->size = 0;
        vec->step = 0;
        vec->last = 0;
        }

/* print out vector items */
void printVec(VectorPtr vec,char* name)

        {
        int i;

        /* check for empty vector */
        if(vec->items == NULL)

                {
                printf("the list is empty\n");
                return;
                }

        /* print out vector elements */
        printf("vector %s: [",name);

        /* loop through all items in vector */
        for(i=0;i<vec->last;i++)

                {
                printf("%d",vec->items[i]); /* print out values */
                if(i < (vec->last-1))putchar(','); /* insert comma */
                }

        printf("]\n");
        }
```

```
/* main program for vector module */
int main()

        {
        Vector vec; /* create vector data structure in memory */
        initVec(&vec,5,2); /* initialize a vector data type size,step */
        printVec(&vec,"vector: "); /* print out vector */

        /* insert 3 items into vector */
        insertVecLast(&vec,6);
        insertVecLast(&vec,3);
        insertVecLast(&vec,9);
        printVec(&vec,"vector: "); /* print out vector */

        /* remove an item */
        removeVec(&vec,8);
        printVec(&vec,"vector: "); /* print out vector */

        /* insert 5 items */
        insertVecLast(&vec,8);
        insertVecLast(&vec,12);
        insertVecLast(&vec,4);
        insertVecLast(&vec,12);
        insertVecLast(&vec,7);
        printVec(&vec,"vector: "); /* print out results */

        /* remove 1 item */
        removeVec(&vec, 6);
        printVec(&vec,"vector: "); /* print out results */

        /* remove 5 items */
        removeVec(&vec, 12);
        removeVec(&vec, 4);
        removeVec(&vec, 7);
        removeVec(&vec, 6);
        removeVec(&vec, 8);
        printVec(&vec,"vector: "); /* print out results */

        /* replace items */
        replaceVec(&vec,13,12); /* replace 12 with 13 */
        printVec(&vec," vector: "); /* print out results */
        replaceVecAt(&vec,9,1); /* replace 1 with 9 */
        printVec(&vec,"vector: "); /* print out results */
        replaceVec(&vec,5,getVecAt(&vec,0)); /* replace item index 0 */
        printVec(&vec," vector: "); /* print out results */
        replaceVecAt(&vec,17,findVec(&vec,5)); /* replace 5 with 17 */
        printVec(&vec," vector: "); /* print out results */

        /* free all items in vector */
        destroyVec(&vec);

        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

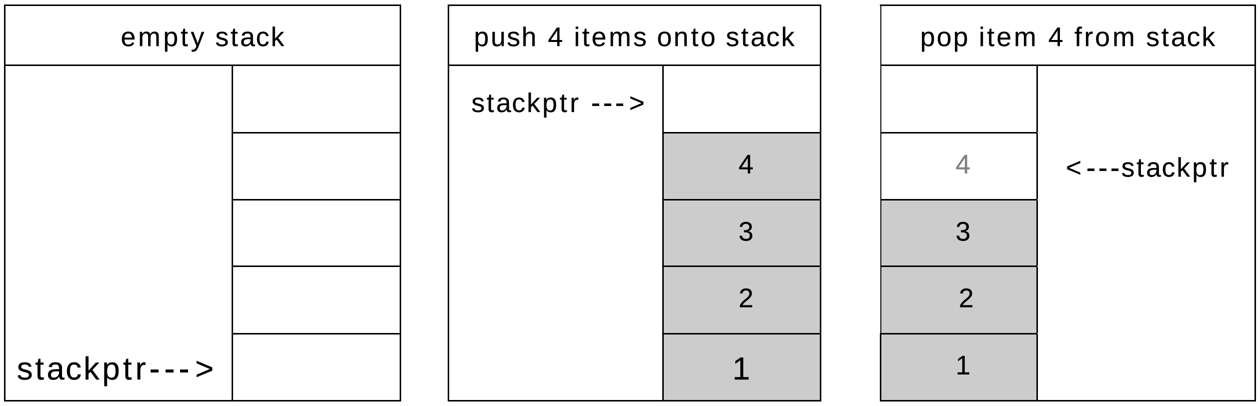
Edit copy paste or type in the Vector module and run it.

LESSON 2 EXERCISE 2

Modify the vector module 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.

STACK ADT MODULE

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.

stack.h	header file	The header file contains the function prototypes.
Stack	data structure	The data structure holds the array and variables and stack pointer to be shared between all Stack functions.
stack.c	code	The code contains the 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.h */

#ifndef __STACK
#define __STACK

/* stack module data items */
typedef int StackData;

/* stack module data structure */
typedef struct

        {
        StackData* items; /* pointer to vector module */
        int stkptr; /* stack pointer */
        int size; /* max size of stack */
        }Stack,*StackPtr;
```

```
/* stack module function prototypes */

/* initialize stack */
void initStk(StackPtr stk,int size);

/* put data item onto stack */
bool push(StackPtr stk,StackData data);

/* get data item from stack */
StackData pop(StackPtr stk);

/* print stack contents */
void printStk(StackPtr stk);

/* deallocate memory for stack */
void destroyStk(StackPtr stk);

#endif
```

Implementation code file:

```
/* stack.c */

#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

/* stack module functions */

/* initialize stack module */
void initStk(StackPtr stk, int size)

        {
        /* allocate memory for vector data structure */
        stk->items=(StackData*)malloc(sizeof(StackData)*size);
        stk->stkptr = 0; /* set stackptr to top of stack */
        stk->size=size; /* set max size of stack */
        }

/* push item into stack */
bool push(StackPtr stk,StackData data)

        {
        if(stk->stkptr == stk->size)return FALSE; /* return false if stack full */
        stk->items[stk->stkptr]=data; /* insert into vector at stpkptr location */
        (stk->stkptr)++; /* increment stack ptr */
        return TRUE;
        }
```

```
/* get item from stack */
StackData pop(StackPtr stk)

        {
        if(stk->stkptr <= 0)return 0; /* check if stack empty */
        --(stk->stkptr); /* decrement stack pointer */
        return stk->items[stk->stkptr]; /* return element */
        }

/* print stack items */
void printStk(StackPtr stk)

        {
        int i;

        /* check for empty stack */
        if(stk->items == NULL)

                {
                printf("the list is empty\n");
                return;
                }

        /* print out stack elements */

        printf("stack: [");
        /* loop through all items in stack */

        for(i=0;i<stk->stkptr;i++)

                {
                printf("%d",stk->items[i]); /* print out values */
                if(i < (stk->stkptr-1))putchar(','); /* insert comma */
                }

        printf("]\n");
        }

/* deallocate memory for stack */
void destroyStk(StackPtr stk)

        {
        free (stk->items);
        stk->stkptr=0;
        stk->size=0;
        }
```

```
/* main function to test stack module */
int main()

        {
        StackData data; /* create stack data structure in memory */
        Stack stk; /* create a stack data structure */
        initStk(&stk,10); /* initialize stack module */
        printStk(&stk); /* print stack contents */
        push(&stk,8); /* push "8" into stack */
        printStk(&stk); /* print stack contents */
        data = pop(&stk); /* get item from stack */
        printf("data from stack: %d\n",data);
        data = pop(&stk); /* get item from stack */
        printf("data from stack: %d\n",data);
        push(&stk,3); /* push "3" into stack */
        push(&stk,7); /* push "7" into stack */
        push(&stk,4); /* push "4" into stack */
        printStk(&stk); /* print stack contents */
        data = pop(&stk); /* get item from stack */
        printf("data from stack: %d\n",data);
        printStk(&stk); /* print stack contents */
        data = pop(&stk); /* get item from stack */
        printf("data from stack: %d\n",data);
        printStk(&stk); /* print stack contents */
        data = pop(&stk); /* get item from stack */
        printf("data from stack: %d\n",data);
        printStk(&stk); /* print stack contents */
        destroyStk(&stk) /* deallocate memory for stack */
        return TRUE
        }
```

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 3

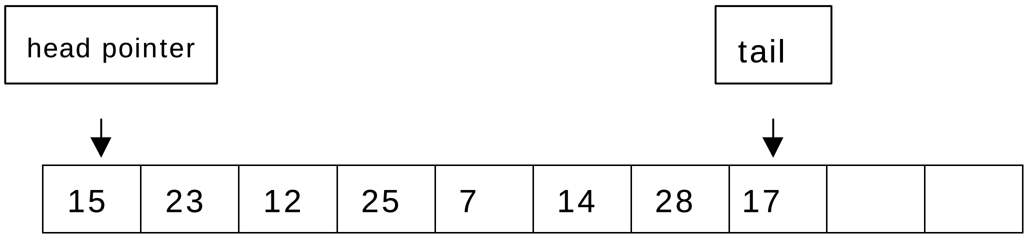
Edit copy paste or type in the Stack module and run it. Add an **isEmptyStk()** function to the Stack. Push some values in the Stack and use the **isEmptyStk()** method to pop all the values from the stack till empty and print them out.

LESSON 2 EXERCISE 4

Use the Vector module to make a Stack.

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.

queue.h	header file	The header file contains the function prototypes.
Queue	data structure	The data structure holds the array and variables and stack pointer to be shared between all Stack functions.
queue.c	code	The code contains the 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;
```

```
/* queue.h */
/* queue module header file */


#ifndef __QUEUE
#define __QUEUE

typedef int QueueData;

/* queue module data structure */
typedef struct

        {
        QueueData* items;
        int head; /* point to first element in queue */
        int tail; /* point to last element in queue */
        int size; /* size of queue */
        }Queue,*QueuePtr;

/* queue module function prototypes */

/* initialize queue */
initQue(QueuePtr que,int size);

/* put item into queue */
bool enqueue(QueuePtr que,QueueData data);

/* remove data item from queue */
QueueData dequeue(QueuePtr que);

/* dealocate memory for queue */
void destroyQue(QueuePtr que);

/* print out contents of queue */
void printQue(QueuePtr que,char* name);
#endif
```

Implementation code file:

```
/* queue.c */

#include <stdio.h>
#include <stdlib.h>
#include "queue.h"
#include "vector.h"
#include "defs.h"
```

```
/* queue module functions */
/* initialize queue */
int initQue(QueuePtr que,int size)
{
/* allocate memory for vector data structure */
que->items=(QueueData*)malloc(sizeof(QueueData)*size);
if(que->items == NULL)return FALSE;
que->head=0;
que->tail=0;
que->size=size;
return TRUE;
}

/* insert items into the queue */

bool enqueue(QueuePtr que,QueueData data)
{
/* insert item at end of vector */
que->items[que->tail]=data;
(que->tail)++; /* increment tail */
que->tail= que->tail % que->size; /* wrap around */
return TRUE;
}

/* remove items from the queue */
QueueData dequeue(QueuePtr que)
{
QueueData data;
/* check if queue empty */
if(que->head==que->tail)return 0;

/* return and remove item at start of vector */
data = que->items[que->head];
(que->head)++; /* increment head */

que->head= que->head % que->size; /* wrap around */

return data;
}

/* deallocate memory for queue */
void destroyQue(QueuePtr que)
{
/* deallocate memory for vector */
free (que->items);
que->items=NULL;
que->head=0;
que->tail=0;
que->size=0;
}
```

```
        /* print out queue items */
        void printQue(QueuePtr que,char* name)
        {
        int i;

        /* check for empty vector */
        if(que->items == NULL)

        {
        printf("the list is empty\n");
        return;
        }

        /* print out queue elements */
        printf("queue: [");

        /* loop through all items in vector */
        for(i=que->head;i!=que->tail;i= (i + 1) % que->size)

        {
        printf("%d",que->items[i]); /* print out values */
        if(i < (que->tail-1))putchar(','); /* insert comma */
        }

        printf("]\n");
        }

        }

/* main function to test queue module */
int main()

        {
        Queue que; /* create queue structure */

        /* initialize queue to 10 elements */
        if(initQue(&que,10)== FALSE)return FALSE;

        enqueue(&que,5); /* put 5 into queue */
        printQue(&que,"queue: "); /* print out queue */

        /* get data item from queue and print out value */
        printf("\n data value: %d\n",dequeue(&que));
        printQue(&que,"queue: "); /* print out queue */

        enqueue(&que,1); /* put 1 into queue */
        printQue(&que,"queue: "); /* print out queue */
        enqueue(&que,2); /* put 2 into queue */
        printQue(&que,"queue: "); /* print out queue */
        enqueue(&que,3); /* put 3 into queue */
        printQue(&que,"queue: "); /* print out queue */
```

```
/* get data item from queue and print out value */
printf("\n data value: %d\n",dequeue(&que));
printQue(&que,"queue: "); /* print out queue */

/* get data item from queue and print out value */
printf("\n data value: %d\n",dequeue(&que));
printQue(&que,"queue: "); /* print out queue */

/* get data item from queue and print out value */
printf("\n data value: %d\n",dequeue(&que));
printQue(&que,"queue: "); /* print out queue */

destroyQue(&que); /* deallocate memory for queue */

return TRUE;
}
```

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 5

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 6

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 7

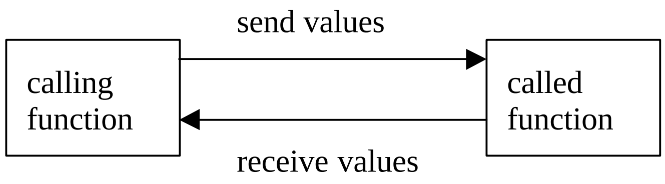
Use the Vector module to make a Queue. Call your modules vqueue.h and vqueue.c.

LESSON 2 EXERCISE 8

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 L2Ex8.c

C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 3

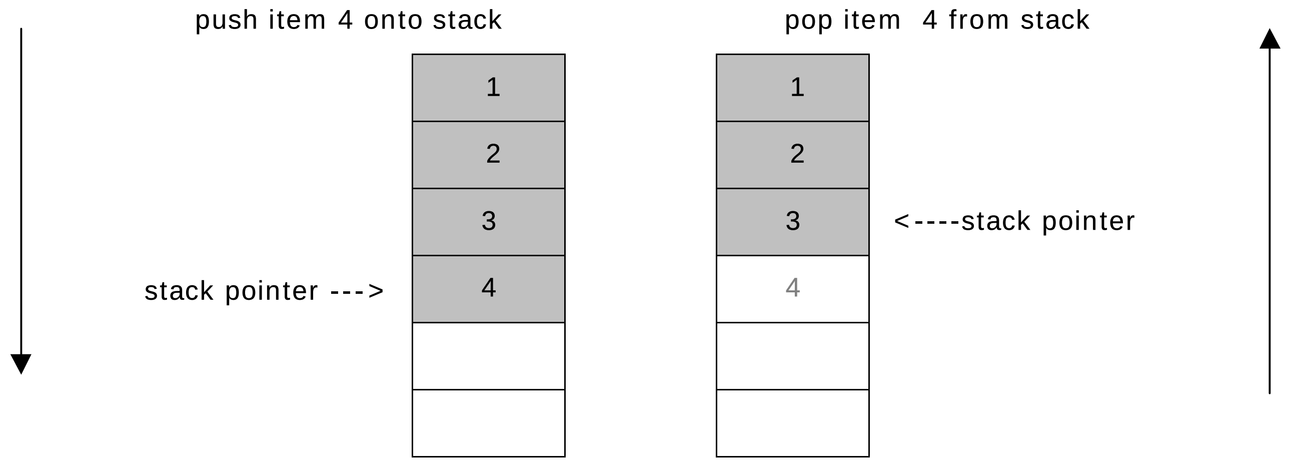
File:	CdsGuideL3.doc
Date Started:	April 15,1999
Last Update:	Dec 22, 2001
Status:	draft



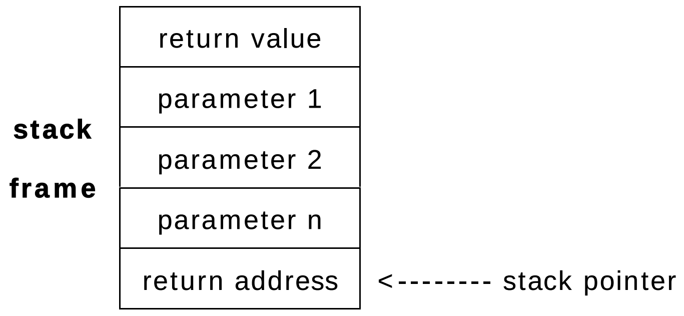
LESSON 3 RECURSION

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.



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 <stdio.h>

/* function prototype */
int mul(int x, int y);

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

/* print out answer */
printf("\n 5 times 4 is: %d",x);
}
```

```
/* 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 */
}
```

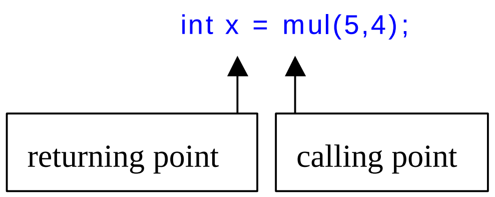
The stack frame when the **mul** function is executing is:

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

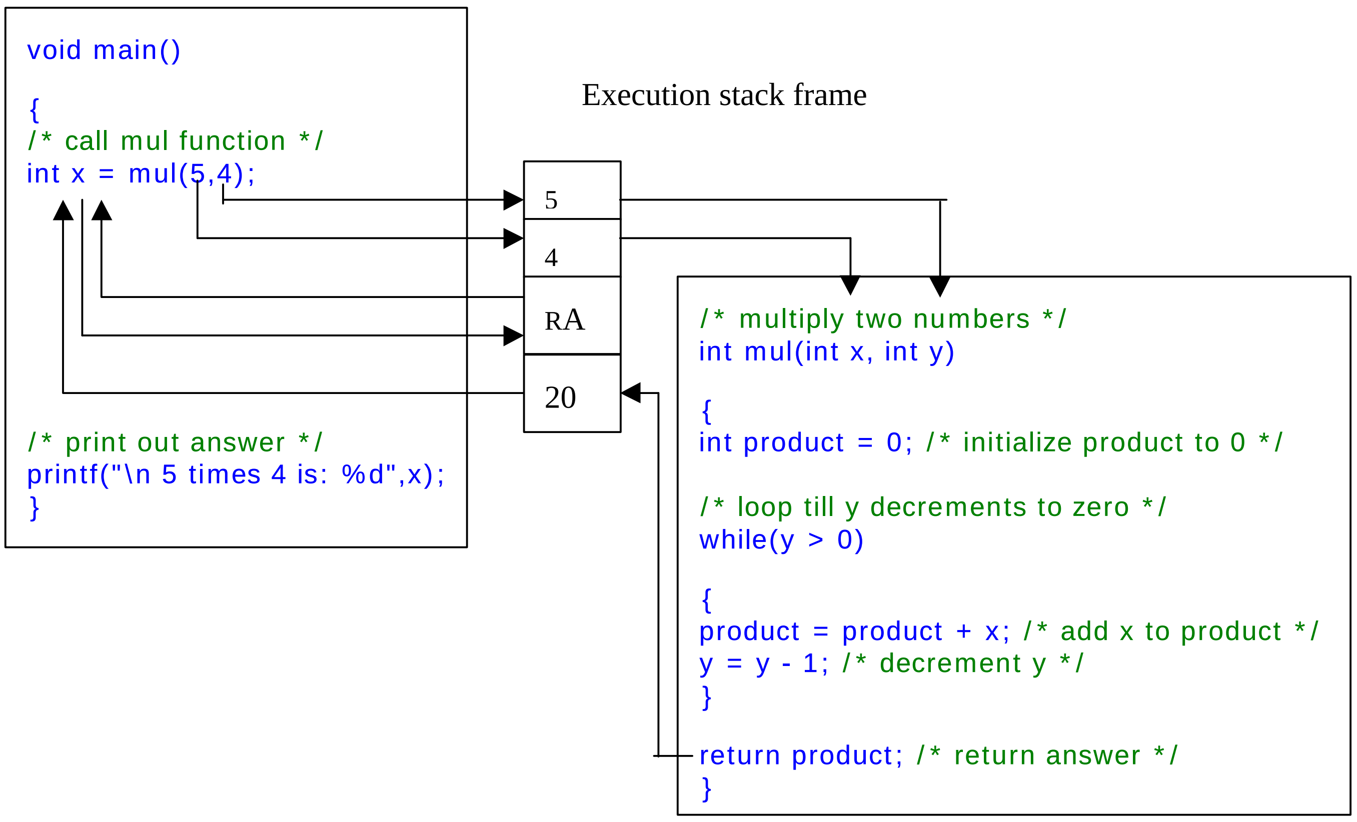
<-------- stack pointer

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.



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

Here is a data flow and operational diagram:



```
void main()

{
/* call mul function */
int x = mul(5,4);

/* print out answer */
printf("\n 5 times 4 is: %d",x);
}
```

```
/* 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 */
}
```

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.

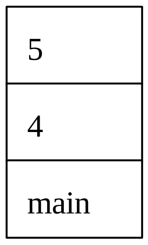
```
/* while loop multiply function */
int void mul(int x, int y)

        {
        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 <stdio.h>

/* function prototypes */
mul(int x, int y);

/* main function */
void main()

        {
        int x = mul(5,4);  /* call mul function */
        printf("\n 5 times 4 is: %d",x); /* 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:



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		
	return address	mul(5,2)	<--	stack pointer

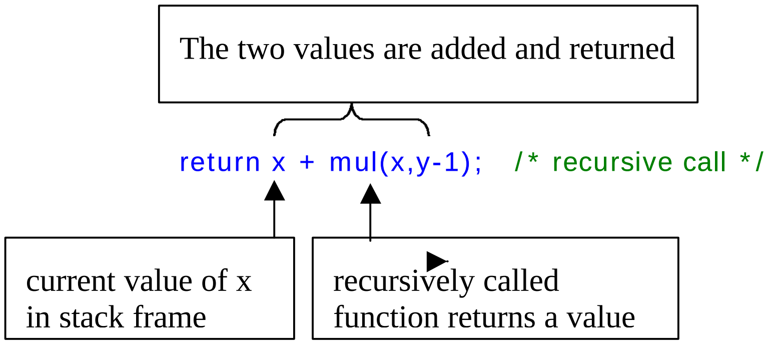
Every time a function is called a stack frame is created. For each time the recursive function terminates(returns) the stack pointer points to the last execution stack frame. Program execution is transferred to the return address stored in the stack frame. The return address can be the recursive function 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

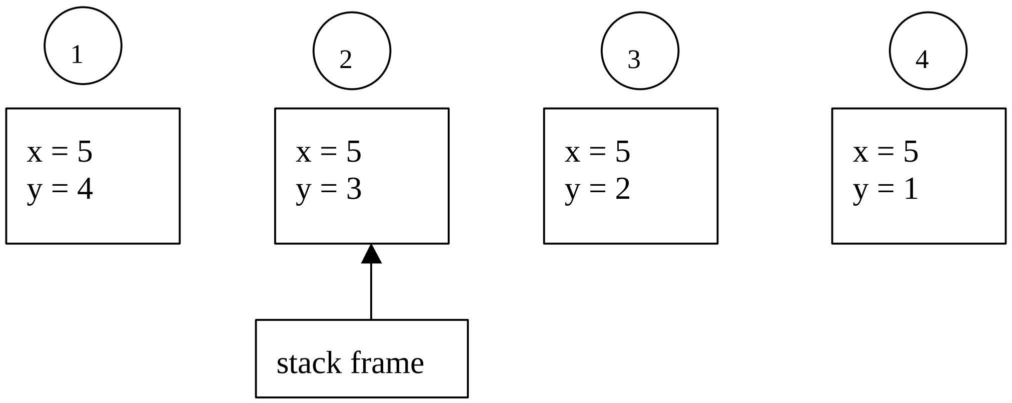
Two important things are happening here The y variable is a counter and is used to count how many recursive calls we need. For each stack frame the y and x values are pushed onto the stack. The y values are decreasing and the x value never changes. When y reaches the value 1 then the function does not 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.


The two values are added and returned
return x + mul(x,y-1); /* recursive call */
current value of x in stack frame
recursively called function returns a value

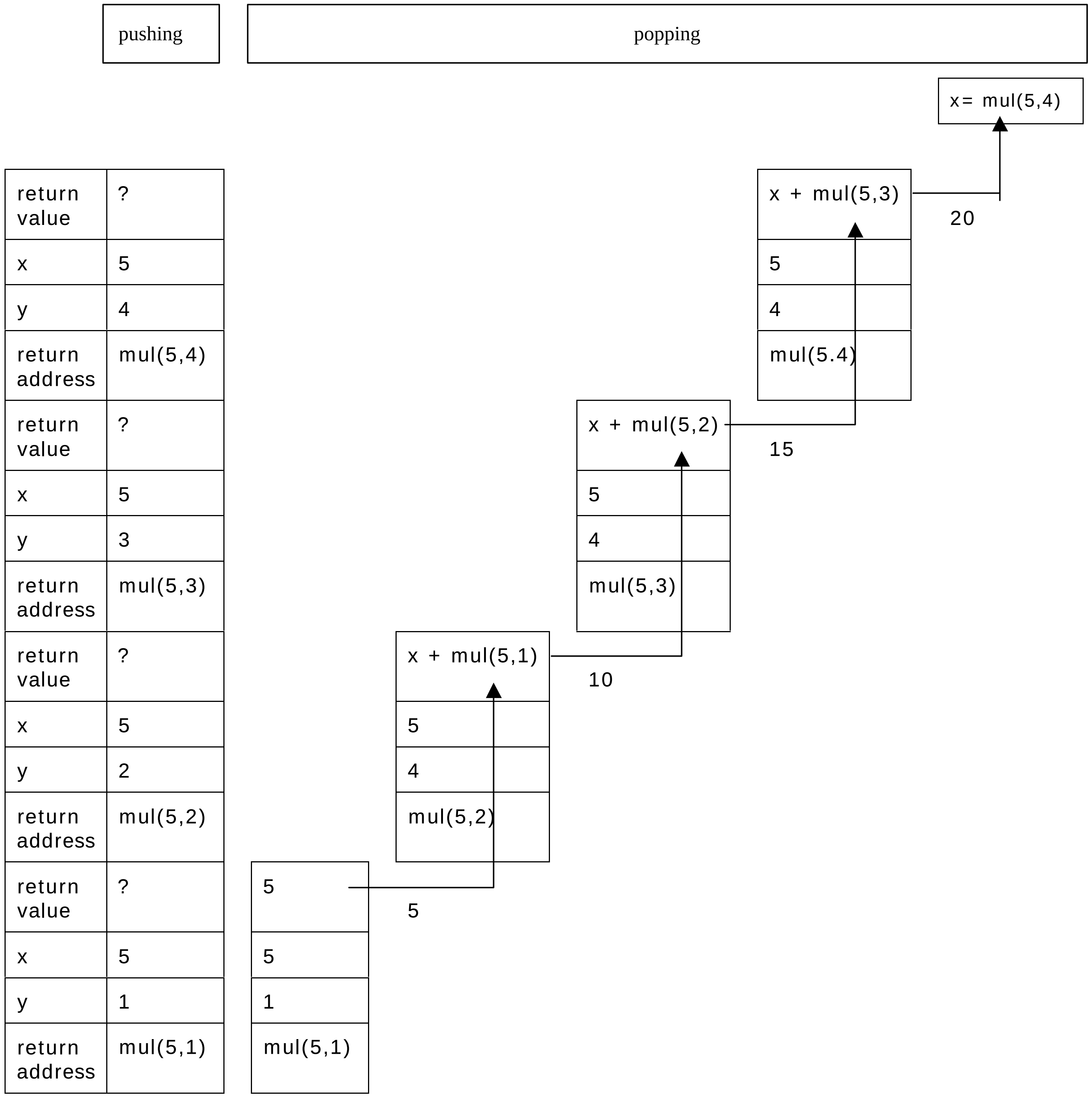
This means the recursive function is returning a value that is calculated from adding a variable value and the return value from the function calling itself. Every time the function calls itself a new stack frame is produced. When a function call itself the return value is not available until the function 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 */
}
```

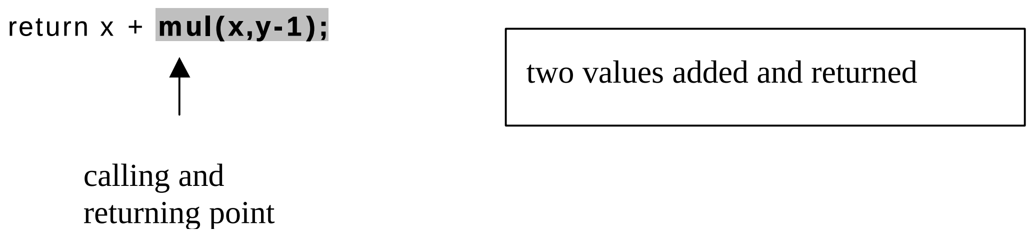
You can also thing of recursion as storing a separate set of values for every recursive call. When the function returns it selects the preceding set of values using the stack frame pointer and discards the current set of values.



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 <u>where it was called</u>. <u>not at the beginning</u> 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.



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	
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	
1	5	4	5,5	10	
2	5	4	5	15	
3	5	4	------	20	20

pushed

popped

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.c  lesson 6 program 3 */
/* this program multiplies two numbers recursively */

#include <stdio.h>
#include <stdlib.h>

/* data structures */
typedef int STACK_DATA;

typedef struct stack_type

        {
        STACK_DATA* items;
        int size;
        int stkptr;
        }STACK,*STACK_PTR;

/* function prototypes */
void initStk(STACK_PTR stk, int size); /* initialize stack module */
int push(STACK_PTR stk,STACK_DATA data);  /* push item into stack */
STACK_DATA pop(STACK_PTR stk);  /* get item from stack */
int mul(STACK_PTR stk,int x, int y);  /* multiply two numbers  */
```

```
/* main function */
void main()

        {
        STACK stk;
        int x;

        initStk(&stk,100);
        x = mul(&stk,5,4);
        printf("\n 5 times 4 is: %d",x);
        }

/* multiply two numbers using a stack */
int mul(STACK_PTR stk,int x, int y)

        {
        int i;
        int sum = 0;

        for(i=0;i<y;i++)push(stk,x);

        for(i=0;i<y;i++)sum = sum + pop(stk);

        return sum;
        }

/* stack code implementation */

/* initialize stack */
void initStk(STACK_PTR stk, int size)

        {
        stk->items=(STACK_DATA*)calloc(size,sizeof(STACK_DATA));
        stk->stkptr = 0;
        stk->size = size;
        }

/* push item into stack */
int push(STACK_PTR stk,STACK_DATA data)

        {
        if(stk->stkptr > stk->size)return 0;
        stk->items[stk->stkptr++] = data;
        return 1;
        }

/* get item from stack */
STACK_DATA pop(STACK_PTR stk)

        {
        if(stk->stkptr <= 0)return NULL;
        else return stk->items[--(stk->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 <stdio.h>

/* function prototypes */
int fib(int n);

/* main function */
void main()

        {
        int x = fib(4);
        printf("the fibonnaci number  is: %d\n",x);
        }

/* 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 <stdio.h>

/* function prototypes */
gcd1(int m, int n);
gcd2(int m, int n);

/* main function */
void main()

{
int x;
x  = gcd1(25,35);
printf("the gcd of 25 and 35 is: %d\n",x);
x  = gcd2(35,25);
printf("the gcd of 35 and 25 is: %d\n",x);
}
```

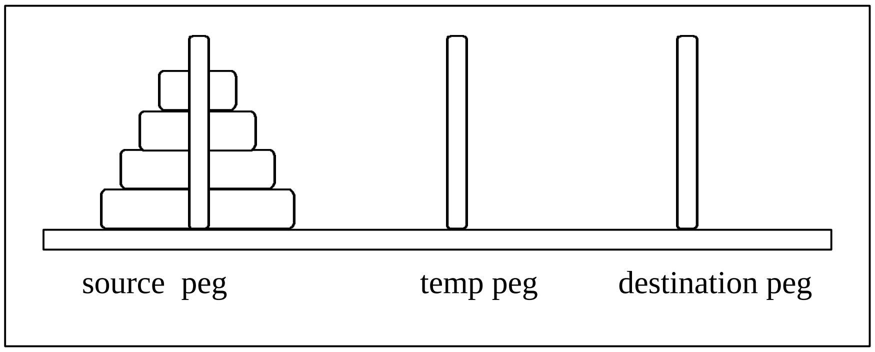
% means successive subtraction

```
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

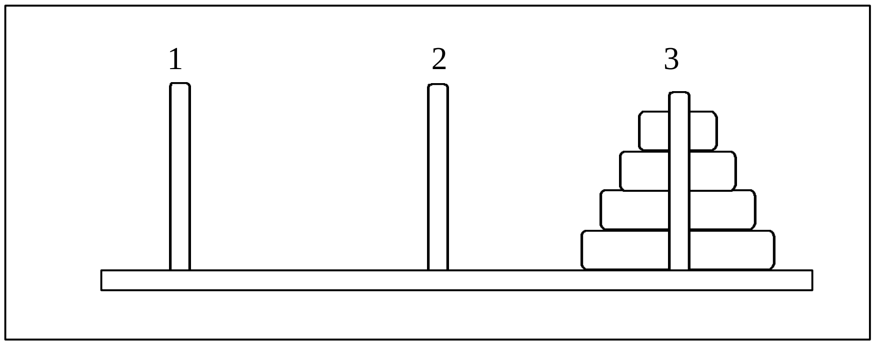
(1) move the n-1 smaller disks to the temporary peg
(2) move the n disk to the target peg
(3) move the n-1 smaller disk from the temporary peg to the target peg

Here is the recursive function for the towers of hanoi:

```
/* towers of hanoi L3p6.c  */
#include <stdio.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);
                printf("move disk %d from peg %d to peg %d\n",n,source,target);
                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

Writ a recursive program that will generate all the possible words for a 7 digit phone number where:

2	**a b c**	**5**	**j k l**	**8**	**t u v**
3	**d e f**	**6**	**m n o**	**9**	**w x y**
4	**g h i**	**7**	**j r s**		

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 $\begin{vmatrix} 4 & -5 \\ -1 & -2 \end{vmatrix}$ 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:	CdsGuideL4.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.

initial array	2	6	5	4	3	1

scan and swap

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

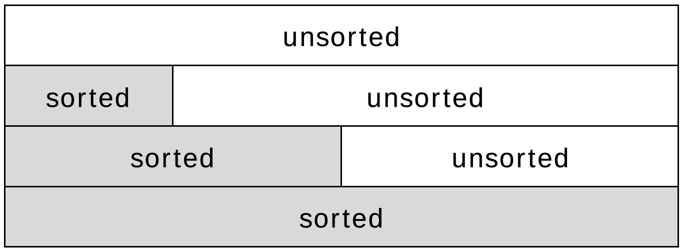
```
/* bubble sort */
#include <stdio.h>

 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 */        } swapping
                                a[j+1] = t; /* assign saved left to right  element */
                                }

                        }

                print(a,n); /* optional print out each sorted iteration */
                }

        }

        /* function to print out an array */
        void print(int a[],int n)

                {
                int i;
                printf("[ ");
                for(i=0;i<n;i++)
                printf("%d ",a[i]);
                printf(" ]\n");
                }
```

```
[ 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

```
/* insertion sort */
#include <stdio.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 */
        }
```

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

```
/* 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;
                printf("[ ");
                for(i=0;i<n;i++)
                printf("%d ",a[i]);
                printf("]\n");
                }
```

```
[ 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 <stdio.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;
                printf("[ ");
                for(i=0;i<n;i++)
                printf("%d ",a[i]);
                printf(" ]\n");
                }
```

```
[ 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  ]
```

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 <stdio.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;

        printf("[ ");

        for(i=s;i<n;i++)printf("%d ",a[i]);

        printf(" ]\n");
        }
```

```
[ 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	--

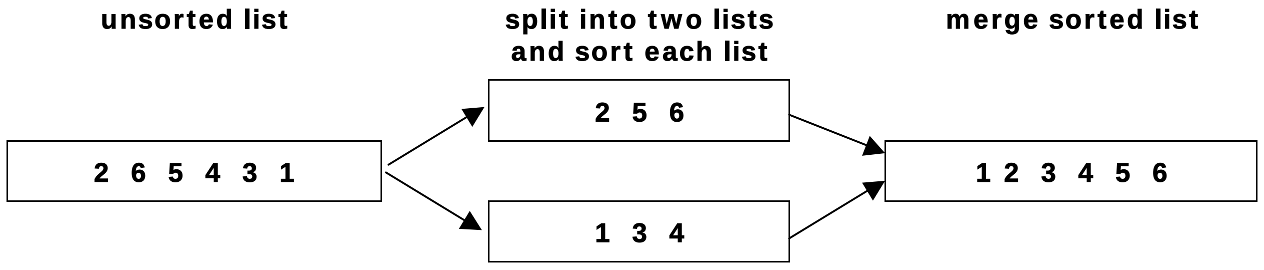
0	1	2	3	4	5
2	6	5	4	3	1
2	6	5	4	3	1
2	6	5	4	3	1
2	6	5	4	3	1
2	6	5	4	3	1
2	6	5	4	6	1
2	3	5	4	6	1
2	3	5	4	6	2
2	3	5	4	6	1
2	3	5	4	6	1
2	3	5	4	6	5
2	3	1	4	6	5
1	3	5	4	5	2
2	3	1	4	6	5
2	3	1	4	6	5
2	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5
1	3	2	4	6	5

i	delta	left	right	key a[right]	left-delta	a[left-delta]	outer loop	inner loop
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
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	3	4	6	5
1	2	3	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.c */
#include <stdio.h>
#include <stdlib.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 = (int*)calloc(N+1,sizeof(int)); /* allocate temp array */
        if(b != NULL)

                {
                print(a,N);
                msort(a,b,0,N-1);  /* call merge sort */
                print(a,N);
                free(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 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;
        printf("[ ");
        for(i=0;i<n;i++)printf("%d ",a[i]);
        printf(" ]\n");
        }
```

merges two sorted arrays

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

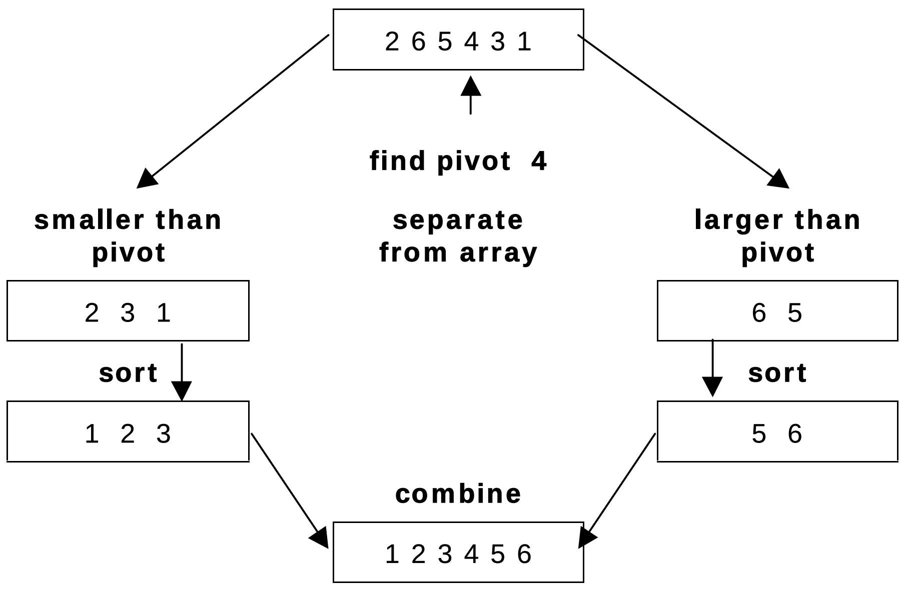
	stack pairs left, right	function	left lpos	middle rpos	right rend	a	b
0	ra	msort	0	---	5	2 6 5 4 3 1	0 0 0 0 0 0
1	ra 0,5	msort	0	2	5	2 6 5 4 3 1	0 0 0 0 0 0
2	ra 0,5 0,2	msort	0	1	2	2 6 5 4 3 1	0 0 0 0 0 0
3	ra 0,5 0,2 0,1	msort	0	0	1	2 6 5 4 3 1	0 0 0 0 0 0
4	ra 0,5 0,2 0,1	msort	0	0	0	2 6 5 4 3 1	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 */
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);
}
```

sort two arrays recursively

```
/* 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;
}
```

choose partition

```
/* print out array */
void print(int a[],int n)

        {
        int i;
        printf("[ ");

        for(i=0;i<n;i++) printf("%d ",a[i]);
        printf(" ]\n");
        }
```

.

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

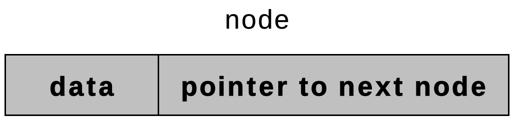
Generate 10000 ransdom numbers. Use quick sort to sort the numbers. Use quick sort to sort the numbers. Which one is fastest?

C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 5

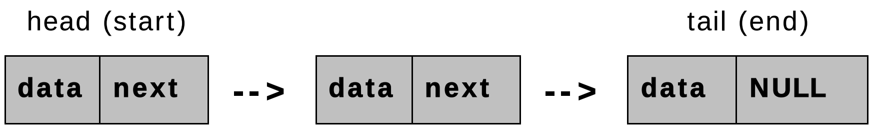
File:	CdsGuideL5.doc
Date Started:	July 24,1998
Last Update:	Nov 22,2001
Status:	proof

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.

list.h	header file	The header file contains the function prototypes.
Node	data structure	The Node data structure holds the data and link to next node.
List	data structure	The List data structure holds the start and end of list pointers and number of nodes in list.
list.c	code	The code contains the function definitions.

Here's the code for the List 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
        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.h */
#include "defs.h"

/* node data */
typedef int NodeData;

/* node data structure */
typedef struct NodeType

        {
        struct NodeType* next;
        NodeData data;
        }Node, *NodePtr;

/* the link list will have head and tail pointers */
typedef struct

        {
        NodePtr head;
        NodePtr tail;
        }List,*ListPtr;
```

```
/* function prototypes for single link list */

/* initialize link list structure */
void initList(ListPtr list);

/* insert item into list */
int insertList(ListPtr list,NodeData data);

/* remove data item from list */
bool removeList(ListPtr list,NodeData data);

/* find data item in list */
/* return pointer to node if found, otherwise return NULL */
NodePtr findList(ListPtr list, NodeData data);

/* find data item in list */
/* return index of node if found, otherwise return -1 */
int findListAt(ListPtr list, NodeData data);

/* print list */
void printList(ListPtr list);

/* de-allocate memory for list */
void destroyList(ListPtr list);
```

Implementation code file for Link List:

```
/* list.c */
/* single link list code implementation */
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

int main()

        {
        List list; /* create a list structure */
        initList(&list); /* initialize link list pointers */
        insertList(&list,10); /* insert 10 into link list */
        printList(&list); /* print out link list */
        insertList(&list,8); /* insert 8 into link list */
        printList(&list); /* print out link list */
        insertList(&list,5); /* insert 5 into link list */
        printList(&list); /* print out link list */
        insertList(&list,20); /* insert 20 into link list */
        printList(&list); /* print out link list */
        if(findList(&list,8))printf("item 8 found\n");
        if(findList(&list,15))printf("item 15 found\n");
        removeList(&list,15); /* remove 15 from list */
        printList(&list); /* print out link list */
        removeList(&list,8); /* remove 8 from list */
        printList(&list); /* print out link list */
        removeList(&list,20); /* remove 20 from list */
        printList(&list); /* print out link list */
        removeList(&list,8); /* remove 8 from list */
        printList(&list); /* print out link list */
        removeList(&list,5); /* remove 5 from list */
        printList(&list); /* print out link list */
        removeList(&list,10); /* remove 10 from list */
        printList(&list); /* print out link list */
        destroyList(&list);/* deallocate memory for list */
        return TRUE;
        } /* 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:
```

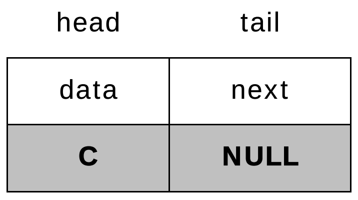
```
/* initialize list */
void initList(ListPtr list)

        {
        list->head = NULL; /* set head to default value */
        list->tail = NULL; /* set tail to default value */
        }

/* destroy list */
void destroyList(ListPtr list)

        {
        NodePtr next;
        NodePtr curr = list->head; /* point to start of list */

        /* loop till end of list */
        while(curr != NULL)

                {
                next = curr->next; /* get pointer to next node */
                free(curr); /* free current node */
                curr = next; /* current points to next node */
                }

        list->head = NULL; /* set head to default value */
        list->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
*/
```

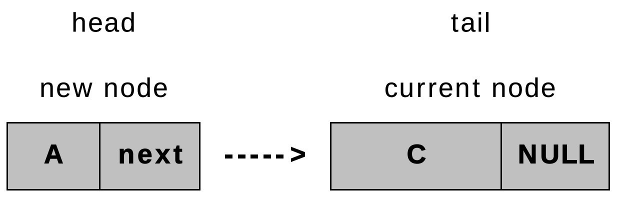
empty list

head	NULL
tail	NULL

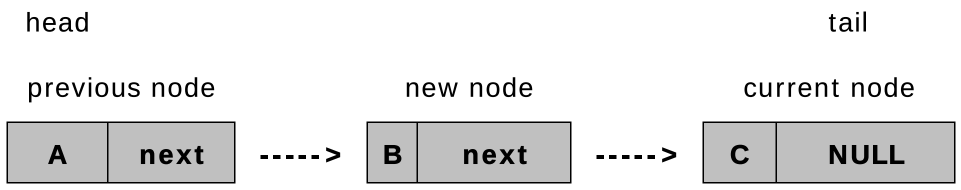
insert "C" into empty list



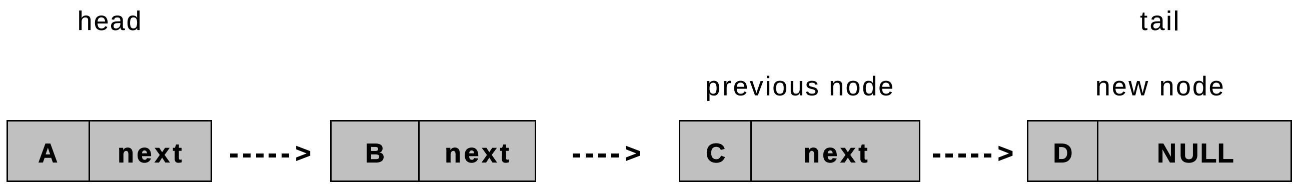
insert "A" at start of list (no previous node)



insert "B" into middle of list

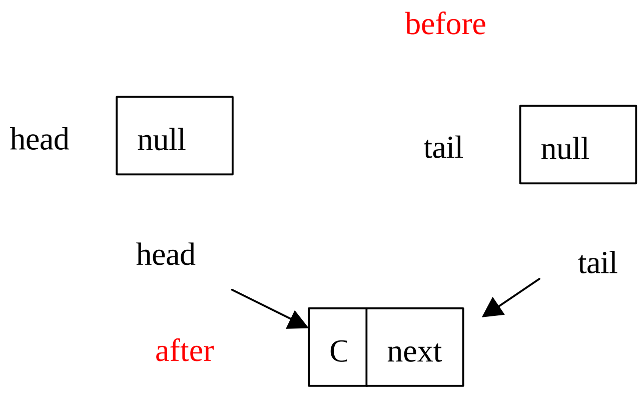


insert "D" at end of list (no current node)



```
int insertList(ListPtr list,NodeData data)

        {
        NodePtr prev = NULL; /* pointer to a previous node */
        NodePtr curr = NULL; /* pointer to a current node */

        /* allocate memory for new node */
        NodePtr node = (NodePtr)malloc(sizeof(Node));
        if (node == NULL)return FALSE;

        node->data = data; /* assign data to node STRUCTURE */
        node->next= NULL; /* assume next node is always NULL */

        /* check for empty list */
        if(list->head == NULL)

                {
                list->head = node; /* set head to point to new node */
                list->tail = node; /* set tail to point to new node */
                return TRUE;
                }
```

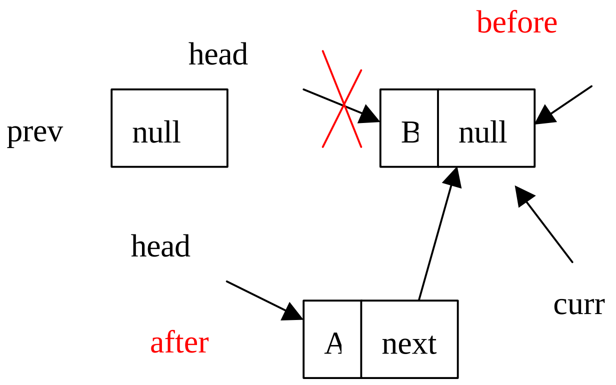
insert C into empty list



```
        /* find out where to insert new node, insert in ascending order */
        curr = list->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 */
                }
```

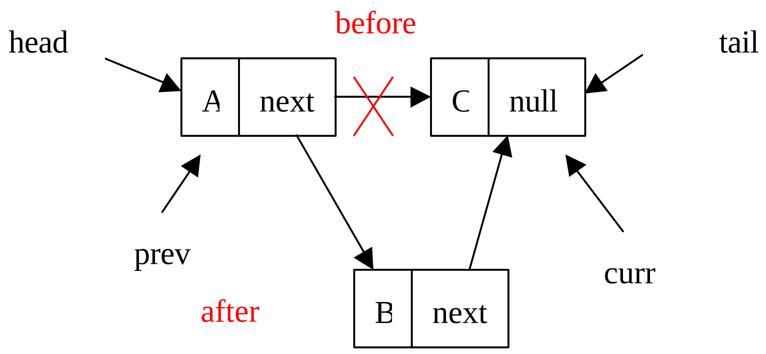
```
/* check for start of list */
/* node points to start of list */
if(prev == NULL)

        {
        node->next = list->head; /* new node points to start of list */
        list->head = node; /* make head point to new node */
        }
```

insert A at beginning of list

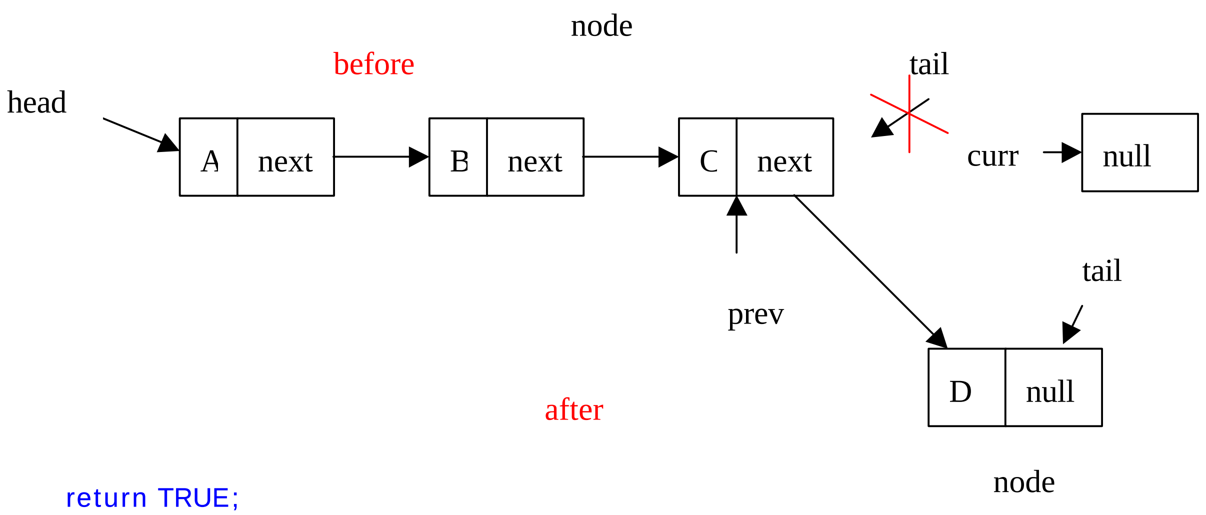


```
/* check for middle of list */
else

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

insert B into middle of list

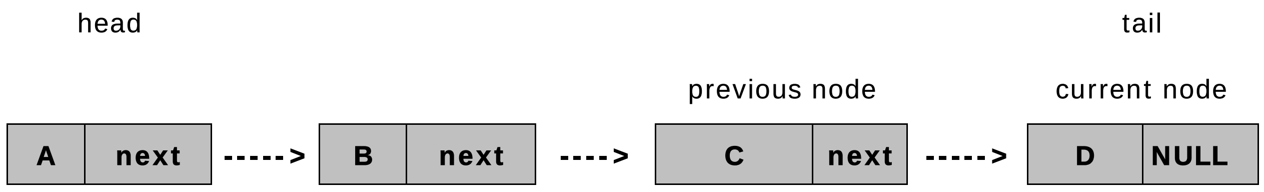


insert D at end of list

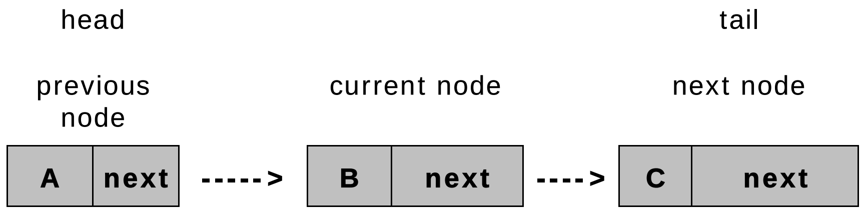


```
        return TRUE;

        } /* 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 */
```

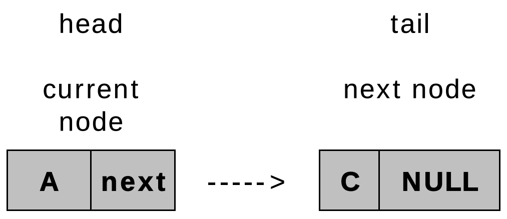
delete "D" at end of list (no next node)



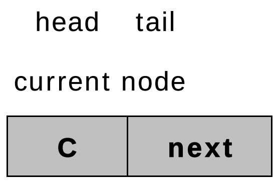
delete "B" at middle of list



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



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

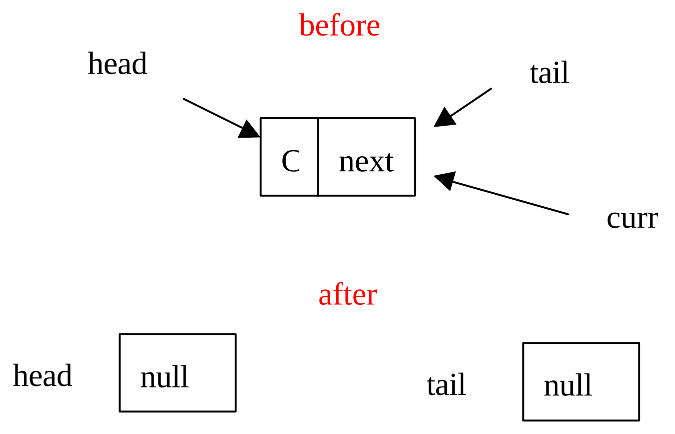


list is empty

head
NULL
tail
NULL

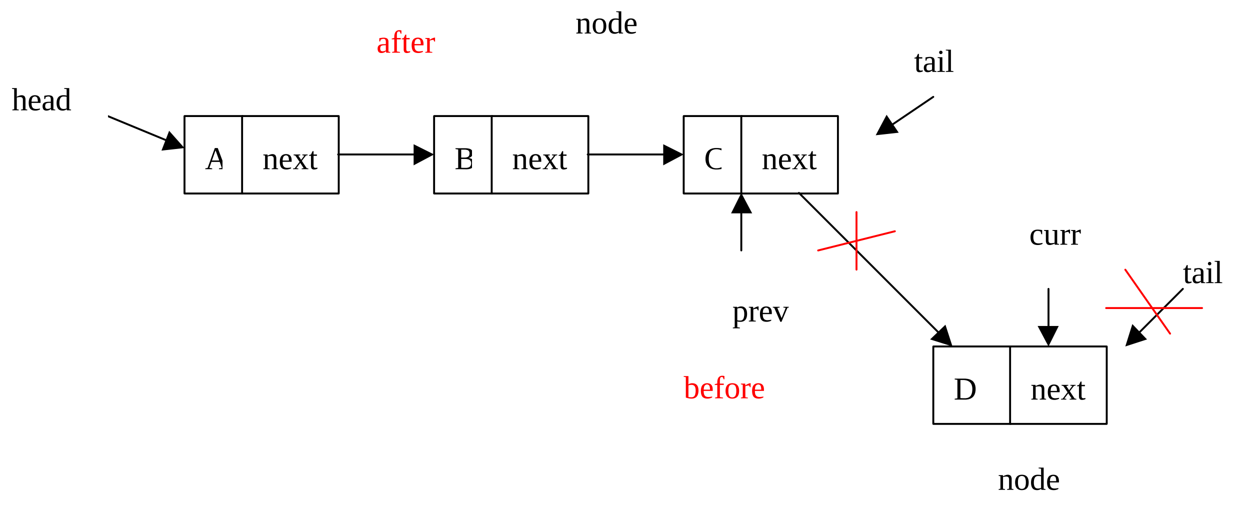
```
/* remove node from list */
bool removeList(ListPtr list,NodeData data)

        {
        NodePtr prev= NULL; /* pointer to a previous node */
        NodePtr curr = list->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(list->head==list->tail)

                {
                /* remove last one node in link list */
                list->head=NULL; /* set head to empty value */
                list->tail=NULL; /* set tail to empty value */
                }
```

remove C from list



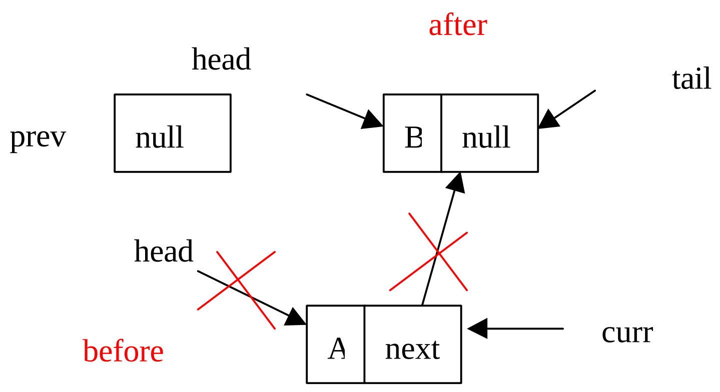
```
/* check for end of list */
else if(curr->next == NULL)

        {
        /* remove node at end of list */
        prev->next=NULL; /* set prev node to point to no node */
        list->tail = prev; /* set tail to point to previous node */
        }
```

remove D from end of list



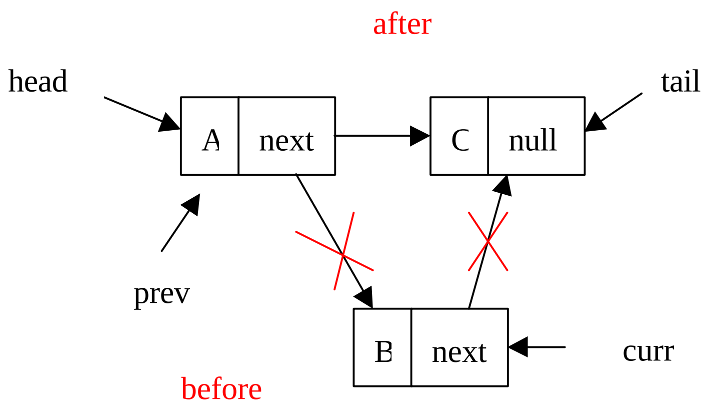
```
/* check for start of list */
else if(prev == NULL)

        /* remove node from start of list */
        list->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



```
free (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 */
NodePtr findList(ListPtr list, NodeData data)

        {
        NodePtr curr = list->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 findListAt(ListPtr list, NodeData data)

        {

        int i = 0; /* set index to zero */
        NodePtr curr = list->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 ERROR;
        }
```

```
/* print a list */
/* print data elements in list using recursion */
/* function calls itself until last node reached */
void printList(ListPtr list)

	{
	NodePtr curr = list->head; /* point to start of list */
	printf("list: ");

	/* loop till end of list */
	while(curr != NULL)

		{
		printf("%d ",curr->data); /* print out item value */
		curr = curr->next; /* point to next data item */
		}

	putchar('\n');
	}
```

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 functionon **iterate()** that intializes a node pointer to the start of the list. Write funvtions **next()** tht returns the curRent node and sets the node pointer to the next node.

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.h and Listr.c.

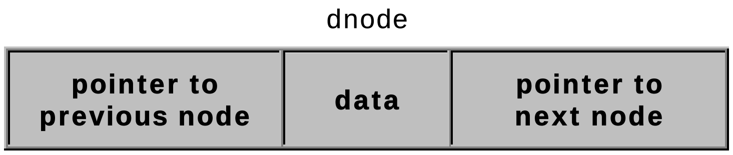
C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 6

File:	CdsGuideL6.doc
Date Started:	July 24,1998
Last Update:	Nov 22, 2001
Status:	proof

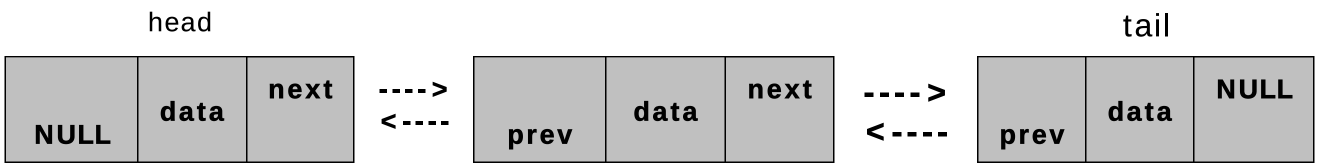
LESSON 6 DOUBLE LINK LISTS

DOUBLE LINK LIST

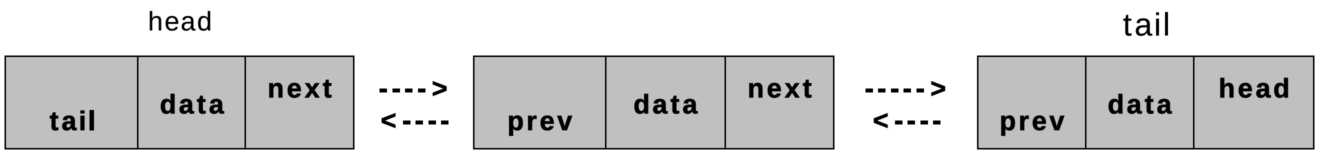
The double link list is similar to the single link list, but it has link pointer to nodes in both directions. There is now a previous pointer and a next pointer.



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.

dlist.h	header file	The header file contains the function prototypes.
DNode	data structure	The DNode data structure holds the data and links to next and previous nodes.
DList	data structure	The List data structure holds the start and end of list pointers and number of nodes in list.
dlist.c	code	The code contains the function definitions.

Here's the code for the DList 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 */

        #definre FALSE 0
        #define TRUE 1
        typedef  int  bool;
```

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

```
/* dlist.h */
/* double link list */

#include "defs.h"

/* double link list node data */
typedef int DNodeData;

/* double node stucture */
typedef struct DNodeType

        {
        struct DNodeType  *prev;
        DNodeData data;
        struct DNodeType *next;
        }DNode,*DNodePtr;
```

```
/* double link list pointers */
typedef struct

        {
        DNodePtr head;
        DNodePtr tail;
        }DList,*DListPtr;

/* prototypes */

/* initialize double link list pointers */
void initDList(DList* dlist);

/* free double link list nodes */
void destroyDList(DListPtr dlist);

/* insert item into double link list */
int insertDList(DListPtr dlist,DNodeData data);

/* remove item from double link list */
bool removeDList(DListPtr dlist,DNodeData data);

/* find item in double link list */
/* return address of double node if found otherwise NULL */
DNodePtr findDList(DListPtr dlist,DNodeData data);

/* find item in double link list */
/* return index if found otherwise -1 */
int findDListAt(DListPtr dlist,DNodeData data);

/* print out double link list items from start of list */
void printStart(DListPtr list);

/* print out double link list items from end of list */
void printEnd(DListPtr list);
```

Implementation code file for Double Link List:

```
/* dlist.c */

/*double link list */

#include <stdio.h>
#include <stdlib.h>
#include "dlist.h"

/* main program */
int main()

        {
        DList dlist;
        initDList(&dlist);
        insertDList(&dlist,10);
        printStart(&dlist);
        printEnd(&dlist);
        insertDList(&dlist,8);
        printStart(&dlist);
        printEnd(&dlist);

        insertDList(&dlist,5);
        printStart(&dlist);
        printEnd(&dlist);
        insertDList(&dlist,20);
        printStart(&dlist);
        printEnd(&dlist);
        printf("found 8 at index %d\n",findDListAt(&dlist,8));
        printf("found 15 at index %d\n",findDListAt(&dlist,15));
        removeDList(&dlist,10);
        printStart(&dlist);
        printEnd(&dlist);
        removeDList(&dlist,15);
        printStart(&dlist);
        printEnd(&dlist);
        removeDList(&dlist,20);
        printStart(&dlist);
        printEnd(&dlist);
        removeDList(&dlist,8);


        printStart(&dlist);
        printEnd(&dlist);
        removeDList(&dlist,5);
        printStart(&dlist);
        printEnd(&dlist);
        destroyDList(&dlist);
        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
```

```
/* initialize dlist pointers */
void initDList(DList* dlist)

        {
        dlist->head = NULL; /* set head to empty value */
        dlist->tail = NULL; /* set tail to empty value */
        }
```

```
/* free nodes in dlist */
void destroyDList(DListPtr dlist)

        {
        DNodePtr next; /* pointer to a next dnode */
        DNodePtr curr = dlist->head; /* point to start of list */

        /* loop till end of list */
        while(curr != NULL)

                {
                next = curr->next; /* point to next node */
                free(curr); /* free current dnode */
                curr = next; /* set curr to next dnode */
                }

        dlist->head = NULL; /* set head to empty value */
        dlist->tail = NULL; /* set tail to empty value */
        }
```

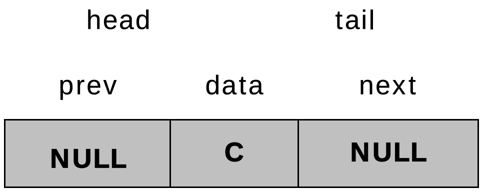
When insert an item into double link list there are 4 conditions to watch out for :

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

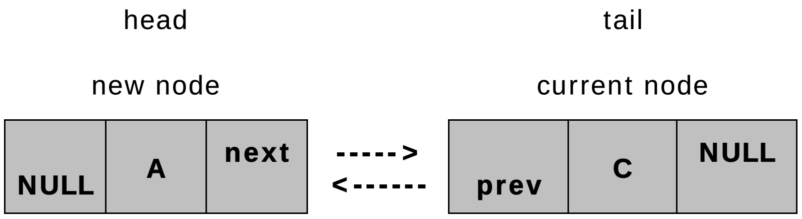
empty list

head
NULL
tail
NULL

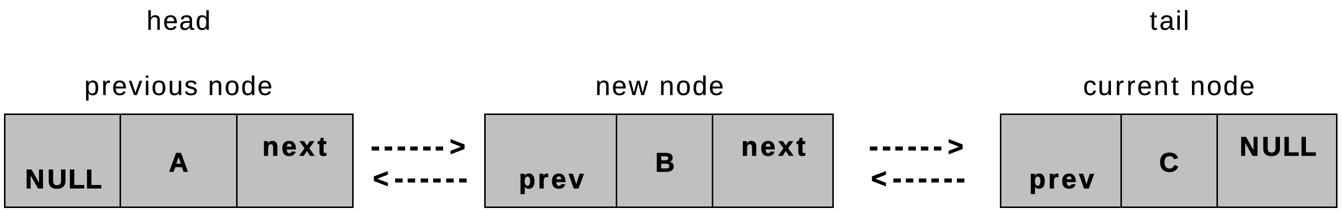
insert "C" into empty list



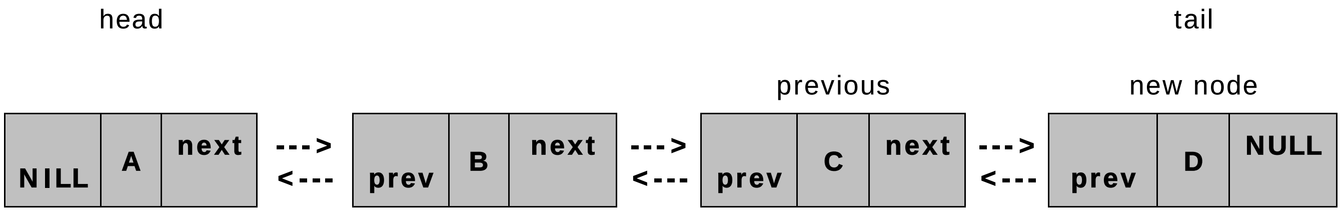
insert at "A" start of list (no previous node)



insert "B" in middle of list

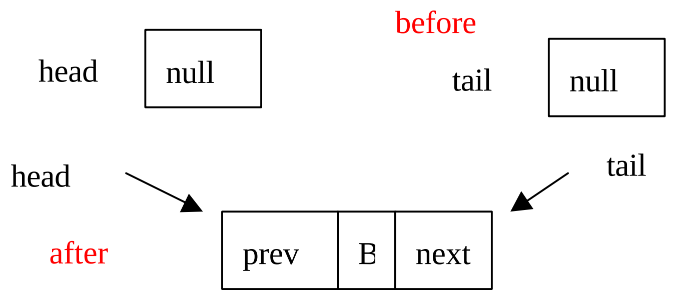


insert "D" at end of list (no current node)



```
/* 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
*/

 int insertDList(DListPtr dlist,DNodeData data)

        {
        DNodePtr curr=NULL; /* pointer to a current dnode */
        DNodePtr prev=NULL; /* pointer to a previous dnode */
        DNodePtr next=NULL; /* pointer to a next dnode */

        /* allocate memory for new dnode */
        DNodePtr dnode = (DNodePtr)malloc(sizeof(DNode));
        if (dnode == NULL)return FALSE;

        /* assign data to dnode structure */
        dnode->data = data;
        dnode->next = NULL;
        dnode->prev = NULL;

        /*check for empty list */
        if(dlist->head == NULL)

                {
                dlist->head = dnode; /* head points to new node */
                dlist->tail = dnode; /* tail points to new node */
                return TRUE;
                }
```

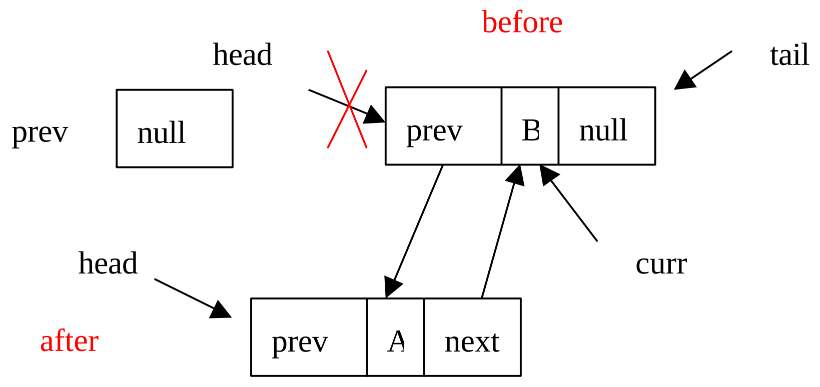
insert C into empty list



```
// find out where to insert new dnode
```

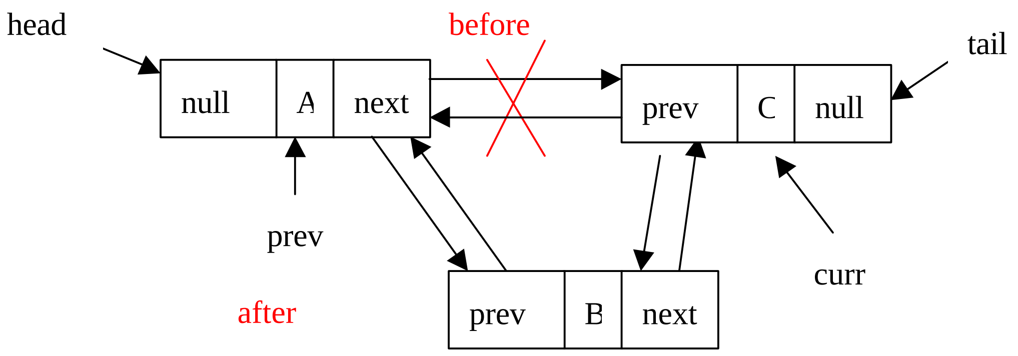
```
/* find out where to insert new node */
curr = dlist->head; /* point to start of dlist */

/* 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 */
        dlist->head = dnode; /* head points to new dnode */
        }
```

insert A at beginning of list

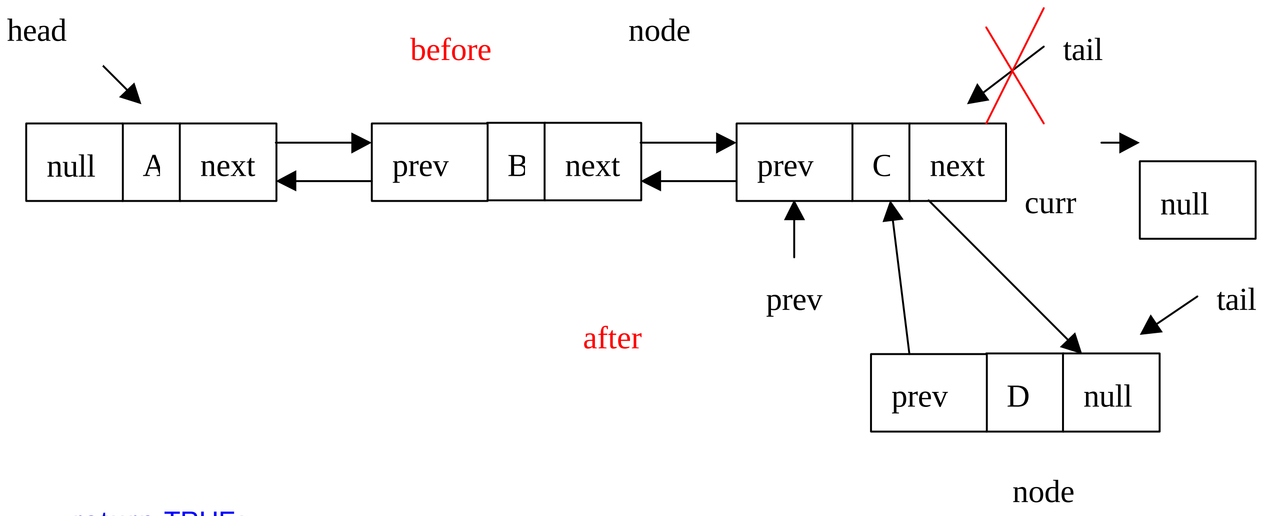


```
/* check for middle of list */
else

        {
        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)dlist->tail = dnode;
        /* current previous points to new dnode */
        else curr->prev = dnode;
        }
```

insert B into middle of list



insert D at end of list

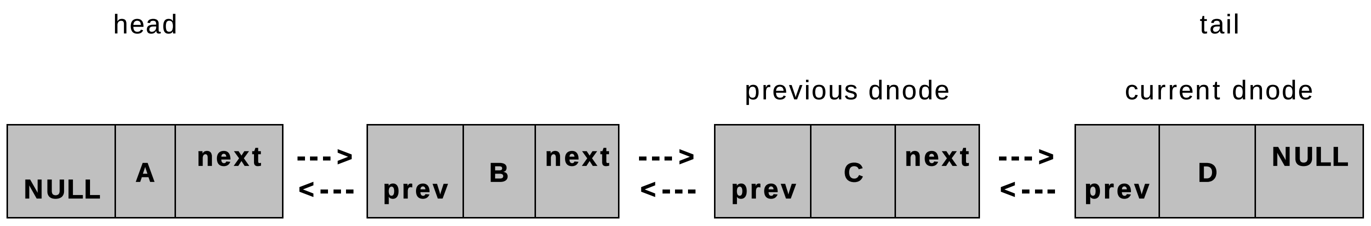


```
 return TRUE;
}
```

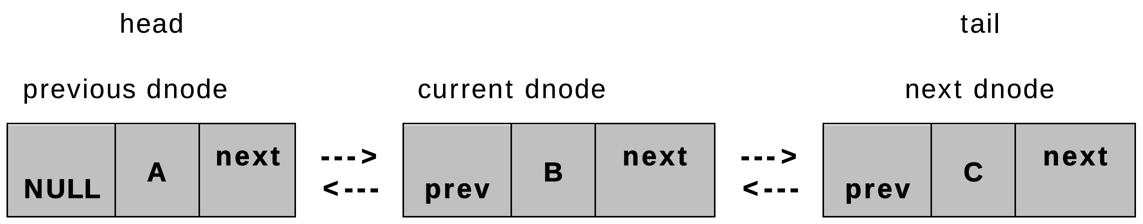
When deleting node item from a double link list there are four conditions to watch out for:

(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

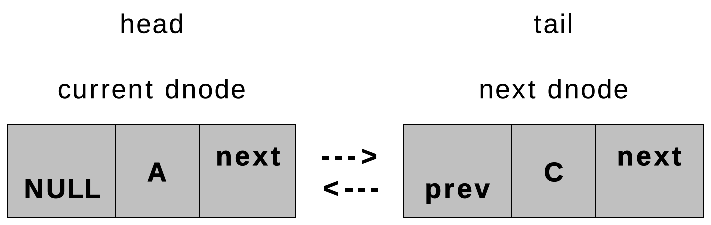
delete "D" at end of list (no next dnode)



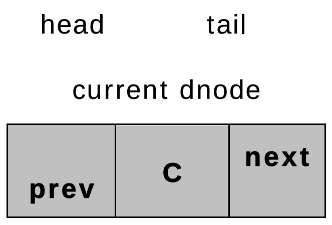
delete "B" at middle of list



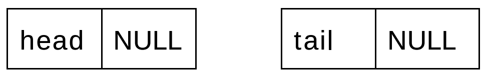
delete "A" at start of link list (no previous dnode)



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

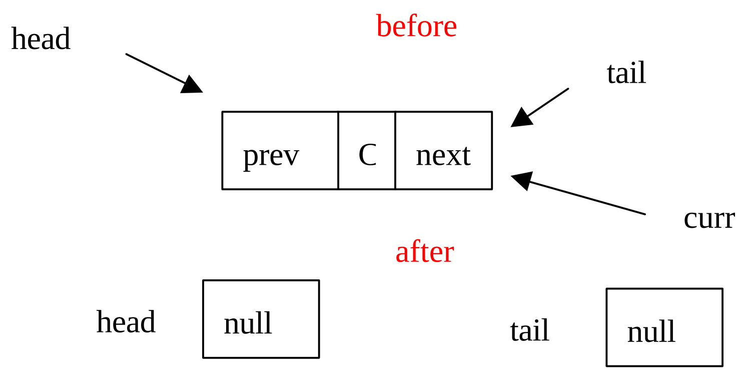


list is empty



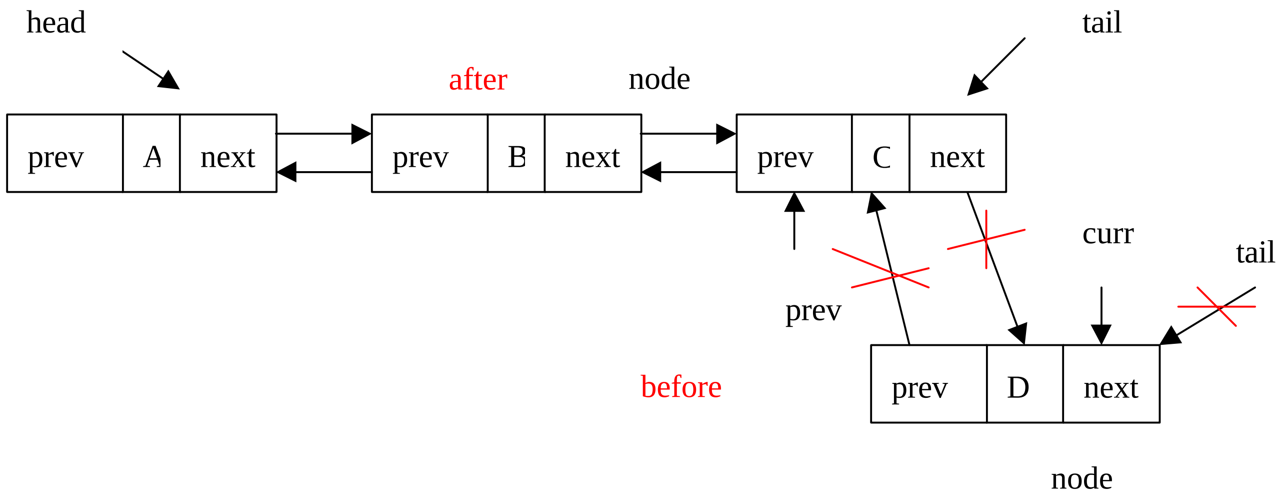
```
/* delete node from dlist
* ----------------------
* four conditions to watch out for
* remove from end of list
* remove from middle of list
* remove from start of list
* remove last item in list
*/

bool removeDList(DListPtr dlist,DNodeData data)

        {
        DNodePtr prev = NULL; /* pointer to a previous dnode */
        DNodePtr next = NULL; /* pointer to a next dnode */
        DNodePtr curr = dlist->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(dlist->head==dlist->tail)

                {
                dlist->head=NULL; /* set head to empty value */
                dlist->tail=NULL; /* set tail to empty value */
                }
```

remove C from list



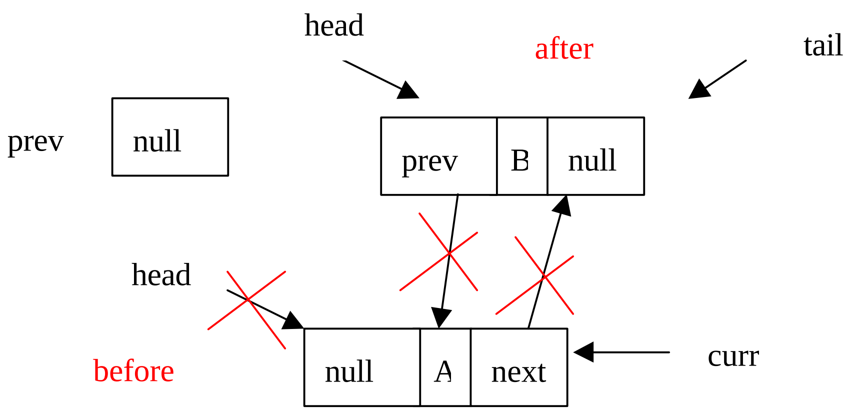
```
/* check for end of list */
else if(curr->next == NULL)

        /* remove dnode from end of list */
        {
        prev->next = NULL; /* set previous dnode next to empty */
        dlist->tail = prev; /* set tail to previous node */
        }
```

remove D from end of list



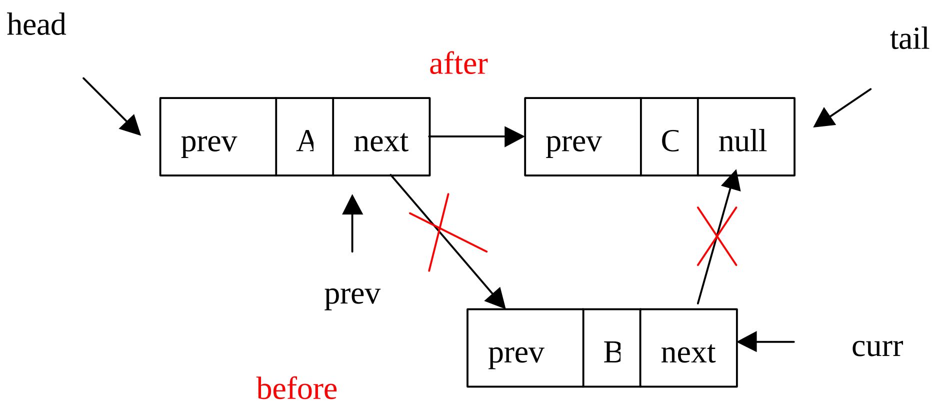
```
/* check for start of list */
else if(prev == NULL)

        /* remove dnode from start of list */
        {
        dlist->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



```
        free(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 findDListAt(DListPtr dlist, DNodeData data)

        {
        int i=0; /* index counter */
        DNodePtr curr = dlist->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 ERROR; /* item not found */
        }
        /* find data element in list */
        /* if found return dnode */
        /* if not found return NULL */
        DNodePtr findDList(DListPtr dlist, DNodeData data)

        {
        DNodePtr curr = dlist->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 */
void printStart(DListPtr dlist)

        {
        DNodePtr curr=dlist->head; /* point to start of dlist */

        /* check if dlist empty */
        if(curr == NULL)

                {
                printf("double list is empty\n");
                return;
                }

        printf("dlist from start: ");

        /* loop till end of list */
        while(curr != NULL)

                {
                printf("%d ",curr->data); /* print out data item */
                curr = curr->next; /* point to next item */
                }

        putchar('\n');
        }
```

```
/* print out items at end of dlist */
void printEnd(DListPtr dlist)

          {
          DNodePtr curr=dlist->tail; /* point to end of dlist */

          /* check if dlist empty */
          if(curr == NULL)

                    {
                    printf("double list is empty\n");
                    return;
                    }

          printf("dlist from end: ");

          /* loop till start of list */
          while(curr != NULL)

                    {
                    printf("%d ",curr->data); /* print out data item */
                    curr = curr->prev; /* point to previous dnode */
                    }

          putchar('\n');
          }
```

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.

C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 7

File:	CdsGuideL7.doc
Date Started:	April 15,1999
Last Update:	Nov 22,2001
Status:	draft

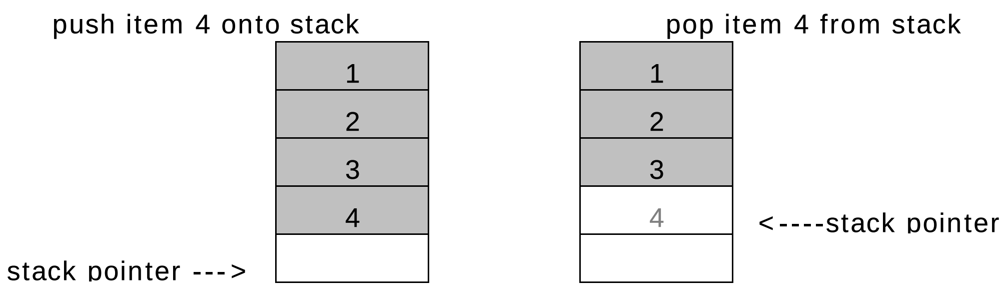
LESSON 7 ABSTRACT DATA TYPES USING LINK LISTS

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.

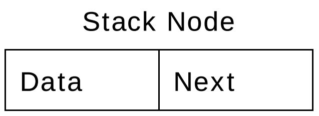
Link List ADT	Implementation Difficulty	Worst Case insertion Time	Worst Case retrieval Time	Memory Requirements
stack	easy	O(1)	O(1)	moderate
queue	easy	O(1)	O(1)	moderate
priority queue	difficult	O(n)	O(1)	moderate
dequeue	easy	O(1)	O(1)	moderate

STACK LIST ADT MODULE

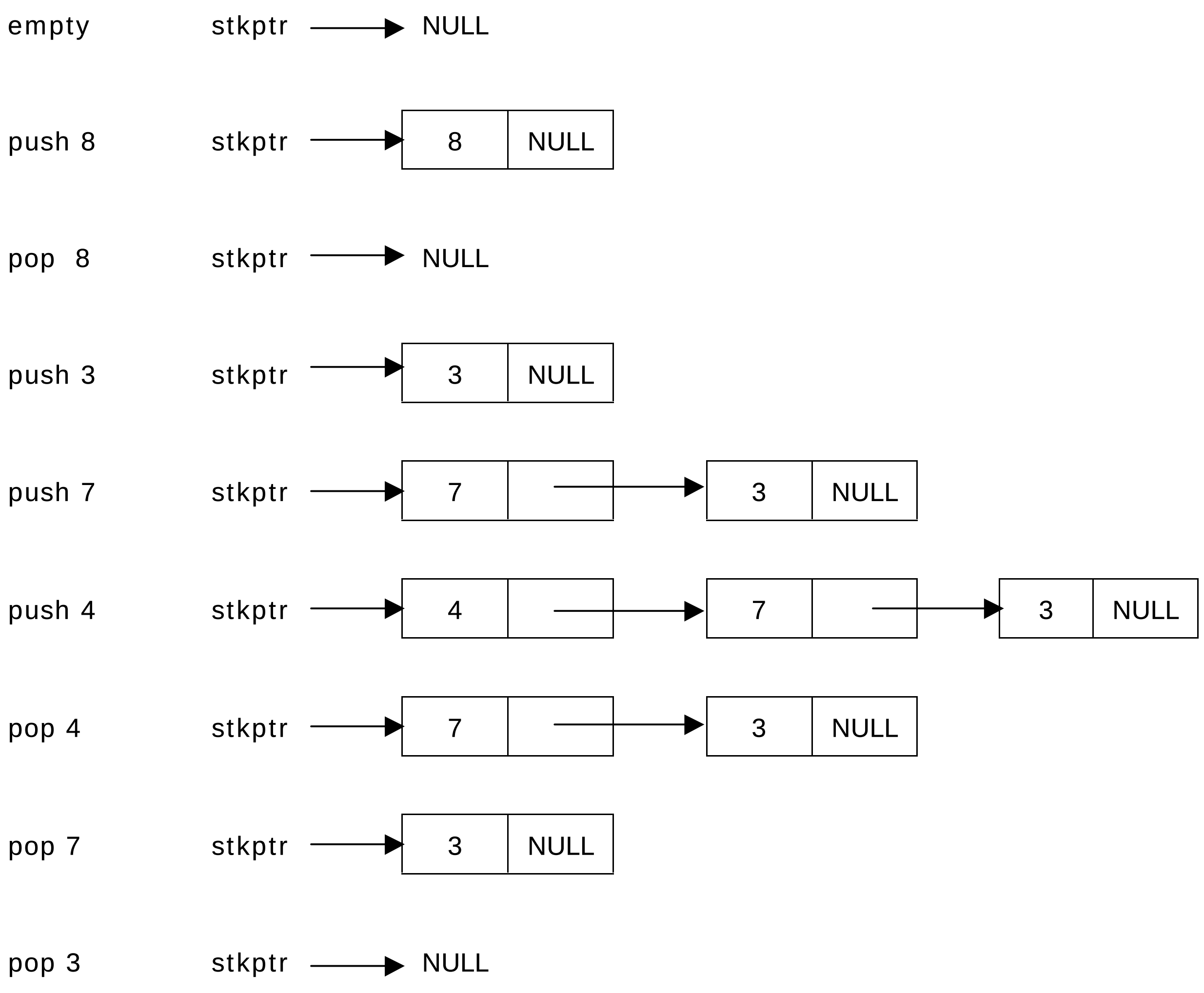
A **Stack** allows you to insert and retrieve items as **last in first out** (LIFO) into a list. This means if you insert the number 1, 2 then 3 you will get back 3, 2 and 1 in the reverse order. Stack operations are known as **push** and **pop**. You use the push operation to insert an item into the 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 **top**. The end of the stack is known as the stack **bottom.** 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.



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.

stklist.h	header file	The header file contains the function prototypes.
STK_NODE	data structure	The STK_NODE data structure holds the data and links to next and previous nodes.
STKLIST	data structure	The STKLIST data structure holds the start of list and number of nodes in list.
stklist.c	code	The code contains the 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
typedef int bool;
#define TRUE  1
#define FALSE 0
#define ERROR -1
#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.h */
#ifndef __STKLIST
#define __STKLIST

#include "defs.h"

/* stack module data items */
typedef int STK_DATA;

typedef struct stk_node_type

{
STK_DATA data;
struct stk_node_type *next;
}STK_NODE,*STK_NODEPTR;
```

```
/* stack module data structure */
typedef struct

        {
        STK_NODEPTR stkptr; /* stack pointer */
        }STKLIST,*STKLIST_PTR;

/* stack module function prototypes */

/* initialize stack */
void initStk(STKLIST_PTR stk);

/* put data item onto stack */
bool push(STKLIST_PTR stk,STK_DATA data);

/* get data item from stack */
STK_DATA pop(STKLIST_PTR stk);

/* print stack contents */
void printStk(STKLIST_PTR stk);
void print(STK_NODEPTR node);

/* deallocate memory for stack */
void destroyStk(STKLIST_PTR stk);
void destroy(STK_NODEPTR node);

#endif
```

Stack module implementation code file:

```
/* stklist.c */

#include <stdio.h>
#include <stdlib.h>
#include "stklist.h"
#include "defs.h"

/* stack module functions */

/* initialize stack module */
void initStk(STKLIST_PTR stk)

        {
        stk->stkptr = NULL; /* set stackptr to top of stack */
        }

/* push item into stack */
bool push(STKLIST_PTR stk,STK_DATA data)

        {
        /* insert into vector at stpkptr++ location */
        STK_NODEPTR node = (STK_NODEPTR)malloc(sizeof(STK_NODE));



        if(node != NULL)
```

```
                {
                node->data = data;
                node->next = stk->stkptr;
                stk->stkptr = node;
                }

        return node != NULL;
        }

/* get item from stack */
STK_DATA pop(STKLIST_PTR stk)

        {
        STK_DATA data;
        STK_NODEPTR node;
        if(stk->stkptr == NULL)return ERROR; /* check if stack empty */
        data = (stk->stkptr)->data;
        /* remove from item at --stpkptr location */
        node = (stk->stkptr)->next;
        free(stk->stkptr);
        stk->stkptr = node;
        return data;
        }

/* print stack items */
void printStk(STKLIST_PTR stk)

        {
        printf("[ ");
        print(stk->stkptr);
        printf("]\n");
        }

/* print stack items */
void print(STK_NODEPTR node)

        {
        /* print out stack items */
        if(node != NULL)

                {
                printf("%d ",node->data);
                print(node->next);
                }

        }
```

```
/* deallocate memory for stack */
void destroyStk(STKLIST_PTR stk)

        {
        destroy(stk->stkptr);
        stk->stkptr=NULL;
        }

/* deallocate memory for stack */
void destroy(STK_NODEPTR node)

        {
        STK_NODEPTR next;
        if(node != NULL)

                {
                next = node->next; /* deallocate memory for stack item */
                free (node);
                destroy(next);
                }

        }

/* main function to test stack module */
int main()

        {
        STK_DATA data; /* create stack data structure in memory */
        STKLIST stk; /* create a stack data structure */
        initStk(&stk); /* initialize stack module */
        printStk(&stk); /* print stack contents */
        push(&stk,8); /* push "8" into stack */
        printStk(&stk); /* print stack contents */
        data = pop(&stk); /* get item from stack */
        printf("data from stack: %d\n",data);
        data = pop(&stk); /* get item from stack */
        printf("data from stack: %d\n",data);
        push(&stk,3); /* push "3" into stack */
        push(&stk,7); /* push "7" into stack */
        push(&stk,4); /* push "4" into stack */
        printStk(&stk); /* print stack contents */
        data = pop(&stk); /* get item from stack */
        printf("data from stack: %d\n",data);
        printStk(&stk); /* print stack contents */
        data = pop(&stk); /* get item from stack */
        printf("data from stack: %d\n",data);
        printStk(&stk); /* print stack contents */
        data = pop(&stk); /* get item from stack */
        printf("data from stack: %d\n",data);
        printStk(&stk); /* print stack contents */
        destroyStk(&stk); /* deallocate memory for stack */
        return TRUE;
        }
```

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

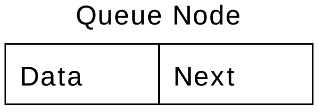
Add an **isEmpty()** method to the stack module that returns true if the stack is empty.

LESSON 7 EXERCISE 2

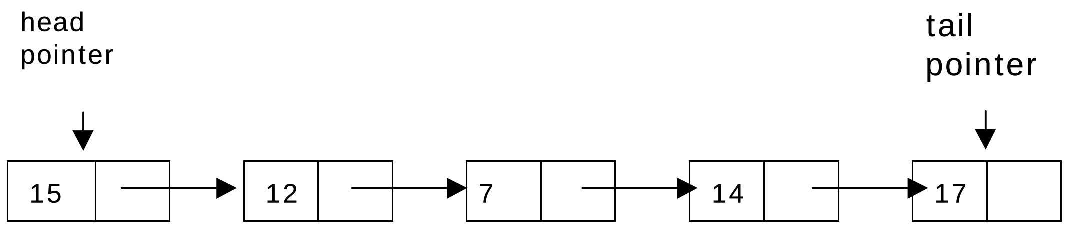
Implement the stack list ADT with a double link list. When printing out the double link list print the stack in reverse order from the top to the bottom. Call your program stkdlist.c and your header file stkdlist.h.

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.

quelist.h	header file	The header file contains the function prototypes.
QUEUE_NODE	data structure	The STK_NODE data structure holds the data and links to next and previous nodes.
QUEUE_LIST	data structure	The STKLIST data structure holds the start of list and number of nodes in list.
stklist.c	code	The code contains the function definitions.

Here's the code for the queList 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
typedef int bool;
#define TRUE  1
#define FALSE 0
#define ERROR -1
#endif
```

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

```
/* quelist.h */
#ifndef __QUELIST
#define __QUELIST

#include "defs.h"

typedef int QUEUE_DATA;

/* queue list node */
typedef struct queue_node_type

        {
        QUEUE_DATA data;
        struct queue_node_type *next;
        }QUEUE_NODE,*QUEUE_NODEPTR;
```

```
/* queue list module data structure */
typedef struct

        {
        QUEUE_NODEPTR head;
        QUEUE_NODEPTR tail;
        }QUEUE_LIST,*QUEUE_LISTPTR;

/* queue module function prototypes */
/* initialize queue */
void initQue(QUEUE_LISTPTR que);

/* put item into queue */
bool enqueue(QUEUE_LISTPTR que,QUEUE_DATA data);

/* remove data item from queue */
QUEUE_DATA dequeue(QUEUE_LISTPTR que);

/* deallocate memory for queue */
void destroyQue(QUEUE_LISTPTR que);
void destroy(QUEUE_NODEPTR node);

/* print out contents of queue */
void printQue(QUEUE_LISTPTR que,char* name);
void print(QUEUE_NODEPTR node);
#endif
```

Queue List Implementation code file:

```
/* quelist.c */
#include <stdio.h>
#include <stdlib.h>
#include "quelist.h"

/* queue module functions */

/* initialize queue */
void initQue(QUEUE_LISTPTR que)

        {
        que->head = NULL;
        que->tail = NULL;
        }

/* insert items into the queue list */
bool enqueue(QUEUE_LISTPTR que,QUEUE_DATA data)

        {
        /* allocate memory for queue node */
        QUEUE_NODEPTR node = (QUEUE_NODEPTR)malloc(sizeof(QUEUE_NODE));
        /* check if memory allocated */
        if(node != NULL)

                {
                node->data = data;
                node->next = NULL;
                if(que->tail != NULL)(que->tail)->next = node;
                else que->head = node;
                que->tail = node;
                }

        return node != NULL;
        }

/* remove items from the queue */
QUEUE_DATA dequeue(QUEUE_LISTPTR que)

        {
        QUEUE_DATA data;
        QUEUE_NODEPTR node;

        /* check if queue empty */
        if(que->head == NULL)return ERROR;
        data = (que->head)->data;

        /* remove from item at head of queue */
        node = (que->head)->next;
        free(que->head);
        que->head = node;

        if(node == NULL)que->tail = NULL;  /* queue is empty */

        return data;
        }

/* deallocate memory for queue */
void destroyQue(QUEUE_LISTPTR que)

        {
        /* deallocate memory for queue list */
        destroy(que->head);
        que->head = NULL;
        que->tail = NULL;
        }
```

```
/* deallocate memory */
void destroy(QUEUE_NODEPTR node)

        {
        if(node != NULL)

                {
                /* deallocate memory for queue item */
                QUEUE_NODEPTR next = node->next;
                free (node);
                destroy(next);
                }

        }

/* print out queue items */
void printQue(QUEUE_LISTPTR que,char* name)

        {
        printf("%s[ ",name);
        print(que->head);
        printf(" ]\n");
        }

/* print out queue items */
void print(QUEUE_NODEPTR node)

        {

        if(node != NULL)

                {
                printf("%d ",node->data);
                print(node->next);
                }

        }

/* main function to test queue module */
int main()

        {
        QUEUE_LIST  que; /* create queue structure */
        /* initialize queue list */
        initQue(&que);
        enqueue(&que,5); /* put 5 into queue */
        printQue(&que,"queue: "); /* print out queue */

        /* get data item from queue and print out value */
        printf("\n data value: %d\n",dequeue(&que));
        printQue(&que,"queue: "); /* print out queue */

        enqueue(&que,1); /* put 1 into queue */
        printQue(&que,"queue: "); /* print out queue */
        enqueue(&que,2); /* put 2 into queue */
        printQue(&que,"queue: "); /* print out queue */
        enqueue(&que,3); /* put 3 into queue */
        printQue(&que,"queue: "); /* print out queue */

        /* get data item from queue and print out value */
        printf("\n data value: %d\n",dequeue(&que));
        printQue(&que,"queue: "); /* print out queue */

        /* get data item from queue and print out value */
        printf("\n data value: %d\n",dequeue(&que));
        printQue(&que,"queue: "); /* print out queue */

        /* get data item from queue and print out value */
        printf("\n data value: %d\n",dequeue(&que));
        printQue(&que,"queue: "); /* print out queue */

        /* get data item from queue and print out value */
        printf("\n data value: %d\n",dequeue(&que));

        printQue(&que,"queue: "); /* print out queue */
        destroyQue(&que); /* deallocate memory for queue */

        return TRUE;
        }
```

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 3

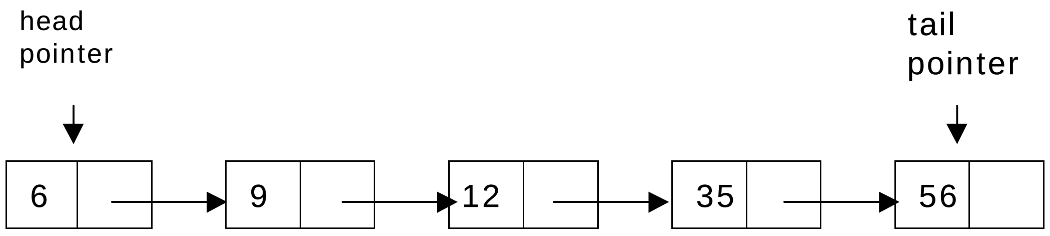
Add an **isEmpty()** method to the queue list module that returns true if the queue is empty.

LESSON 7 EXERCISE 4

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.

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() unction and driver insert() function using recursion

```
/* insert items into the queue list by priority */
bool enqueue(QUEUE_LISTPTR que,QUEUE_DATA data)
{
QUEUE_NODEPTR prev,curr; /* pointers to nodes */

/* allocate memory for queue node */
QUEUE_NODEPTR node = (QUEUE_NODEPTR)malloc(sizeof(QUEUE_NODE));

if (node == NULL)return FALSE;

node->data = data; /* assign data to node STRUCTURE */
node->next= NULL; /* assume next node is always NULL */

/* check for empty list */
if(que->head == NULL)

        {
        que->head = node; /* set head to point to new node */
        que->tail = node; /* set tail to point to new node */
        return TRUE;
        }
/* find out where to insert new node */
/* insert in ascending order */
curr = que->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 = que->head;
        que->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)que->tail = node; /* assign tail to end */
        }

return TRUE;
} /* end enqueue */
```

LESSON 7 EXERCISE 5

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

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:	Nov 25,2001
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.

dlist.h	header file	The header file contains the function prototypes.
DNode	data structure	The DNode data structure holds the data and links to next and previous nodes.
DList	data structure	The List data structure holds the start and end of list pointers and number of nodes in list.
dlist.c	code	The code contains the function definitions.

Here's the code for the DList 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 */

        #define FALSE 0
        #define TRUE 1
        typedef  int  bool;
```

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

```
/* dlist.h */
/* double link list */

#include "defs.h"

/* double link list node data */
typedef int DNodeData;

/* double node structure */
typedef struct DNodeType

        {
        struct DNodeType  *prev;
        DNodeData data;
        struct DNodeType *next;
        }DNode,*DNodePtr;

/* double link list pointers */
```

```
typedef struct

        {
        DNodePtr head;
        DNodePtr tail;
        int length;
        }DList,*DListPtr;

/* prototypes */

/* initialize double link list pointers */
void initDList(DList* dlist);

/* free double link list nodes */
void destroyDList(DListPtr dlist);

/* insert item into double link list */
int insertDLTail(DListPtr dlist,DNodeData data);

/* remove node at start of list */
int removeDLHead(DListPtr dlist);

/* remove node at end of list */
int removeDLTail(DListPtr dlist);

/* test if list is empty */
bool isEmpty(DListPtr dlist);

/* print out double link list items from start of list */
void printStart(DListPtr list);

/* print out double link list items from end of list */
void printEnd(DListPtr list);
```

Implementation code file for Double Link List:

```
/* dlist.c */
/*double link list */

#include <stdio.h>
#include <stdlib.h>
#include "dlist.h"

/* initialize dlist pointers */
void initDList(DList* dlist)

        {
        dlist->head = NULL; /* set head to empty value */
        dlist->tail = NULL; /* set tail to empty value */
        length=0; /* set to no nodes */
        }
```

```
/* free nodes in dlist */
void destroyDList(DListPtr dlist)

        {
        DNodePtr next; /* pointer to a next dnode */
        DNodePtr curr = dlist->head; /* point to start of list */

        /* loop till end of list */
        while(curr != NULL)

                {
                next = curr->next; /* point to next node */
                free(curr); /* free current dnode */
                curr = next; /* set curr to next dnode */
                }

        dlist->head = NULL; /* set head to empty value */
        dlist->tail = NULL; /* set tail to empty value */
        }

/* insert item into dlist at end of list */
 int insertDLTail(DListPtr dlist,DNodeData data)

        {
        /* allocate memory for new dnode */
        DNodePtr dnode = (DNodePtr)malloc(sizeof(DNode));
        if (dnode == NULL)return FALSE;

        /* assign data to dnode structure */
        dnode->data = data;
        dnode->next = NULL;
        dnode->prev = NULL;
        (dlist->length)++;    /* increment node count */

        /*check for empty list */
        if(dlist->head == NULL)

                {
                dlist->head = dnode; /* head points to new node */
                dlist->tail = dnode; /* tail points to new node */
                return TRUE;
                }

                /* attach to end of list */
                (dlist->tail)->next = dnode;
                dnode->prev=dlist->tail;

                /* update tail */
                dlist->tail=dnode;
                return TRUE;
                }

         return TRUE;
        }
```

```
/* remove node at start of list */
bool removeDLHead(DListPtr dlist)

        {
        DNodePtr curr = dlist->head; /* pointer to current dnode */
        DNodePtr next = dlist->head->next; /* pointer to a next dnode */
        DNodeData data = curr->data;
        (dlist->length)--; /* decrement node count */

        /* check for last one item in list */
        if(dlist->head==dlist->tail)

                {
                dlist->head=NULL; /* set head to empty value */
                dlist->tail=NULL; /* set tail to empty value */
                }

        /* remove dnode from start of list */
        else

                {
                dlist->head = next; /* head points to next dnode */
                next->prev = NULL; /* current points to empty node */
                }

        free(curr); /* deallocate memory for current node */
        return data;
        }

/* remove node at start of list */
int removeDLTail(DListPtr dlist)

        {
        DNodePtr curr = dlist->tail; /* pointer to current dnode */
        DNodePtr prev = dlist->tail->prev; /* pointer to a previous dnode */
        DNodeData data = curr->data; /* current is tail dnode */
        (dlist->length)--; /* decrement count */

        /* check for last one item in list */
        if(dlist->head==dlist->tail)

                {
                dlist->head=NULL; /* set head to empty value */
                dlist->tail=NULL; /* set tail to empty value */
                }

        /* remove dnode from end of list */
        else

        {
        dlist->tail->prev = NULL; /* tail points to no node */
        dlist->tail->next = NULL; /* tail points to no node */
        dlist->tail=prev; /* tail now points to previous dnode */
        }
        free(curr); /* deallocate memory for current node */
        return data;
        }

/* test if list is empty */
bool isEmpty(DListPtr dlist)

        {
        return dlist->length == 0;
        }

/* print out items at start of dlist */
void printStart(DListPtr dlist)

        {
        DNodePtr curr=dlist->head; /* point to start of dlist */

        /* check if dlist empty */
        if(curr == NULL)

                {
                printf("double list is empty\n");
                return;
                }

        printf("dlist from start: ");

        /* loop till end of list */
        while(curr != NULL)

                {
                printf("%d ",curr->data); /* print out data item */
                curr = curr->next; /* point to next item */
                }

        putchar('\n');
        }

/* print out items at end of dlist */
void printEnd(DListPtr dlist)

        {
        DNodePtr curr=dlist->tail; /* point to end of dlist */

        /* check if dlist empty */
        if(curr == NULL)

                {
                printf("double list is empty\n");
                return;
                }

        printf("dlist from end: ");
```

```
/* loop till start of list */
while(curr != NULL)

        {
        printf("%d ",curr->data); /* print out data item */
        curr = curr->prev; /* point to previous dnode */
        }

putchar('\n');
}
```

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.c */
#include <stdio.h>
#include "dlist.h"

void bdlsort(DList* dl);

/* driver to test bubble sort of double link list */
void main()

        {
        DList dlist;

        initDList(&dlist);
        insertDLTail(&dlist,2);
        insertDLTail(&dlist,6);
        insertDLTail(&dlist,5);
        insertDLTail(&dlist,4);
        insertDLTail(&dlist,3);
        insertDLTail(&dlist,1);
        printStart(&dlist);
        bdlsort(&dlist);
        printStart(&dlist);
        }
```

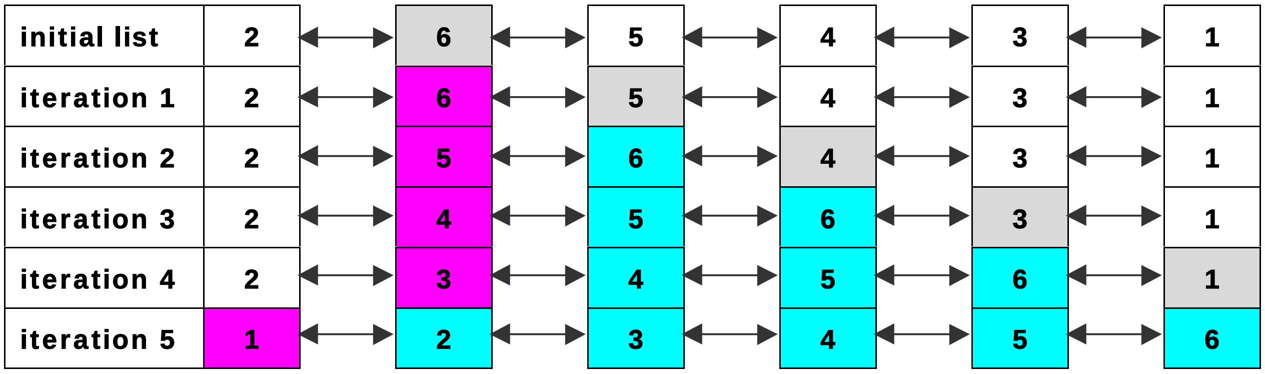
program output:

```
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;
                printStart(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:

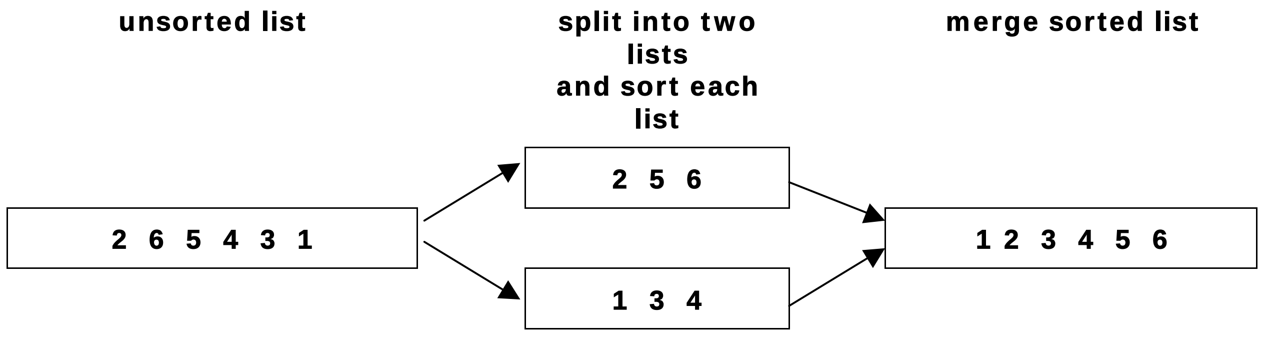
```
dlist from start: 2 6 5 4 3 1

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 <stdio.h>
#include "dlist.h"

void mdlsort(DList* dl);
void merge(DList* dl1, DList* dl2, DList* dl);

/* driver to test  merge  sort */
void main()

        {
        DList dlist;

        initDList(&dlist);
        insertDLTail(&dlist,2);
        insertDLTail(&dlist,6);
        insertDLTail(&dlist,5);
        insertDLTail(&dlist,4);
        insertDLTail(&dlist,3);
        insertDLTail(&dlist,1);
        printStart(&dlist);
        mdlsort(&dlist);
        printStart(&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;

        initDList(&dl1);    /* make list dl1 */
        initDList(&dl2);    /*make list dl2 */

        /* copy for  two lists */
        for (i=1; i <= (n+1)/2; i++)

                {
                d=removeDLHead(dl);
                insertDLTail(&dl1,d);
                }

        for (i=1; i <= n/2; i++)

                {
                d=removeDLHead(dl);
                insertDLTail(&dl2,d);
                }
        printStart(&dl1);  /* optional print list dl1 */
        printStart(&dl2);  /* optional print list dl2 */

        mdlsort(&dl1); /* sort both lists by recursion */
        mdlsort(&dl2);

        printStart(&dl1);  /* optional print list dl1 */
        printStart(&dl2);  /* optional print list dl2 */

        /* merge two lists */
        merge(&dl1,&dl2,dl);
        printStart(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(!isEmpty(dl1) && !isEmpty(dl2))

                {
                d1 = ((DNode*)(dl1->head))->data;
                d2 = ((DNode*)(dl2->head))->data;

                /* check if first element of dl1 is less than dl2 */
                if(d1 < d2)

                        {
                        d = removeDLHead(dl1);
                        insertDLTail(dl,d);
                        }

                else

                        {
                        d = removeDLHead(dl2);
                        insertDLTail(dl,d);
                        }

                }

        /* empty list dl1 into dl */
        if(isEmpty(dl1))

                {
                while(!isEmpty(dl2))

                        {
                        d = removeDLHead(dl2);
                        insertDLTail(dl,d);
                        }

                }
                /* empty list dl2 into dl */
                if(isEmpty(dl2))

                {
                while(!isEmpty(dl1))

                        {
                        d = removeDLHead(dl1);
                        insertDLTail(dl,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

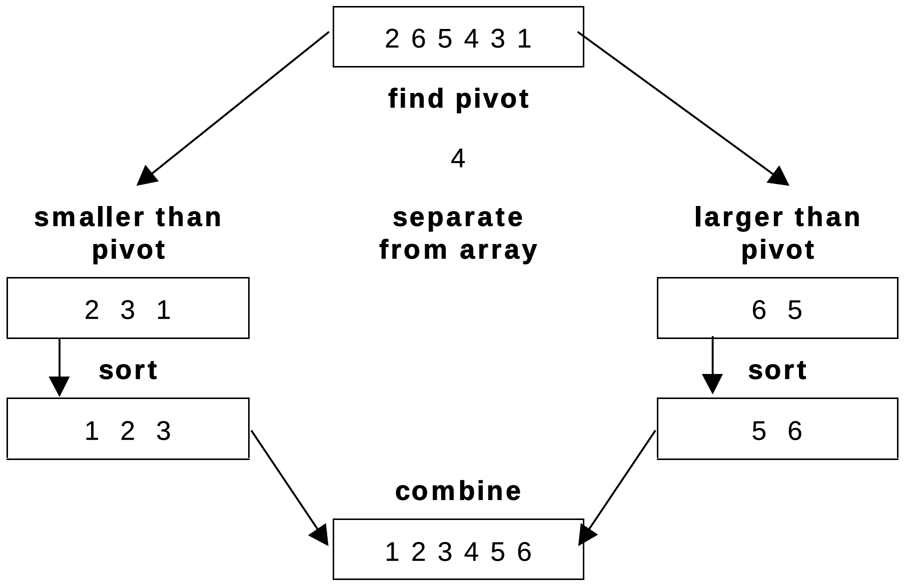
18	ra 0,5 3,5	msort	3	4	5	2 5 6 3 4 1	2 5 6 3 4 0
19	ra 0,5 3,5	msort	5	--	5	2 5 6 3 4 1	2 5 6 3 4 0
20	ra 0,5	**merge**	3	4	5	2 5 6 3 4 1	2 5 6 3 4 0
21	ra	**merge**	0	2	5	2 5 6 1 3 4	2 5 6 1 3 4
22	ra	msort	0	2	5	1 2 3 4 5 6	1 2 3 4 5 6

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.c */
#include <stdio.h>
#include "dlist.h"

/* prototypes */
void qdlsort(DList* dl);

/* test driver for quick sort */
void main()

        {
        DList dlist;
        initDList(&dlist);
        insertDLTail(&dlist,2);
        insertDLTail(&dlist,6);
        insertDLTail(&dlist,5);
        insertDLTail(&dlist,4);
        insertDLTail(&dlist,3);
        insertDLTail(&dlist,1);
        printStart(&dlist);
        qdlsort(&dlist);
        printStart(&dlist);
        }

/* quick sort using double link list */
void qdlsort(DList* dl)

        {
        int pivot;
        int d;
        DList dll;
        DList dlr;

        if (dl->length < 2)return;

        initDList(&dll);
        initDList(&dlr);
        printStart(dl);/* optional print list */
        pivot = removeDLTail(dl);
        printf("pivot: %d\n",pivot);

        /* fill in the left and right elements */
        while ( !isEmpty(dl) )

                {
                /* if next element in the list is less than pivot */
                /* insert into left otherwise insert into right list. */
                d = removeDLHead(dl);
                if ( d < pivot)insertDLTail(&dll,d);
                else insertDLTail(&dlr,d);
                }



                qdlsort(&dll); /* sort the left partition */
                qdlsort(&dlr); /* sort the right partition */
                printStart(&dll); /* optional print list */
                printStart(&dlr); /* optional print list */

        /* insert the left elements, smaller than right  */
        while ( !isEmpty(&dll) )

                {
                d = removeDLHead(&dll);
                insertDLTail(dl,d);
                }

        /* insert pivot , bigger then left smaller than right */
        insertDLTail(dl,pivot);

        /* insert the right end, the largest elements */
        while (!isEmpty(&dlr) )

                {
                d = removeDLHead(&dlr);
                insertDLTail(dl,d);
                }

        printStart(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:	CdsGuideL9.doc
Date Started:	July 24,1998
Last Update:	Nov 29, 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.c */
#include<stdio.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 = 8;
        int a[] = {0,1,2,3,4,5,6,7,8,9};
        printf("find+ing: %d\n",k);
        i = binsrch(a,k,10);
        printf("found: %d\n",a[i]);
        }
```

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(alow,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;
    printf("[ ");
    for(i=s;i<n;i++)
    printf("%d ",a[i]);
    printf(" ]\n");
    }
```

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.c */
#include <stdio.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, k = 7;
int a[] = {0,1,2,3,4,5,6,7,8,9};
printf("finding: %d\n",k);
i = binsrchr(a,k,0,9);
printf("found: %d\n",a[i]);
}
```

```
/* 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;
        printf("[ ");
        for(i=s;i<n;i++)
        printf("%d ",a[i]);
        printf("]\n");
        }
```

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
```

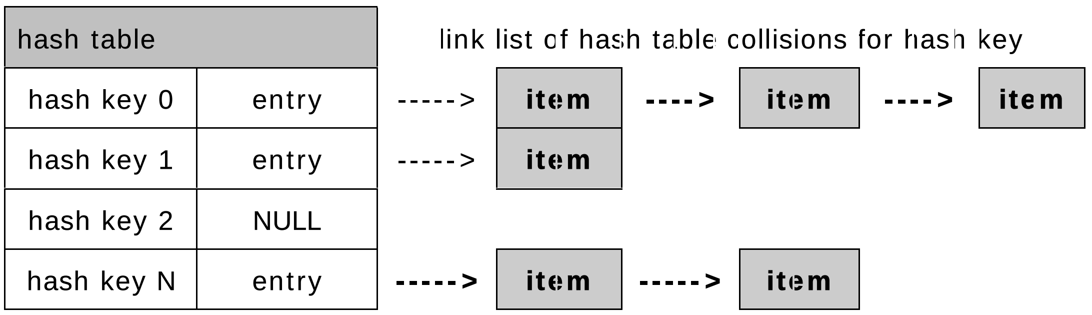
LESSON 9 EXERCISE 1

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 MODULE

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 the modular approach to implement our Hash Table ADT where the Hash structure will store the size of the hash table and the array of link list pointers.

```
/* hash ADT module data structure */
typedef struct
    {
    ListPtr* h; /* pointer to vector module */
    int size; /* size of hash table */
    }Hash,*HashPtr;
```

hash.h	header file	The header file contains the function prototypes.
Hash	data structure	The Hash data structure holds the array of link list pointers and size of the hash table
hash.c	code	The code 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*.

list.h	header file	The header file contains the function prototypes.
Node	data structure	The Node data structure holds the data and link to next node.
List	data structure	The List data structure holds the start and end of list pointers and number of nodes in list.
list.c	code	The code 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
typedef  int  bool;

#endif
```

Here's the Hash Table header file code:

```
/* hash.h */

#include "defs.h"
#include "list.h"
```

```
/* hash table module header file */
/* data structure definitions */

typedef char* HashData; /* hash table data type */

/* hash ADT module data structure */
typedef struct

        {
        ListPtr* h; /* pointer to vector module */
        int size; /* size of hash table */
        }Hash,*HashPtr;

/* initialize hash table */
void initHash(HashPtr hash,int size);

/* calculate hash key from data */
int hashKey(HashPtr hash,HashData data);

/* insert item into hash table */
bool insertHash(HashPtr hash,HashData data);

/* free all memory belonging to hash table */
void destroyHash(HashPtr hash);

/* remove item from hash table */
bool removeHash(HashPtr hash,HashData data);

/* search hash table for entry */
NodePtr  searchHash(HashPtr hash,HashData);

/* print out hash table entries */
void printHash(HashPtr hash);
```

Here's the code for the hash table module implementation file:

```
/* hash.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hash.h"
#include "list.h"

/* initialize hash table */
void initHash(HashPtr hash,int size)

        {
        int i; /* for loop counter */
        ListPtr list; /* pointer to list */
        /* make an array of list pointers */
        hash->h = (ListPtr*)malloc(sizeof(ListPtr)size);

        /* make a new list for each vector entry */
        for(i=0;i<size;i++)

                {
                list=(ListPtr)malloc(sizeof(List)); /* allocate for list pointers */
                hash->h[i]=list; /* assign list to header */
                initList(list); /* initialize list  */
                }

        hash->size=size; /* store hash table size */
        }

/* free all memory belonging to hash table */
void destroyHash(HashPtr hash)

        {
        int i;
        /* loop through hash table deallocating list */
        for(i=0;i<hash-size;i++) destroyList(hash->h[i]);
        }

/* calculate hash key from data */
int hashKey(HashPtr hash,HashData data)

        {
        int i;
        int sum = 0;

        /* add up all characters in data */
        for(i=0;i<strlen(data);i++) sum = sum + data[i];
        return (sum % hash->size); /* calculate and return key */
        }

/* insert item into hash table */
bool insertHash(HashPtr hash,HashData data)

        {
        int key = hashKey(hash,data); /* calculate hash key */
        ListPtr list = hash->h[key]; /* get list */
        return insertList(dlist,data); /* insert item into list */
        }

/* remove item from hash table */
bool removeHash(HashPtr hash,HashData data)

        {
        int key = hashKey(hash,data); /* calculate hash key */
        ListPtr list = hash->h[key]; /* get list */
        return removeList(list,data); /* remove item from list */
        }
```

```
/* search hash table for entry */
NodePtr searchHash(HashPtr hash,HashData data)

        {
        NodePtr node; /* node of item in list */
        int key = hashKey(hash,data); /* calculate hash key */
        ListPtr list = hash->h[key]; /* get list */
        node = findList(list,data); /* find node in list */
        return node;
        }

/* print out hash table entries */
void printHash(HashPtr hash)

        {
        int i;
        printf("\nhash table:\n");
        /* loop through hash table */
        for(i=0;i<hash->size;i++)

                {
                ListPtr list = hash->h[i]; /* get list */
                printf("key %2d: ",i);
                printList(list); /* print put list */
                }

        }

/* main test program */
int main()
{
Hash hash; /* create hash module data structure */
initHash(&hash,5); /* make hash table of size 5 */
insertHash(&hash,"today is cloudy"); /* insert item */
printHash(&hash); /* print hash table */
removeHash(&hash,"how are you?"); /* remove item */
printHash(&hash); /* print hash table */
insertHash(&hash,"it will rain today"); /* insert item */
printHash(&hash); /* print hash table */
insertHash(&hash,"do you have an umbrella?"); /* insert item */
printHash(&hash); /* print hash table */
printf("searching for: today is cloudy"); /* search for an item */
node = searchHash(&hash,"today is cloudy");
if(node==NULL) printf(" : cannot find\n");
else printf(" : found \n");
removeHash(&hash,"today is cloudy"); /* remove item */
printHash(&hash); /* print hash table */
removeHash(&hash,"today is windy"); /* remove item */
printHash(&hash); /* print hash table */
insertHash(&hash,"is it warm today ?"); /* insert item */
printHash(&hash); /* print hash table */
return 0;
}
```

hash module 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

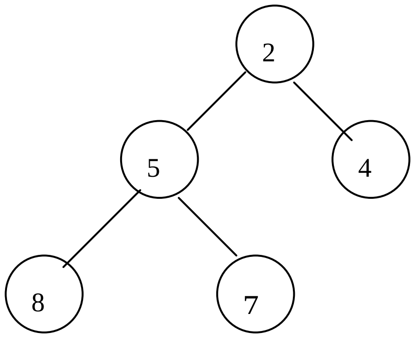
A binary heap is a binary tree stored in an array. A heap has two properties:

(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:

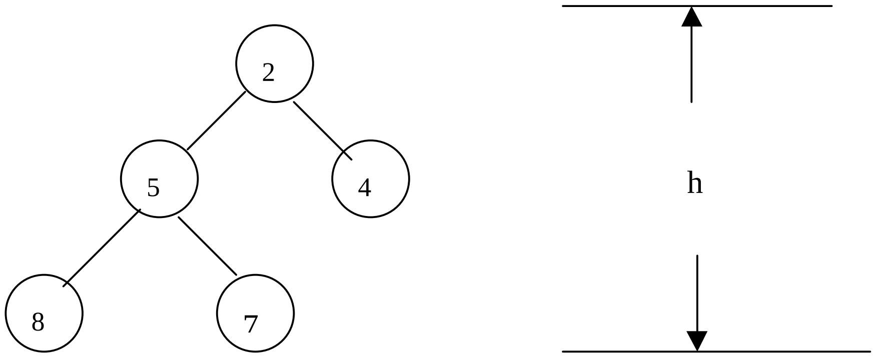
left child	2i
right child	2i + 1
parent of i	i/2

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



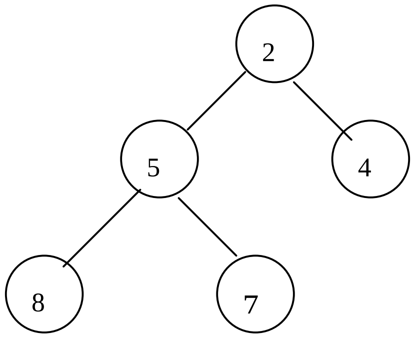
heap order property

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



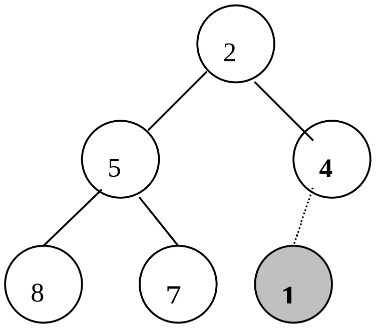
Since a heap is ordered they are used to build priority queues.

HEAP OPERATIONS

inserting an item

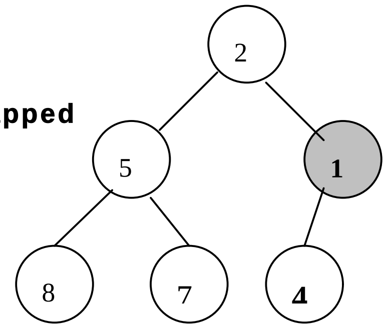
When we insert an item the heap order must be preserved. This is easy to do !. We always insert a new item at the end of the array. Since we have started our heap at **position** 1 the parent of the inserted item will always be **i / 2.** To preserve the heap property we must make sure the parent is 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



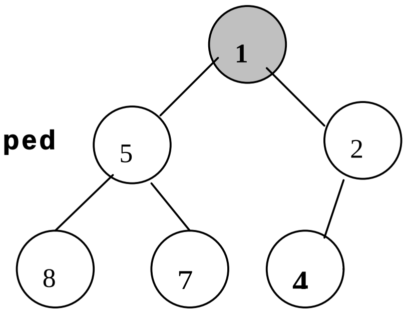
The parent of new item 1 is item 4. Item 4 is larger than the new item 1 so we must swap 1 and 4,

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.

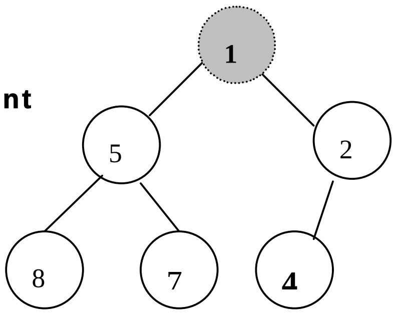
Here's the code to insert an item into a heap.

```
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;
   }
```

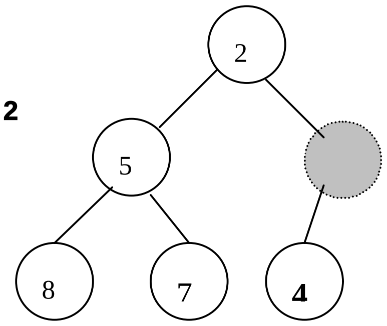
removing item from heap

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.

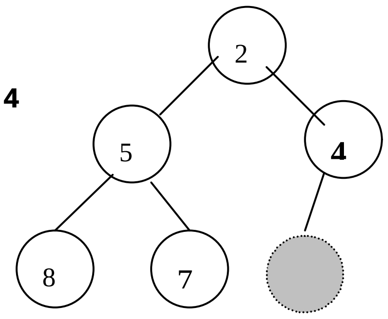
remove first element



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.c
#include <stdio.h>

// prototypes
void insert(int n, int items[], int* count);
int remove(int items[], int* count);
void print(int items[], int count) ;

// insert an item
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;
}

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

        if(*count == 0)
                {
                printf("empty\n");
                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
void print(int items[], int count)
{
        int i = 1;
        int k = 1;
        int j = 0;

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

                for(j=i;(j<i+k) && (j<=count);j++)
                printf("%3d ",items[j]);

                printf("\n");
             i = i + k;
                k = k << 1;
        }
    printf("\n");
    }

// test driver
void main()
{
int count=0;
int items[100];
int i;

for(i=0;i<100;i++)
items[i]=0;
insert(2,items,&count);
insert(5, items,&count);
insert(4, items,&count);
insert(8, items,&count);
insert(7, items,&count);
print(items,count);
insert(1, items,&count);
print(items,count);
remove(items,&count);
print(items,count);
remove(items,&count);
print(items,count);
remove(items,&count);
print(items,count);
remove(items,&count);
print(items,count);
}
```

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.

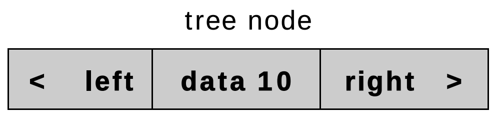
C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 10

File:	CdsGuideL10.doc
Date Started:	July 24,1998
Last Update:	Dec 20, 2001
Status:	proof

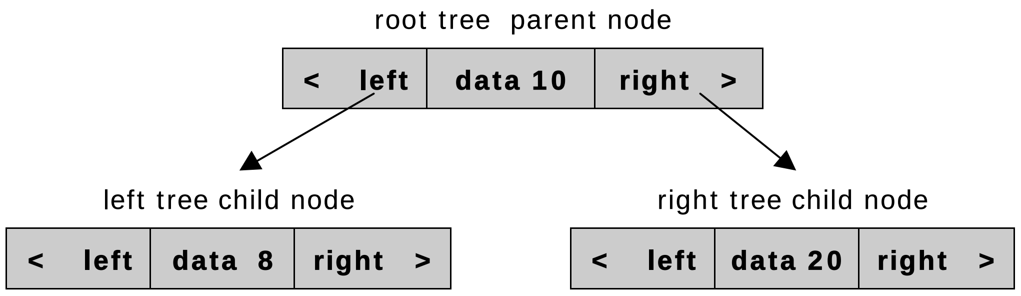
LESSON 10 BINARY SEARCH TREES

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.

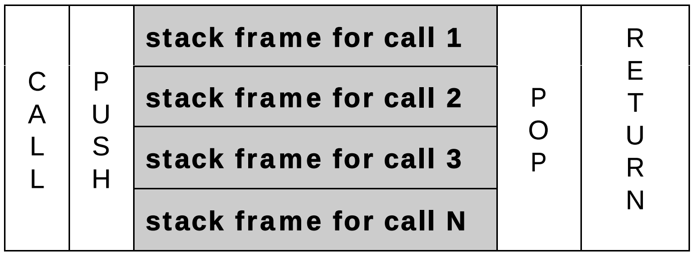


The node that the link points to is known as a **child**. The node that is pointing is known as the **parent**. Nodes that do not have links to other nodes are known as **leaves**. The first node in a tree is known as the **root**. With binary search trees are you can search much faster for items than 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.

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

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:

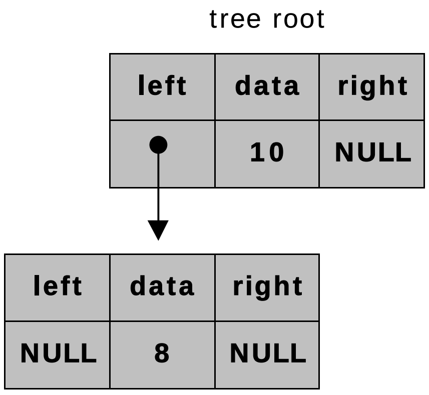
tree	NULL

insert 10 into tree:

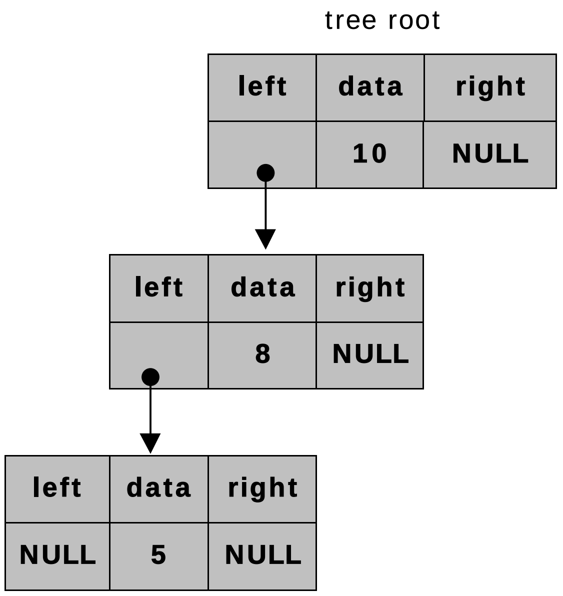
tree root

left	data	right
NULL	10	NULL

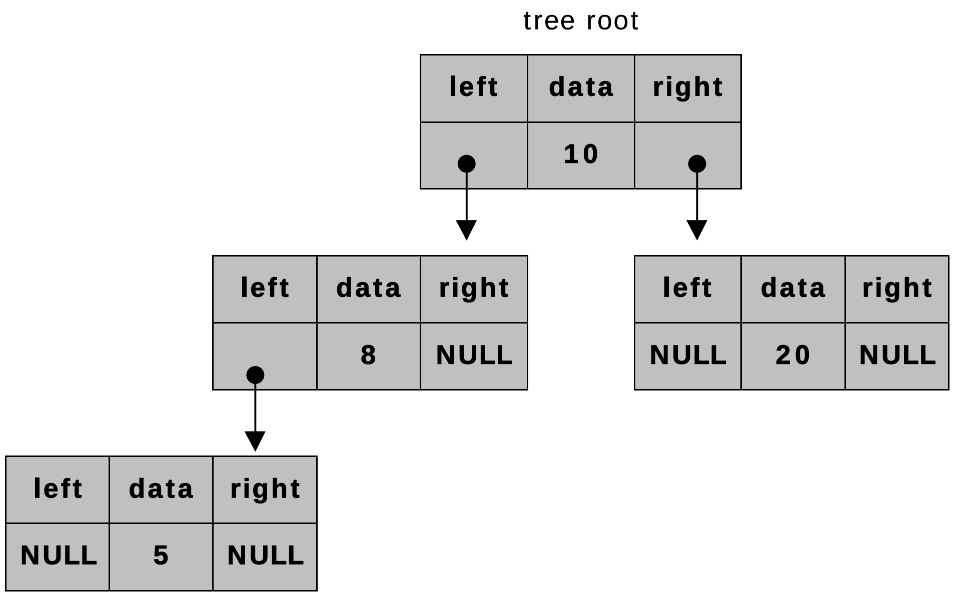
insert 8 into tree



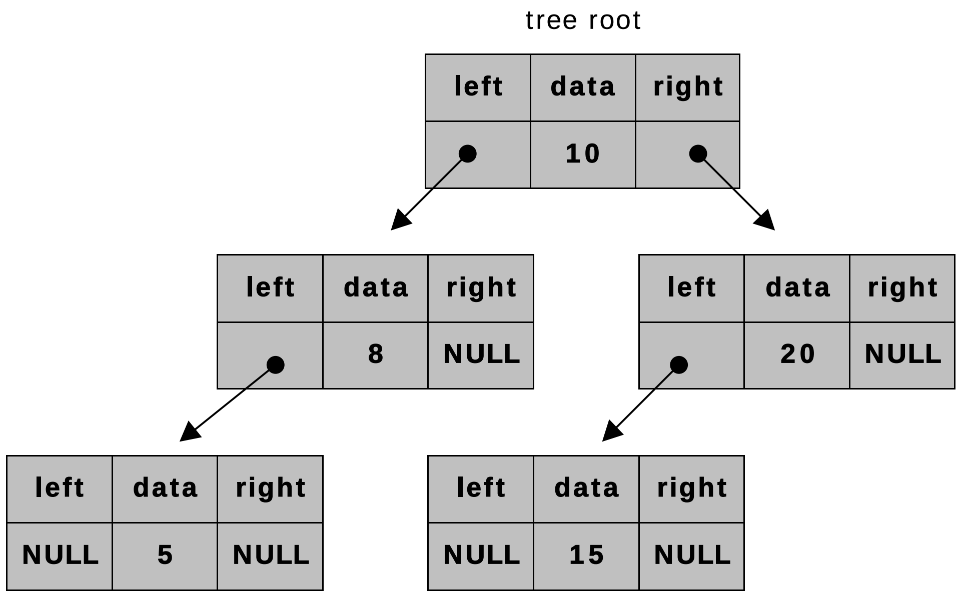
insert 5 into tree



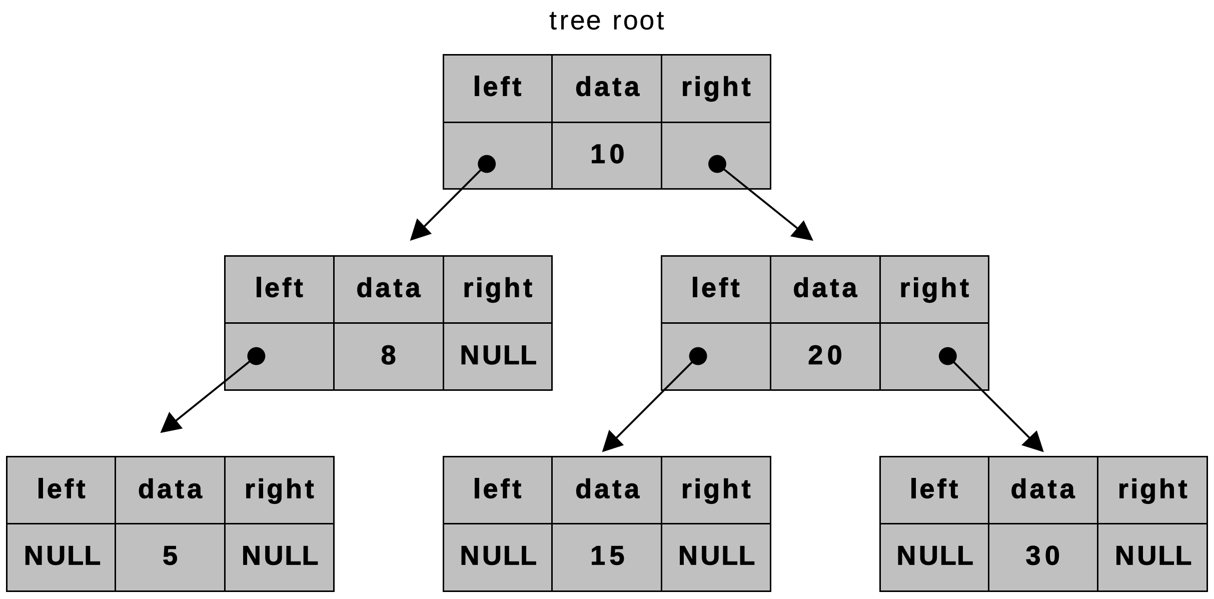
insert 20 into tree



insert 15 into tree



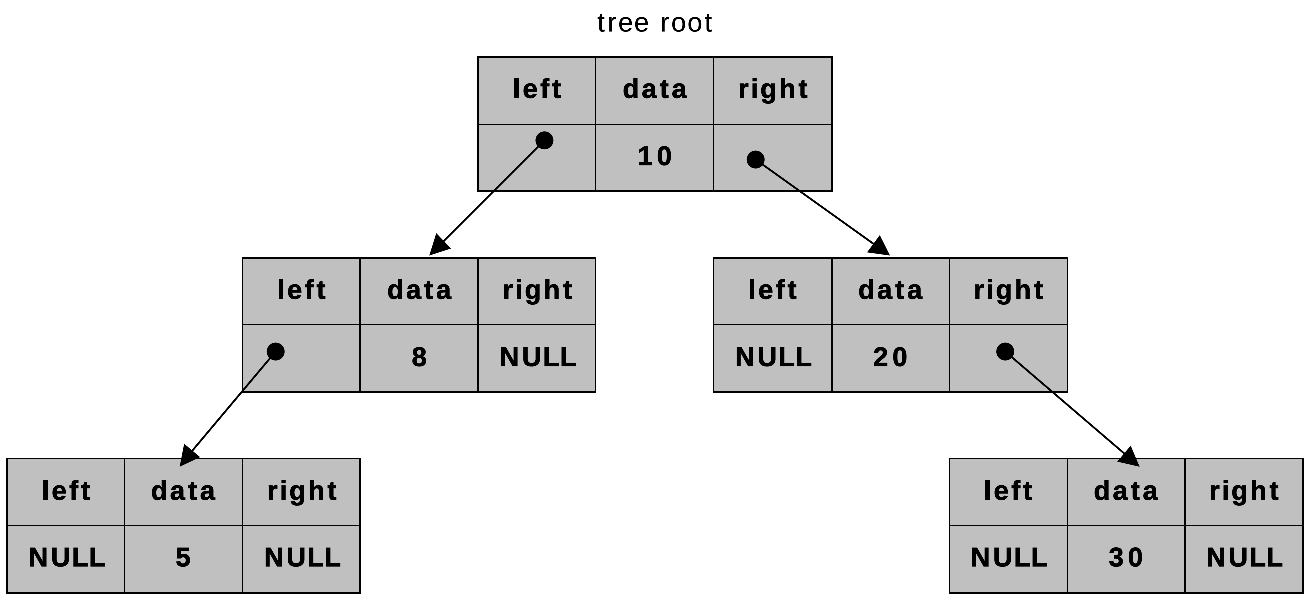
insert 30 into tree



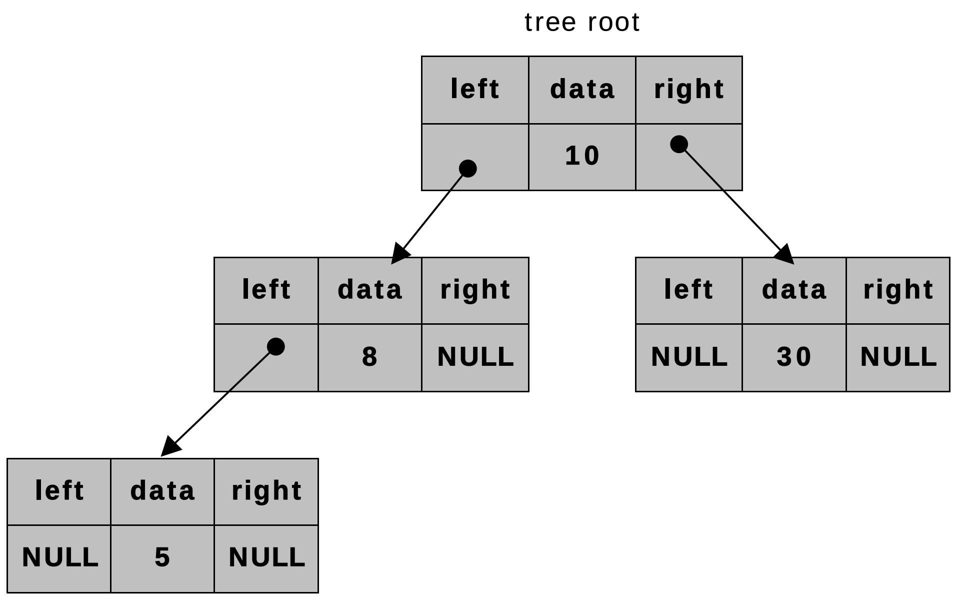
Removing items from binary tree example:

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.

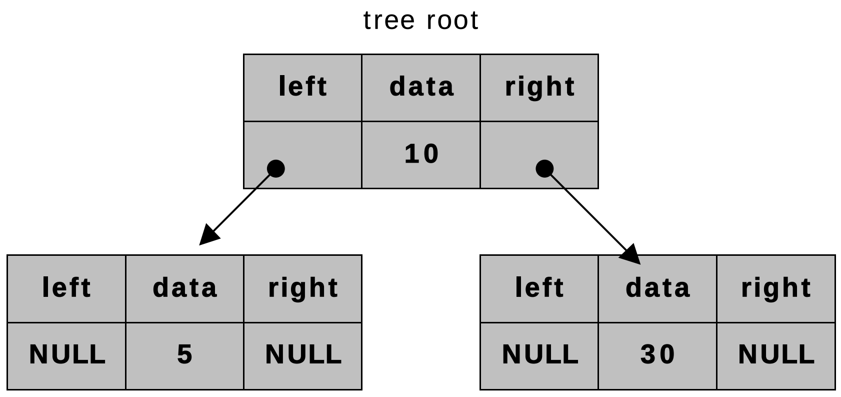
remove 15



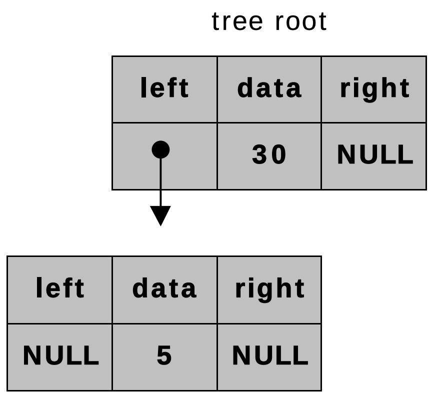
remove 20



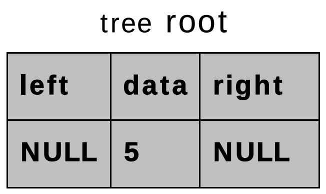
remove 8



remove 10



remove 30



remove 5

The tree is empty

tree	NULL

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. In you main function you will need a pointer to a binary tree node to represent the root of the tree.

tree.h	header file	The header file contains the function prototypes.
Node	data structure	The Tree Node data structure holds the data and link to next node.

```
typedef struct TNodeType
{
struct TNodeType* left; /* left child */
TNodeData data; /* data value */
struct TNodeType* right; /* right child */
}TNode, *TNodePtr;
```

tree.c	code	The code contains the function definitions.

BINARY TREE CODE

Here's the code for the Enhanced Binary Tree 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 tree.h.

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

    #define false 0
    #define true 1
    typedef  int  bool;

    #endif
```

Here's the binary tree header file:

```
/* tree .h */

#include "defs.h"
#ifndef __TREE
#define __TREE

typedef int TNodeData; /* tree node data type */
typedef  struct TNodeType

        {
        struct TNodeType* left; /* left child */
        TNodeData data; /* data value */
        struct TNodeType* right; /* right child */
        } TNode, * TNodePtr;

typedef TNodePtr Tree; /* tree */

/* function prototypes for binary search tree */
/* insert item into tree */
TNodePtr insertTree(TNodePtr tree,TNodeData data);

/* remove data item from tree */
TNodePtr removeTree(TNodePtr tree,TNodeData data);

/* find smallest element in tree */
TNodePtr findMin(TNodePtr tnode);

/* find data item in tree */
TNodePtr findTree(TNodePtr tree,TNodeData data);

/* print tree */
void printTree(TNodePtr tree);

/* de-allocate memory for tree */
void destroyTree(TNodePtr tree);

#endif
```

Here's tree module implementation code:

```
/* tree.c */
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

/* deallocate all memory for tree */
void destroyTree(TNodePtr tnode)

        { if(tnode!=NULL)
        {
        destroyTree(tnode->left);
        destroyTree(tnode->right);
        free(tnode);
        }
        }
```

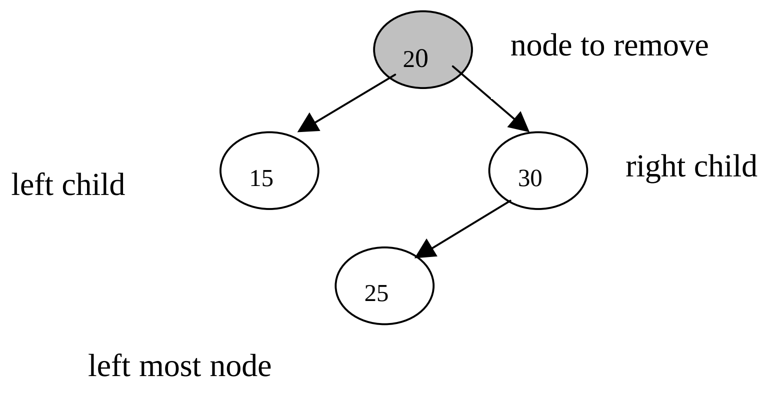
Inserting items into binary tree

We use recursion to insert an item into a binary tree. The function traverses the tree 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. As the function unwinds all the nodes are re-linked.

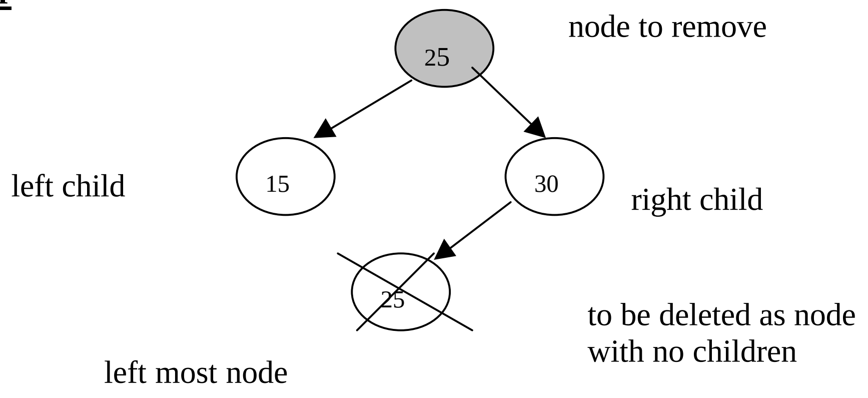
```
/* insert */
/* recursion will find the point to insert the new node */
/* it traverse the tree till it finds where to inset the new node */
TNodePtr insertTree(TNodePtr tnode,TNodeData data)

    {
    /* if empty must make a node */
    if(tnode == NULL)

        {
        /* allocate memory for a new tree node */
        tnode = (TNodePtr)malloc(sizeof(TNode));
        if (tnode == NULL)return NULL;

        /* put data into tree node */
        tnode->data = data;
        tnode->left= NULL;
        tnode->right = NULL;
        }

    /* find where to insert item by traversing tree */
    else if(data < tnode->data)
    tnode->left = insertTree(tnode->left,data); /* copy left */
    else if(data > tnode->data)
    tnode->right = insertTree(tnode->right,data); /* copy right */
    return 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.



after



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.

```
/* delete node from tree */
/* removing item from binary tree
* case 1 : two children
* case 2 : one child */
TNodePtr removeTree(TNodePtr tnode,TNodeData data)

        {
        TNodePtr temp, cnode; /* pointers to tree nodes */

        /*if tree not empty */
        if(tnode != NULL)

                {
                /* re-link and traverse left */
                if(data < tnode->data)tnode->left=removeTree(tnode->left,data);




                /* re-link and traverse right */
```

```
          else if(data > tnode->data)
             tnode->right=removeTree(tnode->right,data);

          /* found data item */
          `else

               {
               /* check for two children */
               if(tnode->left && tnode->right)

                    {
                    temp = findMin(tnode->right); /* find min node */
                    tnode->data = temp->data; /* set node data smaller */
                    /* continue traversal right */
                    tnode->right = removeTree(tnode->right,tnode->data);
                    }

               /* one child */
               else

                    {
                    temp = tnode; /* point to current tree node */
                    /* get pointer to child */
                    if(tnode->left == NULL) cnode = tnode->right;
                    if(tnode->right == NULL) cnode = tnode->left;
                    free (tnode); /* deallocate current node */
                    return cnode; /* return pointer to child */
                    }

               }
               return tnode;

          }

     } /* end removeTree */

/* finMin */
/* find smallest data item in binary tree */
/* the smallest element must be the most left item */
TNodePtr findMin(TNodePtr tnode)

     {
     /* if tree not empty */
     if(tnode != NULL)

          {
          if(tnode->left==NULL)return tnode; /* left leaf */
          else return (findMin(tnode->left)); /* continue left */
          }

     return tnode; /* return left most node */
     }
```

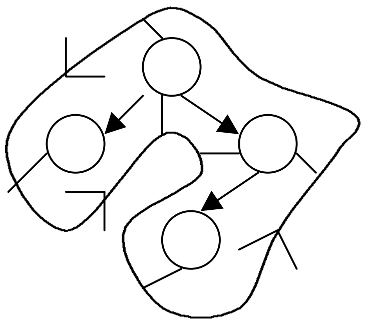
Finding elements in binary tree

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 traverse left. if the data item is larger than the current node then we traverse 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 */
TNodePtr findTree(TNodePtr tnode, TNodeData data)

        {
        /* tree not empty */
        if(tnode != NULL)

                {
                /* search and traverse left */
                if(data < tnode->data)return (findTree(tnode->left,data));

                /* search and traverse right */
                else if(data > tnode->data) return (findTree(tnode->right,data));
                }

        return 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 printf statement.

order	code	position of printf statement

pre-order	`printf("%d",tnode->data);`
	`printTree(tnode->left);`
	`printTree(tnode->right);`
		top

in-order	`printTree(tnode->left);`
	`printf("%d ",tnode->data);`
	`printTree(tnode->right);`
		middle

post-order	`printTree(tnode->left);`
	`printTree(tnode->right);`
	`printf("%d ",tnode->data);`
		bottom

```
 /* print a tree */
/* print tree data elements in order using recursion */
/* function calls itself until last node reached */
void printTree(TNodePtr tnode)

     {
     /* check for empty tree */
     if(tnode != NULL)

          {
          printTree(tnode->left); /* print out left data item */
          printf("%d ",tnode->data); /* middle inorder */
          printTree(tnode->right); /* print out right data item */
          }

     }

/* main program */
int main()

     {
     Tree tree=NULL;
     tree = insertTree(tree,10);
     printf("\n tree: ");
     printTree(tree);
     tree = insertTree(tree,8);
     printf("\n tree: ");
     printTree(tree);
     tree = insertTree(tree,5);
     tree = insertTree(tree,20);
     if(findTree(tree,8))printf("\n found tree item 8");
     if(findTree(tree,15))printf("\n found tree item 10");
     tree = insertTree(tree,15);
     tree = insertTree(tree,30);
```

```
        tree = removeTree(tree,10);
        printf("\n tree: ");
        printTree(tree);
        tree = removeTree(tree,15);
        printf("\n tree: ");
        printTree(tree);
        tree=removeTree(tree,20);
        tree=removeTree(tree,8);
        printf("\n tree: ");
        printTree(tree);
        destroyTree(tree);
        return 0;
        }
```

tree program output:

```
tree: 10

tree: 8 10

found tree item 8

tree: 5 8 15 20 30

tree: 5 8 20 30

tree: 5 30
```

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 the destroyTree routine to deallocate all tree nodes. Hint use postfix.

Lesson 10 Exercise 3

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:

```
                        10

               6                  20

        5                   15           30
```

Lesson 10 Exercise 4

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 10 Exercise 5

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

Lesson 10 Exercise 6

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

Lesson 10 Exercise 7

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

Lesson 10 Exercise 8

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

Lesson 10 Exercise 9

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

Lesson 10 Exercise 10

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

Lesson 10 Exercise 11

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

Lesson 10 Exercise 12

Write a program that will print out the number of nodes per level.

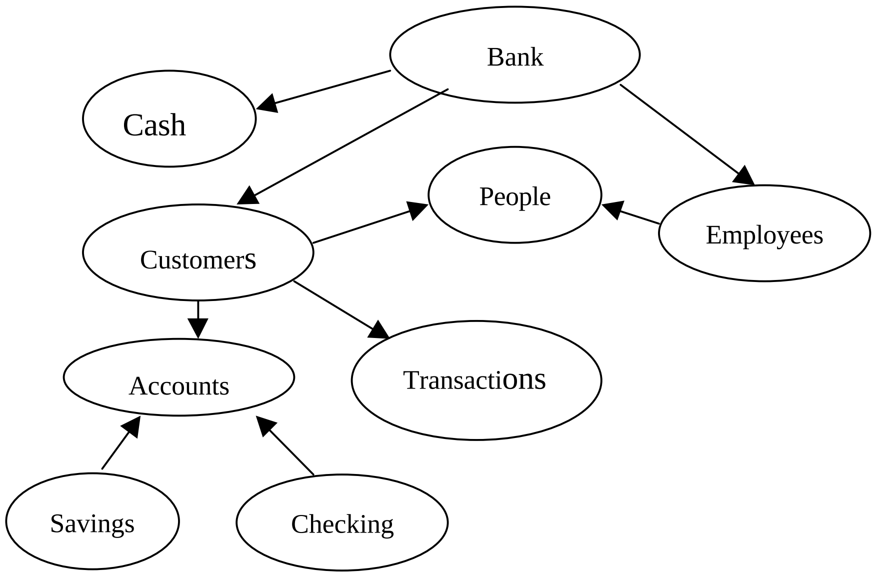
C DATA STRUCTURES PROGRAMMERS GUIDE PROJECT

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 modules. A Bank is a very good example of modular 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 module. Modules may be sub modules of other modules. Modules may also use and include other modules. An up ↑ arrows means a module is a sub module and contains the preceding module, A down ↓ arrow means a module will use another module as a variable. The following is our bank programming model.



SPECIFICATIONS

Before you can write any program you need to have **specifications**. A <u>specification</u> 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 modular programming each sub component can be a module. Modular programming is ideal to break down large complicated programming problems 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 module

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. You will need the Tree and Hash Table modules from previous lessons. The bank module will be a **typedef** structure having the following member variables. If you put the structure in the bank.c file than all member variables will only be known to the bank functions. If you put the structure in the bank.h file than all member variables will known to all functions in the project.

member	**data type**	**base module**	**description**
customers	Tree of Customer*	Person	represents an Binary Search Tree of pointers to Customer module
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 module
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

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 receive more customers the hash table will double in size frequently or even vice-versa. Collisions are stored in a link list sorted by account number.

Bank module Operations

A bank needs to perform operations, each implemented by the following functions:

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

Person module

A Person represents a Customer or Employees name, address, phone and SIN, all represented by String objects. You can use the String module from this course or the one supplied by your compiler. The Person module will be a **typedef** structure having the following member variables. If you put the structure in the person.c file than all member variables will only be known to the Person module functions. If you put the structure in the person.h file than all member variables will known to all functions in the project.

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

Person module operations

For this module you just need an **init()** function, a **getName()** function and functions to get and print out the person information. The **getName()** function must combine the first, initial and last name into one character string. The get function will be used to get customer information from the keyboard or file. The print function will be used to print out customer information to the screen or file.

function prototype	**description**
void initPerson(Person* p);	initialize person module
char* getName(Person* p);	get persons full name
void getPerson(Person* p);	get person info
void printPerson(Person* p);	print person info

Customer module

The Customer module includes the Person module. You just need a few more things for the customer module. 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. The Customer module will be a **typedef** structure having the following member variables. If you put the structure in the person.c file than all member variables will only be known to the Customer module functions. If you put the structure in the customer.h file than all member variables will known to all functions in the project.

member	data type	description
bank	Bank*	pointer to bank module
business phone	char[]	work phone number
accounts	LinkList of Account* modules	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* modules	double link list of pointers to Transaction object
maxTransactions	int	maximum number of transactions
numTransactions	int	number of transactions if Transaction array
person	Person	person info

Customer module operations

You need a default constructor , the constructor must also call the Persons base module default constructor. You need an **equals()** 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 functions to get and print out the Customer information. The Customer module initializing function must also initialize the Persons base module.

function prototype	description
void initCustomer();	initialize customer module
bool equals (Customer* customer1, Customer* customer2);	compare two customer objects by full name
bool addAccount (Customer* customer, Account* account)	add new customer account
bool viewAccount(Customer* customer, long number);	view customer account
bool deleteAccount (Customer* customer, Account* account)	delete customer account
bool addTransaction (Customer* customer, Account* account)	add new customer account
bool viewTransactions (Customer* customer, long number);	search for customer account
void get(Customer* customer);	get customer info
void put(Customer* customer);	print customer info

Employee module

The Employee module includes the Person module. You just need a few more things for the Employee module like a pointer to the bank object, salary and total salary paid so far. The Employee module will be a **typedef** structure having the following member variables. If you put the structure in the employee.c file than all member variables will only be known to the Employee module functions. If you put the structure in the employee.h file than all member variables will known to all functions in the project.

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

Employee module operations

You will need an initializing function that accepts a pointer to a Employee module. The Employee module initializing function must also call the Persons base module initializing function. You need an **equals()** function to compare two employee objects by search key. You also need a function called **pay()** to pay the employees. You need an get function and print function to print out the Employee information.

function prototype	description
void initEmployee(Employee* employee);	initialize employee module
bool equals(Employee* employeee1, Employee* employee2);	compare two employee objects by full name
void pay(Employee*);	pay this employee
void get(Employee* employee);	get employee info
void put(Employee* employee);	print employee info

Account module

The account module contains all the common information about an account like Account number, pointer to Customer module, balance and service fees. The Account module will be a **typedef** structure having the following member variables. If you put the structure in the account.c file than all member variables will only be known to the Account module functions. If you put the structure in the account.h file than all member variables will known to all functions in the project.

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 module operations

You will need an initializing function that accepts a pointer to a Account module, service fee and an initial balance. The account number will be internally generated by the Checking and Savings sub modules. You need an **equals()** function to compare two Account objects by search key. You will need functions to get and print out the Account information.

function prototype	description
void initAccount(Account* account, Customer* customer, float serviceFee, float balance);	initializing constructor
bool equals(Account* account1, Account account2);	compare two account modules by account number
getAccount(Account* account);	get account info
putAccount(Account* account);	print account info

Checking module

The Checking module includes an Account module. A checking account has an overdraft but does 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. The Checking module will be a **typedef** structure having the following member variables. If you put the structure in the checking.c file than all member variables will only be known to the Checking module functions. If you put the structure in the checking.h file than all member variables will known to all functions in the project.

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
account	Account	account module

Checking module operation

The initializing function must also receive the overdraft and transaction fee and call initialize function the account base module. It is inside the Checking initialize function where the account number is assigned and incremented. The Checking module 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 functions to get and print out the Checking information**.**

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

Savings module

The Savings module inherits the Account module. 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. The Checking module will be a **typedef** structure having the following member variables. If you put the structure in the savings.c file than all member variables will only be known to the Savings module functions. If you put the structure in the savings.h file than all member variables will known to all functions in the project.

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

A Savings module operation

The initializing function must also receive the overdraft and interest rate and call the Account base module initializing function. It is inside the Savings module where the account number is assigned and incremented. The savings module 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 function's to get and print out the Savings information.

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

Transaction module

We must keep track of the transactions that each customer makes. A transaction module will have the following members. The Transaction module will be a **typedef** structure having the following member variables. If you put the structure in the transaction.c file than all member variables will only be known to the Savings module functions. If you put the structure in the transaction.h file than all member variables will known to all functions in the project.

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 module operation

You will need an initializing function to initialize transaction type, accounts and amount. All Transaction modules are added to the Customer module where the account resides in a double link list. 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 function's to get and print out the Transaction information.

function prototype	description
Transaction(Transaction* transaction, TransactionType type, long from, long to, float balance); float serviceFee, float balance, float transactionFee);	initializing constructor
get(Transaction* transaction);	get transaction info
print(Transaction* transaction);	print transaction 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

module summary

You will have the following modules. Put the **main()** function in the Bank module.

header files:	**implementation files:**
Bank.h	Bank.c
Person.h	Person.c
Customer.h	Customer.c
Employee.h	Employee.c
Account.h	Account.c
Savings.h	Savings.c
Checking.h	Checking.c
Transaction.h	Transaction.c

header files:	**implementation files:**
link.h	link.c
dlink.h	dlink.c
hash.h	hash.c
tree.h	tree.c

main function

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

Hints on completing Project:

(1) Sketch an operational model of all modules showing how all the modules interact.

(2) Write all the module definitions first top down. Start at the top of the module model and proceed to the bottom module Write the Base module definitions first.

(3) Write each module implementation one at a time bottom up. Start at the bottom module of the module model and then finish up at the top. Always write the base modules before writing the derived modules.

(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 h file includes another h file and this h file includes the one that called it. The only way to avoid tins situation is to use a **forward reference.** For example the **bank.h** file needs to include **customer.h** file. The **customer.h** file needs to include **bank.h** file. The result is a circular #include reference. The only way out of this dilemma is to use a **forward reference.**

In the account module just say:

```
struct BankType* bank;
```

This will solve all your problems. Now the compiler knows that Bank is a pointer to a Bank object.

Marking Scheme:

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

ENHANCEMENTS

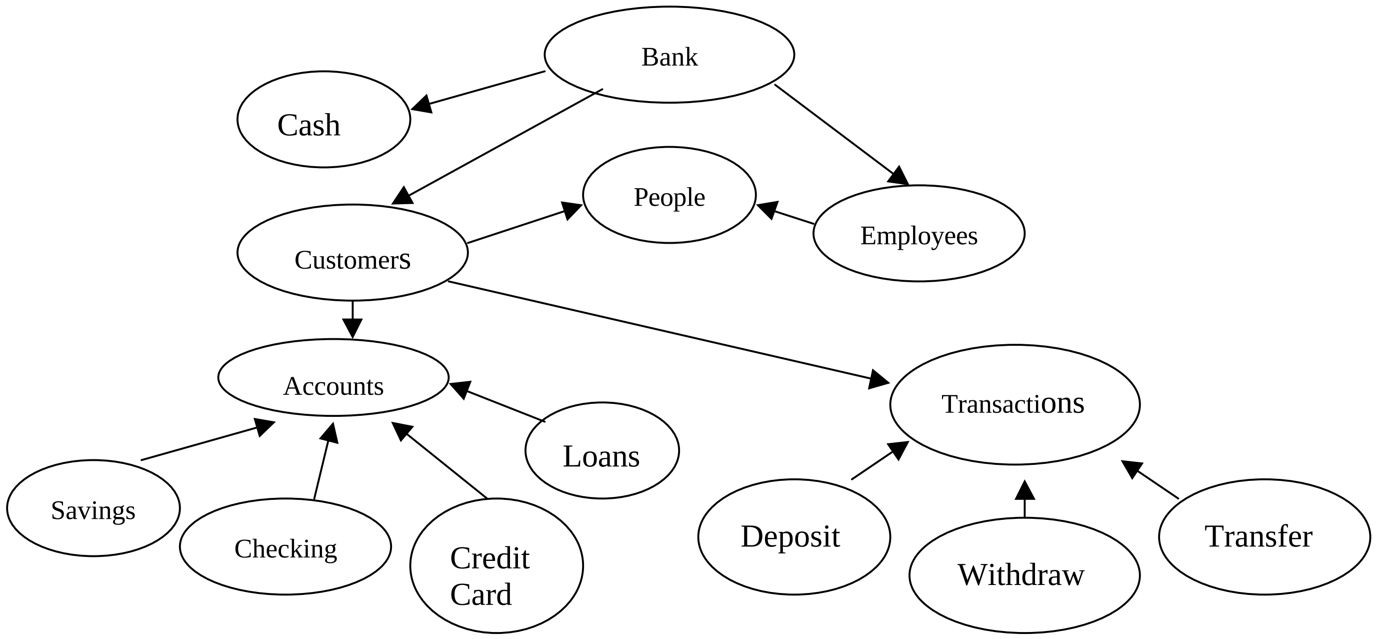
(1) Write all customer accounts, employees and transaction objects to a file. When your program starts up it will read the file and update the arrays used to store our objects. Call your file bank.dat.

(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 modules. Deposit, Withdraw, and Transfer.

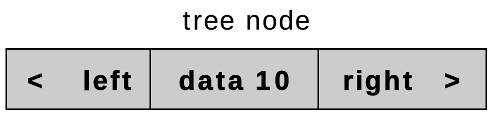


C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 12

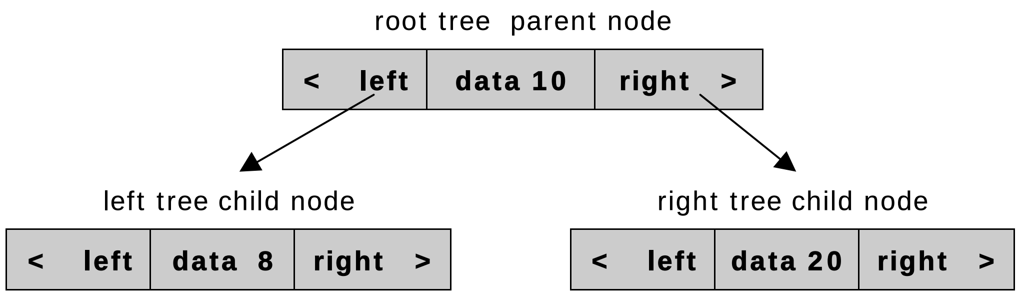
File:	CdsGuideL12.doc
Date Started:	July 24,1998
Last Update:	Dec 20, 2001
Status:	proof

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 you:

function	purpose
TNodePtr insertTree (TNodePtr tree,TNodeData data);	insert item into tree
TNodePtr removeTree (TNodePtr tree,TNodeData data);	remove data item from tree
TNodePtr findMin(TNodePtr tnode);	find smallest element in tree
TNodePtr findTree (TNodePtr tree,TNodeData data);	find node in tree

int num_nodes(TNodePtr tree);	calculate number of nodes in a tree
int num_leaves(TNodePtr tree);	calculate number of leaves in a tree
int num_full_nodes(TNodePtr tree);	calculate number of full nodes in a tree
bool has_neg(TNodePtr tree);	return true if tree has a negative value node
bool has_pos(TNodePtr tree);	return true if tree has a positive value node
bool has_value(TNodePtr tree, TNodeData data);	return true if data value in tree
int max_value(TNodePtr tree);	return maximum value of tree
int min_value(TNodePtr tree);	return minimum value of tree
int sum_tree(TNodePtr tree);	return sum of all nodes in tree
int max_depth(TNodePtr tree);	return max depth of tree
bool is_balanced(TNodePtr tree);	return true if tree balanced
bool is_complete(TNodePtr tree);	return true if tree complete
void print_tree(TNodePtr tree);	print tree by indentation
void print_level(TNodePtr tree);	print tree by level
void destroyTree(TNodePtr tree);	de-allocate memory for tree

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. In you main function you will need a pointer to a binary tree node to represent the root of the tree.

tree.h	header file	The header file contains the function prototypes.
Node	data structure	The Tree Node data structure holds the data and link to next node.

```
typedef struct TNodeType
{
struct TNodeType* left; /* left child */
TNodeData data; /* data value */
struct TNodeType* right; /* right child */
}TNode,*TNodePtr;
```

tree.c	code	The code contains the function definitions.

BINARY TREE CODE

Here's the code for the Enhanced Binary Tree 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 tree.h.

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

#define false 0
#define true 1
typedef  int  bool;

#endif
```

Here's the binary tree header file:

```
/* tree .h */
#include "defs.h"
#ifndef  __TREE
#define  __TREE

typedef int TNodeData; /* tree node data type */

/*' tree node data structure */
struct TNodeType

        {
        TNodePtr left; /* left child */
        TNodeData data; /* data value */
        TNodePtr right; /* right child */
        }TNode,*TNodePtr;

typedef TNodePtr Tree; /* tree */

/*** function prototypes for binary search tree ***/

/* insert item into tree */
TNodePtr insertTree(TNodePtr tree,TNodeData data);

/* remove data item from tree */
TNodePtr removeTree(TNodePtr tree,TNodeData data);

/* find smallest element in tree */
TNodePtr findMin(TNodePtr tnode);

/* find node in tree */
TNodePtr findTree(TNodePtr tree,TNodeData data);

/* print tree */
void printPreorder(TNodePtr  tree);
void printInorder(TNodePtr  tree);
void printPostorder(TNodePtr  tree);
```

```
/* calculate number of nodes of tree */
int num_nodes(TNodePtr tree);

/* calculate number of leaves of tree */
int num_leaves(TNodePtr tree);

/* calculate number of full nodes of tree */
int num_full_nodes(TNodePtr tree);

/* has a negative value node */
bool has_neg(TNodePtr tree);

/* has a positive value node */
bool has_pos(TNodePtr tree);

/* has value node */
bool has_value(TNodePtr tree, TNodeData data);

/* return maximum value */
int max_value(TNodePtr tree);

/* return minimum value */
int min_value(TNodePtr tree);

/* sum of tree */
int sum_tree(TNodePtr tree);

/* max depth of tree */
int max_depth(TNodePtr tree);

/* return true if tree balanced */
bool is_balanced(TNodePtr tree);

/* return true if tree complete */
bool is_complete(TNodePtr tree);

/* print tree by indentation */
void print_level(TNodePtr tree,int level);

/* print tree by level */
void print_level(TNodePtr tree);

/* de allocate memory for tree */
void destroyTree(TNodePtr  tree);

#endif
```

Here's tree module implementation code:

```
/* tree.c */
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"
#include "quelist.h"
```

Destroying binary tree

By using post fix we can safely delete each tree node, because this is the last time we pass it.

```
/* deallocate all memory for tree */
void destroyTree(TNodePtr tnode)
{

        if(tnode!=NULL)
        {
        destroyTree(tnode->left);
        destroyTree(tnode->right);
        free(tnode);
        }

}
```

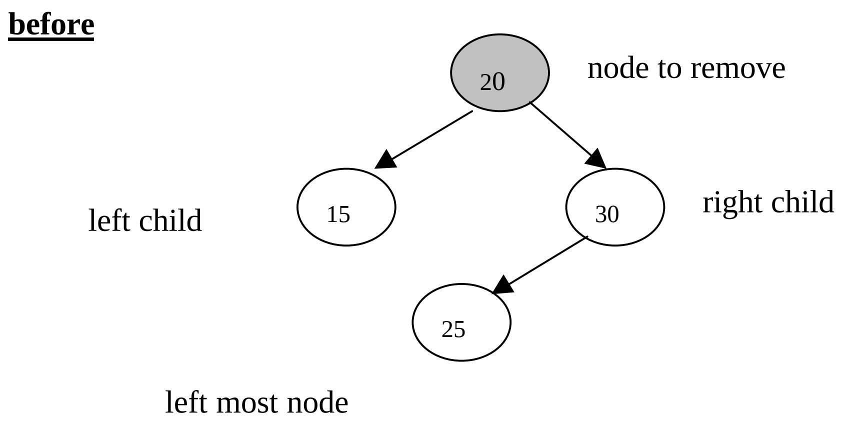
Inserting items into binary tree

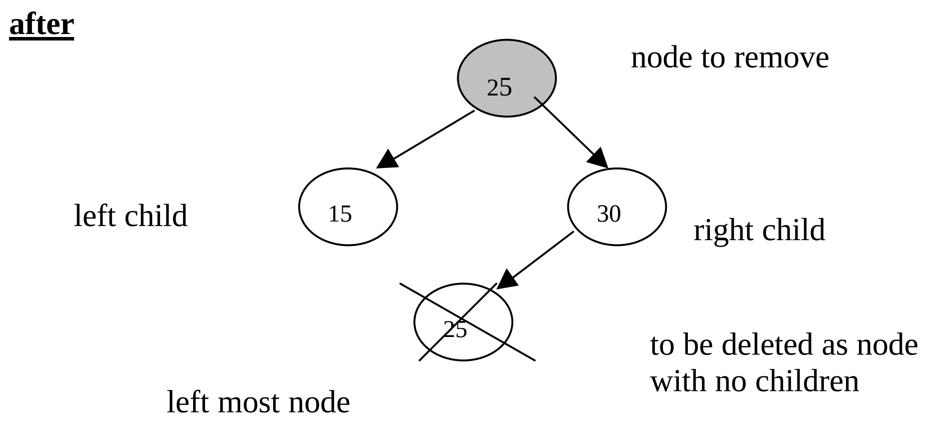
We use recursion to insert an item into a binary tree. The function traverses the tree by **comparing** the data to be inserted with the data value at each node it encounters. If the value is **smaller** it traverse **left**, if the data value is **larger** it traverse **right**. When it comes to a leaf a new node is allocated and attached to the left or right of the leaf node. As the function unwinds all the nodes are re-linked.

```
/* insert */
/* recursion will find the point to insert the new node */
/* it traverse the tree till it finds where to inset the new node */
TNodePtr insertTree(TNodePtr tnode,TNodeData data)

        {
        /* if empty must make a node */
        if(tnode == NULL)

                {
                /* allocate memory for a new tree node */
                tnode = (TNodePtr)malloc(sizeof(TNode));
                if (tnode == NULL)return NULL;

                /* put data into tree node */
                tnode->data = data;
                tnode->left= NULL;
                tnode->right = NULL;
                }

        /* find where to insert item by traversing tree */
        else if(data < tnode->data)
        tnode->left = insertTree(tnode->left,data); /* copy left */
        else if(data > tnode->data)
        tnode->right = insertTree(tnode->right,data); /* copy right */
        return 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.

```
/* delete node from tree */
/* removing item from binary tree
* case 1 : two children
* case 2 : one child */
TNodePtr removeTree(TNodePtr tnode,TNodeData data)

        {
        TNodePtr temp, cnode; /* pointers to tree nodes */

        /*if tree not empty */
        if(tnode != NULL)

                {

                /* copy and traverse left */
                if(data < tnode->data)tnode->left=removeTree(tnode->left,data);

                /* copy and traverse right */
                else if(data > tnode->data)
                   tnode->right=removeTree(tnode->right,data);

                /* found data item */
                else

                        {
                        /* check for two children */
                        if(tnode->left && tnode->right)

                                {
                                temp = findMin(tnode->right); /* find min node */
                                tnode->data = temp->data; /* set node data smaller */

                                /* continue traversal right */
                                tnode->right = removeTree(tnode->right,tnode->data);

                                /* one child */
                                else

                                        {
                                        temp = tnode; /* point to current tree node */
                                        /* get pointer to child */
                                        if(tnode->left == NULL) cnode = tnode->right;
                                        if(tnode->right == NULL) cnode = tnode->left;
                                        free (tnode); /* deallocate current node */
                                        return cnode; /* return pointer to child */
                                        }

                                }

                        }
                        return tnode;

        } /* end removeTree */
```

```
/* finMin */
/* find smallest data item in binary tree */
/* the smallest element must be the most left item */
TNodePtr findMin(TNodePtr tnode)

    { /* if tree not empty */
    if(tnode != NULL)

        {
        if(tnode->left==NULL)return tnode; /* left leaf */
        else return (findMin(tnode->left)); /* continue left */
        }

    return tnode; /* return left most node */
    }
```

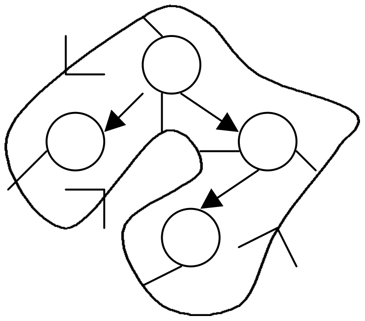
Finding elements in binary tree

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 traverse left. if the data item is larger than the current node then we traverse right. we stop traversing when we find the item we are looking for.

```
/* find data element in tree */
/* if found return node in tree, if not found return NULL */
TNodePtr findTree(TNodePtr tnode, TNodeData data)

    {
    /* tree not empty */
    if(tnode != NULL)

        {
        /* search and traverse left */
        if(data < tnode->data)return (findTree(tnode->left,data));
        /* search and traverse right */
        else if(data > tnode->data) return (findTree(tnode->right,data));
        }

    return 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

```
/* print tree data elements in pre-order using recursion */
void printPreorder(TNodePtr tnode)

        {
        /* check for empty tree */
        if(tnode != NULL)

                {
                printf("%d ",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 printInorder(TNodePtr tnode)

        {
        /* check for empty tree */
        if(tnode != NULL)

                {
                printInOrder(tnode->left); /* print out left data item */
                printf("%d ",tnode->data); /* middle inorder */
                printInOrder(tnode->right); /* print out right data item */
                }

        }

/* print tree data elements in post order using recursion */
void printPostorder(TNodePtr tnode)

        {
        /* check for empty tree */
        if(tnode != NULL)

                {
                printPostOrder(tnode->left); /* print out left data item */
                printPostOrder(tnode->right); /* print out right data item */
                printf("%d ",tnode->data); /* last  post order */
                }

        }
```

Find Data Element in Tree

This function traverses through the tree looking for a specific data element. If found then a pointer to the node is returned. I f not found null is returned.

```
    /* find data element in tree */
    /* if found return position in tree */
    /* if not found return error */
    TNodePtr findTree(TNodePtr tnode, TNodeData data)

        {
        if(tnode != NULL)

            {
             if(data < tnode->data)
            return (findTree(tnode->left,data))
            else if(data > tnode->data)
            return (findTree(tnode->right,data));
            }

        return 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 num_nodes(TNodePtr tree)

    {
    if(tree != NULL)

        {
        /* add 1 to count as traverse to tree */
        return (num_nodes(tree->left) + num_nodes(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 num_leaves(TNodePtr tree)

        {
        if(tree != NULL)

            {


            /* add 1 to count if no children as traverse to tree */
```

```
                if((tree->left == NULL) && (tree->right==NULL))
                return (num_leaves(tree->left) + num_leaves(tree->right) + 1);
                /* traverse without adding 1 */
                else return (num_leaves(tree->left) + num_leaves(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 num_full_nodes(TNodePtr tree)

        {
        if(tree != NULL)

                {
                /* add 1 to count if two children as traverse to tree */
                if((tree->left != NULL) && (tree->right!=NULL))
                return (num_full_nodes(tree->left) + num_full_nodes(tree->right) + 1);
                /* traverse without adding 1 */
                else return (num_full_nodes(tree->left) + num_full_nodes(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 */
int has_neg(TNodePtr tree)

        {
        if(tree != NULL)

                {
                return ((tree->data < 0) | has_neg(tree->left) | has_neg(tree->right));
                }

        else return 0;
        }
```

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 */
int has_pos(TNodePtr tree)

        {
        if(tree != NULL)

                {
                return ((tree->data >= 0) | has_pos(tree->left) | has_pos(tree->right));
                }

        else return 0;
        }
```

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 */
int has_value(TNodePtr tree,TNodeData data)

        {
        if(tree != NULL)
        {
        return ((tree->data == data) | has_value(tree->left,data) | has_value(tree->right,data));
        }

        else return 0;
        }
```

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 max_value(TNodePtr tree)

                {
                if(tree != NULL)

                        {
                        int max_left = max_value(tree->left);
                        int max_right = max_value(tree->right);
                        if((tree->data > max_left) && (tree->data > max_right))
                                return tree->data;


                        else if (max_left > max_right)
                return max_left;
            else return
                max_right;
            }

        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 min_value(TNodePtr tree)

    {
    if(tree != NULL)
    {
    int max_left = max_value(tree->left);
    int max_right = max_value(tree->right);
    if((tree->data < max_left) && (tree->data < max_right))
        return tree->data;
    else if (max_left < max_right)
        return max_left;
    else
        return max_right;
    }
    else return 10000;
    }
```

Sum up all node values in a tree

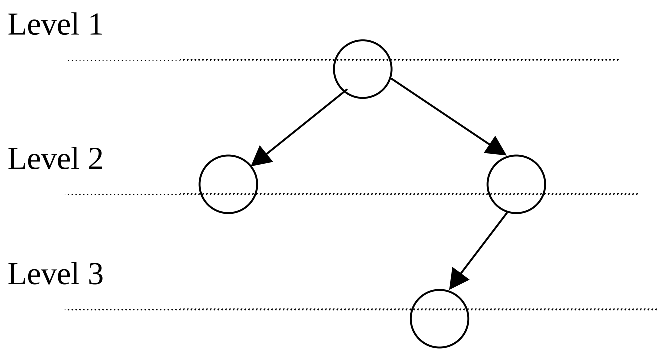
This function traverses every node in a tree adding all the node values.

```
    /* sum of tree */
    int sum_tree(TNodePtr tree)

        {
        if(tree != NULL)
        {
        return tree->data + sum_tree(tree->left) + sum_tree(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.

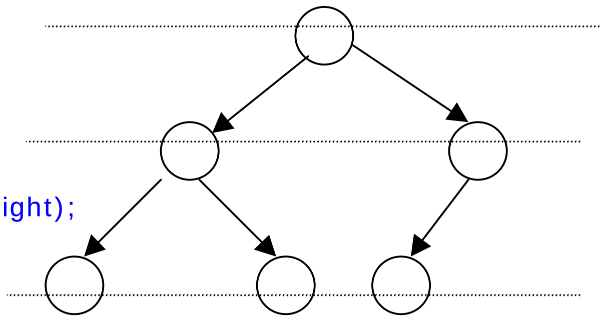
```
/* max depth of tree */
int max_depth(TNodePtr tree)

          {
          if(tree != NULL)
          {
          int depth_left = max_depth(tree->left);
          int depth_right = max_depth(tree->right);
          if(depth_left > depth_right)
           return depth_left + 1;
          else
          return depth_right + 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.

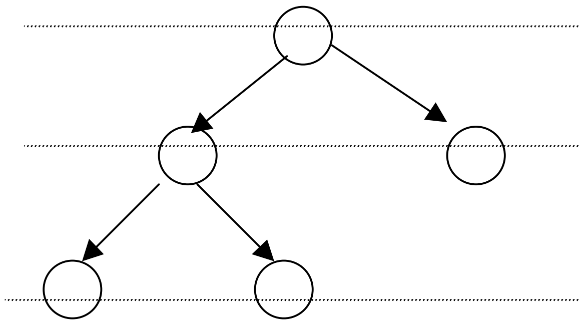
```
/* return true if tree balanced */
int is_balanced(TNodePtr tree)

          {
          if(tree != NULL)
          {
          return max_depth(tree->left) == max_depth(tree->right);
          }
          else return 0;
          }

          }
```



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 */
int is_complete(TNodePtr tree)

          {
          if(tree != NULL)
          {
          /* add 1 to count if no children as traverse to tree */
          return ((is_complete(tree->left) == NULL) ^ (is_complete(tree->right)==NULL));
          }
          else return 0;
          }
```



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 print_tree(TNodePtr tnode, int level)
{
int i;
level++; /* increment level */
if(tnode != NULL)
{
print_tree(tnode->left,level);
for(i=0;i<10-level;i++)
printf("   ");
printf("%2d\n",tnode->data);
print_tree(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 print_level(TNodePtr tree)
{
int i=1,j,k; /* counters */
int space = 6; /* space distance */
TNodePtr tnode;
QUEUE_LIST que; /* queue */
initQue(&que);
enqueue(&que,tree); /* push tree root into queue */
/* print leading space */
for(j=0;j<space/2;j++)printf("  ");
/* loop till queue is empty */
while(!isEmpty(&que))
{
tnode = dequeue(&que); /* get tree node */
/* print tree node data value */
if(tnode != NULL)
printf("%2d",tnode->data);
else
printf("  ");
/* print space between data */
for(j = 0;j<space;j++)printf("  ");
/* push all children of this node into queue */
if(tnode->left!=NULL)enqueue(&que,tnode->left);
if(tnode->right!=NULL)enqueue(&que,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)
{
printf("\n");
space = space/2; /* new space distance */
/* print leading space */
for(j=0;j<space/2;j++) printf("  ");
}
} /* end while */
}
```

Here is the main test function:

```
/* main program */
int main()
{
Tree tree=NULL;
TNode* tnode;

/* insert */
tree = insertTree(tree,10);
tree = insertTree(tree,8);
tree = insertTree(tree,5);
tree = insertTree(tree,20);
tree = insertTree(tree,15);
tree = insertTree(tree,30);

/* print tree by order */
printf("\n preorder: ");
printPreorder(tree);
printf("\n inorder: ");
printPreorder(tree);
printf("\n postorder: ");
printPostorder(tree);

/* find nodes */
tnode = findTree(tree,8);
if(tnode->data==8)

printf("\n found tree item 8");
tnode=findTree(tree,15);
if(tnode->data==15)
printf("\n found tree item 15");

/* remove */
tree = removeTree(tree,15);
tree = removeTree(tree,20);
tree = removeTree(tree,8);
tree = removeTree(tree,10);
tree = removeTree(tree,30);
tree = removeTree(tree,5);
destroyTree(tree);
/* insert tree again */
tree = insertTree(tree,10);
tree = insertTree(tree,8);
tree = insertTree(tree,5);
tree = insertTree(tree,20);
tree = insertTree(tree,15);
tree = insertTree(tree,30);

/* counting */
printf("number nodes: %d\n",num_nodes(tree));
printf("number leaves: %d\n",num_leaves(tree));
printf("number full nodes: %d\n",num_full_nodes(tree));

/* has value */
printf("has negative nodes: %d\n",has_neg(tree));
printf("has positive nodes: %d\n",has_pos(tree));
printf("has value node: %d\n",has_value(tree, 8));

/* max/min value */
printf("max value: %d\n",max_value(tree));
printf("min value: %d\n",min_value(tree));

/* sum of tree */
printf("sum of tree: %d\n",sum_tree(tree));

/* max depth */
printf("max depth of tree: %d\n",max_depth(tree));

/* tree balanced */
printf("tree balanced: %d\n",is_balanced(tree));

 /* tree complete */
printf("tree complete: %d\n",is_complete(tree));

/* make tree complete */
tree = insertTree(tree,9);
printf("tree complete: %d\n",is_complete(tree));

/* print tree using indentation */
printf("\n print tree by identation:\n");
print_tree(tree,10);

/* print tree by level */
printf("\n print tree by level:\n");
print_level(tree);

/* destroy tree */
destroyTree(tree);

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

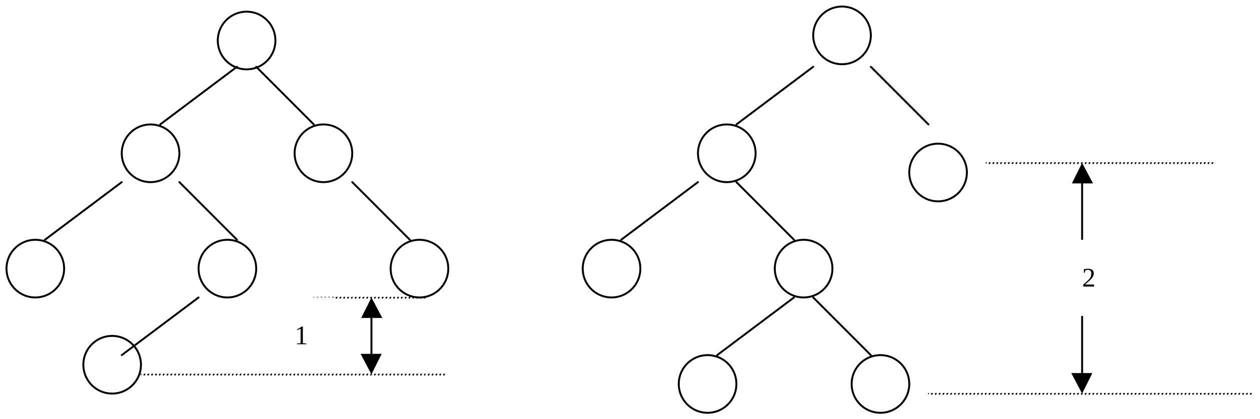
Write a program that will print out all the right child nodes in a tree

File:	CdsGuideL13.doc
Date Started:	Oct 24,1999
Last Update:	Dec 22, 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 subtrees 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 data structure is as follows:

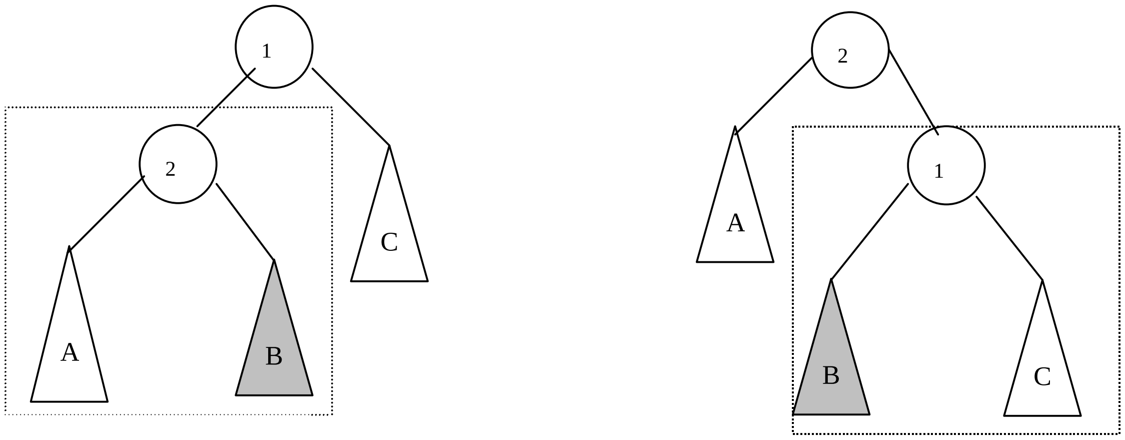
```
typedef int DATATYPE;

/* avl tree node */
typedef struct avlnode_type
{
struct avlnode_type *left;
DATATYPE data;
int height;
struct avlnode_type *right;
}AVLNODE,*AVLPTR;
```

INSERTING A NODE INTO A AVL TREE

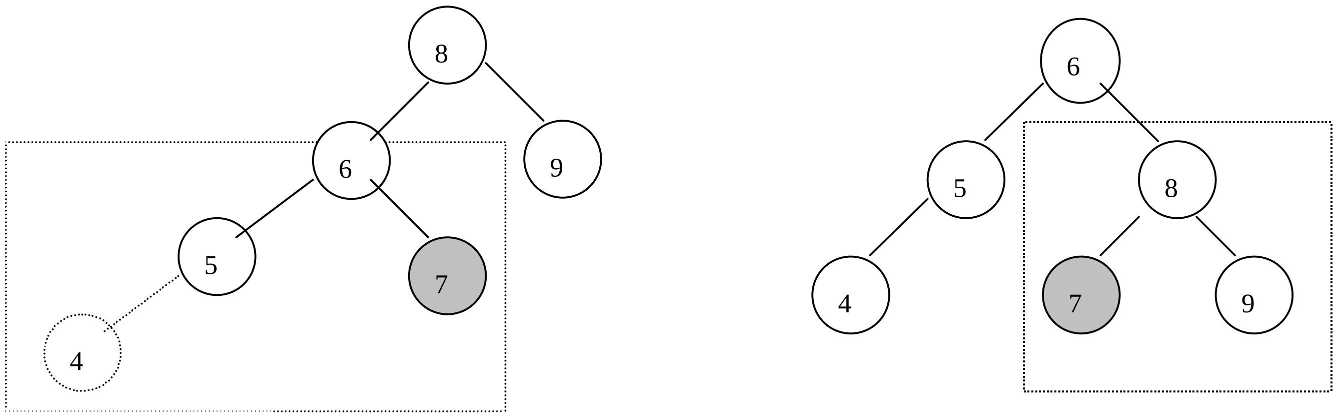
When we insert a Node into a AVL tree the tree must be re-balanced if the insertion routine causes the properties of the AVL tree to be violated. Balancing a tree is quite simple all we have to do is rotate sub-trees. We will have left rotation and right rotation. When we only rotate once this is called 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.



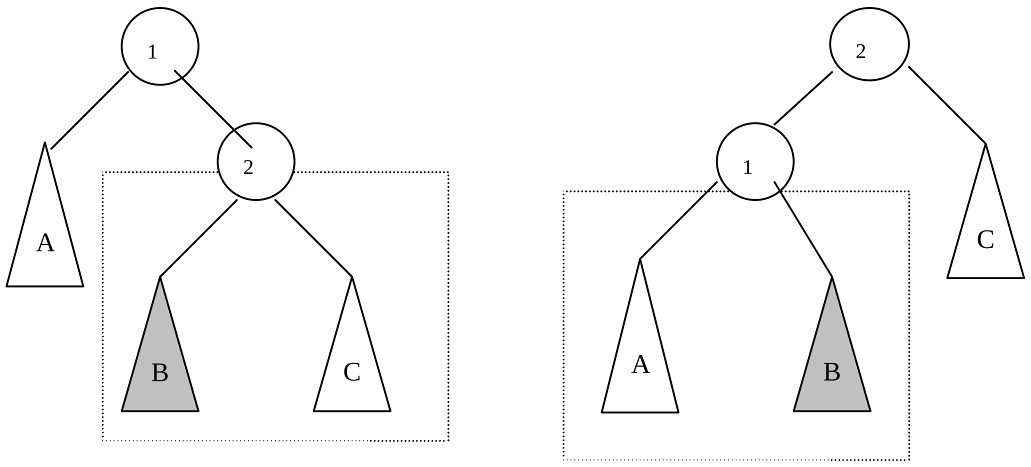
We use single left rotation to insert the node inserted on the left



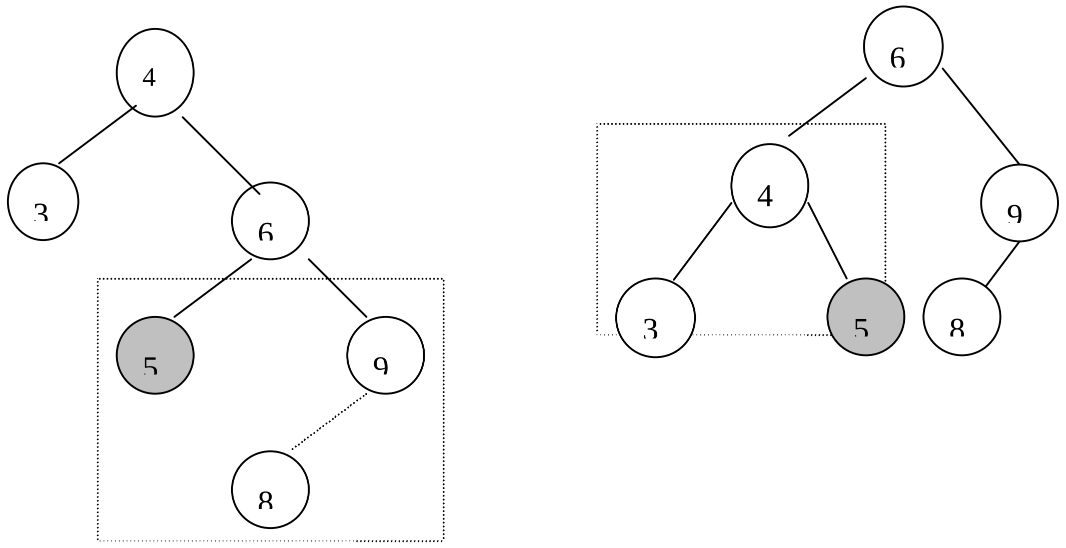
```
/* single rotate left */
AVLPTR rleft_single(AVLPTR T2)
{
AVLPTR T1;
T1 = T2->left;
T2->left = T1->right;
T1->right = T2;
T2->height = max(height(T2->left),height(T2->right))+1;
T1->height = max(height(T1->left),T2->height)+1;
return T1;
}
```

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

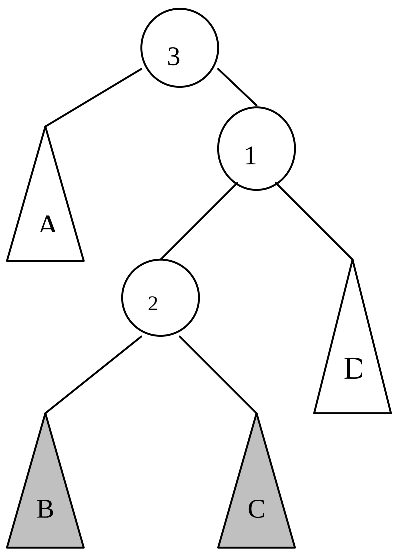


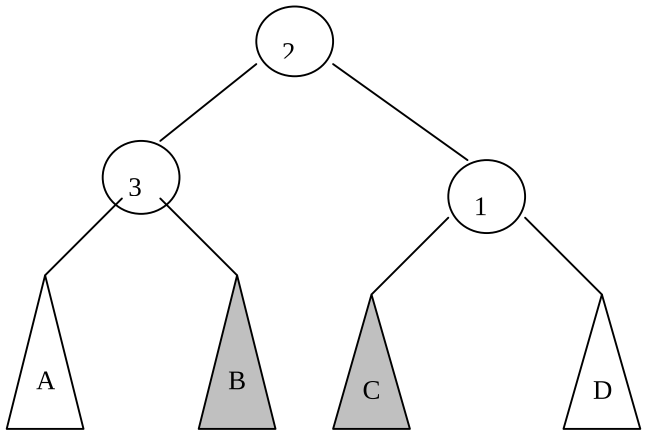
```
/* single rotate right */
AVLPTR rright_single(AVLPTR T2)
{
AVLPTR T1;
T1 = T2->right;
T2->right = T1->left;
T1->left = T2;
T2->height = max(height(T2->right),height(T2->left))+1;
T1->height = max(height(T1->right),T2->height)+1;
return T1;
}
```

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.

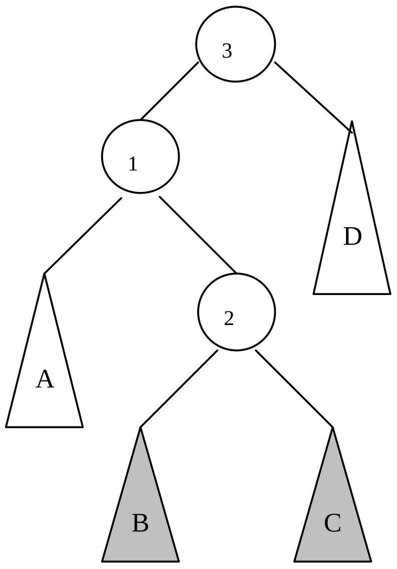
right-left double rotation

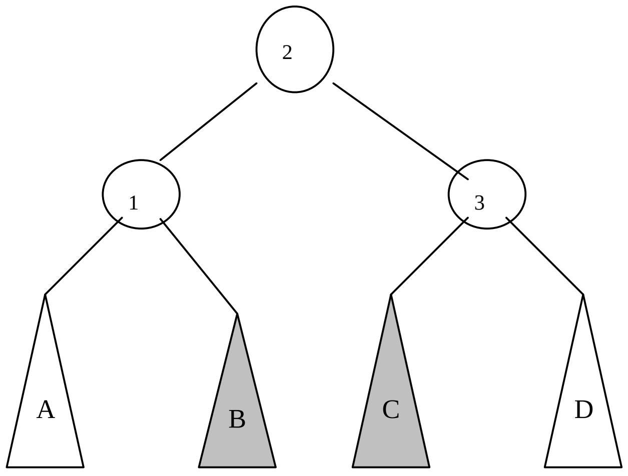




```
/* double rotate right-left */
AVLPTR rleft_double(AVLPTR T3)
{
T3->left = rright_single(T3->left);
return(rleft_single(T3));
}
```

left-right double rotation





```
/* double rotate left-right */
AVLPTR rright_double(AVLPTR T3)
{
T3->right = rleft_single(T3->right);
return(rright_single(T3));
}
```

Here is the AVL Code:

```
/* avl.c */
#include <stdio.h>
#include <stdlib.h>

typedef int DATATYPE;

/* avl tree node */
typedef struct avlnode_type
{
DATATYPE data;
struct avlnode_type *left;
struct avlnode_type *right;
int height;
}AVLNODE,*AVLPTR;

/* prototypes */
AVLPTR insert(AVLPTR T,DATATYPE data);
AVLPTR rleft_single(AVLPTR T2);
AVLPTR rright_single(AVLPTR T2);
AVLPTR rleft_double(AVLPTR T2);
AVLPTR rright_double(AVLPTR T2);
int height(AVLPTR N);
void print_tree(AVLPTR T);
void destroyTree(AVLPTR T);

/* avl test */
void main()
{
AVLPTR T=NULL;
printf("\n tree: ");
T = insert(T,10);
print_tree(T);
printf("\n tree: ");
print_tree(T);
T = insert(T,8);
printf("\n tree: ");
print_tree(T);
T = insert(T,5);
T = insert(T,20);
T = insert(T,15);
T = insert(T,30);
printf("\n tree: ");
print_tree(T);
destroyTree(T);
}
```

```
      /* avl functions */
      /* insert data into avl tree */
      AVLPTR insert(AVLPTR T,DATATYPE data)
      {
      if(T == NULL)

            {
            /* allocate tree node */
            T = (AVLPTR)malloc(sizeof(AVLNODE));
            if(T == NULL)
            {
            printf("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 */
AVLPTR rleft_single(AVLPTR T2)
{
AVLPTR 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 */
AVLPTR rleft_double(AVLPTR T3)
{
T3->left = rright_single(T3->left);
return(rleft_single(T3));
}

/* single rotate right */
AVLPTR rright_single(AVLPTR T2)
{
AVLPTR 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 */
AVLPTR rright_double(AVLPTR T3)
{
T3->right = rleft_single(T3->right);
return(rright_single(T3));
}

/* return height of node */
int height(AVLPTR 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(AVLPTR T)
     {

           if(T != NULL)
           {
           print_tree(T->left);
           printf("%d height: %d ",T->data,T->height);
           print_tree(T->right);
           }

     }

/* removes all nodes in tree */
void destroyTree(AVLPTR T)
{

     if( T != NULL )
     {
     destroyTree( T->left );
     destroyTree( T->right );
     free (T);
     }

}
```

Program Output

```
tree: 10 height: 0

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

Write a function to delete a node element from an AVL Tree. Remember if you delete a node you have to make sure the AVL tree is still balanced.

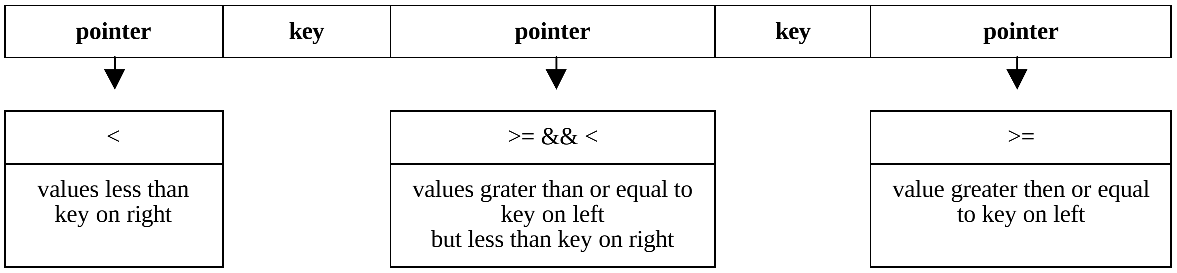
C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 14

File:	CdsGuideL14.doc
Date Started:	Apr 20,1999
Last Update:	Dec 22, 2001
Status:	proof

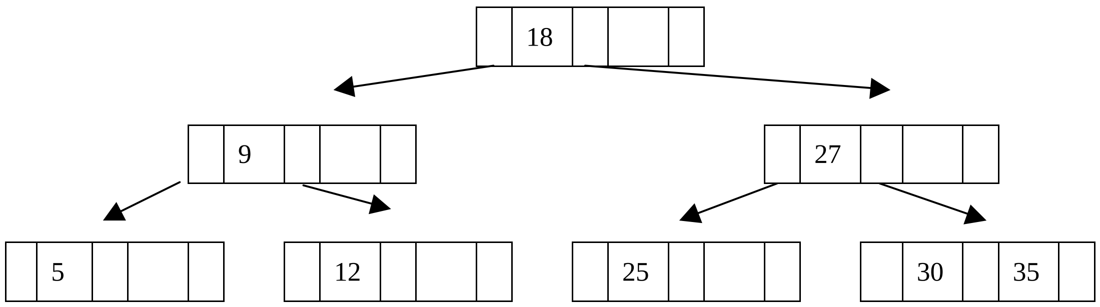
LESSON 14 B-TREES

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 pointer s 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


pointer
key
pointer
key
pointer
<
values less than key on right
>= && <
values grater than or equal to key on left but less than key on right
>=
value greater then or equal to key on left

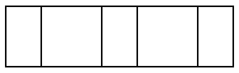
complete B -Tree



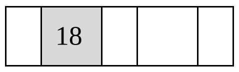
INSERTING KEYS IN A B-TREE

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.

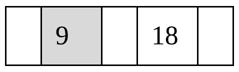
Initially the B-Tree is empty



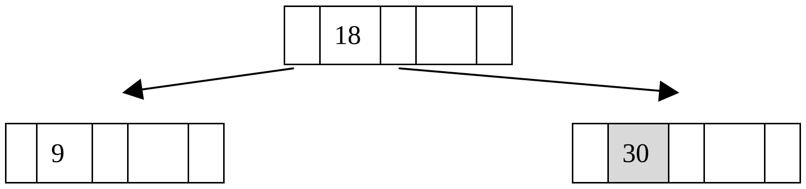
insert 18



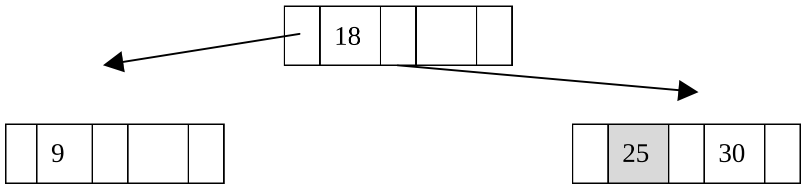
insert 9



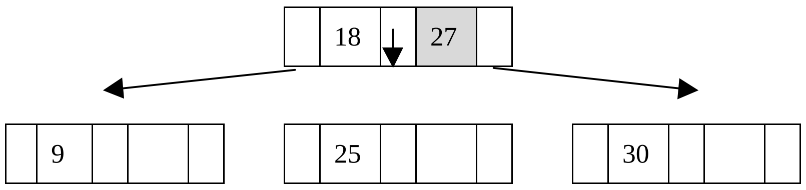
insert 30



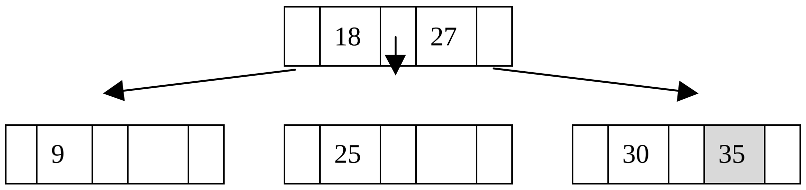
insert 25



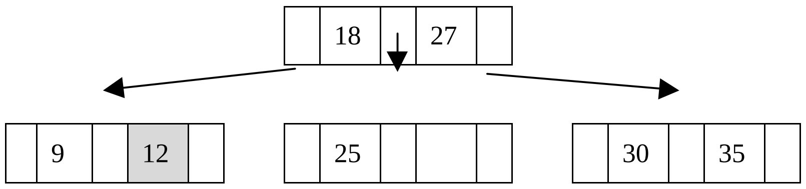
insert 27



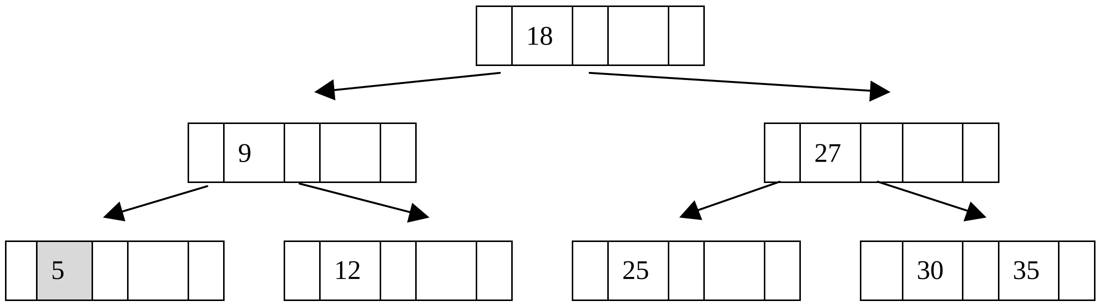
insert 35



insert 12



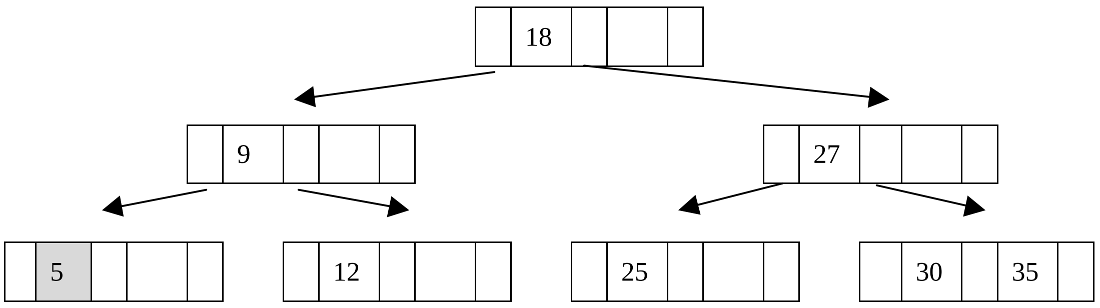
insert 5



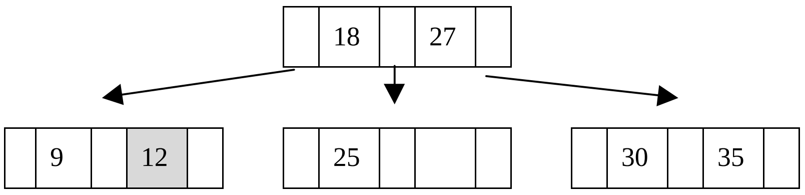
DELETING NODES IN A B-TREE

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.

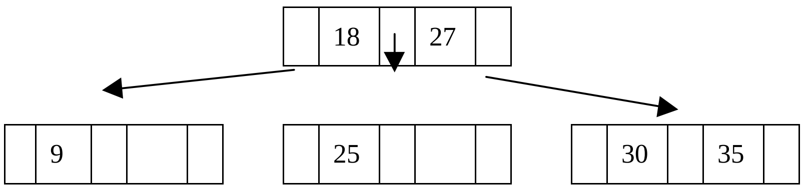
delete 5



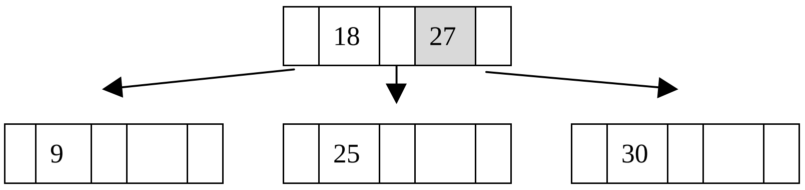
delete 12



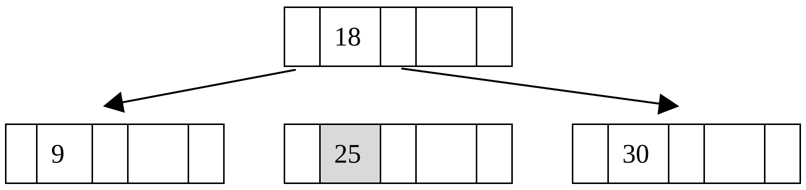
delete 35



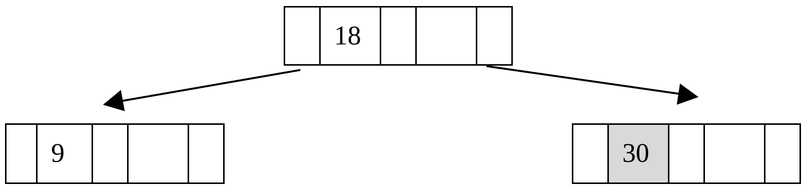
delete 27



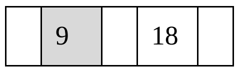
delete 25



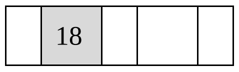
delete 30



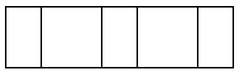
delete 9



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. You will need four main routines:

function	description

search_tree	search a B-Tree the find node to insert key into
search_node	search a node to insert key in the correct position
insert_node	insert key into node
split_node	when a node is filled it needs to be split in two and a new node formed from 1/2 of the original node.
delete_node	delete key from node
print_tree	print out nodes in binary tree

We have provided B-Tree code for you. we have examined many b-tree algorithms this is the best one we have found. There are two data structures a BTNODE and a ENTRY:

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

B-Tree code:

Here the B-Tree header file:

```
/* btree. h */

#define TRUE 1
#define FALSE 0

/* entry includes key and optional data */
typedef struct

      {
      int key;
      }ENTRY;
```

```
/* btree node */
```

```
typedef struct btree_node

      {
      int numentries; /* number of data elements in node */
      ENTRY *entry; /* keys */
      struct btree_node **link; /* links to other nodes */
      }BTNODE;

/* function prototypes */

ENTRY *make_entry(int key); /* make entry */
int search_node(int key, BTNODE *root, int *pos); /* search node for key */
void insert_node  /* insert entry into node */
(ENTRY split_entry, BTNODE *split_right,BTNODE *current, int pos);
/* insert entry into b-tree */
BTNODE* insert_btree(ENTRY newentry, BTNODE *current, int order); search_btree /*
search b-tree for entry */
(ENTRY entry,BTNODE *current,ENTRY *split_entry,BTNODE **split_right,int order);
void split_node  /* split node */
(BTNODE *current, ENTRY split_entry,BTNODE *split_right, int pos,
BTNODE**newnode,ENTRY *newmid, int order);
void print_btree(BTNODE *root,int level);  /* print b-tree */
```

Here the B-Tree implementation file:

```
/* bteee.c */

#include<stdio.h>
#include<stdlib.h>
#include "btree.h"

/* test driver for btree.c */
void main()

      {
      BTNODE *root=NULL;
      int order = 2;

      /*insert entry into b_tree*/
      root = insert_btree(*make_entry(18), root,order);
      printf("\nb tree: \n");   /*print b_tree*/
      print_btree(root,0);

      root = insert_btree(*make_entry(9), root,order);
      printf("\nb tree: \n");   /*print b_tree*/
      print_btree(root,0);

      root = insert_btree(*make_entry(30), root,order);
      printf("\nb tree: \n");   /*print b_tree*/
      print_btree(root,0);

      root = insert_btree(*make_entry(25), root,order);
      printf("\nb tree: \n");   /*print b_tree*/
      print_btree(root,0);

      root = insert_btree(*make_entry(27), root,order);
      printf("\nb tree: \n");   /*print b_tree*/
      print_btree(root,0);
      root = insert_btree(*make_entry(35), root,order);
      printf("\nb tree: \n");   /*print b_tree*/
      print_btree(root,0);

      root = insert_btree(*make_entry(12), root,order);
      printf("\nb tree: \n");   /*print b_tree*/
      print_btree(root,0);

      root = insert_btree(*make_entry(5), root,order);
      printf("\nb tree: \n");   /*print b_tree*/
      print_btree(root,0);
      }

/* B-Tree functions */

/* make an entry from key */
ENTRY *make_entry(int key)

       {
       ENTRY *entry = (ENTRY *)malloc(sizeof(ENTRY));
       entry->key = key;
       return entry;
       }

/* insert entry into B-tree */
BTNODE* insert_btree(ENTRY newentry, BTNODE *root,int order)

      {
      ENTRY   split_entry;
      BTNODE  *split_right,*newnode;

      /* search tree to place entry */
      if(search_btree(newentry, root, &split_entry, &split_right,order))

            {
            /* allocate memory for node and node entries and links */
            newnode = (BTNODE *)calloc(1,sizeof(BTNODE));
            newnode->entry = (ENTRY *)calloc(order+1,sizeof(ENTRY));
            newnode->link = (BTNODE**)calloc(order+1,sizeof(BTNODE*));
            newnode->entry[1] = split_entry;
            newnode->link[0] = root;
            newnode->link[1] = split_right;
            newnode->numentries = 1;
            return newnode;
            }

      return root;
      }
```

```
/* Search tree to find where to insert new entry */
/* returns true if found otherwise false */
int search_btree(ENTRY newentry, BTNODE *current,ENTRY *split_entry,
BTNODE **split_right, int order)

      {
      int pos, pos_value;
      /* insertion point found */
      if (!current)

            {
            *split_entry = newentry;
            *split_right = NULL;
            return TRUE;
            }

      else

            {
            /* do not insert duplicate keys */
            if (search_node(newentry.key, current,&pos))

                   {
                   printf("cannot insert duplicate key");
                   return FALSE;
                   }

      pos_value = search_btree
      (newentry, current->link[pos],split_entry,split_right,order);

      /* insert info into this node */
      if(pos_value)

            {
             /* check if node is filled */
            if(current->numentries < order)

                  {
                  insert_node(*split_entry,*split_right,current, pos);
                  return FALSE;
                  }

            /* node is filled split node */
            else

                  {
                  split_node(current,*split_entry,*split_right,pos,
                  split_right,split_entry,order);
                  return TRUE;
                  }

             }
            return FALSE;
            }

      }
```

```
/* Search node for key  return true if found */
int search_node(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 insert_node
(ENTRY split_entry, BTNODE *split_right,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] = split_entry;
      current->link[pos+1] = split_right;
      current->numentries++;
      return;
      }
```

```
/* split current node into two nodes */
void split_node (BTNODE *current, ENTRY split_entry,BTNODE *split_right,
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 = (BTNODE*) calloc(1,sizeof(BTNODE));
      (*newnode)->entry = (ENTRY *)calloc(order+1,sizeof(ENTRY));
      (*newnode)->link = (BTNODE**)calloc(order+1,sizeof(BTNODE*));

      /* 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) insert_node(split_entry, split_right,current, pos);
      else insert_node(split_entry,split_right, *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 print_btree(BTNODE *root,int level)

      {
      int i,j;
      if (!root)return;

      else

            {
            level++;
            for(i=0; i<=root->numentries; i++)
                 print_btree(root->link[i],level);
            printf("level %d: " , level);
            for(j=1; j<=root->numentries; j++)
                printf("%d ", root->entry[j].key);
            printf("\n");
            }

      }
```

progam 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.

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

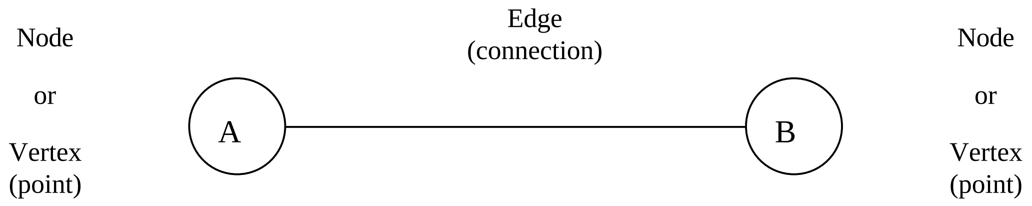
Now that you know how the insert routines work. write the function to delete keys in the B-Tree.

C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 15

File:	CguideL15.doc
Date Started:	April 15,1999
Last Update:	Dec 22, 2001
Status:	draft

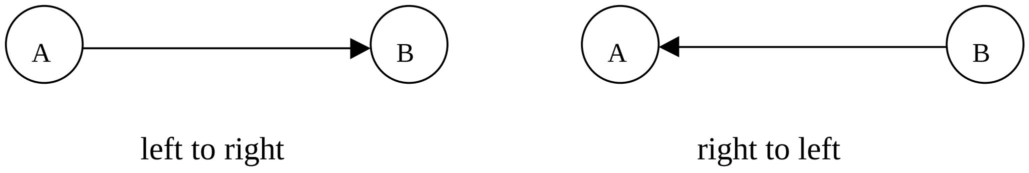
LESSON 15 CONSTRUCTING GRAPHS

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**.

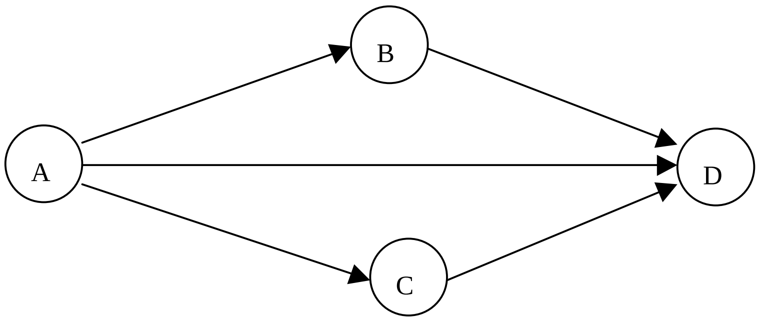


The above graph is said to be **undirected** meaning the direction of travel can be left to right or right to left. Graphs can also be **directed** either forcing direction from left to right or from right to left.

Directed Graphs



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.

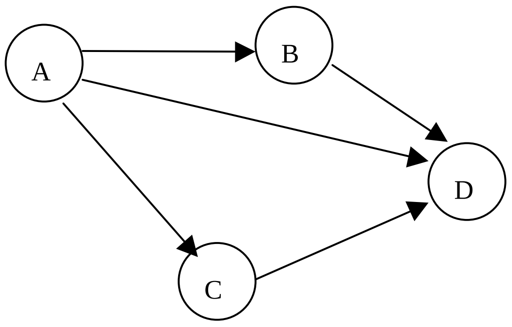
term	description
graph	connection of points
vertex or node	points on a graph, each vertex has a name
edge	connection between vertices
adjacent vertices	two vertices connected together by an edge vertex A connected by an edge to vertex B
path	sequence of connecting edges from one vertex to another
visit 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.

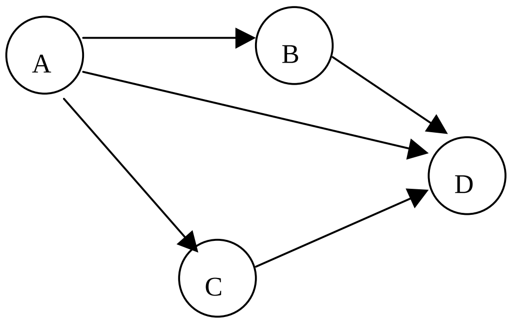


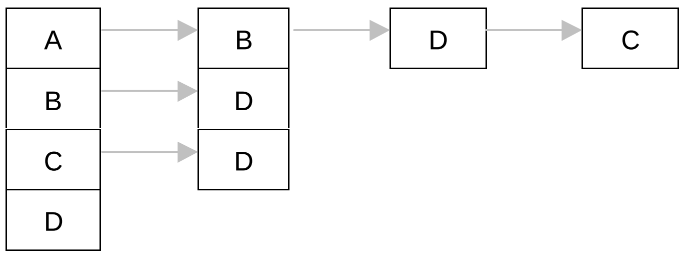
	A	B	C	D
A	0	1	1	1
B	0	0	0	1
C	0	0	0	1
D	0	0	0	0

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 modular programming techniques. By using good modular 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 functions 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 vertexes. etc. The function s 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 a character or string name, rather than a numeric value. We would need the following functions:

operation	description
initialization	initialize graph to size
add a vertex	add vector to graph
add an edge	connect to vertices v1 and v2 together
find a vertex	search for vertex in graph
remove a vertex	remove vertex from graph
remove an edge	remove edge from graph
get edge	get edge weight for specified vertices
print a graph	print graph

GRAPH ADT USING ADJANCEY MATRIX

We first need a structure called Graph to hold all the information about the graph module. The most obvious thing we need is a pointer to 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. Notice we have typedef's for vertices, edges and weight. Making typedef's is smart because you have the freedom to change the data type any time you want. After the structure definition we list all the function prototypes belonging to the graph module using an adjacency matrix. If your compiler does define true, false and bool you can use the following "defs.h" file.

```
/* defs.h */

#define TRUE 1
#define FALSE 0
#define ERROR -1
typedef int bool;

/* graph.h */

#include "defs.h"
typedef char Vertex;
typedef int Edge;
typedef int Weight;

typedef struct
      {
      Weight **m;  /* pointer to array rows of columns */
      int s;       /* max size of  graph */
      Vertex *v;   /* array of vertices names */
      int n;       /* number of vertices in graph */
      } Graph;
```

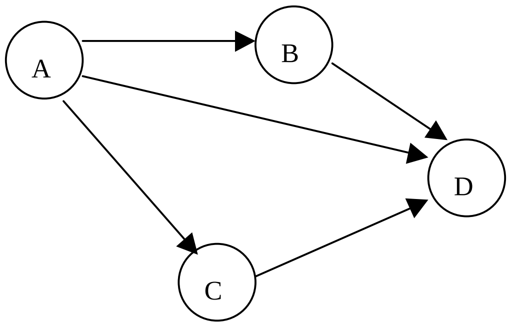
```
/* function prototypes */

/* initialize graph */
void init_graph(Graph* g, int size);
/* deallocate memory for graph module */
void destroy_graph(Graph* g);
/* add a vertex to a graph */
bool add_vertex(Graph* g, Vertex v);
/* add an edge between two vertices */
bool add_edge(Graph* g, Vertex v1, Vertex v2,Weight w);
/* find a vertex in a graph */
int find_vertex(Graph* g, Vertex v);
/* get weight of an edge between two vertices */
Edge get_edge(Graph* h,Vertex v1, Vertex v2);
/* remove vertex from graph */
bool remove_vertex(Graph* g, Vertex v);
/* remove an edge between two vertices */
bool remove_edge(Graph* g, Vertex v1, Vertex v2);
/* print out a graph */
void print_graph(Graph* g);
```

Here's the code to implement a graph using an adjacency matrix. We allocate memory with calloc rather tan malloc because we want the memory cleared once it is allocated.

```
/* graph.c */

#include "graph.h"

/* allocate memory for a graph */
void init_graph(Graph* g, int size)

        {
        int i;

        /* allocated memory for a 2 dimensional array */
        g->m = (Weight**)calloc(size,sizeof(Weight*));

        for(i=0;i<size;i++)g->m[i] = calloc(size,sizeof(Weight));

        g->v = (Vertex*)calloc(size,sizeof(Vertex)); /* names of vertices */
        g->s = size;   /* size of matrices */
        g->n = 0;  /* number of vertices in graph */
        }

/* deallocate memory for graph */
void destroy_graph(Graph* g)

        {
        int i;
        for(i=0;i<g->s;i++)free (g->m[i]);
        free(g->m);
        free(g->v);
        }
        /* add vertex to graph */
        bool add_vertex(Graph* g, Vertex v)

                {
                if(find_vertex(g,v)>=0)return FALSE;

                g->v[g->n] = v;
                g->n++;
                return TRUE;
                }

        /* add edge to a graph */
        bool add_edge(Graph* g, Vertex v1, Vertex v2, Weight w)

                {
                int i = find_vertex(g,v1);
                int j = find_vertex(g,v2);

                if(i > g->n || j > g->n)return FALSE;

                g->m[i][j] = w;
                return TRUE;
                }

        /* find a vertex in a graph */
        int find_vertex(Graph* g, Vertex v)

                {
                int i;

                for(i=0;i<g->n;i++)if(v == g->v[i])return i;
                return ERROR;
                }

/* get an edge weight from a graph */
Edge get_edge(Graph* g,Vertex v1, Vertex v2)

        {
        int i = find_vertex(g,v1);
        int j = find_vertex(g,v2);
        if(i < 0 ||j < 0)return 0;
        return g->m[i][j];
        }

 /* remove edge between to vertices */
bool remove_edge(Graph* g, Vertex v1, Vertex v2)

        {
        int i = find_vertex(g,v1);
        int j = find_vertex(g,v2);
        if(i < 0 || j  < 0)return FALSE;
        g->m[i][j] = 0;
        return TRUE;
        }
```

```
/* print out a graph */
void print_graph(Graph* g)

        {
        int i,j;

        printf("  ");
        /* print out vertices names */
        for(i=0;i<g->n;i++) printf("%c ",g->v[i]); printf("\n");

        for(i=0;i<g->n;i++)

                {
                printf("%c ",g->v[i]);
                /* print out edge weights */
                for(j=0;j<g->n;j++)printf("%d ",g->m[i][j]); printf("\n");
                }

        printf("\n");
        }

        /*  test driver for adjacency matrix graph */
        void main()

                {
                Graph g;   /* declare a graph structure */
                init_graph(&g,4); /* initialize graph structure */
                add_vertex(&g,'a'); /* add vertices */
                add_vertex(&g,'b');
                add_vertex(&g,'c');
                add_vertex(&g,'d');
                add_edge(&g,'a','b',1); /* add edges */
                add_edge(&g,'a','c',1);
                add_edge(&g,'a','d',1);
                add_edge(&g,'b','d',1);
                add_edge(&g,'c','d',1);
                print_graph(&g); /* print out graph */
                }
```

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 main. Use the print graph function to print them out. Write the remove vertex function. If you remove a vertex in one row all occurrences need to be needs to be removed in all rows. Call your project L15ex1 and main file L15ex1.c

GRAPH ADT USING ADJANCY LIST

We need to uses the link list module with the graph module. Each node in the link list module will have a vertex name and a weight and a pointer to the next node.

vertex name	edge weight	next adjacent vertex

We have to modify the link list module . We will rename our node GNodes and the link list module GList. The basic functions of the list module do not change. Only the insert functions because we now the list nodes have names and weights. The structure for the GList module needs an additional member the vertex name. Here's the header file for the GList module:

```
/* glist.h */
#include "defs.h"

#ifndef __GLIST
#define __GLIST
typedef struct GNodeType GNode;
typedef struct GNodeType* GNodePtr;
typedef int Vertex;
typedef int Weight;

/* structure for graph node */
struct GNodeType

        {
        GNodePtr next;
        int v;
        Weight w;
        };
```

The list module structure retains the head and tail pointers but we also needs to add all the things we need for the graph. like vertex name.

```
/* glist module structure */
typedef struct

        {
        GNodePtr head;
        GNodePtr tail;
        char v;
        }GList,*GListPtr;
```

```
/* function prototypes for single link list */
/* initialize link list */
void initGList(GListPtr list);
/* insert item into list */
int insertGList(GListPtr list,Vertex data, Weight w);
/* insert item into list at head */
int insertGHead(GListPtr list,Vertex data, Weight w);
/* insert item into list at tail */
int insertGTail(GListPtr list,Vertex data, Weight w);
/* remove data item from list */
bool removeGList(GListPtr list,Vertex data);
/* find data item in list */
GNodePtr findGList(GListPtr list, Vertex data);
/* print list */
void printGList(GListPtr list);
/* de allocate memory for list */
void destroyGList(GListPtr list);
#endif
```

Here's the code for the link list module used for the graph module

```
 /* glist.c */
/* single link list code implementation */
#include <stdio.h>
#include <stdlib.h>
#include "glist.h"

/* initialize list */
void initGList(GListPtr list)

          {
          list->head = NULL;
          list->tail = NULL;
          }

// deallocate memory per node in list
void destroyGList(GListPtr list)

         {
         GNodePtr next;
         GNodePtr node = list->head; /* point to start of list */

         while(node != NULL)

                  {
                  next = node->next;
                  free(node);
                  node = next;
                  }

         list->head = NULL;
         list->tail = NULL;
         }

/* insert node at end of list */
```

```
int insertGTail(GListPtr list,Vertex v,Weight w)

        {
        /* make new node */
        GNodePtr node = (GNodePtr)malloc(sizeof(GNode));

        if (node == NULL)return FALSE;

        node->v = v;
        node->w = w;
        node->next= NULL;

        /* check for empty list */
        if(list->head == NULL)

                {
                list->head = node;
                list->tail = list->head;
                }

        else

                {
                (list->tail)->next=node;
                list->tail=node;
                }

        return TRUE;
        }


/* find v element in list */
/* if found return position in list, if not found return null */
GNodePtr findGList(GListPtr list, Vertex v)

        {
        GNodePtr curr = list->head;
         while(curr != NULL)

                {
                if(v == curr->v)return curr;
                curr=curr->next;
                }

        return NULL;
        }
```

Once we have our glist module. We first need a structure called GraphL to hold all the information about the graph list module. 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 graph header file:

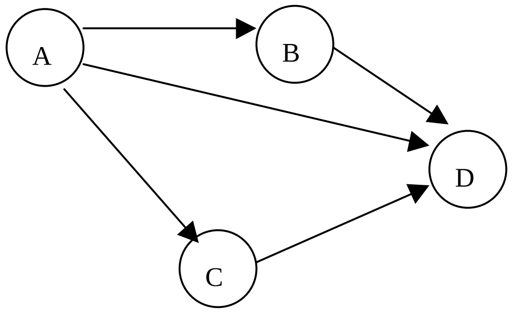
```
/* graphL.h. */
```

```
#ifndef __GRAPHL
#define __GRAPHL
#include "defs.h"
#include "glist.h"

/* structure to hold information about graph using adjacency lists */
typedef struct

        {
        GList *h; /* pointer to array rows of nodes */
        int s;      /* max size of  graph */
        int n;      /* number of vertices in graph */
        } GraphL;

/* function prototypes for graph list module  */

/* initialize graph */
void init_graph(GraphL* g, int size);
/* deallocate memory for graph module */
void destroy_graph(GraphL* g);
/* add a vertex to a graph */
bool add_vertex(GraphL* g, Vertex v);
/* add edge */
bool add_edge(GraphL* g, Vertex v1, Vertex v2,Weight w);
/* find a vertex in a graph */
int find_vertex(GraphL* g, Vertex v);
/* remove vertex from a graph */
bool remove_vertex(GraphL* g, Vertex v);
/* remove an edge between two vertices */
bool remove_edge(GraphL* g, Vertex v1, Vertex v2);
/* get weight of an edge between two vertices */
Edge get_edge(GraphL* h,Vertex v1, Vertex v2);
void print_graph(GraphL* g);  /* print out a graph */
#endif
```

Here's the code for the graph module using an adjacency list:

```
/* graphl.c */
#include <stdio.h>
#include <stdlib.h>
#include "defs.h"
#include "graphl.h"
#include "glist.h"

/* allocate memory for a graph */
void init_graph(GraphL *g, int size)

        {   /* allocated memory for an array of GList's */
        g->h = (GList*)calloc(size,sizeof(GList));
        g->s = size;   /* size of graph list */
        g->n = 0;  /* number of vertices in graph */
        }

/* deallocate memory for graph */
```

```
void destroy_graph(GraphL *g)

        {
        int i;
        /* de-allocate memory for each list */
        for(i=0;i<g->n;i++)destroyGList(&(g->h[i]));
        free (g->h);
        }

/* add vertex to graph */
bool add_vertex(GraphL* g, Vertex v)

        {
        int i = find_vertex(g,v);
        if(i >= 0)return FALSE;
        if(g->n>=g->s)return FALSE;
        g->h[g->n].v = v;
        g->n++;
        return TRUE;
        }

/* add edge to a graph */
bool add_edge(GraphL* g, Vertex v1, Vertex v2, Weight w)

        {
        int i = find_vertex(g,v1);
        int j = find_vertex(g,v2);
        if(i > g->n || j > g->n)return FALSE;
        insertGTail(&(g->h[i]),v2,w);
        return TRUE;
        }

/* find a vertex in a graph */
int find_vertex(GraphL* g, Vertex v)

        {
        int i;
        for(i=0;i<g->n;i++)if(v == (g->h[i]).v)return i;
        return ERROR;
        }

/* print out a graph */
void print_graph(GraphL* g)

        {
        int i;
        GNode* n;
        printf("\n");
/* loop for all vertices */
for(i=0;i<g->n;i++)

            {
            printf("%c: ",g->h[i].v);
            n = g->h[i].head;
            /* loop till end of list */
            while(n != NULL)

                  {
                  printf("%c ",n->v);
                  n = n->next;
                  }

            printf("\n");
            }

      printf("\n");
      }

/* get an edge weight from a graph */
Edge get_edge(GraphL* g,Vertex v1, Vertex v2)

      {
      int i;
      GNode* n;

      if(v1 > g->s || v2 > g->s)return 0;

      i = find_vertex(g,v1);
      if(i < 0)return 0;
      n  = findGList(&(g->h[i]),v2);

      if(n!= NULL)return n->w;
      else return 0;
      }

/* driver to test graph using adjacency list */
void main()

      {
      GraphL g;   /* declare a graph structure */
      init_graph(&g,4); /* initialize graph structure */
      add_vertex(&g,'a'); /* add vertices */
      add_vertex(&g,'b');
      add_vertex(&g,'c');
      add_vertex(&g,'d');
      add_edge(&g,'a','b',1); /* add edges */
      add_edge(&g,'a','c',1);
      add_edge(&g,'a','d',1);
      add_edge(&g,'b','d',1);
      add_edge(&g,'c','d',1);
      print_graph(&g); /* print out graph */
      }
```



Program Output:

```
a: b c d
b: d
c: d
d:
```

LESSON 15 EXERCISE 2

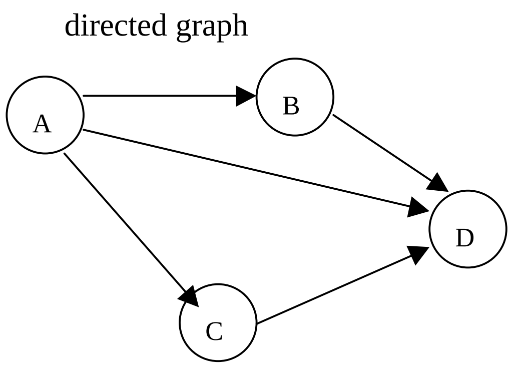
Type in the graph list module and get it going.. Write the remove edge and remove vertex functions. If you remove a vertex in one list it needs to be removed in all lists. Call your project L15ex2 and main file L15ex2.c

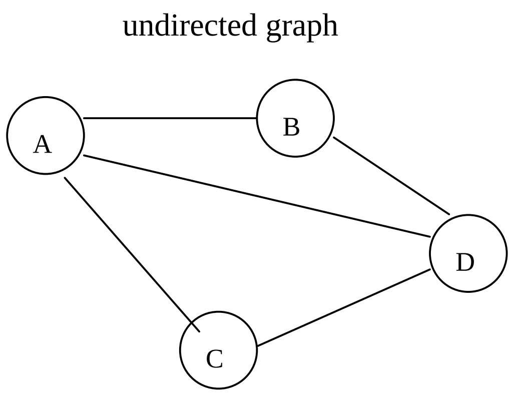
C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 16

File:	CdsGuideL16.doc
Date Started:	April 15,1999
Last Update:	Dec 22, 2001
Status:	draft

LESSON 16 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 adjacent vertex, either from left to right or from right to left. Undirected graph assume edges between adjacent vertices can travel in either direction.





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 list's 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. We also have an 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				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 a module called galgo. Each algorithm's will be a function. By using a separate module for the algorithm we can constructs graphs using adjacency matrix or adjacency list. There is no data structure associated with the galgo module because the graph module and the graph list module has all their own structures. It is very important the algorithms are independent of data structures. The algorithms will call functions from the graph or graph list modules. The galgo algorithms are not concerned if a graph is represented an adjacency matrix or adjacency list. You just need to include the graph.h module for adjacency matrix or the graphl.h module for adjacency list from last lesson. We have two supporting functions **print_paths()** and **print_path(**). Print_paths() will print all paths for each vertex in a graph, where as print_path() will just print a path associated from a start vertex. Here are our graph algorithm functions.

function	description
depth first search	find shortest path by searching by depth
breath first search	find shortest path by searching by breadth
shortest path	
un-weighted 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

Here is the header file for the graph.h algorithm functions. You choose which graph to use by including and defining the appropriate header for adjacency matrix or adjacency list

```
/* galgo.h */
#ifndef __GALGO
#define __GALGO

#include "graphl.h"              /* include to use adjacency list */
/* #include "graph.h" */     /* include to usde adjacency matrix */

#include "list.h"

/* #define GraphL Graph  */    /* define if you want to use adjacency matrix */

#define MAX_DIST 1000

void dfs(GraphL* h,Vertex v, List *list); /* traverse depth first search */
void bfs(GraphL* g,Vertex v, List *list);  /* traverse breath first search */
bool shortest_path(GraphL* h,Vertex v); /* find shortest path */
bool dijkstra(GraphL* g, Vertex v);  /* dikstras algorithm for weighted graphs */
int  longest_path(GraphL* g, Vertex v); /* find longest path */
void topsort(GraphL* g,Vertex v,List *list); /* topological sort */
void print_paths(GraphL* g); /* print paths */
void print_path(GraphL* g,Vertex v);  /* print path */
#endif
```

MODULES

Hare are all the modules we are using:

module	header file	implementation file	description
Galgo	galgo.h	galgo.c	graph algorithms
Graph	graph.h	graph.c	graph using adjacency matrix
GraphL	graphl.h	graphl.c	graph using adjacency list
GList	glist.h	glist.c	graph link list
List	list.h	glist.h	link list

You need to add additional variables to the graph and graph modules from previous lesson. The additional functions will be explained as we use them in the graph algorithms. Here is the new header file for the adjacency matrix module:

```
/* graph.h */
#ifndef __GRAPH
#define __GRAPH
#include "defs.h"
#include "list.h"
```

```
typedef char Vertex;
typedef int Edge;
typedef int Weight;

typedef struct
{
Weight **m;  /* pointer to array rows of columns */
int s;          /* max size of  graph */
Vertex *v;   /* array of vertices names */
int n;       /* number of vertices in graph */
int *vs;     /* visited nodes */
int *d;      /* number of in degrees */
int *p;      /* print array */
int itr;     /* iterator */
} Graph;

/* function prototypes */
/* initialize graph */
void init_graph(Graph* g, int size);
/* deallocate memory for graph module */
void destroy_graph(Graph* g);
/* add a vertex to a graph */
bool add_vertex(Graph* g, Vertex v);
/* add an edge between two vertices */
bool add_edge(Graph* g, Vertex v1, Vertex v2,Weight w);
/* set edge to a graph */
bool set_edge(Graph* g, Vertex v1, Vertex v2, Weight w);
/* find a vertex in a graph */
int find_vertex(Graph* g, Vertex v);
/* get weight of an edge between two vertices */
Edge get_edge(Graph* h,Vertex v1, Vertex v2);
/* remove vertex from graph */
bool remove_vertex(Graph* g, Vertex v);
/* remove an edge between two vertices */
bool remove_edge(Graph* g, Vertex v1, Vertex v2);
/* print out a graph */
void print_graph(Graph* g);
/* get next adjacent vertex not in list */
Vertex next_vertex_list(Graph *g,Vertex v,List *list);
/* initialize distance and path tables */
bool init_dist(Graph *g,Vertex v,int max);
/* get distance for vertex */
int get_dist(Graph *g,Vertex v);
/* set distance for vertex */
bool set_dist(Graph *g,Vertex v,int d);/
/* get weight between current adjacent vertices */
Weight get_weight(Graph *g,Vertex v);
/* set a pointer to first adjacent vertex */
bool iterate(Graph *g,Vertex v);
/* get vertex at iteration pointer */
Vertex get_vertex(Graph *g, Vertex v);
```

```
/* increment iteration pointer, return vertex at iteration pointer */
Vertex next_vertex(Graph *g, Vertex v);
/* weight of edge between  current adjacent vertices */
Weight get_weight(Graph *g,Vertex v);
/* print out paths */
void print_paths(Graph* g);
/* print out path */
void print_path(Graph* g,Vertex v);
/* get vertex at graph index */
Vertex vertex_at(Graph *g,int i);
/* get path at graph index */
Vertex path_at(Graph *g,int i);
/* set path */
bool set_path(Graph *g,Vertex w,Vertex v);
/* clear visited */
void clear_visited(Graph* g);
/* set vertex as being visited */
bool set_visited(Graph* g,Vertex v);
/* check if vertex is visited */
bool is_visited(Graph* g,Vertex v);
/* find minimum unknown distance */
Vertex min_xdist(Graph *g);
#endif
```

Here is the new header file for the adjacency list module.

```
/* graphL.h*/
#ifndef __GRAPHL
#define __GRAPHL
#include "defs.h"
#include "glist.h"
#include "list.h"

/* structure to hold information about graph using adjacency lists */
typedef struct
{
GList *h; /* pointer to array rows of nodes */
int s;      /* max size of  graph */
int n;      /* number of vertices in graph */
char *v;    /* vertex names */
int *vs;    /* visited nodes */
int *d;     /* distance */
int *p;     /* print array */
} GraphL;

/* function prototypes for graph list module  */
/* initialize graph */
void init_graph(GraphL* g, int size);
/* deallocate memory for graph module */
void destroy_graph(GraphL* g);
/* add a vertex to a graph */
bool add_vertex(GraphL* g, Vertex v);
```

```
/* add edge */
bool add_edge(GraphL* g, Vertex v1, Vertex v2,Weight w);
/* set existing edge */
bool set_edge(GraphL* g, Vertex v1, Vertex v2,Weight w);
/* find a vertex in a graph */
int find_vertex(GraphL* g, Vertex v);
/* remove vertex from a graph */
bool remove_vertex(GraphL* g, Vertex v);
/* remove an edge between two vertices */
bool remove_edge(GraphL* g, Vertex v1, Vertex v2);
/* get weight of an edge between two vertices */
Edge get_edge(GraphL* h,Vertex v1, Vertex v2);
/* print out a graph */
void print_graph(GraphL* g);
/* get next adjacent vertex not in list */
Vertex next_vertex_list(GraphL *g,Vertex v,List *list);
/* initialize distance and path tables */
bool init_dist(GraphL *g,Vertex v,int max);
/* get distance for vertex */
int get_dist(GraphL *g,Vertex v);
/* set distance for vertex */
bool set_dist(GraphL *g,Vertex v,int d);
/* get weight between current adjacent vertices */
Weight get_weight(GraphL *g,Vertex v);
/* set a pointer to first adjacent vertex */
bool iterate(GraphL *g,Vertex v);
/* get vertex at iteration pointer */
Vertex get_vertex(GraphL *g, Vertex v);
/* increment iteration pointer, return vertex at iteration pointer */
Vertex next_vertex(GraphL *g, Vertex v);
/* weight of edge between  current adjacent vertices */
Weight get_weight(GraphL *g,Vertex v);
/* print out paths */
void print_paths(GraphL *g);
/* print out path */
void print_path(GraphL *g,Vertex v);
/* get vertex at graph index */
Vertex vertex_at(GraphL *g,int i);
/* get path at graph index */
Vertex path_at(GraphL *g,int i);
/* set path */
bool set_path(GraphL *g,Vertex w,Vertex v);
/* clear all visited vertices */
void clear_visited(GraphL* g);
/* set vertex as being visited */
bool set_visited(GraphL* g,Vertex v);
/* check if vertex is visited */
bool is_visited(GraphL* g,Vertex v);
/* get minimum unknown distance */
Vertex min_xdist(GraphL *g);
#endif
```

You need to add an Iterator variable to the GList module.

```
typedef struct
{
GNodePtr head;  /* point to start of glidt*/
GNodePtr tail;  /* point to end of glist */
char v;     /* vertex that owns list */
GNodePtr itr;  /* GNode iterartor */
}GList,*GListPtr;
```

GRAPH ALGORITHMS

We will now discus the operation of each graph algorithm separately.

GRAPH TRAVERSALS

A graph traversal visits every vertex from a start vertex to 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,

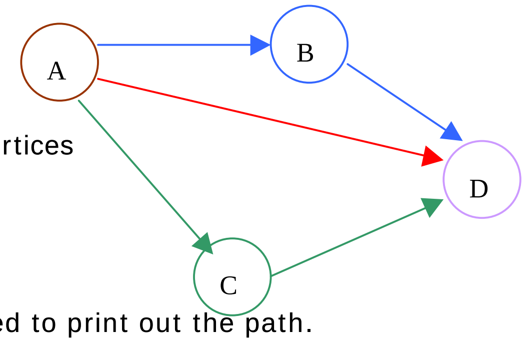
Threes are two types of traversals:

- depth first search
- breath first search

DEPTH FIRST SEARCH

In depth first search we follow the path as far as possible, we then backtrack to search other path choice. 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



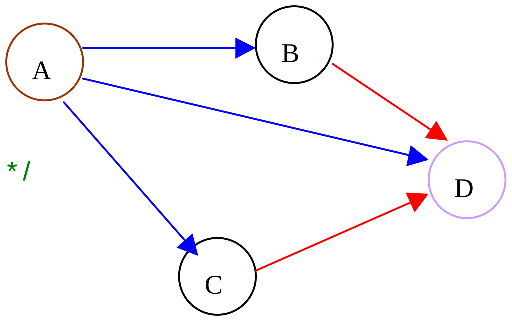
All nodes visited are kept in a list. The list is also used to print out the path.

Here's the code for depth first algorithm:

```
/* depth first search */
void dfs(GraphL* g,Vertex v, List *list)

        {
        Vertex v2;
        insertTail(list,v);   /* insert vertex in list  */
        v2 = next_vertex_list(g,v,list);  /* get next adjacent vertex not in list */
        /* loop till no more available adjacent vertices */
        while(v2 != ' ')

                {
                dfs(g,v2,list);  /* call dfs to search path by depth */
                v2 = next_vertex_list(g,v,list); /* get next adjacent vertex not in list */
                }

        }
```

BREADTH FIRST SEARCH

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. Here's the code for the breadth first algorithm:



```
        /* breath first search */
        void bfs(GraphL* g,Vertex v, List *list)

                {
                Vertex v2;
                QUEUE_LIST q;  /* make queue */
                initQue(&q);   /* initialize queue */
                enqueue(&q,v);   /* insert vertex into queue */
                insertTail(list,v); /* insert vertex into list */

                /* loop while queue not empty */
                while(!isEmptyQue(&q))

                        {
                        v = dequeue(&q);   /* get vertex from queue */
                        v2 = next_vertex_list(g,v,list); /* get next adjacent vertex not in list */

                        /* loop till no more available adjacent vertices */
                        while(v2 != ' ')

                                {
                                enqueue(&q,v2);   /* get vertex from queue */
                                insertTail(list,v2); /* insert vertex into list */
                                v2 = next_vertex_list(g,v,list);  /* get next adjacent vertex not in list */
                                 }

                        }

                destroyQue(&q); /* deallocate memory for queue */
                }
```

The dfs and bsf algorithms use the **next_vertex_list ()** function. This function gets the next adjacent vertex not in the list. These functions need to be added to the graph and graph list modules. Each module function will be different. The graph algorithms functions must be **data structure independent**. All functions 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 the **next_verex_list()** function for the adjacency matrix. You will need add it to the graph module.

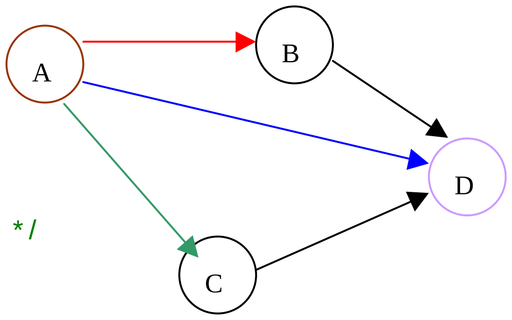
```
/* get next adjacent vertex not in list */
Vertex next_vertex_list(Graph *g,Vertex v,List *list)

        {
        int j;
        int i=find_vertex(g,v);  /* get vertex index */

        if(i < 0)return ERROR;  /* exit if vertex not in graph */

        /* loop through all adjacent vertices */
        for(j=0;j<g->n;j++)

                {
                if( list,g->m[i][j] != 0)   /* check if edge */
                /* return vertex if not in list */
                if(findList(list,g->v[j])==NULL)return g->v[j];
                }

        return ' ';  /* no more vertices */
        }
```

Here's the code for the **next_verex_list()** function for the adjacency list. You will need to add it to the graph list module.

```
/* get next adjacent vertex from v not in list */
Vertex next_vertex_list(GraphL *g,Vertex v,List *list)

        {
        GNode* n;

        int i=find_vertex(g,v);  /* find vertex in graph */
        if(i < 0)return ERROR;  /* exit if vertex not in graph */
        n = g->h[i].head;  /* point to start of list */

        /* loop through all adjacent vertices */
         while(n != NULL)

                {
                if(n->w != 0)  /* check if edge */
                if(findList(list,n->v)==NULL)return n->v; /* check if vertex in list */
                n = n->next;   /* get next vertex in list */
                }

        return ' ';  /* no more vertices in 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 having 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.



```
/* shortest path */
bool shortest_path(GraphL* g,Vertex v)

        {
        Vertex w;
        QUEUE_LIST q;  /* create queue */
        initQue(&q);  /* initialize queue */

        /* initialize distance table  with max distance */
        if(!init_dist(g,v,MAX_DIST))return FALSE;

        enqueue(&q,v);   /* put vertex in queue */

        /* loop while queue not empty */
         while(!isEmptyQue(&q))

                {
                v = dequeue(&q);  /* get vertex from queue */
                 iterate(g,v);     /* get adjacent vertices from this vertex */
                 w = get_vertex(g,v);  /* get fist adjacent vertex */

                /* loop till no more vertices in list  */
                while(w != ' ')

                        {
                        /* get distance for this vertex */
                        if(get_dist(g,w)==MAX_DIST)

                        {
                        set_dist(g,w,get_dist(g,v) + 1); /* set w distance to v vertex + 1 */
                        set_path(g,w,v); /* set path w to v */
                        enqueue(&q,w);  /* put w vertex in queue */
                        }

                w = next_vertex(g,v); /* get next vertex in adjacency list */
                }

        } /* end while */

destroyQue(&q);    /* deallocate memory for queue */
return TRUE;
}
```

The shortest path algorithm also needs functions to initialize the distance table and to , set and get distances from the table. The functions are both the same for the adjacency matrix and the adjacency list.

```
        /* initialize distance and path tables */
        bool init_dist(Graph *g,Vertex v,int max)

                {
                int i;

                /* loop for all distances */
                for(i=0;i<g->n;i++)

                        {
                        g->d[i] = max;  /* set to max */
                        g->p[i] = ' '; /* set to empty vertex */
                        }

                i = find_vertex(g,v);
                if(i < 0)return FALSE;
                g->d[i]= 0;

                return TRUE;
                }

        /* get distance for vertex */
        int get_dist(Graph *g,Vertex v)

                {
                int i = find_vertex(g,v);
                if(i < 0)return  0; /* no distance */
                return g->d[i];  /* return distance */
                }

        /* set distance for vertex */
        bool set_dist(Graph *g,Vertex v,int d)

                {
                int i = find_vertex(g,v);
                if(i < 0)return ERROR;
                g->d[i]= d;
                return TRUE;
                }
```

We also need functions to set the path and print the path. The functions are both the same for the adjacency matrix and the adjacency list.

```
/* set path to this vertex */
bool set_path(Graph *g,Vertex w,Vertex v)

                {
                int i;
                i = find_vertex(g,w);
                if(i < 0)return ERROR;
                g->h[i].p = v;
                return TRUE;
                }
```

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 **print_path()** function will print out the shortest paths from the start vertex to the specified end vertex.

```
/* print out path */
void print_paths(Graph* g)

    {
    int i;

    for(i=0;i<g->n;i++)

        {
        printf("%c: ",vertex_at(g,i));
        print_path(g,path_at(g,i));
        printf("\n");
        }

    }

/* print out path */
void print_path(GraphL* g,Vertex v)

    {
    int i = find_vertex(g,v);
    if(i >= 0) print_path(g,path_at(g,i));
    printf("%c ",v);
    }
```

We need to get the path using the **path_at()** function and vertex name at a certain index **vertex_at()** function. The functions are both the same for the adjacency matrix and the adjacency list.

```
/* get vertex at graph index */
Vertex vertex_at(GraphL *g,int i)
{
return g->v[i];
}

/* get path at graph index */
Vertex path_at(GraphL *g,int i)
{
return g->p[i];
}
```

The **iterate()** function sets a pointer to the start of an adjacency list and **get_vertex()** gets the vertex at the iteration pointer. The **next_vertex()** function increments the pointer and gets the next vertex in the adjacency list. We need separate get_vertex() and next-vetex() functions because we need to get additional data at the iteration point like the weight of a edge. By using these supporting functions the galgo module is data structure independent. This means it can be used for graphs implemented adjacency matrix is graph implement adjacency lists. These functions are different for adjacency matrix and adjacency list.

Here are the functions for the adjacency matrix.

```
/* set a pointer to first adjacent vertex */
bool iterate(Graph *g,Vertex v)

        {
        int i = find_vertex(g,v);  /* get vertex position */
        if(i < 0)return FALSE;   /* vertex not found */
        g->itr = 0;  /* set pointer to first vertex */
        return TRUE;
        }

/* get vertex at iteration pointer */
Vertex get_vertex(Graph *g, Vertex v)

        {
        int i = find_vertex(g,v);
        if(i < 0)return ' ';    /* no vertex */
        if(g->itr >= g->n)return ' ';  /* no vertex */
        return g->v[g->itr];  /* return vertex */
        }

/* increment iteration pointer, return vertex at iteration pointer */
Vertex next_vertex(Graph *g, Vertex v)

        {
        int i = find_vertex(g,v); /* find vertex position */
        if(i < 0)return ' ';  /* no vertex */
        g->itr++;   /* increment iteration pointer */
        while(g->itr<g->n)

                {
                if(g->m[i][g->itr])return g->v[g->itr];
                g->itr++; /* increment iteration pointer */
                }

        return ' ';  /* no more vertices */
        }
```

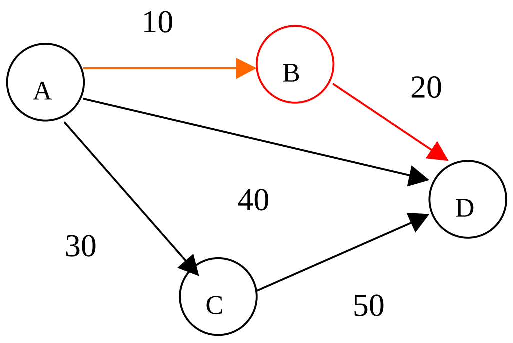
Here are the functions for the adjacency list.

```
/* set a pointer to first adjacent vertex */
bool iterate(GraphL *g,Vertex v)
{
int i = find_vertex(g,v);
if(i < 0)return FALSE;
g->h[i].itr = g->h[i].head;
return TRUE;
}
```

```
/* get vertex at iteration pointer */
Vertex get_vertex(GraphL *g, Vertex v)
{
int i = find_vertex(g,v);
if(i < 0)return ' ';
if(g->h[i].itr == NULL)return ' ';
return g->h[i].itr->v;
}

/* increment iteration pointer, return vertex at iteration pointer */
Vertex next_vertex(GraphL *g, Vertex v)
{
int i = find_vertex(g,v);
if(i < 0)return ' ';
g->h[i].itr = g->h[i].itr->next;
if (g->h[i].itr == NULL)return ' ';
return g->h[i].itr->v;
}
```

SHORTEST PATH WEIGHTED (Dikstra's algorithm)

When each edge has a weight then we need an algorithm to fin the path with the minimum weight.



Dikstra'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. Here's the code for dikstras algorithm:

```
/* dikstras shortest path algorithm for weighted graph's */|
bool dijkstra(GraphL* g, Vertex v)
{
Vertex w;

 /* set distance table to maximum distance */
if(!init_dist(g,v,MAX_DIST))return FALSE;
 clear_visited(g);    /* clear visited array */
/* loop till all vertices visited */
 while(TRUE)

          {
         v = min_xdist(g);  /* get minimum unknown vertex distance */
          if(v == ' ')return TRUE;  /* no vertices available */
          set_visited(g,v);  /* set vertex visited */
          iterate(g,v);  /* iterate on this vertex */
          w = get_vertex(g,v);   /* get first vertex in adjacent vertex list */

          /* loop till end of adjacent vertex  */
         while(w != ' ')

                   {
                   /* if w not visited */
                  if(!is_visited(g,w))

                            {
                             /* check costs */
                             if(get_dist(g,v) + get_weight(g,v) < get_dist(g,w))

                                      {
                                      /* update distance table to minimum accumulative distance */
                                      set_dist(g,w,get_dist(g,v) + get_weight(g,v));
                                      set_path(g,w,v);  /* store path for print out */
                                      }

                            }

                  w = next_vertex(g,v); /* get next vertex */
                  } /* end while w */

         } /* end while true */

} /* end dikstras */
```

The only new functions are the get Weight() function. it retrieves the weight on the edge between two vertices at the iteration point. These functions are both different for the adjacency matrix and the adjacency list.

```
/* get weight between current adjacent vertices */
Weight get_weight(Graph *g,Vertex v)

         {
         int i = find_vertex(g,v);
         if(i < 0)return 0;
         return g->m[i][g->itr];
         }
```

```
/* get weight between current adjacent vertices */
Weight get_weight(GraphL *g,Vertex v)

	{
	int i = find_vertex(g,v);
	if(i < 0)return ERROR;
	return g->h[i].itr->w;
	}
```

The dikstra's algorithm uses additional functions from the graph and graph list modules. We have an array of visited vertices and functions to clear and set the visited vertices.

```
/* clear all visited vertices */
void clear_visited(Graph* g)

	{
	int i;
	for(i=0;i<g->n;i++)g->vs[i] = 0;   /* set all not visited */
	}

/* set vertex as being visited */
bool set_visited(Graph* g,Vertex v)

	{
	int i = find_vertex(g, v); /* find vertex position */
	if(i >= 0)

		{
		g->vs[i] = 1;  /* set vertex visited */
		return TRUE;  /* success */
		}

	else return FALSE;  /* vertex not in graph */
	}

/* check if vertex is visited */
bool is_visited(Graph* g,Vertex v)

	{
	int i = find_vertex(g, v);   /* find vertex position */
	if(i >= 0) return g->vs[i] != 0;   /* check if vertex visited */
	else return FALSE;  /* vertex not in graph */
	}
```

The dikstras function also needs a function to find the minimum unknown distance in the distance table.

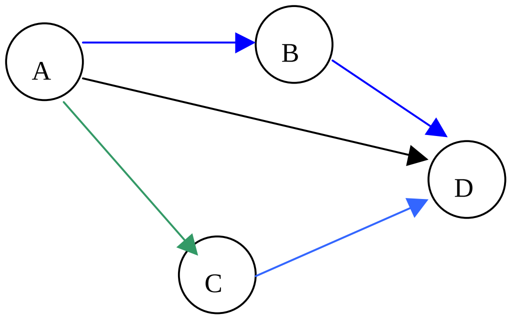
```
/* find minimum unknown distance */
Vertex min_xdist(Graph *g)

        {
        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<g->n;i++)if(g->d[i] > mind)mind = g->d[i];

        /* loop through table */
        for(i=0;i<g->n;i++)

                {
                /* check if visited */
                if(g->vs[i] == 0)



                        {
                        /* keep track of minimum distance and exit */
                        if(g->d[i] < mind)

                        {
                        mini = i;  /* store minimum index */
                        mind = g->d[i];  /* store minimum value */
                        }

                }


        }

if(mini < 0)return ' ';   /* no minimum distance found */|
return g->v[mini];  /* return vertex having minimum distance */
}
```

You need to add one more function to the graph and graghl modules that allows you to set the edge to a new value.

```
/* set edge to a graph */
bool set_edge(Graph *g, Vertex v1, Vertex v2, Weight w)

{
int i = find_vertex(g,v1);
int j = find_vertex(g,v2);
if(i < 0 || j < 0)return FALSE;
g->m[i][j] = w;
return TRUE;
}
```

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 the choices of the path and chooses the longest and increments a counter.. recursing it counts from the end of the paths to the beginning blue is the choice for the longest path length



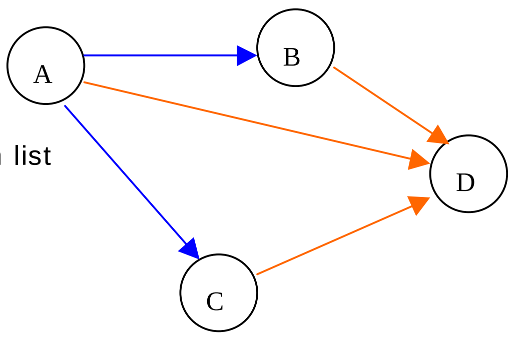
Here's is the code for the longest path algorithm:

```
/* find longest path */
int longest_path(GraphL* g , Vertex v)

    {
    int k = 0; /* check if a path found */
    Vertex w; /* adjacent vertex */
    int d; /* distance */
    int s = 0; /* path counter  */

    set_visited(g,v); /* set all vertices unvisited */
    iterate(g,v); /* set iteration to start of v */
    w = get_vertex(g,v); /* get first adjacent vertex */

    /* loop for all vertices w adjacent to v */
    while(w != ' ')

        {
        /* check if visited  */
        if(is_visited(g,w)==FALSE)

            {
            k++;
            d = longest_path(g,w); /* get preceding longest length of path */
            if(d > s)s = d; /* store longest path */
            }

        w = next_vertex(g,v); /* get next adjacent vertex w */
        }

    clear_visited(g);

    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:

```
/* topological sort */
void topsort(GraphL* g,Vertex v, List *list)

    {
    Vertex v2 = next_vertex_list(g,v,list); /* get next vertex not in list */

    /* loop till all vertices visited */
    while(v2 != ' ')

        {
        topsort(g,v2,list);
        v2 = next_vertex_list(g,v,list); /* get next vertex not in list */
        }

    insertHead(list,v); /* insert vertex in list */
    }
```

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.

```
/* galgo.c */

#include <stdio.h>
#include "list.h";
#include "defs.h"
#include "quelist.h"
#include "galgo.h"

#define MAX_DIST 1000
```

```
void main()

        {
        GraphL g;
        List list;
        int dist;

        /* make graph */
        init_graph(&g,4);
        add_vertex(&g,'a');
        add_vertex(&g,'b');
        add_vertex(&g,'c');
        add_vertex(&g,'d');
        add_edge(&g,'a','b',1);
        add_edge(&g,'a','c',1);
        add_edge(&g,'a','d',1);
        add_edge(&g,'b','d',1);
        add_edge(&g,'c','d',1);
        print_graph(&g);

        printf("testing graph algorithms :\n");

        /* dfs */
        initList(&list);
        dfs(&g,'a',&list);
        printf("depth first search: ");
        printList(&list);
        destroyList(&list);

        /* bfs */
        initList(&list);
        bfs(&g,'a',&list);
        printf("breath first search: ");
        printList(&list);
        destroyList(&list);

        /* shortest path */
        clear_visited(&g);
        shortest_path(&g,'a');
        printf("shortest path:\n");
        print_paths(&g);

        /* dikdtras */
        printf(" adding weights to edges \n");
        set_edge(&g,'a','b',10);
        set_edge(&g,'a','c',30);
        set_edge(&g,'a','d',40);
        set_edge(&g,'b','d',20);
        set_edge(&g,'c','d',50);
        print_graph(&g);
        dijkstra(&g,'a');
        printf("dijkstra's path:\n");
        print_paths(&g);
        clear_visited(&g);
        dist = longest_path(&g,'a');
        printf("longest path: %d\n", dist);

        /* top sort */
        initList(&list);
        topsort(&g,'a',&list);
        printf("topological sort: ");
        printList(&list);
        destroyList(&list);
        destroy_graph(&g);
        }
```

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

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

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

dijkstra's path:

a:
b:   a
c:   a
d:   a b

longest path: 2

topological sort: list: a c b d
```

LESSON 16 EXERCISE 1

Type in or copy the galgo module. You will need to add the functions to the graph list and graph list modules. the functions will be different for each module. we have only shown examples for the graph module not the graph list module. you will have to add the distance (d) and visited (vs) and path (p) arrays and iteration pointer (itr) to each module. for the graph list module the arrays can be part of each graph list header node or just part of the graph module. the graph module used an adjacency matrices where the graph list module uses a adjacency list. both modules cannot be present in a program because all functions have the same name. you need to write the functions for the graph list module.

LESSON 16 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 16 EXERCISE 3

Add a print table to the longest path algorithm so that we can print out the longest paths.

LESSON 16 EXERCISE 4

Test your algorithms from known graphs solutions you find in text books. See if you get the same answer as them.

C DATA STRUCTURES PROGRAMMERS GUIDE LESSON 18

File:	CguideL18.doc
Date Started:	April 15,1999
Last Update:	Dec 22, 2001
Status:	draft

LESSON 18 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 there 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 8 program 2 */
#include <stdio.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 */
        printf("%d",num);   /* print out random number */
        num = (rand() % 100) + 1; /* random number between 1 and 100 */
        printf("%d",num);   /* print out random number */
        fnum = rand()/MAX_RAND; /* random number between 0 and 1.0 */
        printf("%f",fnum);   /* print out random number */
        fnum = rand()/(MAX_RAND-1); /* random number between 0 and 0.9999  */
        printf("%f",fnum);   /* print out random number */
        }
```

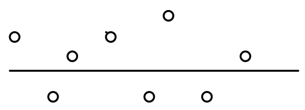
MEAN VARIANCE AND STANDARD DEVIATION

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

$$\boxed{\text{mean} = \frac{x1 + x2 + x3 + ... + xn}{n}}$$

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.

$$\boxed{\text{variance} = \sum_{n}^{i=0} \left[(x\,i - \text{mean})^{2} \right]}$$



The **standard deviation** (SD)is simply the square root of the variance.

LESSON 18 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

$y = 1 - \exp(-rx)$

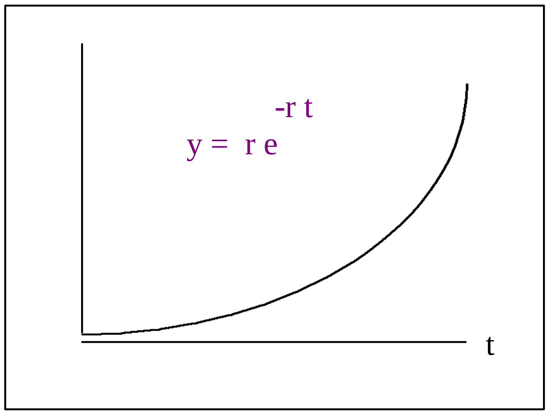
take ln of both sides:

$\ln y = \ln 1 + rx \qquad \ln(y-1) = -rx$

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))

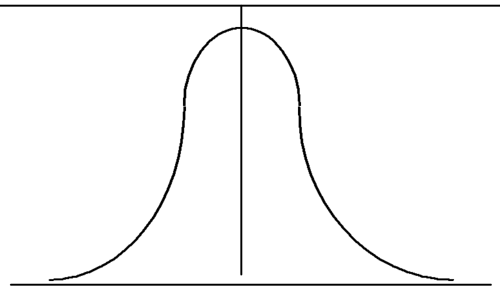
y =   1 / (b - a) where a <= x <= b

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:

```
              1                  -(x - mean)squared /2 *SD squared
y =   ---------------------- e
      SD sqrt(2 * PI )
```



We have to approximate with the following algorithm. Figure out how it works.

```
  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;
```

Random

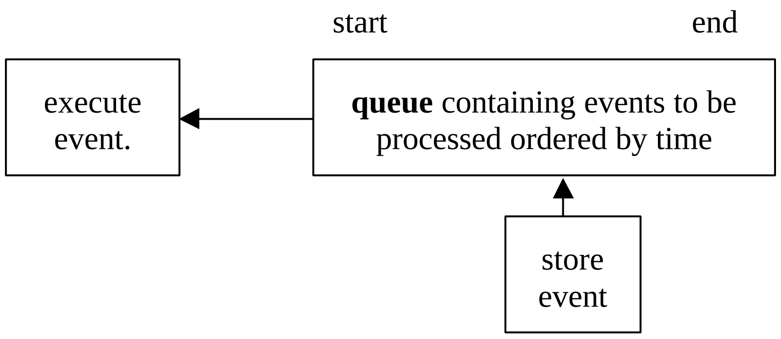
A random distribution is just random numbers with no concentration.

LESSON 18 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

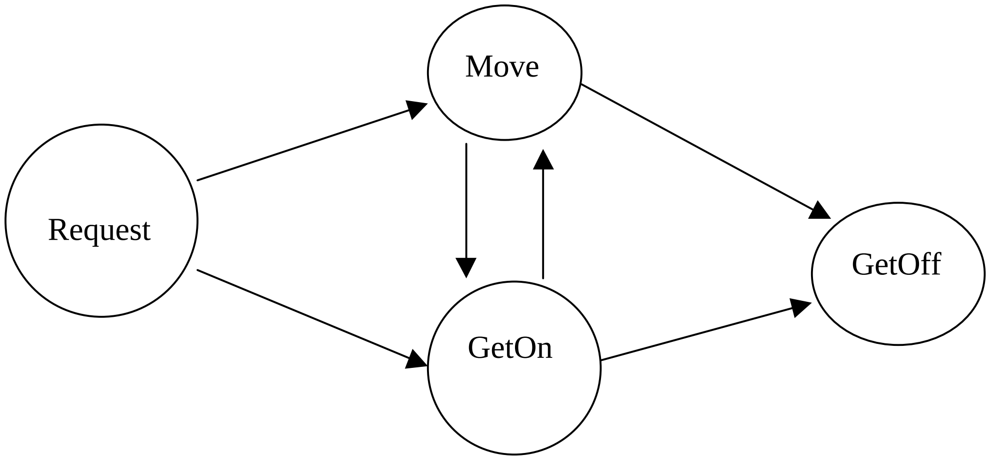
What events do our elevators have ?

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. 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

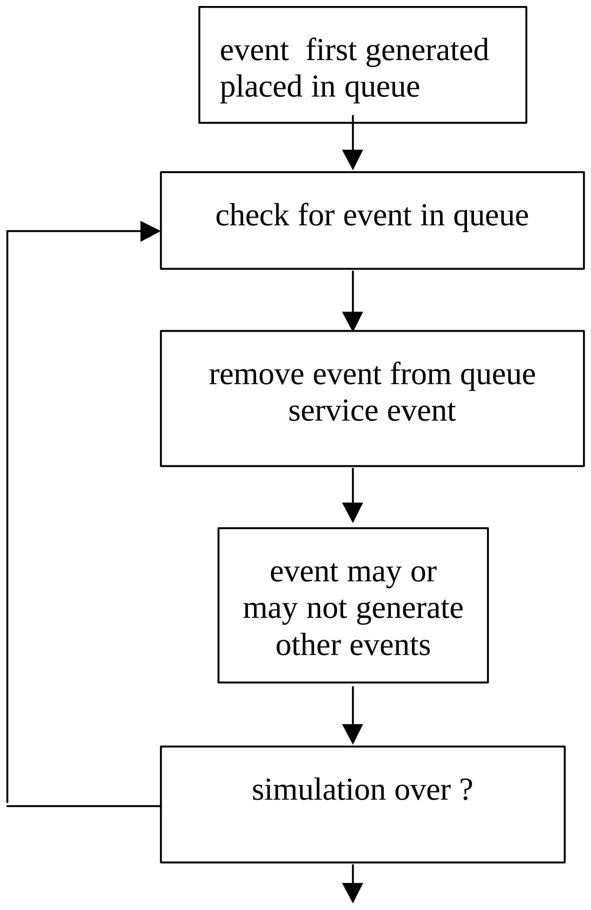
STATE DIAGRAM

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.


event first generated placed in queue
check for event in queue
remove event from queue service event
event may or may not generate other events
simulation over ?

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	printf(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.

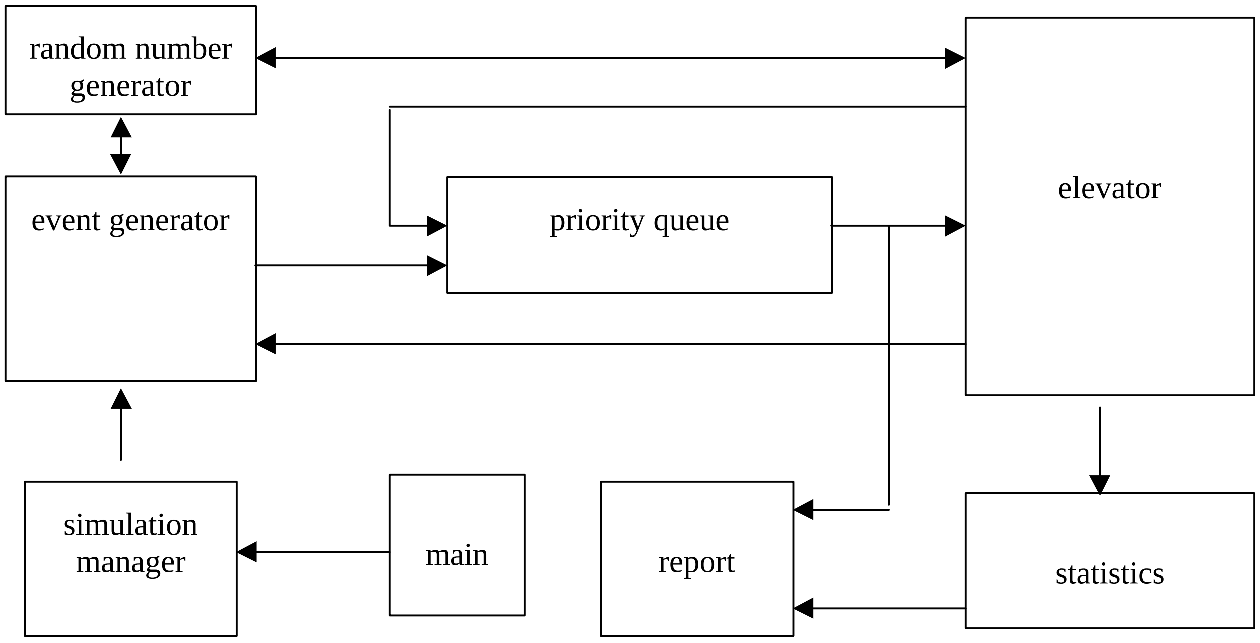
```
/* lesson 17 program 1 */

#include <stdio.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 */
    printf("the starting time is %s:",ctime(start);

    for(i=0;i<100000;i++);  /* long delay */

    end = time(NULL); /* get end time */
    printf("the ending time is %s:",ctime(end);

    diff = difftime(end,start) /* calculate difference in time */
    printf("The time for the calculation is: %d seconds",diff);

    tm = localtime(&time(NULL));

    printf("The time is: %s %s\n",asctime(tm));

    gettime(&tm);

    printf("The time is: %s %s\n",asctime(tm));
    }
```

MODULES

We need a simulation manager module to take care of executing events placed in the event queue and run the operations. We need an elevator module to simulate the elevator actions. We need a **request event generator** to generate requests. We need a **random generator module** to generate random distributions. We will also need a **statistics module** and **report module**.

module	purpose	header file	implementation
random number generator	generate a random number to a specified distribution	rgen.h	rgen.c
event generator	generates random events based on data and time	eventgen.h	eventgen.c
priority queue	stores events sorted in ascending order by time	pqueue.h	pqueue.c
elevator	elevator operations	elevator.h	elevator.c
statistics	keep track of day to day operation statistics	stats.h	stats.c
reports	give a detailed report from statistic data	reports.h	reports.c
simulation manager	control system operation	simmgr.h	simmgr.c
main	get things rolling		main.c

elevator simulation system model



MAIN FUNCTION

The main function will create the simulation manager, call the initialize simulation function, the call the run function with he simulation time, max floors, max patrons and exponential generator rate.

```
/* lesson 22 simulation */
#include "simulmgr.h"

void main()

        {
        SimulMgr sim;  /* create simulator manager */
        /*sim, maxTime,maxFloors,maxPatrons,double rate */
        initSM(&sim,10,5,10,0.5); /* initialize simulation manager */
        run(&sim);   /* run simulation */
        printf("\n simulation over\n");
        }
```

PRIORITY QUEUE MODULE

the priority queue module will store the time in ascending order. This meanst5he 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.h */
/* priority queue list module header file */

#ifndef __PQUEUE
#define __PQUEUE

/* priority queue module data structures */
#include "eventgen.h"
#include "defs.h"

/* priority queue list node */
typedef struct pqueue_node_type

        {
        Event* data;
        struct pqueue_node_type *next;
        }PQNode;

/* priority queue list module */
typedef struct

        {
         PQNode* head;
         PQNode* tail;
         int count;
        }PQueue;
```

```
/* priority queue module function prototypes */

/* initialize priority queue */
void initPQ(PQueue *que);

/* put item into priority queue */
bool enqueue(PQueue *que,Event* data);

/* check if queue is empty */
bool isEmpty(PQueue *que);

/* look at item at start of queue */
Event* peekTail(PQueue *que);

/* look at item at end of queue */
Event* peekHead(PQueue *que);

/* insert node into queue */
PQNode* insert(PQueue *que,PQNode *node,Event* data);

/* remove data item from priority queue */
Event* dequeue(PQueue *que);

/* delicate memory for priority queue */
void destroyPQ(PQueue *que);

/* print out contents of priority queue */
void destroyPQList(PQNode *node);

/* print out contents of priority queue */
void printQue(PQueue *que,char *name);

/* print out info about queue node */
void print(PQNode *node);
#endif

/* pqueue.c */

#include <stdio.h>
#include <stdlib.h>
#include "pqueue.h"
#include "defs.h"

/* initialize priority queue */
void initPQ(PQueue* que)

        {
        que->head = NULL;
        que->tail = NULL;
        que->count = 0;
        }
```

```
/* insert items at end of priority queue */
bool enqueue(PQueue* que,Event* data)

        {
        PQNode* node = insert(que,que->head,data);

        /* check for empty queue */
        if(que->head==NULL)

                {
                que->head=node;
                que->tail=node;
                }

         que->count++;
        return TRUE;
         }

 /* insert node into list recursively in ascending order */
PQNode* insert(PQueue* que,PQNode* node,Event* data)

        {
        /* find where to insert item */
        if((node == NULL) || (compare(data,node->data)< 0))

                {
                /* allocate memory for priority queue node */
                PQNode* tnode = (PQNode*)malloc(sizeof(PQNode));
                (Event*)tnode->data = data;
                (PQNode*)tnode->next = node;

                /* adjust head and tail pointers */
                if((que->head == NULL) || (node != NULL && node == que->head)) que-
                >head=tnode;
                if(tnode->next == NULL)que->tail=tnode; /* last */
                node = tnode;
                }

        else node->next = insert(que,node->next,data);
        return node;
        }

/* remove items from the head of queue */
Event* dequeue(PQueue* que)

        {
        Event* data;
        PQNode* next;

        /* check if priority queue empty */
        if(que->head == NULL)return NULL;
        data = (que->head)->data;
        /* remove from item at head of priority queue */
        next = (que->head)->next;
        if(next!=NULL)

                {
                que->head->next = NULL;
                free(que->head);  /* delete link node */
                }

        que->head = next;
        if(next == NULL)que->tail = NULL;  /* priority queue is empty */
        que->count--;
        return data; /* return data element */
        }

/* look at item at start of queue */
Event* peekTail(PQueue *que)

        {
        if(que->tail==NULL)return NULL;
        return que->tail->data;
        }

/* look at item at start of queue */
Event* peekHead(PQueue *que)

        {
        if(que->head==NULL)return NULL;
        return que->head->data;
        }

/* check if queue is empty */
bool isEmpty(PQueue *que)

        {
         return (que->count == 0);|
        }

/* delicate memory for priority queue driver */
void destroyPQ(PQueue* que)

        {
        /* delicate memory for vector */
        destroyPQList(que->head);
        que->head = NULL;
        que->tail = NULL;
        }
```

```
/* deallocate memory */
void destroyPQList(PQNode* node)

        {
        if(node != NULL)

                {
                /* delicate memory for priority queue item */
                PQNode* next = node->next;
                free (node);
                destroyPQList(next);
                }

        }

/* print out priority queue items driver */
void printQue(PQueue* que,char* name)

        {
        printf("%s[ ",name);
        print(que->head);
        printf(" ]\n");
        }


/* print out priority queue items */
void print(PQNode* node)

        {
        if(node != NULL)

                {
                printf("%d ",node->data);
                print(node->next);
                }

        }
```

LESSON 18 EXERCISE 3

Write a small test program. Type in or copy and paste the priority queue module. Make an event structure to handle a **time_t** and a sequential event number. In a loop generate random numbers to represents time as random events to put these times into the priority que. Try using the different time distributions. When the event is ready to be executed remove from the queue. Print out the results on the screen when the events were executed.

C PROGRAMMERS GUIDE LESSON 19

File:	CdsGuideL19.doc
Date Started:	April 15,1999
Last Update:	Sept 2,1999
Status:	draft

LESSON 19 SIMULATION TECHNIQUES II

SIMULATION MANAGER MODULE

The simulation manager module will be responsible for ruining the elevator simulation operation. The **initialization()** function will create the elevator, random number generator, request event generator modules. The **run()** function has the main time driven event loop that checks if there are any events in the priority queue. If there are it checks if the event is less than or equal to the current time. If it is the event is removed from the priority queue and executed. We are using real time we use the functions from time.h to get the real time and compare the real-time. The main time driven event loop uses a switch statement where each case is an Event id. For every event id there is a event function. **request()**, **getOn()**, **move()** and **getOff()**. An alternative is to store the address of each function has a function pointer in the event structure. Since we have few evens the switch statement is not too cumbersome.

```
/* simulmgr.h */
#ifndef __SIMULMGR
#define __SIMULMGR

#include "pqueue.h"
#include "eventgen.h"
#include "elevator.h"
#include "randgen.h"

/* exit floors */
typedef enum {PARKING,GROUND}ExitFloors;

/* simulation manager variables */
typedef struct
{
PQueue *pq; /* pointer to priority queue */
Event *event; /* current event being executed */
EventGen *evtgen; /* pointer toevent generator */
Elevator *elev; /* pointer to elevator */
RandGen *rgen; /* pointer to random number generator */
int maxTime; /* maximum simulation time in seconds */
time_t startTime; /* simulation start time */
time_t time; /* current time */
time_t endTime; /* simulation end time */
}SimulMgr;

/* initialize simulation manager */
```

```
void initSM(SimulMgr *sim,double maxTime,int maxFloors,int maxPatrons,double rate);

/* run simulation */
void run(SimulMgr *sm);

/* request an elevator */
void request(SimulMgr *sm);

/* get on an elevator */
void getOn(SimulMgr *sm);

/* get off an elevator */
void getOff(SimulMgr *sm);

/* move an elevator */
void move(Event* event,Elevator* elev,PQueue *pq,time_t time);
#endif

/* simulmgr.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "simulmgr.h"
#include "pqueue.h"
#include "elevator.h"
#include "eventgen.h"
#include <dos.h>

/* initialize simulation manager */
void initSM(SimulMgr *sim,double maxTime,int maxFloors,int maxPatrons,double rate)

        {
        /* store start and end simulation times */
        sim->maxTime = maxTime;
        sim->startTime = time(NULL);
        sim->endTime = sim->startTime + maxTime;

        /* allocate memory for modules */
        sim->pq = (PQueue*)malloc(sizeof (PQueue));
        sim->evtgen = (EventGen*)malloc(sizeof (EventGen));
        sim->elev = (Elevator*)malloc(sizeof(Elevator));
        sim->rgen  = (RandGen *)malloc(sizeof(RandGen));

        /* initialize modules */
        initPQ(sim->pq);
        initEvtGen(sim->evtgen,GROUND);
        initElev(sim->elev,maxFloors,maxPatrons);
        initRandGen(sim->rgen,0,maxFloors,rate);
        }
```

```
/* run simulation */
void run(SimulMgr * sim)
{
 /* generate request event depending on time of day  */
genEvent(sim->evtgen,REQUEST,sim->rgen,sim->pq);
sim->time = time(NULL);  /* get time */

 /* loop for simulation time */
while(sim->time < sim->endTime)

        {
        sim->event = peekHead(sim->pq);  /* check event in queue */
         /* get event if queue not empty */
        if(!isEmpty(sim->pq))

                {
                 /* check if time to service this event */
                if(difftime(sim->event->time,sim->time)<=0)

                        {
                        dequeue(sim->pq); /* remove event from queue */

                        /* service event */
                        switch(sim->event->id)

                                {
                                case REQUEST: request(sim); break; /* request elevator */
                                case GETON: getOn(sim); break;  /* get on a elevator */
                                 /* move elevator */
                                case MOVE: move(sim->event,sim->elev,sim->pq,sim->time);
                                break;
                                case GETOFF: getOff(sim); break;  /* get off elevaor */
                                default:

                                        {
                                        printf("\n event not known to this simulation\n");
                                        printf("please check your number and call
                                        again.thankyou\n");
                                        return;
                                        }

                                } /* end switch */

                        } /* end if time */

                } /* end if empty */

        /* generate request event depending on time of day  */
        genEvent(sim->evtgen,REQUEST,sim->rgen,sim->pq);
        sim->time = time(NULL);  /* get time */
        } /* end while */

} /* end run */
```

```
/* process events */

/* request an elevator */
void request(SimulMgr *sim)

        {
        Event* event;
        int on;
        printf("request floor %d from  %d %s\n",sim->event->to,sim->event-
        >from,ctime(&sim->time));

        /* put event in elevator on queue */
        setElevOn(sim->elev,sim->event);
        on =  checkOn(sim->elev); /* check if anyone can get on this floor */

        /* only generate 1 geton event per floor  */
        if(on == 1 && event->from == sim->elev->floor)

                {
                event=copy(sim->event,GETON);
                enqueue(sim->pq,event);
                }

        /* 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 == STOPPED && on == 0)

                {
                if(sim->elev->dir == STOPPED)sim->elev->dir = REQUESTED;
                event=copy(sim->event,MOVE);
                enqueue(sim->pq,event);
                }

        }

        /* check to move this elevator */
        void move(Event* event,Elevator* elev,PQueue *pq,time_t t)

                {
                bool on,off;
                printf("at floor %d %s\n",elev->floor,ctime(&t));
                event->time = time(NULL);
                off = checkOff(elev);  /* check if anyone can get off this floor */
                on = checkOn(elev);   /* check if anyone can get on this floor */

                /* any body wants to get on or off this elevator ? */
                if(on | off)

                        {
                        if(off)

                                {
                                event->id = GETOFF;   /* generate get off event */
                                enqueue(pq,event);
                                }

                        if(on)

                                {
                                if(off)event = copy(event,GETON);   /* generate get on event */
                                event->id = GETON;
                                enqueue(pq,event);
                                }

                        }

        /* move elevator to next floor */
         else

                 {
                moveTo(elev,event);
                enqueue(pq,event);
                 }

        }

/* get on elevator */
void getOn(SimulMgr *sim)

        {
        sim->elev->patrons++; /* count people getting on elevator */
        printf("person get on floor %d %s\n",sim->elev->floor,ctime(&sim->time));

        removeOn(sim->elev,sim->event);  /* take out of elevator on queue */

        /* check if all passengers get on */
        if(checkOn(sim->elev)==0)

                {
                if(sim->elev->dir == STOPPED)sim->elev->dir = REQUESTED;
                sim->event->id = MOVE; /* generate move event */
                sim->event->time = time(NULL) + (long)gaussian(2,5);
                enqueue(sim->pq,sim->event);
                }

        }

/* get off an elevator */
void getOff(SimulMgr *sim)

        {
        printf("%d people get off floor %d %s\n",checkOff(sim->elev),
        sim->elev->floor,ctime(&sim->time));

        removeOff(sim->elev); /* take out of elevator off queue */



        /* check if any more requests */
        if(moreUp(sim->elev)||moreDown(sim->elev))
```

```
                {
                sim->event->id = MOVE;  /* generate move event */
                sim->event->time = time(NULL);
                enqueue(sim->pq,sim->event);
                }

        /* no more requests */
        else

            {
            /* delete this even if ground or parking */
            if(sim->elev->floor == GROUND || sim->elev->floor == PARKING)

                {
                free(sim->event);
                 return;
                }

            }

        /* more floors to service */
        if(!(sim->elev->floor == GROUND || sim->elev->floor == PARKING))

            {
            /* generate an request event when they leave */
            sim->event->id = REQUEST;
            sim->event->from = sim->elev->floor;
            sim->event->to = uniform(sim->rgen);
            if(sim->event->to > sim->event->from)sim->event->dir = UP;
            else sim->event->dir = DOWN;
            sim->event->time= time(NULL) + exponential(sim->rgen);
            enqueue(sim->pq,sim->event);
            }

        }
```

random number generator module

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 module */
        #include <stdlib.h>
        #include <time.h>
        #ifndef __RANDGEN
        #define __RANDGEN
```

```
 // random number generator module structure
typedef struct

        {
        int a;   /* uniform a */
        int b;   /* uniform b */
        double rate; /* exponential rate */
        }RandGen;

/* initialize random number generator */
void initRandGen(RandGen *rand,int a,int b,double rate);
/* generate gaussian distribution random number */
int gaussian(int min,int max);
/* generate exponential distribution random number */
long exponential(RandGen *rand);
/* generate uniform distribution random number */
double uniform(RandGen *rand);

#endif

/* randgen.c */
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include "randgen.h"

/* initialize random number distribution generator */
void initRandGen(RandGen* rgen,int a, int b, double rate)

        {
        rand((unsigned) time(NULL));  /* seed random number generator */
        rgen->a = a;   /* store a min uniform */
        rgen->b = b;   /* store b max uniform */
        rgen->rate = rate;  /* store exponential rate */
        }

/* gaussian distribution */
int 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 exponential(RandGen* rgen)

                {
                double d = (double)rand()/(double)RAND_MAX;
                d = (1.0 - d);
                d = d/rgen->rate;
                d = -log(d);
                d * 100;
                return (long)d;
                }

/* uniform distribution */
/* (a + (b - a)  *  (random() / (RANDOM_MAX - 1.0)) */
double uniform(RandGen * rgen)

        {
        double d =  rgen->a + (rgen->b - rgen->a) * (rand() / (RAND_MAX - 1.0));
        return d;
        }
```

REQUEST EVENT GENERATOR MODULE

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 going down and going up event. At the end of the day the request event generator will generate requests going down. At the night the event generator will generate 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 . This module also handles the event structure and functions for comparing and copying events.

```
        /* eventgen.h */

        #include "randgen.h"
        #include "pqueue.h"

        #ifndef __EVENTGEN
        #define __EVENTGEN




        typedef enum {STOPPED,REQUESTED,UP,DOWN}Dir;
```

```
typedef enum {REQUEST,GETON,GETOFF,MOVE}Events;

/* event */
typedef struct

        {
        Events id; /* event id */
        long time; /* event time */
        int to; /* to floor */
        int from; /* from floor */
        Dir dir; /* direction */
        int num;  /* event number for tracking */
        } Event;

/* event generator */
typedef struct

        {
        int num;  /* current event number */
        int startFloor;  /* starting floor */
        }EventGen;

/* initialize event generator */
void initEvtGen(EventGen * evtgen,int startFloor);

/* copy an existing event */
Event *copy(Event  *event,Events id);

/* generate an request event */
int genEvent(EventGen * evtgen,Events id,RandGen* rgen,struct PQueue* pq);

/* compare if two events equal */
int compare(Event* evt1, Event* evt2);

#endif

/* eventgen.c */
#include <stdlib.h>
#include <time.h>
#include <dos.h>
#include "eventgen.h"
#include "simulmgr.h"
#include "pqueue.h"
#include "mem.h"

/* initialize event generator */
void initEvtGen(EventGen *evtgen,int startFloor)

        {
        evtgen->num = 0; /* current event number */
        evtgen->startFloor=startFloor;
        }
```

```
/*generate event considering time of day */
int genEvent(EventGen *evtgen,Events id,RandGen *rgen,struct PQueue* pq)

        {
        tm t;  /* time in hours/min/seconds */
        time_t tim; /* time in seconds */
        Event *event; /* pointer to event */
        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 hour clock */

        /* morning */
        if(t.tm_hour >= 7 && t.tm_hour <= 9)

                {
                max = 10;
                min = 2;
                interval = 10;
                dir = UP;
                }

        /* lunch */
        else if(t.tm_hour >= 11 && t.tm_hour <= 13)

                {
                max = 5;
                min = 1;
                interval = 50;
                dir = DOWN;
                }

        /* evening */
        else if(t.tm_hour >= 16 && t.tm_hour <= 18)

                {
                max = 10;
                min = 5;
                interval = 10;
                dir = DOWN;
                }
        /* all other times */
        else

                {
                max = 2;
                min = 1;


        interval = 100;
        dir = STOPPED;
        }

tim = time(NULL); /* get current time */

/* generate first event (num means event number) */
if(!(evtgen->num == 0)||((tim % interval)== 0))return 0;

patrons = gaussian(min,max); /* get number of patrons */

/* loop for number of patrons  requesting elevator */
for(i=0;i<patrons;i++)

        {
        event=(Event*)malloc(sizeof(Event));
        event->id = id;
        event->time = time(NULL);

        /* if morning or after lunch go up from ground or parking */
        if(evtgen->num == 0 || dir == UP)

                {
                event->from = gaussian(PARKING,GROUND);
                event->to = uniform(rgen);
                }

        /* lunch time or evening go down */
        else if(dir == DOWN)

                {
                event->from = uniform(rgen);
                event->to = gaussian(PARKING,GROUND);
                }

         /* all other times completely random */
        else

                {
                 event->to = uniform(rgen);
                event->from = uniform(rgen);
                }
        /* set event direction from floors */
        if(event->to >=  event->from) event->dir = UP;
        else event->dir = DOWN;

        event->num = ++(evtgen->num); /* set event number */
        enqueue(pq,event); /* put event in queue */
        }

return patrons;
}

/* compare 2 time events */
int compare(Event* evt1, Event* evt2)

        {
        return difftime(evt1->time,evt2->time);
        }

/* copy an event */
Event *copy(Event  *event,Events id )

        {
        Event *event2 =(Event*)malloc(sizeof(Event));
        memcpy(event2,event,sizeof(Event));
        event2->id=id;
        return event2;
        }
```

ELEVATOR MODULE

The elevator module 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 module 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 for the direction the elevator is traveling. Example if the elevator is going up, we don't want to pickup people who have requested to go down.

```
/* elevator.h */
#ifndef __ELEVATOR
#define __ELEVATOR

#include "defs.h"
#include "eventgen.h"
#include "pqueue.h"

/* elevator status */
typedef enum {OPERATING,OUT_OF_SERVICE,ON_SERVICE}ElevStatus;
```

```
/*elevator module structure */
typedef struct

{
int capacity;     /* number of people allowed in elevator */
int patrons;      /* number of people in elevator */
int floor;        /* floor elevator is at */
PQueue *upon;     /* queue for people requesting to going up */
PQueue *downon;   /* queue for people requesting to going down */
PQueue *upoff;   /* queue for people getting off going up */
PQueue *downoff; /* queue for people getting off going down */
int status;       /* status of elevator */
Dir dir;          /* direction of elevator */
int maxFloors;    /* number of floors in elevator */
}Elevator;

/* initialize elevator */
void initElev(Elevator* elev,int maxFloors, int capacity);

/* move to next floor */
void moveTo(Elevator* elev,Event* event);

/* set on requests */
void setElevOn(Elevator *elev,Event* event);

/* set off requests */
void setElevOff(Elevator *elev,Event* event);

/* check if people can get on from this floor */
int checkOn(Elevator *elev);

/* check if people can get off to this floor */
int checkOff(Elevator *elev);

/* remove on requests from elevator */
void removeOn(Elevator* elev,Event* event);

/* remove off requests from elevator */
void removeOff(Elevator* elev);

/* check if elevator needs to move up */
bool moreUp(Elevator* elev);

/* check if elevator needs to move down */
bool moreDown(Elevator* elev);

#endif
```

```
/* elevator.c */
#include <stdio.h>
#include <stdlib.h>
#include "elevator.h"
#include "eventgen.h"
#include "simulmgr.h"
#include "pqueue.h"

/* initialize elevator */
void initElev(Elevator* elev,int maxFloors,int capacity)

         {
        int i;
        elev->floor = GROUND;
        elev->status = OPERATING;
        elev->dir = STOPPED;
        elev->maxFloors = maxFloors;
        elev->capacity=capacity;
        elev->patrons = 0;
        elev->upon = (PQueue*)calloc(maxFloors,sizeof(PQueue));
        elev->downon = (PQueue*)calloc(maxFloors,sizeof(PQueue));
        elev->upoff = (PQueue*)calloc(maxFloors,sizeof(PQueue));
        elev->downoff = (PQueue*)calloc(maxFloors,sizeof(PQueue));

        /* initialize all queues */
        for(i=0;i<maxFloors;i++)

                {
                initPQ(&elev->upon[i]);
                initPQ(&elev->downon[i]);
                initPQ(&elev->upoff[i]);
                initPQ(&elev->downoff[i]);
                }

        }

/* move elevator to required floor */
void moveTo(Elevator* elev,Event* event)

        {
        /* check if elevator is operating */
        if(elev->status == OPERATING)

                {
                /* elevator moving up */
                if(elev->dir == UP && elev->floor < elev->maxFloors)elev->floor++;

                /* elevator moving down */
                else if(elev->dir == DOWN && elev->floor > 0)elev->floor--;
                /* elevator not moving */
                else

                        {
                         /* check to go up */
                        if(moreUp(elev))

                                {
                                elev->dir = UP;
                                elev->floor++;
                                }

                        /* check to go down */
                        else if(moreDown(elev))

                                {
                                elev->dir = DOWN;
                                elev->floor--;
                                }

                /* we are here */
                else printf("elevator already at this floor\n");
                }

        printf("move to %d with %d people \n",elev->floor,elev->patrons);
        }

}

/* add floor requests to elevator */
void setElevOn(Elevator *elev,Event *event)

        {
        /* elevator moving up */
        if(event->dir == UP)enqueue(&elev->upon[event->from],event);

        /* elevator moving down */
        else if(event->dir == DOWN)
        enqueue(&elev->downon[event->from],event);
        }

/* check if elevator should stop at this floor */
int checkOn(Elevator *elev)

        {
        int up,down;
        int i=elev->floor;

        up = elev->upon[elev->floor].count;
        down = elev->downon[elev->floor].count;
        if(elev->dir == UP && up)return up; /* elevator moving up */

        /* elevator moving down or stationary */
        else if(elev->dir == DOWN && down)return down;
        /* no direction */
        else

                {
                if(up)return up;
                if(down)return down;
                }

        return 0;
        }

/* check if elevator should stop at this floor */
bool checkOff(Elevator *elev)

        {
        int up,down;
        int i=elev->floor;
        up = elev->upoff[elev->floor].count;
        down = elev->downoff[elev->floor].count;
        /* elevator moving up or stationary */
         if(elev->dir == UP && up)return up;
        /* elevator moving down or stationary */
        else if(elev->dir == DOWN && down)return down;

        /* no direction */
        else

                {
                if(up)return up;
                if(down)return down;
                }

        return 0;
        }

/* remove requests to get on from this floor , add to off list */
void removeOn(Elevator* elev,Event* event)

        {
        /* elevator moving up */
        if(event->dir == UP)
        {
        event = dequeue(&elev->upon[elev->floor]);
        /* loop till all up on events requests removed *./
        while(event != NULL)

                {

                /* put in off list */
                enqueue(&elev->upoff[event->to],event);
                event = dequeue(&elev->upon[elev->floor]);
                }

        }
```

```
/* elevator moving down or stationary */
else

        {
        event = dequeue(&elev->downon[elev->floor]);

        /* loop till all down on events requests removed *./
        while(event != NULL)

                {
                /* put in off list */

                enqueue(&elev->downoff[event->to],event); event = dequeue(&elev-
                >downon[elev->floor]);
                }

        }

}

/*  remove requests to get off from this floor */
void removeOff(Elevator* elev)

        {
        Event *event;

        /* elevator moving up or stationary */
        if(elev->dir == UP || elev->dir == STOPPED)

                {
                event = dequeue(&elev->upoff[elev->floor]);

                /* loop till all down up off  events requests removed *./
                 while(event != NULL)

                        {
                        elev->patrons--;
                        free(event);
                        /* put in off list */
                        event = dequeue(&elev->upoff[elev->floor]);
                        }

                }
        /* elevator moving down  */
        else

                {
                event = dequeue(&elev->downoff[elev->floor]);
                /* loop till all down ooff events requests removed */
                while(event != NULL)

                        {
                        /* put in off list */
                        elev->patrons--;
                        free(event);
                        event = dequeue(&elev->downoff[elev->floor]);
                        }

                }

        /* set elevator to no direction if everyone got off */
        if(elev->patrons == 0)elev->dir = STOPPED;
        }

/* check if more elevator requests from this floor going up */
bool moreUp(Elevator* elev)

        {
        int i;
        /* check from this floor going up */
        for(i=elev->floor;i<elev->maxFloors;i++)

                {
                if(elev->upoff[i].count || elev->upon[i].count)return true;
                if(elev->downoff[i].count || elev->downon[i].count)return true;
                }

        return false;
        }

/* check if any requests from this floor going down
bool moreDown(Elevator* elev)

        {
        int i;
        /* check from this floor going down */
        for(i=elev->floor;i>=0;i--)

                {
                if(elev->downoff[i].count || elev->downon[i].count)return true;
                if(elev->upoff[i].count || elev->upon[i].count)return true;
                }

        return false;
        }
```

STATISTICS MODULE

The statistics module must keep track the awaiting and average waiting times of the people requesting an elevator, We know the time they request, we know the time when they get on and we know the tine when they get off. The statistics module must have a data structure to keep track of all these things.

REPORT MODULE

The report module will just report the results of the statistics module at convenient time. A repot event will cause another report event to be generated at the next report time interval.

LESSON 19 EXERCISE 1

Type in or paste and copy all the elevator simulation modules. Use separate header *.h files and implementation files *.cpp. Run a few simulations and convince your self everything is working okay.

LESSON 19 EXERCISE 2

Write the Statistics and Report modules and integrate them into the elevator simulation program.

LESSON 19 EXERCISE 3

Replace the switch statement on the Run() module by using function pointers in the event structure.

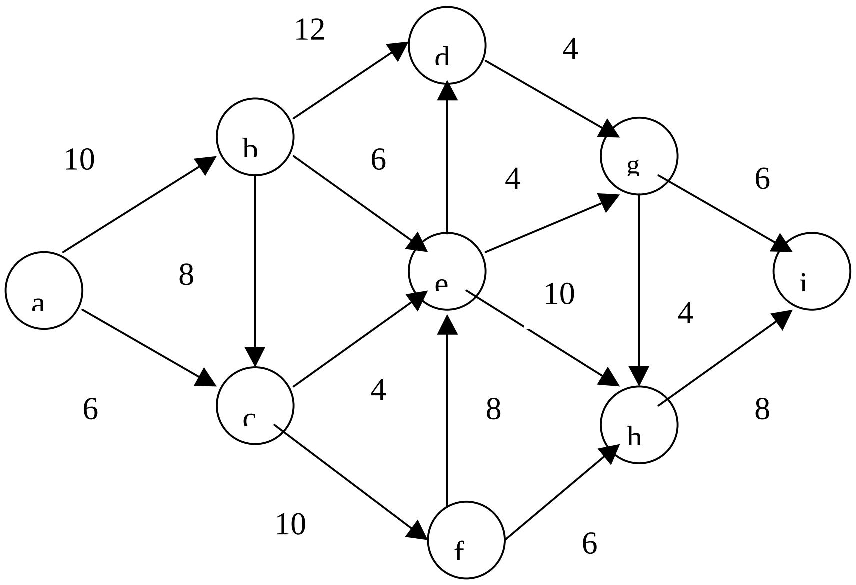
LESSON 19 EXERCISE 4

Add two more elevators and generate more request events by changing the interval times.

C DATA STRUCTURES PROGRAMMERS GUIDE PROJECT II

File:	CGuideL20.doc
Date Started:	July 24,1998
Last Update:	Dec 22, 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

students@cstutoring.com