本人答案整理自

<http://www.cise.ufl.edu/~sahni/fdsc2ed/>

没有进行校对,仅为方便自己断网或网络不好时使用.如有纰漏,望请见谅.

因见多有人寻此资源不果,遂分享之,以报多年来众驴友分享于我之万一.

本答案为 英文版 数据结构基础(C语言版) 即

FUNDAMENTALS OF DATA STRUCTURES IN C 部分习题答案.

答案所有文字描述均为英文.且仅有大部分习题答案.少数习题答案网站没有给出,因而我也没有办法.

本人排版水平不佳,全部复制粘贴网站内容,几乎未经过任何排版,大家就将就看下吧.

目录

[CHAPTER 1 2](#_Toc246109837)

[CHAPTER 2 24](#_Toc246109838)

[CHAPTER 3 44](#_Toc246109839)

[CHAPTER 4 63](#_Toc246109840)

[CHAPTER 5 79](#_Toc246109841)

[CHAPTER 6 96](#_Toc246109842)

[CHAPTER 7 125](#_Toc246109843)

[CHAPTER 8 139](#_Toc246109844)

[CHAPTER 9 140](#_Toc246109845)

[CHAPTER 10 142](#_Toc246109846)

[CHAPTER 11 147](#_Toc246109847)

[CHAPTER 12 170](#_Toc246109848)

# CHAPTER 1

Chapter 1, Pages 16-17

Problem 1.a This statement is a poorly phrased version of Fermat's last theorem. We know that we can find n > 2 for which ‚the equation holds. Fermat scribbled a note on a text margin indicating that he had found the solution for n = 2. Unfortunately, the proof died with him. Because the statement is phrased as a question‚ rather than a clearly defined algorithm, it lacks definiteness.

Problem 1.b This statement violates not only the rules of mathematics, but the criterion of effectiveness. We can compute only those things that are feasible, and division by zero is mathematically undefined

Page 17, Exercise 3

#include <stdio.h>

#include <math.h>

#include <string.h>

#define TRUE 1

#define FALSE 0

#define MAX\_STRING 100

void truth\_table(int);

int main(){

int n;

printf("n:(>=0): ");

scanf("%d", &n);

while (n <= 0)

{ /\*error loop \*/

printf("n:(>=0): ");

scanf("%d", &n);

}

truth\_table(n);

}

void truth\_table(int n)

{/\* generate a truth\_table by transforming # of

permutations into binary \*/

int i,j, div, rem;

char string[MAX\_STRING];

for (i=0; i < pow(2,n); i++)

{/\*number of permutations or rows in the table \*/

strcpy(string ,"\0");

div = i;

for (j = n; j > 0; j--)

{/\*number of bits needed for each row\*/

rem = div%2;

div = div/2;

if (!rem) strcat(string,"FALSE ");

else strcat(string, "TRUE ");

}

printf("%s\n", string);

}

}

Page 17, Exercise 4

#include <stdio.h>

int min(int, int);

#define TRUE 1

#define FALSE 0

int main()

{

int x,y,z;

printf("x: "); scanf("%d", &x);

printf("y: "); scanf ("%d", &y);

printf("z: "); scanf("%d", &z);

if (min(x,y) && min(y,z)) {/\*x is smallest \*/

printf("%d ", x);

if (min(y,z)) printf ("%d %d\n", y,z);

else printf("%d%d\n", z, y);

}

else if (min(y,x) && min(y,z)){ /\*y is the smallest \*/

printf("%d ", y);

if (min(x,z)) printf ("%d %d\n", x,z);

else printf("%d%d\n", z,x);

}

else

printf("%d %d %d\n", z, y, x);

}

int min(int a, int b) {

if (a < b) return TRUE;

return FALSE;

}

Page 17, Exercise 7

#include <stdio.h>

double iterFact(int);

double recurFact(int);

int main(){

int n;

printf("n:(>=0): ");

scanf("%d", &n);

while (n < 0)

{ /\*error loop \*/

printf("n:(>=0): ");

scanf("%d", &n);

}

printf("%d factorial is %f.\n", n, iterFact(n));

printf("%d factorial is %f.\n", n, recurFact(n));

}

double recurFact(int n)

{ /\*recursive version \*/

if ((n==0) || (n==1)) return 1.0;

return n\*recurFact(n-1);

}

double iterFact(int n)

{/\* find the factorial, return as a double

to keep it from overflowing \*/

int i;

double answer;

if ((n == 0) || (n == 1)) return 1.0;

answer = 1.0;

for (i = n; i > 1; i--)

answer \*= i;

return answer;

}

Page 17, Exercise 8

#include <stdio.h>

int iterFib(int);

int recurFib(int);

int main(){

int n;

printf("n:(>=0): ");

scanf("%d", &n);

while (n < 0)

{ /\*error loop \*/

printf("n:(>=0): ");

scanf("%d", &n);

}

printf("%d Fibonacci is %d.\n", n, iterFib(n));

printf("%d Fibonacci is %d.\n", n, recurFib(n));

}

int recurFib(int n)

{ /\*recursive version \*/

if ((n==0) || (n==1)) return 1;

return recurFib(n-1) + recurFib(n-2);

}

int iterFib(int n)

{/\* find the factorial, return as a double

to keep it from overflowing \*/

int i;

int fib, fib1, fib2;

if ((n == 0) || (n == 1)) return 1;

fib1 = fib2 = 1;

for (i = 2; i <=n; i++) {

fib = fib1+fib2;

fib2 = fib1;

fib1 = fib;

}

return fib;

}

Page 17, Exercise 9

#include <stdio.h>

double recurBinom(int, int);

double iterBinom(int, int);

double recurFact(int n);

int main(){

int n,m;

printf("n:(>=0): ");

scanf("%d", &n);

while (n < 0)

{ /\*error loop \*/

printf("n:(>=0): ");

scanf("%d", &n);

}

printf("m:(>=0): ");

scanf("%d", &m);

while (m < 0)

{ /\*error loop \*/

printf("m:(>=0): ");

scanf("%d", &m);

}

printf("n: %d m: %d Recursive Binomial coefficient is %f.\n", n, m, recurBinom(n,m));

printf("n: %d m: %d Iterative Binomial coefficient is %f.\n", n, m, iterBinom(n,m));

}

double iterBinom(int n, int m)

{/\* defined as n!/(m! - (n-m)!)\*/

int i;

double nFact, mFact, nMinusMFact;

if (n == m) return 1;

if ((n==0) || (n == 1)) nFact = 1;

else {

nFact = 1;

for (i = n; i > 1; i--)

nFact \*= i;

}

if ((m==0) || (m == 1)) mFact = 1;

else {

mFact = 1;

for (i = m; i > 1; i--)

mFact \*= i;

}

if ( ((n-m) == 0) || ((n-m) == 1)) nMinusMFact = 1;

else {

nMinusMFact = 1;

for (i = n-m; i > 1; i--)

nMinusMFact \*= i;

}

return nFact/(mFact\*nMinusMFact);

}

double recurFact(int n)

{ /\*recursive version \*/

if ((n==0) || (n==1)) return 1.0;

return n\*recurFact(n-1);

}

double recurBinom(int n, int m)

{ /\*recursive version \*/

return recurFact(n)/(recurFact(m)\*recurFact(n-m));

}

Page 17, Exercise 11

#include

#define Tower1 1

#define Tower2 2

#define Tower3 3

void towers\_of\_hanoi(int, int, int, int);

int main(){

int n\_disks;

printf("Number of disks: ");

scanf("%d", &n\_disks);

printf("Disk, source, and destination towers listed below\n");

printf("%12s%10s%15s\n", "Disk No", "Source","Destination");

towers\_of\_hanoi(n\_disks,Tower1,Tower3,Tower2);

}

void towers\_of\_hanoi(int n\_disks, int source, int dest, int spare){

if (n\_disks == 1 )

printf("%10d%10d%10d\n", n\_disks, source, dest);

else {

/\*move a disk from source to spare\*/

towers\_of\_hanoi(n\_disks-1,source,spare,dest);

printf("%10d%10d%10d\n", n\_disks, source, dest);

/\*move a disk from spare to destination tower\*/

towers\_of\_hanoi(n\_disks-1,spare,dest,source);

}

}

Page 21, Exercise 1

ADT NaturalNumber is objects:

an ordered subrange of the integers starting at zero and ending at the

maximum integer (INT\_MAX) on the computer functions: for

all x, y ∈ NaturalNumber; TRUE, FALSE ∈ Boolean

and where +, −, <, and == are the usual integer operations

NaturalNumber Zero( ) ::= 0

Boolean IsZero(x) ::= if (x) return FALSE

return TRUE

Boolean Equal(x, y) ::= if (x == y) return TRUE

return FALSE

NaturalNumber Successor(x) ::= if (x == INT\_MAX) return x

return x + 1

NaturalNumber Add(x, y) ::= if ((x + y) < INT\_MAX) return x + y

if ((x + y) == INT\_MAX) return x + y

return INT\_MAX

NaturalNumber Subtract(x, y) ::= if (x < y) return 0

return x − y

NaturalNumber Predecessor(x) ::= if (x < 0) return ERROR

if (x == 0) return 0

return x - 1

Boolean IsGreater(x, y) := if ((x-y)) < 0) return ERROR

if ((x-y)) == 0) return FALSE

return TRUE

NaturalNumber mult(x, y) ::= if (x < 0) return ERROR

if (y < 0) return ERROR

if (y == 1) return x

return x + mult(x, y-1)

NaturalNumber div(x, y) ::= if (x < 0) return ERROR

if (y < 0) return ERROR

if (y == 0) return ERROR

if (y == 1) return 1

return x - div(x, y-1)

end NaturalNumber

Page 21, Exercise 3

ADT Set is objects:

a subrange of the integers starting at (INT\_MIN) and ending at the

maximum integer (INT\_MAX) on the computer functions: for

all x, y ∈ Set; TRUE, FALSE ∈ Boolean and the Boolean operations defined in Problem 4 are available (not, and, or,...)

and where ==, Ø, +, head(s), tail(s) are the usual set operations,

where == return TRUE if tw set elements are the same, and TRUE otherwise.

Ø is the empty set.

+ adds and element to a set.

head(s) extracts the first member in the set.

tail(s) extracts a list of all other elements in the set. An empty set contains no tail. A set with only one element has the emtpy set has it tail.

Set Create(s) ::= Ø

Boolean IsEmpty(s) ::= if (s ==Ø ) return TRUE

return FALSE

Boolean IsIn(x, s) ::= if (IsEmpty(s)) return FALSE

if (x == head(s) return TRUE

return IsIn(x, Tail(s))

Set Insert(x,s) ::= if (IsEmpty(s)) return x + s

if (IsIn(a,s)) return s

return x + s

Set Remove(x, s) ::= if (x == head(s)) return tail(s)

return Remove(x, tail(s))

Set Union(x, s1, s2) ::= if IsEmpty(s1) return s2

if IsIn(head(s1), s2)) return Union(x, tail(s1), s2)

return head(s1) + Union(x, tail(s1), s2)

set Intersection(x, s1,s2) ::= if IsEmpty(s1) return Ø

if IsIn(head(s1), s2)) return head(s1) + Intersection(x, tail(s1), s2)

return Intersection(x, tail(s1), s2)

Boolean Difference(s1,s2) ::= if IsEmpty(s1) return Ø

if IsIn(head(s1), s2)) return Difference(x, tail(s1), s2)

return head(s1) + Difference(x, tail(s1), s2)

end Set

Page 21, Exercise 3

ADT Bag is objects:

a subrange of the integers starting at (INT\_MIN) and ending at the

maximum integer (INT\_MAX) on the computer functions: for

all x, y ∈ Bag; TRUE, FALSE ∈ Boolean and the Boolean operations defined in Problem 4 are available (not, and, or,...)

and where ==, Ø, +, head(s), tail(s) are the usual Bag operations,

where == return TRUE if tw set elements are the same, and TRUE otherwise.

Ø is the empty Bag.

+ adds and element to a Bag.

head(s) extracts the first member in the Bag.

tail(s) extracts a list of all other elements in the Bag. An empty bag contains no tail. A bag with only one element has the emtpy bag has it tail.

Bag Create(b) ::= Ø

Boolean IsEmpty(b) ::= if (b ==Ø ) return TRUE

return FALSE

Boolean IsIn(x, b) ::= if (IsEmpty(b)) return FALSE

if (x == head(b) return TRUE

return IsIn(x, Tail(b))

Bag Insert(x,s) ::= if (IsEmpty(b)) return b

return x + b

Bag Remove(x, s) ::= if (IsEmpty(b)) return b

if (x == head(b)) return tail(b)

return Remove(x, tail(b))

end bag

**Page 21, Exercise 4**

**ADT** *Boolean* is objects:   
TRUE, FALSE ∈ *Boolean* and the Boolean== the usual boolean operation.   
where == return TRUE if tw set elements are the same, and TRUE otherwise.

|  |  |
| --- | --- |
| *Boolean* not(*x*) | ::= **if** (*x*) **return** TRUE    **return** FALSE |
| *Boolean* and(*x,y*) | ::= **if**(not(*x*)) **return** FALSE     **if** (not(*y*)) **return** FALSE     **return** TRUE |
| *Boolean* or(*x, s*) | ::= **if**(*x*) **return** TRUE     **if** (*y*) **return** TRUE     **return** FALSE |
| *Boolean* xor(*x, s*) | ::= **if**(and(not(*x*), not(*y*))) **return** FALSE    **if** (and(*x*, *y*)) **return** FALSE     **return** TRUE |
| *Boolean* implies(*x, s*) | ::= **if** (and(*x*, *y*)) **return** TRUE    **if** (and(not(*x)*,not(*y*))**) return** TRUE     **return** FALSE |
| *Boolean* equivalent(*x, s*) | ::= **if**(and(not(*x*), not(*y*))) **return** TRUE    **if** (and(*x*, *y*)) **return** TRUE     **return** FALSE |

end *Set*

**Page 25, Exercise 1**

Iterative Factorial Function, SiterFact(I) = 0

Recursive Function: recurFact()

|  |  |  |
| --- | --- | --- |
| Type | Name | Number of Bytes |
| Parameter   (int) | n | 4 |
| Local Variables   (int)   (double) | i answer | 4 6 |
| Return Type (double) |  | 6 |
| TOTAL | | 20 for each recursive call if (n == MAX\_SIZE) SrecurFact(I) = (20•MAX\_SIZE) |

**Page 21, Exercise 3**

**ADT** *Set* is objects:   
a subrange of the integers starting at **(INT\_MIN)** and ending at the  
maximum integer (**INT\_MAX**) on the computer functions: for  
all x, y ∈ *Set*; TRUE, FALSE ∈ *Boolean* and the Boolean operations defined in Problem 4 are available (not, and, or,...)  
and where ==, Ø, +, head(s), tail(s) are the usual set operations,   
where == return TRUE if tw set elements are the same, and TRUE otherwise.   
Ø is the empty set.  
+ adds and element to a set.  
head(s) extracts the first member in the set.   
tail(s) extracts a list of all other elements in the set. An empty set contains no tail. A set with only one element has the emtpy set has it tail.

|  |  |
| --- | --- |
| *Set* Create(*s*) | ::= Ø |
| *Boolean* IsEmpty(*s*) | ::= **if** (*s ==*Ø ) **return** *TRUE*      **return** *FALSE* |
| *Boolean* IsIn(*x, s*) | ::= **if** (IsEmpty(*s*)) **return** FALSE      **if** (*x* == head(s) **return** *TRUE*      **return** IsIn(*x,* Tail(s)) |
| *Set* Insert(*x,s*) | ::= **if** (IsEmpty(*s*)) **return** *x + s*       **if** (IsIn(a,s)) **return** *s*      **return** *x + s* |
| *Set* Remove(*x, s*) | ::= **if** (*x* == head(*s*)) **return** tail(*s*)       **return** Remove(*x*, tail(*s*)) |
| *Set* Union(*x, s1, s2*) | ::= **if** IsEmpty(*s1*) **return** *s2*     **if** IsIn(head(*s1*), *s2*)) **return** Union(*x*, tail(*s1*), *s2*)      **return** head(*s1*) + Union(*x*, tail(*s1*), *s2*) |
| *set* Intersection(*x, s1,s2*) | ::= **if** IsEmpty(*s1*) **return**Ø     **if** IsIn(head(*s1*), *s2*)) **return** head(*s1*) + Intersection(*x*, tail(*s1*), *s2*)      **return** Intersection(*x*, tail(*s1*), *s2*) |
| *Boolean* Difference(*s1,s2*) | ::= **if** IsEmpty(*s1*) **return**Ø     **if** IsIn(head(*s1*), *s2*)) **return** Difference(*x*, tail(*s1*), *s2*)      **return** head(*s1*) + Difference(*x*, tail(*s1*), *s2*) |

end *Set*

**Page 21, Exercise 3**

**ADT** *Bag* is objects:   
a subrange of the integers starting at **(INT\_MIN)** and ending at the  
maximum integer (**INT\_MAX**) on the computer functions: for  
all x, y ∈ *Bag*; TRUE, FALSE ∈ *Boolean* and the Boolean operations defined in Problem 4 are available (not, and, or,...)  
and where ==, Ø, +, head(s), tail(s) are the usual *Bag* operations,   
where == return TRUE if tw set elements are the same, and TRUE otherwise.   
Ø is the empty Bag.  
+ adds and element to a Bag.  
head(s) extracts the first member in the Bag.   
tail(s) extracts a list of all other elements in the Bag. An empty bag contains no tail. A bag with only one element has the emtpy bag has it tail.

|  |  |
| --- | --- |
| *Bag* Create(*b*) | ::= Ø |
| *Boolean* IsEmpty(*b*) | ::= **if** (*b ==*Ø ) **return** *TRUE*      **return** *FALSE* |
| *Boolean* IsIn(*x, b*) | ::= **if** (IsEmpty(*b*)) **return** FALSE      **if** (*x* == head(b) **return** *TRUE*      **return** IsIn(*x,* Tail(b)) |
| *Bag* Insert(*x,s*) | ::= **if** (IsEmpty(*b*)) **return** *b*     **return** *x + b* |
| *Bag* Remove(*x, s*) | ::= if (IsEmpty(*b*)) return *b* **if** (*x* == head(*b*)) **return** tail(*b*)       **return** Remove(*x*, tail(*b*)) |

end *bag*

**Page 32, Exercise 4**: PrintMatrix

**a. Counts**

void Printmatrix(int matrix[][MAX\_SIZE], int rows, int cols)

{

int i,j;

int count = 0;

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

{

count ++; /\*for i loop \*/

for (j = 0; j<cols; j++) {

printf("%5d",matrix[i][j]);

count++; /\*for j loop \*/

}

count++; /\* last time of j \*/

printf("\n");

}

count++; /\* last time of i \*/

printf("Print Count: %d\n", count);

}

**b. Simplified Counts**

void Printmatrix(int matrix[][MAX\_SIZE], int rows, int cols)

{

int i,j;

int count = 0;

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

{

count ++; /\*for i loop \*/

for (j = 0; j<cols; j++) {

printf("%5d",matrix[i][j]);

count++; /\*for j loop \*/

}

count++; /\* last time of j \*/

printf("\n");

}

count++; /\* last time of i \*/

printf("Print Count: %d\n", count);

}

**c. Final Count for 5x5 matrix : 36**

**d. Step Count Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Step Count Table | | | |
| Statement | s/e | f | Total Steps |
| void Printmatrix(int matrix[][MAX\_SIZE], int rows, int cols)  {  int i,j;  for (i = 0; i<rows; i++)  {  for (j = 0; j<cols; j++)  printf("%5d",matrix[i][j]);  printf("\n");  }  } | 0  0  0  1  0  1  0  0  0 | 0  0  01  n+1  0  n+1  0  0  0 | 0  0  0  n+1  0  n+1  0  0  0 |
| Total | | | 2n+2 |

**Page 32, Exercise 5 Multiplication**

**a. Counts**

void mult(int a[][MAX\_SIZE], int b[][MAX\_SIZE], int c[][MAX\_SIZE])

{

int i, j,k;

int count = 0;

for (i = 0; i < MAX\_SIZE; i++) {

count ++; /\*for i loop \*/

for (j = 0; j < MAX\_SIZE; j++) {

c[i][j] = 0; count++; /\* for assignment \*/

count++; /\*for j loop \*/

}

count++; /\* last time of j \*/

for (k=0; k < MAX\_SIZE; k++){

c[i][j] += (a[i][k] \* b[k][j]); count++; /\* for assignment \*/

count++; /\* for k loop\*/

}

count++; /\*last time of k \*/

}

count++; /\* last time of i \*/

printf("Multiplication Count: %d\n", count);

}

**b. Simplified Counts**

void mult(int a[][MAX\_SIZE], int b[][MAX\_SIZE], int c[][MAX\_SIZE])

{

int i, j,k;

int count = 0;

for (i = 0; i < MAX\_SIZE; i++) {

for (j = 0; j < MAX\_SIZE; j++) {

c[i][j] = 0;

count+=2;

}

for (k=0; k < MAX\_SIZE; k++){

c[i][j] += (a[i][k] \* b[k][j]);

count+=2;

}

count+=3;

}

count++; /\* last time of i \*/

printf("Multiplication Count: %d\n", count);

}

**c. Final Count for 5x5 matrix: 119**

**d. Step Count Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Step Count Table | | | |
| Statement | s/e | f (MAX\_S) = MAX\_SIZE | Total Steps |
| void mult(int a[][MAX\_SIZE], int b[][MAX\_SIZE],  int c[][MAX\_SIZE])  {  int i, j,k;  for (i = 0; i < MAX\_SIZE; i++)  for (j = 0; j < MAX\_SIZE; j++)  {  c[i][j] = 0;  for (k=0; k < MAX\_SIZE; k++)  c[i][j] += (a[i][k] \* b[k][j]);  }  } | 0  0  0  0  1  1  0  1  1  1  0  0 | 1  1  1  1  MAX\_S+1  MAX\_S·MAX\_S + MAX\_S  MAX\_S·MAX\_S  MAX\_S·MAX\_S  MAX\_S·MAX\_S·MAX\_S + MAX\_S·MAX\_S·MAX\_s  MAX\_S·MAX\_S·MAX\_S  MAX\_S·MAX\_S  MAX\_S | 0  0  0  0  MAX\_S+1  MAX\_S·MAX\_S + MAX\_S  0  MAX\_S·MAX\_S  MAX\_S·MAX\_S·MAX\_S + MAX\_S·MAX\_S·MAX\_S  MAX\_S·MAX\_S·MAX\_S  0  0 |
| Total : 3·MAX\_SIZE3 + 2·MAX\_SIZE2 + 2·MAX\_SIZE + 1 = O(MAX\_SIZE)3 | | | |

**Page 32, Exercise 4: Product**

**a. Counts**

void prod(int a[][MAX\_SIZE], int b[][MAX\_SIZE], int c[][MAX\_SIZE], int rowsA, int colsB, int colsA)

{

int i, j,k;

int count = 0;

for (i = 0; i < rowsA; i++) {

count ++; /\*for i loop \*/

for (j = 0; j < colsB; j++) {

c[i][j] = 0; count++; /\* for assignment \*/

count++; /\*for j loop \*/

for (k = 0; k < colsA; k++) {

count++; /\* for k loop\*/

c[i][j] += (a[i][k] \* b[k][j]); count++; /\*for assignment \*/

}

count++; /\*last time of k \*/

}

count++; /\* last time of j \*/

}

count++; /\* last time of i \*/

printf("Prod Count: %d\n", count);

}

**b. Simplified Counts**

void prod(int a[][MAX\_SIZE], int b[][MAX\_SIZE],

int c[][MAX\_SIZE], int rowsA, int colsB, int colsA)

{

int i, j,k;

int count = 0;

for (i = 0; i < rowsA; i++) {

for (j = 0; j < colsB; j++) {

c[i][j] = 0; count++;

count+=3;

for (k = 0; k < colsA; k++) {

count++;

c[i][j] += (a[i][k] \* b[k][j]);

count+=2;

}

}

count+=3;

}

count++; /\* last time of i \*/

printf("Prod Count: %d\n", count);

}

**c. Final Count for 5x5 matrix**

void prod(int a[][MAX\_SIZE], int b[][MAX\_SIZE], int c[][MAX\_SIZE],

int rowsA, int colsB, int colsA)

{

int i, j,k;

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

for (j = 0; j < colsB; j++) {

c[i][j] = 0;

for (k = 0; k < colsA; k++)

c[i][j] += (a[i][k] \* b[k][j]);

}

}

**d. Step Count Table : 336**

|  |  |  |  |
| --- | --- | --- | --- |
| Step Count Table | | | |
| Statement | s/e | f (A = rows A, B = rowsB) | Total Steps |
| void prod(int a[][MAX\_SIZE], int b[][MAX\_SIZE],  int c[][MAX\_SIZE], int rowsA, int colsB, int colsA)  {  int i, j,k;  for (i = 0; i < rowsA; i++)  for (j = 0; j < colsB; j++)  {  c[i][j] = 0;  for (k = 0; k < colsA; k++)  c[i][j] += (a[i][k] \* b[k][j]);  }  } | 0  0  0  0  1  1  0  1  1  1  0  0 | 1  1  1  1  A+1  A•B+ B+1  A•B  A•B  A•B•A + A•B+ A+1  A•B•A + A•B+ A  A•B  A | 0  0  0  0  A+1  A•B+ B+1  0  A•B  A•B•A + A•B+ A+1  A•B•A + A•B+ A  0  0 |
| Total : 2A•B•A + 3A•B + 2A + B + 3 | | | |

**Page 32, Exercise 4**

**a. Counts**

void transpose(int a[][MAX\_SIZE])

{

int i, j;

int temp;

int count = 0;

for (i = 0; i & lt; MAX\_SIZE-1; i++) {

count ++; /\*for i loop \*/

for (j = i+1; j < MAX\_SIZE; j++) {

count ++; /\*for j loop \*/

SWAP(a[i][j],a[j][i],temp); count+=3; /\*for swap \*/

}

count++; /\* last timne of j \*/

}

count++; /\* last time of i \*/

printf("Transpose Count: %d\n", count);

}

**b. Simplified Counts**

void transpose(int a[][MAX\_SIZE])

{

int i, j;

int temp;

int count = 0;

for (i = 0; i < MAX\_SIZE-1; i++) {

for (j = i+1; j < MAX\_SIZE; j++) {

count ++; /\*for j loop \*/

SWAP(a[i][j],a[j][i],temp); count+=3; /\*for swap \*/

}

count+=2;

}

count++; /\* last time of i \*/

printf("Transpose Count: %d\n", count);

}

**c. Final Count for 5x5 matrix : 49**

**d. Step Count Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Step Count Table | | | |
| Statement | s/e | f | Total Steps |
| void transpose(int a[][MAX\_SIZE])  {  int i, j;  int temp;  for (i = 0; i =< MAX\_SIZE-1; i++)  for (j = i+1; j < MAX\_SIZE; j++)  SWAP(a[i][j],a[j][i],temp);  } | 0  0  0  0  1  1  3  0 | 1  1  1  1  MAX\_S  MAX\_S(MAX\_S-1)+MAX\_S  MAX\_S(MAX\_S-1)  1 | 0  0  0  0  MAX\_S  MAX\_S•MAX\_S-1 + MAX\_S  3(MAX\_S(MAX\_S-1))  0 |
| Total : 4MAX\_SIZE•MAX\_S-1 + 2MAX\_SIZE | | | |

**Page 32, Exercise 4**

**a. Counts**

void transpose(int a[][MAX\_SIZE])

{

int i, j;

int temp;

int count = 0;

for (i = 0; i & lt; MAX\_SIZE-1; i++) {

count ++; /\*for i loop \*/

for (j = i+1; j < MAX\_SIZE; j++) {

count ++; /\*for j loop \*/

SWAP(a[i][j],a[j][i],temp); count+=3; /\*for swap \*/

}

count++; /\* last timne of j \*/

}

count++; /\* last time of i \*/

printf("Transpose Count: %d\n", count);

}

**b. Simplified Counts**

void transpose(int a[][MAX\_SIZE])

{

int i, j;

int temp;

int count = 0;

for (i = 0; i < MAX\_SIZE-1; i++) {

for (j = i+1; j < MAX\_SIZE; j++) {

count ++; /\*for j loop \*/

SWAP(a[i][j],a[j][i],temp); count+=3; /\*for swap \*/

}

count+=2;

}

count++; /\* last time of i \*/

printf("Transpose Count: %d\n", count);

}

**c. Final Count for 5x5 matrix : 49**

**d. Step Count Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Step Count Table | | | |
| Statement | s/e | f | Total Steps |
| void transpose(int a[][MAX\_SIZE])  {  int i, j;  int temp;  for (i = 0; i =< MAX\_SIZE-1; i++)  for (j = i+1; j < MAX\_SIZE; j++)  SWAP(a[i][j],a[j][i],temp);  } | 0  0  0  0  1  1  3  0 | 1  1  1  1  MAX\_S  MAX\_S(MAX\_S-1)+MAX\_S  MAX\_S(MAX\_S-1)  1 | 0  0  0  0  MAX\_S  MAX\_S•MAX\_S-1 + MAX\_S  3(MAX\_S(MAX\_S-1))  0 |
| Total : 4MAX\_SIZE•MAX\_S-1 + 2MAX\_SIZE | | | |

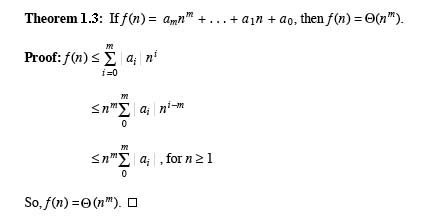
**Page 40, Exercise 1**

1. Show that the following statements are correct:  
(a) 5n2 − 6n = θ(n2) Sincen 2 is the leading exponent this is correct.   
(b) n! = O(nn) n! is the leading exponent and its time is a non-deterministic polynomial, so this is correct.  
(c) n2+ n log n = θ(n2 ) n2 is the leading exponent.  
(f) n2n + 6 . 2n = θ(n2n) n2n is the leading exponent.  
(g) n2 + 103n2 = θ(n3) Since 103 is a constant and the time is based on the leading exponent for the variable, this should be n2.  
(h) 6n3 / (log n + 1) = O(n3) n3 is the leading exponent.  
(i) n1.001 + n log n = θ(n1.001 ) This is debatable because n logn actually grows at a faster rate than n raised to 1.0001 except for n < 2.  
(j) nk + n + nklog n = θ(nklogn) for all k ≥ 1. nklogn grows at a faster rate.  
(k) 10n3 + 15n4 +100n22n = O(n22n) n22nis the leading exponent.

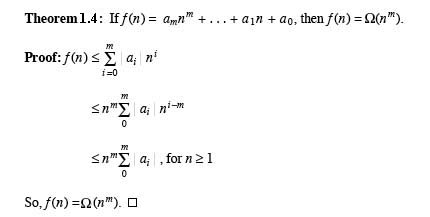
**Page 40, Exercise 2**

Show that the following statements are incorrect:  
(a) 10n2 + 9 = O(n). The leading exponent is n2 so the correct answer is O n2)  
(b) n2 logn = θ(n2). The leading exponent is n2 logn so the time is θ(n2 logn).  
<  
(c) n2 / logn = θ(n2 ). The leading expoonent is n2 / logn so the time is θ( n2 / logn).  
(d) n3 2n + 6n2 3n = O(n2 2n). It doesn't make much difference because either way this time is a non-deterministic polynomial. The time should be O( nn).  
(e) 3nn = O(2n ). The time should be O( nn).

Page 41, Exercise 3



**Page 41, Exercise 4**



**Page 41, Exercise 5**

Program 1.19 prints out a 2-dimensional array. The outerloop iterates n times as does the inner loop. The time is n•n; therefore, the worst case time is O(n2 ).

**Page 41, Exercise 6**

Program 1.19 multiplies a 2-dimensional array. It requires 3-nested loops each iteratint n times. The time is n•n•n; therefore the worst case time is O(n3). Strassen's Matrix Multiplication method reduces the complexity to : O(n2.76).

**Page 41, Exercise 7**

a: O(n2)

b : 20n+ 4

The answer is obvious because 20•20 = 2020. (a) will exceed b at n>20.

# CHAPTER 2

**Page 58, Exercise 1**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int\*\* make2dArray(int, int);  #define MALLOC(p,s) \  if (!((p) = malloc(s))) {\  fprintf(stderr, "Insufficient memory"); \  exit(EXIT\_FAILURE); \  }  int main(){  int i,j;  int \*\*myArray;  myArray = make2dArray(5,10);  myArray[2][4]=6;  for (i = 0; i<5; i++){  for (j = 0; j < 10; j++){  /\*myArray[i][j] =0;\*/  printf("%d ", myArray[i][j]);  }  printf("\n");  }  }  int\*\* make2dArray(int rows, int cols)  {/\*create a 2-d array rows x cols \*/  int \*\*x, i;  MALLOC(x,rows\*sizeof(\*x)); /\*get memory for row pointers\*/  for(i = 0; i < rows; i++){  MALLOC(x[i], cols \* sizeof(\*\*x)); /\*get memory for col pointers\*/  \*x[i]=0;  }  return x;  } |

**Page 64, Exercise 1**

|  |
| --- |
| #include <stdio.h>  #include <string.h>  typedef struct {  int month;  int day;  int year;  } date;  typedef struct {;}noMarriage;  typedef struct { date dom, dodivorce;}divorce;  typedef struct {date dom, dodeath;}widow;  typedef struct {  enum tagField {single, married, widowed, divorced} status;  union {  noMarriage nom;  date dom;  divorce divorceStuff;  widow widowStuff;  } u;  } maritalStatus;  typedef struct {  char name[10];  int age;  float salary;  date dob;  maritalStatus ms;  }humanBeing;  void printPerson(humanBeing);  int main()  {  humanBeing p1;  p1.dob.month = 5;  p1.dob.day = 16;  p1.dob.year = 1978;  strcpy(p1.name, "Fiona");  p1.salary = 1.00;  p1.ms.status = married;  p1.ms.u.dom.month = 10;  p1.ms.u.dom.day = 31;  p1.ms.u.dom.year = 1999;  printPerson(p1);  }  void printPerson(humanBeing p)  {/\*print out the information on a person \*/  printf("Name: %s\n", p.name);  printf("DOB: %d/%d/%d\n", p.dob.month, p.dob.day, p.dob.year);  printf("Salary: %5.2f\n", p.salary);  switch(p.ms.status){  case married:  printf("Marriage Date: %d/%d/%d\n", p.ms.u.dom.month, p.ms.u.dom.day, p.ms.u.dom.year);  }  } |

**Page 64, Exercise 3**

|  |
| --- |
| #include <stdio.h>  #include <string.h>  typedef struct {double length, width;} rectStuff;  typedef struct {double base, height;} triStuff;  typedef struct {  char name[10];  enum tagField {rectangle, triangle, circle} shapeType;  union {  rectStuff rect;  triStuff tri;  double radius;  } stats;  }shape;  void printShape(shape);  int main()  {  shape s1, s2, s3;  strcpy(s1.name, "rectangle");  s1.shapeType = rectangle;  s1.stats.rect.length = 10;  s1.stats.rect.width = 20;  strcpy(s2.name, "triangle");  s2.shapeType = triangle;  s2.stats.tri.base = 102;  s2.stats.tri.height = 450;  strcpy(s3.name ,"circle");  s3.stats.radius = 2.5;  /\* printf("%f\n",s3.stats.radius);\*/  printShape(s1);  printShape(s2);  printShape(s3);    }  void printShape(shape s)  {/\*print out the information on a shape \*/  printf("Name: %s\n", s.name);  switch(s.shapeType){  case rectangle:  printf("\tLength:%f\n", s.stats.rect.length);  printf("\tWidth:%f\n\n", s.stats.rect.width);  break;  case triangle:  printf("\tBase:%f\n", s.stats.tri.base);  printf("\tHeight:%f\n\n", s.stats.tri.height);  break;  case circle:  printf("Radius:%f\n",s.stats.radius);  break;  }  } |

**Page 72, Exercise 2: readPoly() & printPoly()**

void printPoly(polynomial terms [], int n)

{/\*print the polynomial\*/

int i;

for (i = 0;i < n-1; i++)

printf("%5.2fx^%d +", terms[i].coef, terms[i].expon);

printf("%5.2fx^%d\n", terms[n-1].coef, terms[n-1].expon);

}

void readPoly(polynomial terms [], int \*n)

{/\*read in a polynomial\*/

int i,expon;

float coef;

printf ("Enter the number of terms in your polynomial: ");

scanf("%d", n);

while (\*n >= MAX\_TERMS) {

printf("Too many terms in the polynomial\n");

printf("Number of terms: ");

scanf("%d", n);

}

for (i= 0; i < \*n; i++) {

printf("Coefficient: ");

scanf("%f", &coef);

printf("Exponent: ");

scanf("%d", &expon);

terms[i].coef = coef;

terms[i].expon = expon;

}

}

**Page 72, Exercise 3: pmult()**

void pmult(polynomial a [] , polynomial b [], polynomial c [], int na, int nb, int \*nc)

{/\*multiply two polynomials \*/

int i, j;

\*nc = 0;

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

for (j = 0; j < nb; j++) {

c[\*nc].coef = a[i].coef \* b[j].coef;

c[(\*nc)++].expon = a[i].expon + b[j].expon;

}

/\*put simplification here if you wish \*/

}

**Page 72, Exercise 4: peval()**

double peval(float x, polynomial terms[], int n)

{/\*evaluate the polynomial for x \*/

int i;

float answer = 0;

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

answer = answer + terms[i].coef\*pow(x, terms[i].expon);

return answer;

}

**Page 72, Exercise 5: 2-d array of polynomials**

Advantage: This implementation allows automatic processing of polynomials through the use of for statements.

Disadvantage: Using a 2 dimensional array wastes a tremendous amount of storage. This 2d array contains @64k cells!

|  |
| --- |
| /\* \*  \* Created by Susan Anderson-Freed on 10/2/07.  \*\*/  #include  #include  #define MAX\_TERMS 100  #define MAX\_POLYS 15  typedef struct {  float coef;  int expon;  } polynomial;  void readPoly(polynomial [][MAX\_TERMS], int [], int \*);  void printPoly(polynomial [], int );  int main()  {  float x;  polynomial terms[MAX\_POLYS][MAX\_TERMS];  int nes[MAX\_POLYS];  int totalPolys = 0;  readPoly(terms, nes, &totalPolys);  readPoly(terms, nes, &totalPolys);  printPoly(terms[0],nes[0]);  printPoly(terms[1],nes[1]);  }  void printPoly(polynomial terms [], int n)  {/\*print the polynomial\*/  int i;  for (i = 0;i < n-1; i++)  printf("%5.2fx^%d +", terms[i].coef, terms[i].expon);  printf("%5.2fx^%d\n", terms[n-1].coef, terms[n-1].expon);  }  void readPoly(polynomial terms [][MAX\_TERMS], int nes[], int \*total)  {/\*read in a polynomial\*/  int i,expon, nterms;  float coef;  printf ("Enter the number of terms in your polynomial: ");  scanf("%d", &nterms);  while (nterms >= MAX\_TERMS) {  printf("Too many terms in the polynomial\n");  printf("Number of terms: ");  scanf("%d", &nterms);  }  for (i= 0; i < nterms; i++) {  printf("Coefficient: ");  scanf("%f", &coef);  printf("Exponent: ");  scanf("%d", &expon);  terms[\*total][i].coef = coef;  terms[\*total][i].expon = expon;  }  nes[\*total] = nterms;  (\*total)++;  } |

**Page 72, Exercise 5: 2-d array of polynomials**

Advantage: This implementation allows automatic processing of polynomials through the use of for statements.

Disadvantage: Using a 2 dimensional array wastes a tremendous amount of storage. This 2d array contains @64k cells!

|  |
| --- |
| /\* \*  \* Created by Susan Anderson-Freed on 10/2/07.  \*\*/  #include  #include  #define MAX\_TERMS 100  #define MAX\_POLYS 15  typedef struct {  float coef;  int expon;  } polynomial;  void readPoly(polynomial [][MAX\_TERMS], int [], int \*);  void printPoly(polynomial [], int );  int main()  {  float x;  polynomial terms[MAX\_POLYS][MAX\_TERMS];  int nes[MAX\_POLYS];  int totalPolys = 0;  readPoly(terms, nes, &totalPolys);  readPoly(terms, nes, &totalPolys);  printPoly(terms[0],nes[0]);  printPoly(terms[1],nes[1]);  }  void printPoly(polynomial terms [], int n)  {/\*print the polynomial\*/  int i;  for (i = 0;i < n-1; i++)  printf("%5.2fx^%d +", terms[i].coef, terms[i].expon);  printf("%5.2fx^%d\n", terms[n-1].coef, terms[n-1].expon);  }  void readPoly(polynomial terms [][MAX\_TERMS], int nes[], int \*total)  {/\*read in a polynomial\*/  int i,expon, nterms;  float coef;  printf ("Enter the number of terms in your polynomial: ");  scanf("%d", &nterms);  while (nterms >= MAX\_TERMS) {  printf("Too many terms in the polynomial\n");  printf("Number of terms: ");  scanf("%d", &nterms);  }  for (i= 0; i < nterms; i++) {  printf("Coefficient: ");  scanf("%f", &coef);  printf("Exponent: ");  scanf("%d", &expon);  terms[\*total][i].coef = coef;  terms[\*total][i].expon = expon;  }  nes[\*total] = nterms;  (\*total)++;  } |

**Page 84, Exercise 4 : Fast Transpose Function**

The fast transpose function achieves its efficiency through the use of two auxiliary arrays both indexed by the number of columns in the original matrix (a). Array rowTerms holds the number of columns in the original matrix (a); array startingPos holds the starting position of the column, which will become the row, in the new matrix (b). Using the example from the text, arrays a, startingPos, and rowTerms will hold the following values:

|  |  |  |
| --- | --- | --- |
| (a)  r c v  [0] 6 6 8  [1] 0 0 15  [2] 0 3 22  [3] 0 5 −15  [4] 1 1 11  [5] 1 2 3  [6] 2 3 −6  [7] 4 0 91  [8] 5 2 28 | rowTerms  [0] 1  [1] 2  [2] 2  [3] 2  [4] 0  [5] 1 | startingPos  [0] 1  [1] 2  [2] 4  [3] 6  [4] 8  [5] 8 |

Reading down the **c column of a verifies that column 2 contains two entries. By cumulating the entries in rowTerms, we obtain the starting position for each row in the transposed matrix.**

**As the book indicates, the computing time is 0(n+t) for sparse matrices and O(mn) for dense ones. Improving the efficiency of transposition requires a data structure that directly accesses the columns of a matrix.**

**Page 84, Exercise 2 : Modify fastTranspose() so it uses only 1 array.**

This modification uses a slightly different array structure for a. Ra†her than using an 1-d array of a struct, it uses a 2-d array where column 0 contains the row value, column 1 contains the column value and column 2 contains the contains of the cell. Using the example array in the book we have :

|  |
| --- |
| Modified array structure |
| Matrix a contains:  [0] [1] [2]  [0] 6 6 8  [1] 0 0 15  [2] 0 3 22  [3] 0 5 -15  [4] 1 1 11  [5] 1 2 3  [6] 2 3 -6  [7] 4 0 91  [8] 5 2 28 |

The function creates a new 2-d array, *row\_starting*, in which the original row\_terms array entries occupy column 0 and the starting\_pos entries occupy column 1.The new matrix, using the example in the book is:

|  |
| --- |
| row\_starting matrix |
| row\_starting contains:  [0] [1]  row\_terms starting\_pos  [0] 1 1  [1] 2 2  [2] 2 4  [3] 2 6  [4] 0 8  [5] 1 8 |

The new function defintion appears below. The function call is unchanged.

|  |
| --- |
| void fast\_transpose(int a[][3], int b[][3])  {/\* the transpose of a is placed in b and is found in O(n+t) time,  where n is the number of columns and t is the numbers of terms \*/  int row\_starting[MAX\_COL][2];  int i,j, num\_cols = a[0][1], num\_terms = a[0][2];  b[0][0] = num\_cols; b[0][1] = a[0][0];  b[0][2] = num\_terms;  if (num\_terms > 0) { /\* nonzero matrix \*/  for (i = 0; i <= num\_cols; i++)  row\_starting[i][0] = 0;  for (i = 1; i <= num\_terms; i++)  row\_starting[a[i][1]][0]++;  row\_starting[0][1] = 1;  for (i = 1; i <= num\_cols; i++)  row\_starting[i][1] = row\_starting[i-1][1] + row\_starting[i-1][0];  for (i = 1; i <= num\_terms; i++) {  j = row\_starting[a[i][1]][1];  b[j][0] = a[i][1]; b[j][1] = a[i][0];  b[j][2] = a[i][2];  row\_starting[a[i][1]][1]= j+1;  }  }  } |

**Page 84, Exercise 4 : Fast Transpose Function**

The fast transpose function achieves its efficiency through the use of two auxiliary arrays both indexed by the number of columns in the original matrix (a). Array rowTerms holds the number of columns in the original matrix (a); array startingPos holds the starting position of the column, which will become the row, in the new matrix (b). Using the example from the text, arrays a, startingPos, and rowTerms will hold the following values:

|  |  |  |
| --- | --- | --- |
| (a)  r c v  [0] 6 6 8  [1] 0 0 15  [2] 0 3 22  [3] 0 5 −15  [4] 1 1 11  [5] 1 2 3  [6] 2 3 −6  [7] 4 0 91  [8] 5 2 28 | rowTerms  [0] 1  [1] 2  [2] 2  [3] 2  [4] 0  [5] 1 | startingPos  [0] 1  [1] 2  [2] 4  [3] 6  [4] 8  [5] 8 |

Reading down the **c column of a verifies that column 2 contains two entries. By cumulating the entries in rowTerms, we obtain the starting position for each row in the transposed matrix.**

**As the book indicates, the computing time is 0(n+t) for sparse matrices and O(mn) for dense ones. Improving the efficiency of transposition requires a data structure that directly accesses the columns of a matrix.**

**Page 86, Exercise 3** Column Major Form

The addressing formula is: α + *j*·upper0 + *i*  
where *i* is the row subscript, *j* is the column subscript and upper0 is the size of the first  
dimension, that is, the number of rows.

|  |  |
| --- | --- |
| Example: 2x3 Matrix | |
| Address = Contents | Addressing Scheme |
| [0] = [0][0]  [1] = [1][0]  [2] = [0][1]  [3] = [1][1]  [4] = [0][2]  [5] = [1][2] | 0·2 + 0 = 0  0·2 + 1 = 1  1·2 + 0 = 2  1·2 + 1 = 3  2·2 + 0 = 4  2·2 + 1 = 5 |

**Page 98, Exercise 1** : Count the characters in a string

|  |
| --- |
| **Call:** char s1[MAX\_SIZE] = {"hello, world!"}; countChars(s1); |
| void countChars(char \*s)  { /\*Exercise 1\*/  int counts[26];  int i;  char ch;  for (i = 0; i < 26; i++) /\* set counts to 0 \*/  counts[i] =0;    for (i = 0; i < strlen(s); i++)  if ( (isupper(s[i])) || (islower(s[i]))) {  ch = toupper(s[i]); /\* make case insensitive \*/  counts[(int)ch-65]++; /\*Subtract ASCII value for A \*/  }  for (i = 0; i < 26; i++)  printf("[%c: %d] ", (char)(65+i), counts[i]);  printf("\n");  } |

**Page 98, Exercise 2** : **strndel()**

|  |
| --- |
| Call:  char s1[MAX\_SIZE] = {"hello, world!"};  strndel(s1, 5, 7, s2); |
| void strndel(char \* s, int start, int length, char \* cpy)  {/\*Exercise 2: create a string with start to length -1 characters  removed from s \*/  int i,n;  for (i = 0; i < start; i++)  cpy[i] = s[i];  n = start;  printf("start + length: %d\n", start+length);  for (i = start+length; i < strlen(s); i++)  cpy[n++] = s[i];  cpy[n] = '\0';  } |

**Page 98, Exercise 3** **strdel()**

|  |
| --- |
| Call:  strndel("doghouse",0,3,s2); |
| void strdel(char \*s, char ch)  {/\*Exercise 3: remove first occurence of c from s \*/  char cpy[MAX\_SIZE];  int i= 0;  while (i < strlen(s))  if (s[i] == ch) break;  else cpy[i] = s[i++];  if (i < strlen(s)) {  i++; /\*skip ch \*/  while (i < strlen(s))  cpy[i-1] = s[i++];  }  cpy[i] = '\0';  strcpy(s,cpy);  } |

**Page 98, Exercise 4** : **strpos1()**

|  |
| --- |
| Call:  pos = strpos1(s1,ch);  if (pos >= 0)  printf("%c found at position %d\n", ch, pos);  else  printf ("%c is not in the string\n", ch); |
| int strpos1 (char \*s, char c)  {/\* return the position of c in s  This version is case sensitive  Exercise 4 \*/  int i;  for (i = 0; i < strlen(s); i++)  if (c == s[i]) return i;  return -1;  } |

**Page 98, Exercise 5 : strchr1() function**

|  |
| --- |
| char s1[MAX\_SIZE] = {"hello, world!"};  char \*chPtr;chPtr = strchr1(s1,ch);  if (chPtr)  printf("%c found at position %s\n", ch, chPtr); |
| char \*strchr1 (char \*s, char c)  { /\*return a pointer of c in s  This version is case sensitive  Exercise 5 \*/  int i;  for (i = 0; i < strlen(s); i++)  if (c == s[i]) return &s[i];  return NULL;  } |

**Page 98, Exercise 6 :Rewrite Program 2.12 without temp**

|  |
| --- |
| void strnins(char \*s, char \*t, int i)  {/\* revised version of program 2.12 \*/  char string[MAX\_SIZE];  if( (i < 0) || (i > strlen(s)))  printf("Position is out of bounds \n");  else {  if (!(strlen(s)))  strcpy(s,t);  else if (strlen(t)) {  strncpy(string, s,i);  string[i]='\0';  printf("After strncpy: %s\n", string);  strcat(string,t); printf("After strcat: %s\n", string);  strcat(string, (s+i));  strcpy(s, string);  }  }  } |

**Page 98, Exercise 9 : failure function**

String: ababdababc

Failure function for (aaaab) is: -1 0 1 2 3

Failure function for (ababaa) is: -1 0 0 1 2 3

Failure function for (abaabaab) is: -1 0 0 1 1 2 3 4

**Page 99, Exercise 3**

To illustrate the addressing formula for lower triangular matrices, we’ll use the following table. It contains a 4x4 matrix with the lower triangular portion highlighted. It also shows the mapping of this matrix into array .

Lower Triangular Matrix B [1] [2] [3]   
(0,0)(0,1)(0,2)(0,3) [0] 0 0 Value   
(1,0)(1,1)(1,2)(1,3) [1] 1 0 Value  
(2,0)(2,1)(2,2)(2,3) [2] 1 1   
(3,0)(3,1)(3,2)(3,3) [3] 2 0   
[4] 2 1   
[5] 2 2   
[6] 3 0   
[7] 3 1   
[8] 3 2   
[9] 3 3

To obtain the addressing formula, we notice the entries/row. For row 0, we have 1 entry; for row 1 we have 2 entries; for row 3 we have 3 entries, and so on. So to obtain the offset to the row, we simply count from 0 to the row add the entries. If the row subscript is the same as the column subscript, we’re done. If the row subscript is less than the column subscript we add the column subscript. So the addressing formula written as code is:

base = 0;  
for (i = 0; i <= row; i++)   
base += i;  
if (row == column)  
base += row;  
else base = base + column;

Upper Triangular System  
(0,0)(0,1)(0,2)(0,3) [0] 0 0 Value   
(1,0)(1,1)(1,2)(1,3) [1] 0 1 Value  
(2,0)(2,1)(2,2)(2,3) [2] 0 2   
(3,0)(3,1)(3,2)(3,3) [3] 0 3   
[4] 1 1   
[5] 1 2   
[6] 1 3   
[7] 3 1   
[8] 2 2   
[9] 3 3

For the upper triangular system, we begin with the maximum row subscript and add one to obtain the number of entries on that row. So if the max subscript is 3, there will be four entries on the first row. The next row will have 3 entries and so on. We add the entries for each row to give the highest subscript for that row. To obtain the column offset we subtract the column position from the row entries, and then subtract that result from the base address. The address formula written as C code is:

entries = maxRow+1; base = 0; /\*number of entries on that row \*/  
for (i = row; i >=0; i--)   
base = base + entries--;  
base = base - (maxRow + 1 - column);

**Page 100, Exercise 5 : Triadiagonal Matrix**

|  |  |
| --- | --- |
| Sample Matrix | B |
| [0] [1] [2] [3]  [0] 1 1 0 0  [1] 1 1 1 0  [2] 0 1 1 1  [3] 0 0 1 1 | [0] = a[0][0]  [1] = a[0][1]  [2] = a[1][0]  [3] = a[1][1]  [4] = a[1][2]  [5] = a[2][1]  [6] = a[2][2]  [7] = a[2][3]  [8] = a[3][2]  [9] = a[3][3] |

|  |
| --- |
| Addressing Algorithm |
| if (row == 0)  b[i] = column  else {  i = 2;  for (j = 1; j < row; j++)  i+= j;  if (column == row) i++;  if (column == row+1) i+= 2; |

# CHAPTER 3

**Page 119, Exercise 1**

**Array Queue Implementation:** Since all of the work is done in the addq() and deleteq() functions, we only need to print and error message and exit on error or return. Here’s the code with the addq() and deleteq() functions.

|  |
| --- |
| void queue\_empty()  {/\* check for empty queue is in the deleteq() function \*/  fprintf(stderr, “The queue is empty\n”);  }  void queue\_full()  {/\* code to check for a full queue is in the addq() function \*/  fprintf(stderr, “The queue is Full\n”);  }  void addq(element item)  {/\* add an item to the global queue rear points to the current end of the queue \*/  if (rear == MAX\_QUEUE\_SIZE-1)  queue\_full();  else  queue[++rear] = item;  }  element deleteq()  {/\* remove element at the front of the queue \*/  if (front == rear)  queue\_empty();  else  return queue[++front];  } |

**Problem 2, p 119**. Similarly all the work for the circular queue is done in the addq() and deleteq() methods. Here’s the two methods with the code for printing error message when the queue is empty or full. You can also exit on error, but this makes it difficult to check the correctness of the code to add or delete.

|  |
| --- |
| void queue\_full(){ printf(“The queue is Full \n”);}  void queue\_empty(){ printf(“The queue is empty\n”);}  void addq(element item)  {/\* add an item to the global queue front and rear mark the two queue ends \*/  if ((rear+1 == front) || ((rear == MAX\_QUEUE\_SIZE-1) && !front))  queue\_full();  else {  queue[rear] = item;  rear = (rear+1) % MAX\_QUEUE\_SIZE;  }  }  element deleteq()  {/\*delete and element from the circular queue \*/  int i;  element temp;  /\* remove front element from the queue and put it in item \*/  if (front == rear) queue\_empty();  else {  temp = queue[front];  front = (front+1) % MAX\_QUEUE\_SIZE;  return temp;  }  } |

**Page 119, Exericse 3**

If we start with a full queue, remove an item from the front, and shift the remaining items downward by one array position, it will take O(MAX\_QUEUE\_SIZE) for each delete operation. If we immediately and repeatedly, add one item to the rear of the queue and remove the item at the front, each add/delete combination will take O(MAX\_QUEUE\_SIZE) .

**Page 119, Exercise 4 - Deque**

|  |
| --- |
| Declarations |
| #define MAX\_SIZE 6  typedef struct {  int key; } element;  element deque[MAX\_SIZE]; /\* global queue declaration \*/  int rear = -1;  void deque\_empty()  { fprintf(stderr, "The deque is empty\n");}  void deque\_full()  {fprintf(stderr, "The deque is Full\n");} |

|  |
| --- |
| Adding to the front |
| void addFront(element item)  {/\* add an item to the front of the dequeu \*/  int i;  if (rear == MAX\_SIZE-1)  deque\_full();  else {  rear++;  if (rear > 0) /\* shift if deque isn't empty \*/  for (i = rear; i > 0; i--)  deque[i] = deque[i-1];  deque[0] = item;  }  } |

|  |
| --- |
| Adding to the rear |
| void addRear(element item)  {/\* add an item to the high end of the global deque  rear points to the current end of the deque \*/  if (rear == MAX\_SIZE-1)  deque\_full();  else  deque[++rear] = item;  } |

|  |
| --- |
| Deleting from the front |
| element deleteFront()  {/\* remove element at the front of the deque \*/  int i;  if (rear < 0)  deque\_empty();  else { /\* shift downward \*/  element item = deque[0];  for (i = 1; i <= rear; i++)  deque[i-1] = deque[i];  rear--;  return item;  }  } |

|  |
| --- |
|  |
| Deleting from the rear  element deleteRear()  {/\* remove element at the high end of the deque \*/  if (rear < 0)  deque\_empty();  else  return deque[rear--];  } |

|  |
| --- |
| Driver |
| printf("1. Add to front, 2. Delete front, 3. Add to rear, 4. Delete rear, 0. Quit: ");  scanf("%d",&choice);  while (choice > 0) {  switch (choice) {  case 1: printf("Enter the number to insert: ");  scanf("%d",&item.key);  addFront(item);  break;  case 2: item = deleteFront();  if (rear >= 0)  printf ("%d was deleted from the queue.\n\n", item.key);  break;  case 3: printf("Enter the number to insert: ");  scanf("%d",&item.key);  addRear(item);  break;  case 4: item = deleteRear();  if (rear >= 0)  printf("%d was deleted from the stack \n\n",item.key);  break;  } |

**Page 127, Exericse 1** -- Horizontal & Vertical Borders

The following table shows the maze of Figure 3.8 drawn with only horizontal and vertical borders allowed.

|  |
| --- |
| Maze with Horizontal and Vertical Borders |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp3/mazeHV2.jpg |

Horizontal and vertical borders do not effect the ability to move in any direction, so the allowable moves is unaltered.

|  |
| --- |
| Allowable Moves |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp3/allowableHV.jpg |

|  |
| --- |
| Move Table |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp3/moveTableHV.jpg |

**Page 127, Exericse 2** -- Horizontal & Vertical Borders And Borders at 45 and 135 degree angles.

The following table shows the maze of Figure 3.8 drawn with only horizontal and vertical borders allowed.

|  |
| --- |
| Maze with Horizontal, Vertical Borders and 45, 135 degree New Borders in green |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp3/maze45.jpg |

The addition of the borders at 45 and 135 degrees places restrictions on the allowable moves. The new move move table is:

|  |
| --- |
| Allowable Moves |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp3/allowable45.jpg |

Notice that we can no longer move in the NE or SW directions.

**Page 127, Exericse 4**

The following table shows the maze matrix redrawn as a maze with my path through. Redrawn as a maze the path through it is fairly obvious.

|  |
| --- |
| Maze Path |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp3/mazePathOrig.jpg |

Because I can see the maze and its a simple one, I can take a direct path from start to finish. In contrast, the program tries the moves and backtracks when it reaches a dead end. Its approach is basically trial and error.

**Page 127, Exericse 5** -- Maze

|  |
| --- |
| Maze Program |
| #include <stdio.h>  #define NUM\_ROWS 5  #define NUM\_COLS 3  #define BOUNDARY\_COLS 5  #define MAX\_STACK\_SIZE 100  #define FALSE 0  #define TRUE 1  typedef struct {  short int row;  short int col;  short int dir;  } element;  element stack[MAX\_STACK\_SIZE]; /\* global stack declaration \*/  typedef struct {  short int vert;  short int horiz;  } offsets;  offsets move[9]; /\* array of moves for each direction \*/  static short int maze[][BOUNDARY\_COLS] = {{1,1,1,1,1}, /\* top boundary \*/  {1,0,0,0,1},  {1,1,1,0,1},  {1,0,0,0,1},  {1,0,1,1,1},  {1,0,0,0,1},  {1,1,1,1,1}}; /\* bottom boundary \*/  short int mark[][BOUNDARY\_COLS] = {{0,0,0,0,0},  {0,0,0,0,0},  {0,0,0,0,0},  {0,0,0,0,0},  {0,0,0,0,0},  {0,0,0,0,0},  {0,0,0,0,0}};  int top;  void init\_move();  void add(element);  element delete();  void stack\_full();  void stack\_empty();  void path();  void print\_record(int,int,int);  void print\_maze();  int main ()  {/\* stack represented as an array \*/  init\_move();  print\_maze();  path();  }  void init\_move()  {/\* initial the table for the next row and column moves \*/  move[1].vert = -1; move[1].horiz = 0; /\* N \*/  move[2].vert = -1; move[2].horiz = 1; /\* NE \*/  move[3].vert = 0; move[3].horiz = 1; /\* E \*/  move[4].vert = 1; move[4].horiz = 1; /\* SE \*/  move[5].vert = 1; move[5].horiz = 1; /\* S \*/  move[6].vert = 1; move[6].horiz = 0; /\* SW \*/  move[7].vert = 0; move[7].horiz = -1; /\* W \*/  move[8].vert = -1; move[8].horiz = -1; /\* NW \*/  }  void print\_maze()  {/\* print out the maze \*/  int i,j;  printf("Your maze, with the boundaries is: \n\n");  for (i = 0; i <= NUM\_ROWS+1; i++) {  for(j = 0; j <= NUM\_COLS+1; j++)  printf("%3d",maze[i][j]);  printf("\n");  }  printf("\n");  }  void stack\_full()  {  printf("The stack is full. No item added \n");  }  void stack\_empty()  {  printf("The stack is empty. No item deleted \n");  }  void add(element item)  { /\* add an item to the global stack  top (also global) is the current top of the stack,  MAX\_STACK\_SIZE is the maximum size \*/  if (top == MAX\_STACK\_SIZE)  stack\_full();  else  stack[++top] = item;  }  element delete()  { /\* remove top element from the stack and put it in item \*/  if (top < 0)  stack\_empty();  else  return stack[top--];  }  void print\_record(int row, int col, int dir)  { /\* print out the row, column, and the direction, the direction  is printed out with its numeric equivvalent \*/  printf("%2d %2d%5d", dir,row, col);  switch (dir-1) {  case 1: printf(" N");  break;  case 2: printf(" NE");  break;  case 3: printf(" E ");  break;  case 4: printf(" SE");  break;  case 5: printf(" S ");  break;  case 6: printf(" SW");  break;  case 7: printf(" W ");  break;  case 8: printf(" NW");  break;  }  printf("\n");  }  void path()  {/\* output a path through the maze if such a path exists,  the maze is found in positions 1 to NUM\_ROWS and 1 to NUM\_COLS.  Rows 0 and NUM\_ROWS+1 serve as boundaries, as do Columns  0 and NUM\_COLS+1. \*/  int i, row, col, next\_row, next\_col, dir, found = FALSE;  element position;  mark[1][1] = 1;  /\* place the starting position, maze[1][1] onto the stack  starting direction is 2 \*/  top = 0;  stack[0].row = 1; stack[0].col = 1; stack[0].dir = 2;  while (top > -1 && !found) {  /\* remove position at top of stack, and determine if  there is a path from this position \*/  position = delete();  row = position.row; col = position.col; dir = position.dir;  while (dir <= 8 && !found) {  /\* check all of the remaining directions from the current  position \*/  next\_row = row + move[dir].vert;  next\_col = col + move[dir].horiz;  if (next\_row == NUM\_ROWS && next\_col == NUM\_COLS)  /\* path has been found, exit loop and print it out \*/  found = TRUE;  else if ( !maze[next\_row][next\_col]  && !mark[next\_row][next\_col]) {  /\* current position has not been checked, place it  on the stack and continue \*/  mark[next\_row][next\_col] = 1;  position.row = row; position.col = col;  position.dir = ++dir;  add(position);  row = next\_row; col = next\_col; dir = 1;  }  else  ++dir;  }  }  if (!found)  printf("The maze does not have a path\n");  else {  /\* print out the path which is found in the stack \*/  printf("The maze traversal is: \n\n");  printf("dir# row col dir\n\n");  for (i = 0; i <= top; i++)  print\_record(stack[i].row, stack[i].col, stack[i].dir);  printf(" %2d%5d\n",row,col);  printf(" %2d%5d\n",NUM\_ROWS,NUM\_COLS);  }  } |

**p136, Exercise 1**

(a) ab\*c\*  
(b) a-b+c-d+  
(c) a\*b-c+  
(d) ab+d\*efad\*+/+c+

**Problem 2, p 137**

|  |
| --- |
| void PrintToken(priority token)  {/\* print out the character equivalent of the token \*/  switch (token) {  case plus: printf(“+”);  break;  case minus: printf(“-”);  break;  case divide: printf(“/”);  break;  case times: printf(“\*”);  break;  case mod: printf(“%”);  }  } |

**Page 142, Exercise 1**

The first stack starts at the low end of the memory and grows upward. The second stack starts at the high end of memory and grows downward. The memory is full when the stack pointers collide. The declarations are:

|  |
| --- |
| Stack declarations |
| #define MAX\_SIZE 6  typedef struct {  int key; } element;  element memory[MAX\_SIZE]; /\* global queue declaration \*/  int top [2];  top[0] = -1;  top[1] = MAX\_SIZE;  CALLS: printf("1. Insert stack 0, 2.Delete stack 0, 3. Insert Stack 1, 4. Delete Stack 1, 0. Quit: ");  scanf("%d",&choice);  while (choice > 0) {  switch (choice) {  case 1: printf("Insert in stack 0: ");  scanf("%d",&item.key);  add(0, item);  break;  case 2: item = delete(0);  if (top[0] >= 0 )  printf ("%d was deleted from the stack 0.\n\n", item.key);  break;  case 3: printf("Enter the number to insert: ");  scanf("%d",&item.key);  add(1, item);  break;  case 4: item = delete(1);  if (top[1] < MAX\_SIZE)  printf("%d was deleted from the stack 1 \n\n",item.key);  break;  } |

|  |
| --- |
| Adding to a stack |
| void StackFull()  {  printf("The stack is full. No item added \n");  }  void add(int topNo, element item)  {/\* add an item to the global stack  top (also global) is the current top of the stack,  MAX\_SIZE is the maximum size \*/  if (top[0]+1 >= top[1])  StackFull();  else {  if (!topNo)  memory[++top[0]] = item;  else  memory[--top[1]] = item;  }  } |

|  |
| --- |
| Deleting from a stack |
| void StackEmpty()  {  printf("The stack is empty. No item deleted \n");  }  element delete(int topNo)  {/\* remove top element from the stack and put it in item \*/  if (!topNo) {  if (top[0] < 0)  StackEmpty();  return memory[top[0]--];  }  else { /\*second stack \*/  if (top[1] == MAX\_SIZE)  StackEmpty();  return memory[top[1]++];  }  } |

**Page 142, Exercise 2**

The queue starts at the low end of the memory and grows upward. Only a rear pointer is needed because when an item is deleted the queue is shifted downward. The queue is full when the rear and top pointers meet.

The queue operations are:

|  |
| --- |
| Adding to the queue |
| void queue\_full()  {/\* code to check for a full queue is in the  addq() function \*/  fprintf(stderr, "The queue is Full\n");  }  void addq(element item)  {/\* add an item to the global queue  rear points to the current end of the queue \*/  if (rear >= top-1)  queue\_full();  else  memory[++rear] = item;  } |

|  |
| --- |
| Deleting from the queue |
| void queue\_empty()  {/\* check for empty queue is in the deleteq() function \*/  fprintf(stderr, "The queue is empty\n");  }  element deleteq()  {/\* remove element at the front of the queue \*/  int i;  element item = memory[0];  if (!rear)  queue\_empty();  else { /\* shift downward \*/  for (i = 1; i <= rear; i++)  memory[i-1] = memory[i];  rear--;  return item;  }  } |

The stack starts at the high end of memory and grows downward. It is also full when the rear and top pointers meet. The operations are:

|  |
| --- |
| Adding to the stack |
| void add(element item)  {/\* add an item to the global stack  top (also global) is the current top of the stack,  MAX\_SIZE is the maximum size \*/  if (top <= rear+1)  StackFull();  else  memory[--top] = item;  } |

|  |
| --- |
| Deleting from the Stack |
| void StackEmpty()  {  printf("The stack is empty. No item deleted \n");  }  element delete()  {/\* remove top element from the stack and put it in item \*/  if (top == MAX\_SIZE)  StackEmpty();  else  return memory[top++];  } |

**Page 142, Exericse 4**

Assuming a simple stack implementation such as the one given in Page 142, Exercise 2, there is no combination that will produce O(MAX\_STACK\_SIZE). Since both the push and pop operations require a time of O(1), this is the time for each operation using two stacks with one starting at each end of the array.

# CHAPTER 4

**Page 154, Exercise 1**

|  |
| --- |
| The delete() function with 2 pointers |
| void delete(listPointer \*first, listPointer trail)  {/\* deletefrom the list, trail is the preceding node  and \*first is the front of the list \*/  if (trail)  trail->link = (\*first)->link;  else  \*first = (\*first)->link;  free(x);  } |

**Page 154, Exercise 2**

|  |
| --- |
| search() function |
| Call:  if (search(ptr,key))  printf("The key is in the list\n");  else  printf("The key is not in the list\n"); |
| list\_pointer search(list\_pointer ptr, int key)  {/\* determine if key is in the list \*/  list\_pointer temp;  for (temp = ptr; temp; temp = temp->link)  if (temp->item.key == key) return temp;  return NULL;  } |

**Page 154, Exercise 3**

|  |
| --- |
| Delete Number from a List |
| printf("Enter the number to delete: ");  scanf("%d",&num);  ptr = Deletelist(ptr,num,&found);  if (found)  printf("The item was deleted from the list \n\n");  else  printf("The item was not found in the list\n\n"); |
| listpointer Deletelist(listpointer ptr, int searchNum, int \*found)  /\* search for an element, delete it if it is in the list  The pointer to the head of the list is returned in the function name,  the value of found will be 1, if the entry was deleted, and 0  otherwise \*/  {  listpointer position,temp;  position = search(ptr,searchNum, found);  if (\*found) {  if (!position) {  /\* entry was found at the head of the list, delete  the current head pointer, and return the link field  as the new pointer to the head of the list \*/  temp = ptr->link;  free(ptr);  return temp;  }  else {  /\* entry was not at the head of the list, change the  link pointers and free the storage \*/  temp = position->link;  position->link = temp->link;  free(temp);  return ptr;  }  }  else  /\* item was not found in the list, return the pointer to  the head of the list \*/  return ptr;  } |

|  |
| --- |
| Modification of search that returns True/false |
| listpointer search(listpointer ptr, int searchNum, int \*found)  {  /\* determine if searchNum is in the list, found will hold a value  of TRUE (1) if the item was located, otherwise it will hold a value  of FALSE (0). The function name will hold the pointer to the entry prior  to the one to be deleted. \*/  listpointer lead,trail;  trail = NULL; /\* entry prior to current one \*/  lead = ptr; /\* entry currently being examined \*/  \*found = FALSE;  while (lead && !\*found)  if (lead->data == searchNum)  \*found = TRUE;  else {  trail = lead;  lead = lead->link;  }  return trail;  } |

**Page 154, Exercise 4**

|  |
| --- |
| length() function |
| Call:  printf("The list contains %4d elements:\n\n",length(ptr)); |
| int length(list\_pointer ptr)  {/\* find the length of the list \*/  list\_pointer temp;  int size;  size = 0;  for (temp=ptr; temp; temp = temp->link)  size++;  return size;  } |

**Page 159, Exercise 1**

|  |
| --- |
| Function to convert string |
| void convert\_string(char \*expr , char cleaned[MAX\_STRING\_SIZE])  {/\* change to all lower case  remove spaces and punctuation \*/  int i=0, count=0;  char ch;  while (i < strlen(expr)) {  if (isspace(expr[i])) i++;  else if (ispunct(expr[i])) i++;  else {  ch = expr[i++];  cleaned[count++] = tolower(ch);  }  }  cleaned[count] = '\0';  } |

|  |
| --- |
| The palindrome parser() function |
| int parser(char expr [])  {  int i=0, half, odd;  char ch;  stack\_ptr top = NULL;  half = (int)strlen(expr)/2;  odd = strlen(expr)%2;  while (i < half) /\* push half the letters onto the stack \*/  push(&top, expr[i++]);  if (odd) i++;  while (i < strlen(expr))  {/\* pop character from stack and compare with next character in the string \*/  ch = pop(&top);  if (expr[i] != ch)  {/\* not a palindrome, empty out stack and return false \*/  while (!IS\_EMPTY(top)) {  ch = pop(&top);  }  return FALSE;  }  i++;  return TRUE;  }  } |

|  |
| --- |
| main() Function |
| int main()  {  char expr [MAX\_STRING\_SIZE];  char parse\_string [MAX\_STRING\_SIZE];  int i, half, odd, ok\_pal;  stack\_ptr top = NULL;  do {  printf("Expression: ");  /\* use fgets to read in strings with spaces,  strip off the new line character to get the correct count \*/  fgets(expr, MAX\_STRING\_SIZE,stdin);  if (expr[strlen(expr)-1] == '\n')  expr[strlen(expr)-1] = '\0';  if(strlen(expr)) {  convert\_string(expr, parse\_string);  printf("cleaned %s\n", parse\_string);  if(ok\_pal = parser(parse\_string))  printf("%s IS a palindrome.\n\n", expr);  else  printf("%s IS NOT a palindrome.\n\n", expr);  }  }while (strlen(expr));  } |

**Page 160, Exercise 2**

|  |
| --- |
| int parser(char expr [])  {/\*parser for parethesis language  if left paren, bracket or brace, push  if right paren, bracket, or brace pop and compare  if not a match return false. \*/  int i;  char ch;  stack\_ptr top = NULL;  for (i = 0; i < strlen(expr); i++)  switch (expr[i]) {  case '[': case '{': case '(': push(&top, expr[i]);  break;  case ']': if (IS\_EMPTY(top))  return FALSE;  ch = pop(&top);  if (ch != '[') return FALSE;  break;  case '}': if (IS\_EMPTY(top))  return FALSE;  ch = pop(&top);  if (ch != '{') return FALSE;  break;  case ')': if (IS\_EMPTY(top))  return FALSE;  ch = pop(&top);  if (ch != '(') return FALSE;  }  if (IS\_EMPTY(top)) /\*string is finished, stack contains items \*/  return TRUE;  return FALSE;  } |

**Page 170, Exercise 1**

|  |
| --- |
| Call: a = ReadPoly(); |
| PolyPointer ReadPoly()  {/\*read the polynomial into a chain \*/  PolyPointer front, rear,temp;  float coefficient;  int exponent;  front=rear=NULL;  printf("Enter an exponent Less than 0 to quit: \n");  printf("Coefficient, Exponent: ");  scanf("%f,%d",&coefficient,&exponent);  while (exponent >= 0) {  temp = (PolyPointer)malloc(sizeof(struct PolyNode));  temp->coef = coefficient;  temp->expon = exponent;  temp->link = NULL;  if (!front) front = temp;  else rear->link = temp;  rear = temp;  printf("Coefficient, Exponent: ");  scanf("%f,%d",&coefficient,&exponent);  }  return front;  } |

**Page 170, Exercise 3**

|  |
| --- |
| Call: result = evalPoly(x0,a) |
| float evalPoly(float x0, PolyPointer ptr)  {/\*evaluate the polynomial at point x \*/  PolyPointer temp;  float result = 0;  for (temp = ptr; temp; temp= temp->link)  result = result + temp->coef \* pow(x0,temp->expon);  return result;  } |

**Page 170, Exercise 4**

|  |
| --- |
| Call: a = ReadPoly(); |
| PolyPointer ReadPoly()  {/\*read in the polynomial \*/  PolyPointer node,c;  float coefficient;  int exponent;  node = GetNode();  node->coef = -1.0;  node->expon = -1;  node->link = node;  printf("Enter an exponent < 0 to quit: ");  printf("\nCoefficient, Exponent: ");  scanf("%f,%d",&coefficient,&exponent);  while (exponent >= 0) {  c = GetNode();  c->coef = coefficient;  c->expon = exponent;  c->link = node->link;  node->link = c;  printf("Coefficient, Exponent: ");  scanf("%f,%d",&coefficient,&exponent);  }  return node;  } |

**Page 170, Exercise 6**

|  |
| --- |
| Call: printf("\nAt %f the polynomial is: %5.2f\n", x0, evalPoly(x0,a)); |
| float evalPoly(float x0, PolyPointer ptr)  {/\*evaluate the polynomial \*/  PolyPointer c;  float result = 0;  for (c = ptr->link; c != ptr; c = c->link) {  result = result + c->coef\*pow(x0, c->expon);  printf("%f, %d\n", c->coef,c->expon);  }  return result;  } |

**Page 170, Exercise 7**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <math.h>  #define FALSE 0  #define TRUE 1  #define COMPARE(x,y) ((x) < (y) ? -1 : (x) == (y) ? 0 : 1)  typedef struct PolyNode \*PolyPointer;  struct PolyNode {  float coef;  int expon;  PolyPointer link;  };  void attach(int, int, PolyPointer \*);  PolyPointer GetNode();  void RetNode(PolyPointer);  void cerase(PolyPointer \*);  PolyPointer av = NULL;  PolyPointer ReadPoly();  void PrintPoly(PolyPointer);  PolyPointer cpadd(PolyPointer, PolyPointer);  float evalPoly(float, PolyPointer);  int main()  {  PolyPointer a,b,c;  float x0;  printf("Polynomial A: \n");  a = ReadPoly();  printf("Polynomial B: \n");  b = ReadPoly();  printf("Your polynomials are: \n");  printf("Polynomial a: "); PrintPoly(a);  printf("\nPolynomial b: "); PrintPoly(b);  c = cpadd(a,b);  printf("\n a+b = "); PrintPoly(c);  printf("\n\nx0:");  scanf("%f", &x0);  printf("\nAt %f the polynomial is: %5.2f\n", x0, evalPoly(x0,a));  }  float evalPoly(float x0, PolyPointer ptr)  {/\*evaluate the polynomial \*/  PolyPointer c;  float result = 0;  for (c = ptr->link; c != ptr; c = c->link) {  result = result + c->coef\*pow(x0, c->expon);  printf("%f, %d\n", c->coef,c->expon);  }  return result;  }  PolyPointer ReadPoly()  {/\*read in the polynomial \*/  PolyPointer node,c;  float coefficient;  int exponent;  node = GetNode();  node->coef = -1.0;  node->expon = -1;  node->link = node;  printf("Enter an exponent < 0 to quit: ");  printf("\nCoefficient, Exponent: ");  scanf("%f,%d",&coefficient,&exponent);  while (exponent >= 0) {  c = GetNode();  c->coef = coefficient;  c->expon = exponent;  c->link = node->link;  node->link = c;  printf("Coefficient, Exponent: ");  scanf("%f,%d",&coefficient,&exponent);  }  return node;  }  void PrintPoly(PolyPointer ptr)  {/\*write out the polynomial \*/  PolyPointer c;  for (c = ptr->link; c != ptr; c = c->link)  printf("<%5.2fx^%d>, ",c->coef,c->expon);  }  PolyPointer GetNode()  /\* provide a node for use \*/  {  PolyPointer node;  if (!av)  node = (PolyPointer)malloc(sizeof(struct PolyNode));  else {  node = av;  av = av->link;  }  return node;  }  void attach(int coefficient, int exponent, PolyPointer \*ptr)  {/\* create a new node with coef = coefficient and expon = exponent,  attach it to the node pointed to by ptr. ptr is updated to point  to this new node \*/  PolyPointer node;  node = GetNode();  node->coef = coefficient;  node->expon = exponent;  (\*ptr)->link = node;  \*ptr = node;  }  PolyPointer cpadd(PolyPointer a, PolyPointer b)  {/\* polynomials a and b are singly linked lists, return  a polynomial which is the sum of a and b \*/  PolyPointer startA, c, lastC;  int num, done = FALSE; /\* jump out of switch and do loop \*/  startA = a; /\* record start of a \*/  a = a->link; /\* skip head node for a and b\*/  b = b->link;  c = GetNode(); /\* get a head node for c \*/  c->expon = -1; c->coef = -1;  lastC = c;  do {  switch (COMPARE(a->expon, b->expon)) {  case -1: /\* a->expon < b->expon \*/  attach(b->coef,b->expon,&lastC);  b=b->link;  break;  case 0: /\* a->expon = b->expon \*/  if (startA == a)  done = TRUE;  else {  num = a->coef + b->coef;  if (num) {  attach(num,a->expon,&lastC);  a = a->link;  b = b->link;  }  }  break;  case 1: /\* a->expon > b->expon \*/  attach(a->coef,a->expon,&lastC);  a = a->link;  break;  }  } while (!done);  lastC->link = c;  return c;  } |

**Page 172, Exercise 1**

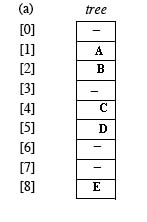
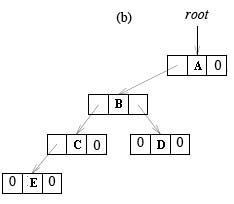
|  |
| --- |
| Call:  if (searchList(x,ptr))  printf("%d is in the list.\n", x);  else printf("%d is Not in the list.\n",x); |
| listPointer searchList(int x, listPointer ptr)  {/\* print out the contents of the list \*/  listPointer temp = ptr;  do {  if (temp->data == x) return temp;  temp=temp->link;  } while (temp != ptr);  return NULL;  } |

**Page 186, Exercise 7**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 25  typedef enum {head,entry} tagfield;  typedef struct MatrixNode \*MatrixPointer;  struct EntryNode {  int row;  int col;  int value;  };  struct MatrixNode {  MatrixPointer down;  MatrixPointer right;  tagfield tag;  union {  MatrixPointer next;  struct EntryNode entry;  } u;  };  MatrixPointer HdNode[MAX\_SIZE];  MatrixPointer mread(void);  void mwrite(MatrixPointer);  void merase(MatrixPointer \*node);  int main()  {  MatrixPointer a,b,c;  a = mread();  mwrite(a);  merase(&a);  if (!a)  printf("The matrix has been erased.\n");  }  MatrixPointer mread(void)  {/\* read in a matrix and set up its linked representation.  An auxilliary global array HdNode is used \*/  int NumRows, NumCols, NumEntries, NumHeads, i;  int row, col, value, CurrentRow;  MatrixPointer temp,last,node;  printf("Enter the number of rows, columns and entries: ");  scanf("%d,%d,%d",&NumRows, &NumCols, &NumEntries);  NumHeads = (NumCols > NumRows) ? NumCols : NumRows;  /\* set up head node for the list of head nodes \*/  node = (MatrixPointer)malloc(sizeof(struct MatrixNode));  node->tag = entry;  node->u.entry.row = NumRows;  node->u.entry.col = NumCols;  if (!NumHeads)  node->right = node;  else {  /\* initialize the head nodes \*/  for (i = 0; i < NumHeads; i++) {  temp = (MatrixPointer)malloc(sizeof(struct MatrixNode));  HdNode[i] = temp;  HdNode[i]->tag = head;  HdNode[i]->right = temp;  HdNode[i]->u.next = temp;  }  CurrentRow = 0;  last = HdNode[0];  for (i = 0; i < NumEntries; i++) {  printf("Enter row, column and value: ");  scanf("%d,%d,%d",&row,&col,&value);  if (row > CurrentRow) {  last->right = HdNode[CurrentRow];  CurrentRow = row;  last = HdNode[row];  }  temp = (MatrixPointer)malloc(sizeof(struct MatrixNode));  temp->tag = entry;  temp->u.entry.row = row;  temp->u.entry.col = col;  temp->u.entry.value = value;  last->right = temp; /\* link into row list \*/  last = temp;  HdNode[col]->u.next->down = temp; /\* link into column list \*/  HdNode[col]->u.next = temp;  }  /\*close last row \*/  last->right = HdNode[CurrentRow];  /\* close all column lists \*/  for (i = 0; i < NumCols; i++)  HdNode[i]->u.next->down = HdNode[i];  /\* link all head nodes together \*/  for (i = 0; i < NumHeads-1; i++)  HdNode[i]->u.next = HdNode[i+1];  HdNode[NumHeads-1]->u.next = node;  node->right = HdNode[0];  }  return node;  }  void mwrite(MatrixPointer node)  {  /\* print out the matrix in row major form \*/  int i;  MatrixPointer temp;  /\* matrix dimensions \*/  printf("\n\nNumRows = %d, NumCols = %d\n",node->u.entry.row,  node->u.entry.col);  printf(" The matrix by row, column, and value: \n\n");  for (i = 0; i < node->u.entry.row; i++)  /\* print out the entries in each row \*/  for (temp = HdNode[i]->right; temp != HdNode[i]; temp = temp->right)  printf("%5d%5d%5d\n",temp->u.entry.row,temp->u.entry.col,  temp->u.entry.value);  }  void merase(MatrixPointer \*node)  {  /\* erase the matrix, return the pointers to the heap \*/  MatrixPointer x,y;  int i, NumHeads;  /\* free the entry pointers by row \*/  for (i = 0; i < (\*node)->u.entry.row; i++) {  y = HdNode[i]->right;  while (y != HdNode[i]) {  x = y;  y = y->right;  free(x);  }  }  /\* determine the number of head nodes and free these pointers \*/  NumHeads = ((\*node)->u.entry.row > (\*node)->u.entry.col) ?  (\*node)->u.entry.row : (\*node)->u.entry.col;  for (i = 0; i < NumHeads; i++)  free(HdNode[i]);  \*node = NULL;  } |

# CHAPTER 5

**Page 201, Exercise 4**

* A: Level 1, Non-Leaf Node (root of tree)
* B: Level 2, Non-Leaf Node
* C: Level 3, Non-Leaf Node
* D: Level 3, Leaf Node
* E: Level 4, Leaf Node
* **Page 204, Exercise 3**
* 
* 

**Page 211, Exercise 1 (a)**

* Preorder: A, B, C, D, E
* Inorder: E, D, C, B, A
* PostOrder: E,D, C, B,A
* LevelOrder: A, B, C, D, E

**Page 211, Exercise 1 (b)**

* Preorder: A, B, D, H, I, E, C, F, G
* Inorder: H, D, I, B, E, A, F, C, G
* PostOrder: H, I, D, E, B, F, G, C, A
* LevelOrder: A, B, C, D,E,F,G,H, I

**Page 211, Exercise 3**

* Preorder: A, B, C, E, D
* Inorder: E, C, B, D, A
* PostOrder: E, C, D, B, A
* LevelOrder: A, B, C, D, E

**Page 216, Exercise 3**

Since PostOrderEval is only a variation of the PostOrder traversal, its computing time is the same as that of the standard   
PostOrder traversal: O(n).

**Page 228, Exercise 1**

The Max Priority Queue is listed in array order:

* Insert 7 : 7
* Insert 16: 16, 7
* Insert 49: 49, 7, 16
* Insert 82: 82, 49, 16, 7
* Insert 5: 82, 49, 16, 7, 5
* Insert 31: 82, 49, 31, 7, 5, 16
* Insert 6: 82, 49, 31, 7, 5, 16, 6
* Insert 2: 82, 49, 31, 7, 5, 16, 6, 2
* Insert 44: 82, 49, 31, 44, 5, 16, 6, 2, 7

(b) The Min Priority Queue is listed in array order:

* Insert 7 : 7
* Insert 16: 7, 16
* Insert 49: 7, 16, 49
* Insert 82: 49, 16, 7, 82
* Insert 5: 5, 7, 49, 82, 16
* Insert 31: 5, 7, 32, 82, 16, 49
* Insert 6: 5, 7, 6, 82, 16, 49, 32
* Insert 2: 2, 5, 6, 7, 16, 49, 32, 82
* Insert 44: 2, 5, 6, 7, 16, 49, 32, 83, 44
* **Page 229, Exercise 2**
* **ADT** *MinPriorityQueue* is  
  **objects:** a collection of n > 0 elements, each element has a key  
  **functions:**for all q ∈ MinPriorityQueue, item ∈ Element, n ∈ integer

|  |  |
| --- | --- |
| *MinPriorityQueue* create(*max\_size)* | ::= creates an empty priority queue |
| *Boolean* isEmpty(*q,n*) | ::= **if**(n>0) **return** TRUE      **return** FALSE |
| *Element* top(*q, n*) | ::= **if** (*!isEmpty(q,n*) **return** an instance of the smallest element in q     **return** ERROR |
| *Element* pop (*q, n*) | ::= **if** (*!isEmpty(q,n*) **return** an instance of the smallest element in q and  remove it from the priority queue.     **return** ERROR |
| *MinPriorityQueue* push(*q, item, n*) | ::= insert *item* into *q* and return the resulting priority queue. |

* end *MinPriorityQueue*
* **Page 230, Exercise 5: Change Priority**

|  |
| --- |
| void changePriority(int priority)  { /\*change priority of an item in the heap \*/  int position, newPriority, parent, child;  element item;  position = searchHeap(priority);  if (!(position)) {  printf("%d is not in the priority queue.\n", priority);  return;  }  printf("New Priority: ");  scanf("%d", &newPriority);  item.key = newPriority;  if (item.key > heap[position/2].key)  {/\* new priority is higher than current priority \*/  while (1)  if ((position == 1) || (item.key <= heap[position / 2].key))  /\* terminate when the root is reached or the element  is in its correct place \*/  break;  else {  /\* check the next lower level of the heap \*/  heap[position] = heap[position/2];  position /= 2;  }  heap[position] = item;  }  else {/\* new priority is lower, so go down the heap \*/  parent=position;  child=position \*2;  while (child<=n) {  if (child< \* (item.key if child++; heap[child+1].key) if(heap[child].key parent current the of child largest find>= heap[child].key)  /\*correct position has been found \*/  break;  else {  /\* move to the next lower level \*/  heap[parent] = heap[child];  parent = child;  child \*= 2;  }  }  heap[parent] = item;  }  } |

**Page 230, Exercise 6: Remove Priority**

|  |
| --- |
| void removePriority(int priority)  {/\*Remove an arbitrary item from the priority queue \*/  int position, newPriority, parent, child;  element item;  position = searchHeap(priority);  if (!(position)) {  printf("%d is not in the priority queue.\n", priority);  return;  }  item = topPriority();  item.key++;  /\* new priority is higher than top priority  sift heap upward\*/  while(1)  if ((position == 1) || (item.key <= heap[position / 2].key))  break;  else {  heap[position] = heap[position/2];  position /= 2;  }  heap[position] = item;  /\* remove it from the heap, since it's now at the top \*/  item = DeleteMaxHeap();  } |

**Page 230, Exercise 7: Heap Search**

|  |
| --- |
| int searchHeap(int x)  {/\* search for x in the heap  Time if 0(n) since heaps aren't  organized for searching \*/  int i;  for (i=1; i<=n; i++)  if (heap[i].key == x) return i;  return 0;  } |

**Page 230, Exercise 9**

|  |
| --- |
| #include <stdio.h>  #define TRUE 1  #define FALSE 0  #define MAX\_ELEMENTS 100  typedef struct {  int key;  /\* other fields \*/  } element;  element heap[MAX\_ELEMENTS];  int n = 0;  void InsertMinHeap(element);  element DeleteMinHeap();  void printHeap();  void HeapFull();  void HeapEmpty();  int searchHeap(int);  void changePriority(int);  void removePriority(int);  element topPriority();    int main()  {  int choice, y, position;  element x;  printf("MIN Heap Operations \n\n");  printf("1. Insert, 2. Delete, 3: Search 4. Top Priority 5. Change Priority 0. Quit:");  scanf("%d",&choice);  while (choice > 0) {  switch (choice) {  case 1: printf("Enter a number: ");  scanf("%d",&x.key);  printf("n = %d\n",n);  InsertMinHeap(x);  printHeap();  break;  case 2: x = DeleteMinHeap();  printf("%d was deleted from the heap\n\n",x.key);  printHeap();  break;  case 3: printf("Search for y: ");  scanf("%d", &y);  position = searchHeap(y);  if (position) printf("%d was FOUND in position %d.\n", y, position);  else printf("%d is not in the heap.\n", y);  break;  case 4: x = topPriority();  printf("The top priority is: %d.\n", x.key);  break;  case 5: printf("Change Priority of: ");  scanf("%d", &y);  changePriority(y);  printHeap();  break;  case 6: printf("Remove Priority: ");  scanf("%d", &y);  removePriority(y);  printHeap();  }  printf("1. Insert, 2. Delete, 3: Search 4. Top Priority 5. Change Priority ");  printf("6. Remove Priority 0. Quit:");  scanf("%d", &choice);  }  }  element topPriority() { return heap[1]; }  int searchHeap(int x)  {/\* search for x in the heap  Time if 0(n) since heaps aren't  organized for searching \*/  int i;  for (i=1; i<=n; i++)  if (heap[i].key == x) return i;  return 0;  }  void printHeap(){  int i;  for (i = 1; i <=n; i++)  printf("[%d] = %d\n", i, heap[i].key);  }  void removePriority(int priority)  {/\*Remove an arbitrary item from the priority queue \*/  int position, newPriority, parent, child;  element item;  position = searchHeap(priority);  if (!(position)) {  printf("%d is not in the priority queue.\n", priority);  return;  }  item = topPriority();  item.key++;  /\* new priority is higher than top priority  sift heap upward\*/  while(1)  if ((position == 1) || (item.key >= heap[position / 2].key))  break;  else {  heap[position] = heap[position/2];  position /= 2;  }  heap[position] = item;  /\* remove it from the heap, since it's now at the top \*/  item = DeleteMinHeap();  }  void changePriority(int priority)  { /\*change priority of an item in the heap \*/  int position, newPriority, parent, child;  element item;  position = searchHeap(priority);  if (!(position)) {  printf("%d is not in the priority queue.\n", priority);  return;  }  printf("New Priority: ");  scanf("%d", &newPriority);  item.key = newPriority;  if (item.key < heap[position/2].key)  {/\* new priority is lower than current priority \*/  while (1)  if ((position == 1) || (item.key >= heap[position / 2].key))  /\* terminate when the root is reached or the element  is in its correct place \*/  break;  else {  /\* check the next lower level of the heap \*/  heap[position] = heap[position/2];  position /= 2;  }  heap[position] = item;  }  else {/\* new priority is lower, so go down the heap \*/  parent=position;  child=position \*2;  while (child<=n) {  if (child heap[child+1].key)  child++;  if (item.key <= heap[child].key)  /\*correct position has been found \*/  break;  else {  /\* move to the next lower level \*/  heap[parent] = heap[child];  parent = child;  child \*= 2;  }  }  heap[parent] = item;  }  }  void HeapFull()  {/\* print an error message if the heap is full \*/  printf("The heap is full, No insertion made \n");  }  void HeapEmpty()  {/\* print an error message if the heap is empty \*/  printf("The heap is empty, No deletion made. \n");  }  void InsertMinHeap(element x)  {/\* insert x into the global max heap [1..MAXELEMENTS-1]  n (also global) is the present size of the heap \*/  int i;  if (n == MAX\_ELEMENTS)  HeapFull();  else {  i = ++n;  while (1)  if ((i == 1) || (x.key >= heap[i / 2].key))  /\* terminate when the root is reached or the element  is in its correct place \*/  break;  else {  /\* check the next lower level of the heap \*/  heap[i] = heap[i/2];  i /= 2;  }  heap[i] = x;  }  }  element DeleteMinHeap()  {/\* delete element with the highest key from the heap \*/  int parent, child;  element x, temp;  if (!n)  HeapEmpty();  else {  /\* save value of the element with the highest key \*/  x = heap[1];  temp = heap[n--]; /\* use last element in heap to adjust heap \*/  parent=1;  child=2;  while (child<=n) {  if (child heap[child+1].key)  child++;  if (temp.key <= heap[child].key)  /\*correct position has been found \*/  break;  else {  /\* move to the next lower level \*/  heap[parent] = heap[child];  parent = child;  child \*= 2;  }  }  heap[parent] = temp;  return x;  }  } |

**Page 240, Exercise 10** (a)  
  
*Note: InsertNode() is called EnterNode in this implementation.*

|  |
| --- |
| void EnterNode(TreePointer \*node, int num)  {/\* Enter an element into the tree \*/  /\* pass in the address of the pointer to the current node \*/  TreePointer ptr;  ptr = \*node;  if (!ptr) {  ptr = (TreePointer)malloc(sizeof(struct TreeNode));  ptr->data = num;  ptr->count = 0;  ptr->LeftChild = ptr->RightChild = NULL;  \*node = ptr;  }  else if (num == ptr->data) ptr->count++;  else if (num < ptr->data)  EnterNode(&ptr->LeftChild, num);  else  EnterNode(&ptr->RightChild, num);  } |

Recursive version of InsertNode -- Mine is called Enter Node

|  |
| --- |
| EnterNode() Function |
| void EnterNode(TreePointer \*node, int num)  {/\* Enter an element into the tree \*/  /\* pass in the address of the pointer to the current node \*/  TreePointer ptr;  ptr = \*node;  if (!ptr) {  ptr = (TreePointer)malloc(sizeof(struct TreeNode));  ptr->data = num;  ptr->LeftChild = ptr->RightChild = NULL;  \*node = ptr;  }  else if (num <= ptr->data)  EnterNode(&ptr->LeftChild, num);  else  EnterNode(&ptr->RightChild, num);  } |

**Page 247, Exercise 1**  
The transformation of a tree into a forest can be defined as follows:  
If T is a Tree with Root R, then the Forest F corresponding to this tree is denoted by T1 .. Tn+1:  
(1) is empty, if R is null.  
(2) T1  has root equal to R, and children equal to C1 .. Cn, where C1 .. Cn are the children of R in T.  
(3) T2 .. Tn+1 are trees consisting of the descendants of C1 .. Cn  in T.  Tk is empty if Ck has no descendants in T.  
The transformation is unique only in that T2 must match the descendants of C1, and so on.  The initial ordering of the children in T is irrelevant.

**Page 247, Exercise 2**  
      
The correspondence between the PreOrder traversal of a binary tree and its equivalent forest representation is fairly obvious if we look at the linked list representation of the tree.  For example, assume that we have the following binary tree:

                               A

                             /   \

                            B     C

                          /  \   /  \

                         D    E F    G

The linked representation of this tree would be: (A(B(D,E)C(F,G)).  This would produce the following forest of trees:

                A                 D                 F

               / \                |                 |

              B   C               E                 G

               T0                 T1                T2

The PreOrder traversal of the binary tree produces the sequence: ABDECFG.  Notice that this corresponds to the list representation.  For the forest, we follow A (Node), B (left child),  Tree T1, C (right child), and Tree T2.

**Page 247 Exercise 3**

There is in fact no natural †correspondence between the InOrder traversal of a binary tree and the InOrder traversal of   
the corresponding forest because there is no obvious place to attach the root of a subtree in the InOrder traversal of a forest.

**Page 264, Exercise 1**

Assume that we have the following tree.

M

/ \

G R

/ \ / \

E L P X

\

Q

The traversals are:

PreOrder: M G E L R P Q X

InOrder : E G L M P Q R X

To create the tree, we process the sequences from the first character to the last character.  For the PreOrder sequence, each time that we place a value into the tree, we get a new PreOrder value.  For example, we know that M is the root of the tree.  place it in the tree and get the next character in the PreOrder sequence.

For the InOrder sequence, we get a new character only if the current character matches the current PreOrder character, or the current character matches a value in the tree.

Continuing with our example, we next compare G, from the PreOrder sequence, and E, from the InOrder sequence.  Since they don't match, we know that G is a left child of M.  We then compare E and E.  Because they match, we have obtained two pieces of information:

1. we know that E is the left child of G, and

2. we know that it is a leaf node.  We now back up to E's parent, and continue by comparing L and L.  They   match which means that L is a right child of G.  We again backup, but this time to the root node.  Since R and P don't match, R is a non-terminal right child of M. Traversal continues with R's left child.  Notice that the next two characters in each sequence match.  The first character (P) is obviously the left child of R, but can we correctly place the second character (Q).  This character must be a right child because the last character was a left child.  Furthermore, it must be the right child of the newly created left child because the InOrder traversal hasn't yet reached R.  The correct positioning of Q is the key to the creation of a unique tree.  Because we can handle cases where the non-terminal node has only one child, we can create a unique tree.

**Page 264, Exercise 2**

Yes, the InOrder and PostOrder sequences uniquely define a tree.  The key to determining a unique sequence is the ability to   
determine the left children from the right children.  For example,  look at the following tree:

M

/ \

G R

/ \ / \

E L P X

\

Q

PostOrder: E L G Q P X R M

  InOrder: E G L M P Q R X

To create this tree from the PostOrder and InOrder sequences, we process the sequences from the last to the first   
characters. For the PostOrder sequence, each time we place a node in the tree we discard it. For example, we know that M must be the root of the tree because it is the last character in the PostOrder sequence.  We place it in the tree, and go to the next character in the PostOrder sequence.  
For the InOrder sequence, we discard a node only if it   
matches the current PostOrder character, or it is the parent of a   
node.

Continuing with our example, we would compare R from the PostOrder sequence and X from the InOrder Sequence.  Since the inverted PostOrder sequence contains nodes in the form NRL and the inverted InOrder sequence contains nodes in the form RNL, we know  that R is a right child of M.   We next compare X and X, since they match we know not only that X is the right child of R, but that we have reached the end of the right children.  We now back up to X's parent, and continue the traversal by comparing P and Q. They obviously don't match, which means that P is the left child of R.  However, because of the traversal patterns, we also know that Q must be P's right child.  Why?  Because the inverted InOrder sequence just found a parent node. The next child must either be the root of the left subtree, or a right child of it.  If it is the root, the InOrder and PostOrder sequences will match.  If it is a right child, they won't.  You can verify this by examining   
the following two sequences.

SEQUENCE 1: Tree traversals with node Q eliminated.

PostOrder: E L G P X R M

  InOrder: E G L M P R X

SEQUENCE 2: Tree traversals with node Q elimated, and Node N added   
as a left child of P.

PostOrder: E L G N P X R M

  InOrder: E G L M N P R X

**Page 264, Exercise 3**

The PreOrder and PostOrder sequences uniquely define a binary tree only if all non-terminal nodes in the original tree have two children.  If a non-terminal node has one child, it is impossible to determine if this child is a left or right child.

**Page 264, Exercise 5**

The following algorithm shows how to construct the tree given the PreOrder and InOrder traversals.  The algorithm assumes   
that Pre or In represents the characters from each sequence that are currently being compared.

Algorithm CreateTree;

begin

GetNext(Pre);

GetNext(In);

Root's value <= Pre; {first character from prefix is the root}

Parent <= Pre;

GetNext(Pre); {dispose of current character, get the next}

While (Pre contains character) and (In contains characters) do begin

{as long as there are characters in Pre and In, continue}

{constructing the tree}

While (In != Parent's value)do begin

{the nodes will be left children until the Parent's value}

{matches In's value}

Parent.Left's value <= Pre;

GetNext(Pre);

If (Pre== In) then

{Parent found, dispose of it}

GetNext(In)

Else

Parent <= Parent.Left

end; {find the left children}

GetNext(In); {dispose of parent}

While (Pre == In) do begin

{find the right children}

Parent .Right's value <= Pre;

GetNext(Pre);

GetNext(In);

Parent <=Parent.Right

end;

{continue with left children again}

Parent <= node with In's value;

GetNext(In);

Parent.Right's value <= Pre;

GetNext(Pre)

end; {there are values left to process}

end; {CreateTree}

**Page 264, Exercise 7**

You will probably want to refer to Figure 5.49 of the text during this discussion.  This figure shows the five legal trees   
obtained by permuting three nodes.  Problem 3.1 gave the formula for determining the number of permutations, and, hence, the number of distinct binary trees.  Here we'll just elaborate by examining the trees appearing in Figure 5.49.

As the figure shows, the PreOrder traversal is always 1,2,3.  However, the InOrder traversals vary widely.  These traversals   
are, from left to right: 123, 132, 213, 231, 321. We can easily reproduce these results by using the railtrack analogy discussed   
in problem 3.1.  Let me indicate how each of the InOrder traversals was obtained using this analogy.

Tree 1:  Each number is added to the stack and then removed from   
the stack.  The stack contains no more than one element at any   
given time.

Tree 2:  The first number is added to, and removed from, the stack.  We then add the remaining two numbers as a unit, and   
remove each of them.

Tree 3: The first two numbers are added to the stack as a unit.  Both numbers are removed before the last number is added.

Tree 4:  The first two numbers are added to the stack.  Only the second number is removed.  The third number is added to the stack,  and the remaining two numbers are removed.

Tree 5:  The three numbers are added to the stack as a unit.  They are then removed individually.

Because the traversals follow the stack's operations certain combinations are illegal.  For example, it is impossible to arrive   
at the combination 312.  Why?  Because there is no way we could remove three from the stack followed by a one.

# CHAPTER 6

**Page 276, Exercise 1**

Since the degree of each vertex is even, there must be an Eulerian walk.  One such walk is:

PATH: (4,3) (3,4), (4,1) (1,2), (2,1),(1,3),(3,4).

**Page 277, Exercise 2 (a)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | Out-Degree |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 2 |
| 2 | 0 | 1 | 0 | 0 | 0 | 1 | 2 |
| 3 | 0 | 0 | 1 | 0 | 1 | 1 | 3 |
| 4 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 1 | 1 | 0 | 0 | 1 | 0 | 3 |
| In-Degree | 3 | 2 | 1 | 1 | 2 | 2 |  |

2(b) Adjacency list

Vertex 0 NULL

Vertex 1 -> 0 -> 3 -> NULL

Vertex 2 -> 1 -> 5 -> NULL

Vertex 3 -> 2 -> 4 -> 5 -> NULL

Vertex 4 -> 0 -> NULL

Vertext 5 -> 0 -> 1 -> 4 -> NULL

2(C) ADJACENCY MULTII-LIST

HeadNodes Multilist Nodes

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0 NULL N1 -> 1 ->0 ->N2 ->NULL edge (1,0)

1 -> N1 N2 -> 1 ->3 ->NULL ->N5 edge(1,3)

2 -> N3 N3 -> 2 ->1 ->N4 ->NULL edge(2,1)

3 -> N5 N4 -> 2 ->5 ->NULL ->N9 edge(2,5)

4 -> N8 N5 -> 3 ->2 ->N6 ->NULL edge(3,2)

5 -> N9 N6 -> 3 ->4 ->N7 ->N8 edge(3,4)

N7 -> 3 ->5 ->NULL ->N9 edge(3,5)

N8 -> 4 ->0 ->NULL ->NULL edge(4,0)

N9 -> 5 ->0 ->N10 ->NULL edge(5,0)

N10 -> 5 ->1 ->N11 ->NULL edge(5,1)

N11 -> 5 ->4 ->NULL ->NULL edge(5,4)

The lists are:

vertex 0: NULL

vertex 1: N1 N2

vertex 2: N3 N4

vertex 3: N5 N6 N7

vertex 4: N8

vertex 5: N9 N10 N11

(e) Since node 0 has an out-degree of 0, it is not strongly connected. Similarly, since node 4's only outward node is adjacent to node 0, it is not strongly connected. The remaining nodes are all strongly connected as the table below illustrates.

V i V j v i ->v j v j -> v i

1 2 <1,3><3,2> <1,2>

1 3 <1,3> <3,2><2,1>

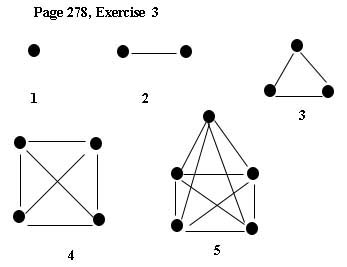
1 5 <1,3><3,5> <5,1>

2 3 <2,1><1,3> <3,2>

2 5 <2,5> <5,1><1,3><3,2>

3 5 <3,5> <5,1><1,3>

**Page 278, Exercise 3**



Proof:

1. **Informal:** The definition of a complete graph states that each vertex is connected to every other vertex.  If there are N   
vertices, then each vertex must be connected to the remaining N-1 vertices.  This gives N•(N-1) edges.  However, since each edge is   
shared by two vertices, we must divide our initial answer by two.  This gives N•(N-1)/2 edges.

**2.Formal:** Alternately, the question can be phrased in terms of combinations.  We are interested in the number of ways to   
combine N objects (vertices) taken two at a time (edges).  This is   
defined as:

N!

\_\_\_\_\_\_\_\_\_\_

2 • (N-2)!

This is equivalent to:

N•(N-1)•(N-2)!

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2 • (N-2)!

The (N-2)! terms cancel out leaving N•N-1)/2. **Page 278, Exercise 4**

The table below lists all of the simple paths. Since there is a path from each node to every other node, the graph is strongly connected.

V i V j V i->V j V j->V i

0 1 <0,1> <1,2><2,0>

0 2 <0,1><1,2> <2,0>

0 3 <0,3> <3,2><2,0>

1 2 <1,2> <2,0><0,1>

1 3 <1,2><2,0>,<0,3> <3,2><2,0><0,1>

2 3 <2,0><0,3> <3,2>

**Page 278, Exercise 5**

**(A) Matrix**

0 1 2 3

-----------------------------

0 0 1 0 1

1 0 0 1 0

2 1 0 0 0

3 0 0 1 0

**(B) Adjacency List**

Headnodes

0 -> 1 -> 3 -> NULL

1 -> 2 -> NULL

2 -> 0 -> NULL

3 -> 2 -> NULL

**(c) Adjacency Multi-List**

headnodes Multi-list nodes

0 -> N0 N0 -> 0 -> 1 -> N1 -> N2 -> edge(0,1)

1 -> N2 N1 -> 0 -> 3 -> NULL -> N4 -> edge(0,3)

2 -> N3 N2 -> 1 -> 2 -> NULL -> N3 -> edge(1,2)

3 -> N4 N3 -> 2 -> 0 -> NULL -> NULL -> edge(2,0)

N4 -> 3 -> 2 -> NULL -> NULL -> edge(3,2)

The lists are:

vertex 0: N0 -> N1

vertex 1: N2

vertex 2: N3

vertex 3: N4

**Page 278, Exercise 6**

The degree of a vertex (di) is defined as "the number of edges incident to that vertex" (p. 271).  Since each edge is   
incident to two vertices, the total number of edges is:

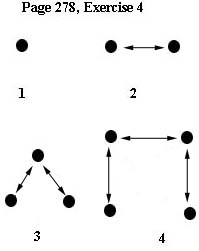
http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp6/p272Form.jpg

Multiplying both sides of the equation by 2 gives:

http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp6/p272bForm.jpg

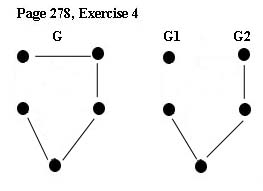
**Page 278 Exercise 7**

An undirected graph with only one vertex is connected.  Such a graph has 0 edges.  Adding a second vertex to this graph requires an additional edge to connect the first vertex to the second.  As each additional vertex is added we must add one edge   
to connect the new vertex to one of the previous vertices.  If this is not done the resulting graph is not connected.   The figure  
below shows the growth of a graph from one vertex to four. With  each addition, only one new edge is required.

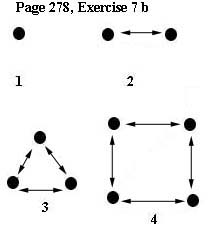


We can demonstrate that the above observation holds for all graphs.  Assume that N = number of vertices and E = Number of   
edges, then formally a graph is connected:  
1. If N=1, and Edgesn = 0  
  
2. For N > 1,  Edgesn = 1 + Edgesn-1

**Proof:** Assume that we have a graph G as given below.  We remove an edge from this graph. This produces the two Graphs G1   
and G2  (also below). The number of edges in G must equal E1+E2+1 since G1 and G2 were produced by removing a single edge from G.  By rule 2, E1=(N1-1) and E2=(N2-1).  Therefore, E = N1-1 + N2-1 +1 = N1 + N2-1 =  N-1.



PART B: A strongly connected digraph requires a minimum of N edges.  The samples below show that all of the resulting graphs are cyclic.



To prove this we use the result of the previous exercise, we know that removing an edge from a graph with N-1 edges produces a graph which is not connected.  This means that graphs with N-1 edges cannot have cross paths. Removing an edge from a graph with N edges, however, does not produce an unconnected graph as the above examples illustrate.  Since no connected graph with fewer than n edges can be cyclic, the examples in are trees.

**Page 278-9 Exercise 8**

The definition in the text states that "A tree is a finite set of one or more nodes such that 1) there is a specially desig­  
nated node called the root; 2) the remaining nodes are partitioned into n ≤ 0 disjoint sets T1, ... ,Tnwhere each of these sets is a   
tree. T1, ... ,Tn are called the subtrees of the root."

This definition is equivalent to statement c since the condition that T1, ... ,Tn be disjoint sets means that there can be only one path between any two vertices.  Since the sets are disjoint there can be no cross-edges in the graph. This means that   
there can not be multiple paths between vertices.

Statement c implies statement d because a connected graph with cross-edges must have more than n edges, and hence is cyclic.  Therefore, the graph must be acyclic, and contain n-1 edges.

Statement d implies statement b because removing an edge from a  cyclic graph produces a connected graph.  Since the graph is not cyclic, removing an edge must produce a graph that is not connected.  In essence, all nodes except leaf nodes are articula­  
tion points.

**Page 278, Exercise 9**

|  |
| --- |
| Data Structure & Call |
| typedef struct Node \*NodePointer;  struct Node  {  int Vertex;  struct Node \*Link;  };  typedef NodePointer Lists[MAX\_SIZE];    Lists AdjList;  CreateList(AdjList); |

|  |
| --- |
| Function Definition |
| void CreateList(Lists AdjList)  {/\* create the adjacency list \*/  int i, Vertex1, Vertex2;  NodePointer temp;  /\* Initialize all of the pointers to NULL, including the  0 position which is never used. Failure to do so  causes the system to blow up !!! \*/  for (i=0; i= 0)  {  /\* place vertex 1 in vertex 2's adjacency list \*/  temp = (NodePointer)malloc(sizeof(struct Node ));  temp->Vertex = Vertex1;  temp->Link = AdjList[Vertex2];  AdjList[Vertex2] = temp;  /\* place vertex 2 in vertex 1's adjacency list \*/  temp = (NodePointer) malloc(sizeof (struct Node));  temp-> Vertex = Vertex2;  temp->Link = AdjList[Vertex1];  AdjList[Vertex1] = temp;  printf("Enter a pair of vertices, 0 0 to quit: ");  scanf("%d%d",&Vertex1,&Vertex2);  }  } |

**Page 278, Exercise 10**

|  |
| --- |
| Data Structure and Call |
| typedef struct edge \*edge\_pointer;  typedef struct edge {  short int marked;  int vertex1;  int vertex2;  edge\_pointer path1;  edge\_pointer path2;  };  edge\_pointer graph[MAX\_VERTICES];  create\_list(graph); |

|  |
| --- |
| Function Definition |
| void create\_list(edge\_pointer graph[])  {/\* create the adjacency multi-list \*/  int i, vertex1, vertex2;  edge\_pointer temp;  /\* Initialize all of the pointers to NULL, including the  0 position which is never used. Failure to do so  causes the system to blow up !!! \*/  for (i = 0; i < MAX\_VERTICES; i++)  graph[i] = NULL;  /\* allow the user to enter vertices, until a 0 vertex is  entered. This could be changed by by convention  vertices are labeled beginning with 1 \*/  printf("Enter a pair of vertices, -1 -1 to quit: ");  scanf("%d%d",&vertex1,&vertex2);  while (vertex1 >= 0) {  /\* place vertex 1 in vertex 2's adjacency list \*/  temp = (edge\_pointer)malloc(sizeof(struct edge ));  temp->vertex1 = vertex1;  temp->vertex2 = vertex2;  temp->marked = FALSE;  temp->path1 = NULL;  temp->path2 = NULL;  graph[vertex1] = temp;  printf("Enter a pair of vertices, -1 -1 to quit: ");  scanf("%d%d",&vertex1,&vertex2);  }  } |

**Page 291 Exercise 3**

According to the definition in the text, "a biconnected component of a graph G is a maximal binconnected subgraph H of G."  That is, "G contains no other subgraph that is both biconnected   
and properly contains H."

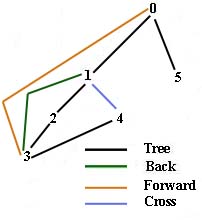
Assume that we have an edge that is in two biconnected components.  This can occur only if a cycle is present.  However, cyclic graphs are not biconnected since removing an edge from a cyclic graph will not cause the graph to break into two distinct components.  Thus, we have a contradiction: If an edge lies in two distinct subgraphs, it must produce a cycle.  But cyclic graphs are not biconnected.

**Page 291 Exercise 4**

Proving that a depth first spanning tree T has no cross edges relative to the graph G, is analogous to proving that T forms a tree.

First, for an undirected graph the proof is trivial since technically speaking the edges of an undirected graph are divided into only tree and back edges.  Thus, there are no cross edges in an undirected graph.

Second,   we note that with a digraph the edges are typically divided into four categories: Tree, Back, Forward, and Cross.  The diagram below shows a graph, with all four types of edges, that you may use as a reference.  The tree edges are <0,1> <1,2> <2,3> <3,4> <0,6>.  Edge <0,3> is forward.  Edge <3,1> is back, and edge <4,1> is cross.



Cross edges are only conceptually distinct from back edges.  Technically, a cross edge, like a back edge, is defined as an edge in which the new vertex has a higher dfn (depth first search  number) than the previously visited vertex to which it is adjacent.  Cross edges are used to refer to edges that cross from right to left on the graph, any other edges with a higher dfn for the new vertex are back edges.

The proof given in problem Page 292, Problem 12 demonstrates that the depth first search eliminates the back edges from a graph during the traversal.  Since cross edges are only conceptually distinct from back edges any traversal which eliminates back edges will also eliminate the cross edges.

**Page 291 Exercise 5**

This version uses an array based version of the stack for ease in programming.

|  |
| --- |
| Declarations |
| int top = -1;  int stack[MAX\_NODES][2];  void add(int,int);  void delete(int \*,int \*); |

|  |
| --- |
| add() |
| void add(int x, int y)  {  stack[++top][0] = x;  stack[top][1] = y;  } |

|  |
| --- |
| delete() |
| void delete(int \*x, int \*y)  {  \*x = stack[top][0];  \*y = stack[top--][1];  } |

|  |
| --- |
| bicon() |
| void bicon(int u, int v)  {  /\* compute dfn and low, and output the edges of G by their  biconnected components, v is the parent (if any) of the spanning  tree of u. dfn[0] to dfn[MAX\_SIZE] is initialized to -1, and  num is set to 0 before the function begins \*/  NodePointer ptr;  int w,x,y;  dfn[u] = low[u] = num++;  for (ptr = graph[u]; ptr; ptr = ptr->link) {  w = ptr->vertex;  if (v != w && dfn[w] < dfn[u])  **add(u,w);**  if (dfn[w] <0) { /\* w has not been visited \*/  bicon(w,u);  low[u] = MIN2(low[u],low[w]);  if (low[w] >= dfn[u]) {  printf("New biconnected component: ");  do {  **delete(&x,&y)**;  printf("\t<%2d,%2d>",x,y);  } while (!((x == u) && (y == w)));  printf("\n");  }  }  else if (w != v)  low[u] = MIN2(low[u],dfn[w]);  }  } |

**Page 291, Exercise 6**

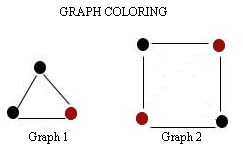
     To demonstrate that bicon works correctly, we must show that no edge appears in more than one biconnected component.  This is equivalent to indicating that bicon partitions the graph into a forest of spanning trees with each tree representing one biconnected component.  
       
To prove that a forest of spanning trees exists, we must establish that each tree in the forest excludes back edges.  Recall that a back edge leads to a previously visited path, and hence introduces a cycle.   
  
The bicon function uses the back edges to signal the start of a new biconnected component.  It prints out the tree edges of a biconnected component until it encounters a back edge.  At this point, it closes the spanning tree for the old biconnected component, and starts a new spanning tree for the next biconnected component.  The result is a forest of spanning trees.

**Page 291, Exercise 8**

Demonstrating that all trees are bipartite is a relatively simple task.  Assume that we mark the root vertex with a one.  Also assume that we mark each of its children with a zero.  Because only one simple path exists between any two vertices in a tree, each vertex will appear on only one adjacency list.  This means that each vertex is either an unmarked child or a previously visited parent.  If it is an unmarked child, it will be given the opposite mark of its parent.  If it is a previously visited parent it will have the opposite sign of the child.  A mismatch can occur only if there is more than one path between any two vertices.  This means that the graph is not a tree, which is a contradiction.

**Page 291, Exercise 9**

When my daughter was 8, I asked her to color each of the graphs below.  I told her that she could use only two colors and that she   
couldn't put the same color on any circles connected by a line. After giving me a chagrined look, she told me that it couldn't be   
done with the first graph.  As is obvious this  graph contains an odd cycle.  To alternate colors,  you must exclude odd cycles   
because the first and last vertices will be of the same color.  Two coloring the second graph, which contains an even cycle, is quite easy.



**Page 291, Exercise 10**

The results of applying the depth first and breadth first   
search to the complete graph with four vertices are identical.    
The paths taken depend only on the order in which the vertices are   
entered into the adjacency lists.  Assuming that the adjacency   
lists have the following structure, the DFS and BFS will be 0, 1, 2, 3.

ADJACENCY LIST

0 -> 1 -> 2 -> 3 -> NULL

1 -> 2 -> 3 -> 0 -> NULL

2 -> 3 -> 1 -> 0 -> NULL

3 -> 2 -> 1 -> 0 -> NULL

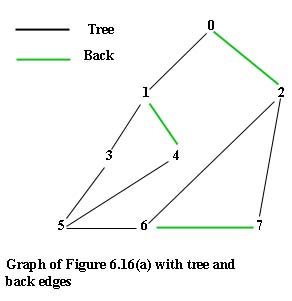
**Page 291 Exercise 11**

The modification appears below.  The function extends dfs() so that it prints out all of the connected components of a graph.  The traverse() function is the original dfs() function.  Itstops when it reaches the first node in a different subgraph.  The main() function checks to see if the current node is not NULL and has not been visited.  If both of these conditions prevail, it restarts traverse with the new node.  It continues in   
this fashion until all of the nodes have been examined.The code appears below:

|  |
| --- |
| int main ()  {  int i,n;  create\_list(graph, &n);  for (i=0; i< MAX\_VERTICES;i++)  visited[i]=FALSE;  printf("CONNECTED SUBGRAPHS\n");  i = 0;  while (graph[i]) {  /\*start the next subgraph\*/  if (!visited[i]){  printf("%3d", i);  traverse(i);  printf("\n");  }  i++;  }  } |
| void traverse(int x1)  {/\*visit each of the vertices that are connected through x1\*/  node\_pointer temp;  visited[x1]=TRUE;  temp= graph[x1];  while (temp)  {/\*visit all vertices adjacent to x1\*/  if (visited[temp->vertex] == FALSE) {  traverse(temp->vertex);  printf("%3d ",temp->vertex);  }  temp=temp->link;  }  } /\*traverse\*/ |

**Page 292 Exercise 12**

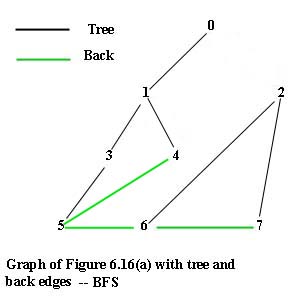
I have redrawn (below) Figure 6.16(a) with the tree edges in solid black lines , and the back edges in solid green lines. This graph illustrates the results of a depth first traversal.  Assume for the purpose of the discussion that follows that v = V0, u = v1, and w = v2.



Recall that a depth first search begins with some vertex v.  This vertex is marked as visited, and the search continues with v's adjacency list.  Assume that an unvisited vertex u is found.  The edge (v,u) is a tree edge because it connects a vertex in the tree (v) with an unvisited vertex (u).  Because u is not currently in the tree, its introduction cannot create a cycle.  The traversal now continues with u's adjacency list.  Assume that at some   
point in our traversal we examine vertex w's adjacency list, and discover that vertex u is on it.   Edge (w,u) is a back edge because u appears higher in the tree than w.  Adding the edge (w,u) to the tree would produce a cycle, as the diagram above illustrates.  The algorithm prevents the introduction of back edges by marking ancestors as visited.  Since edges that contain previously visited vertices are ignored, the algorithm excludes   
cycles.

**Page 292 Exercise 13**

Ihave redrawn (below) figure 6.16(a) with the tree edges in solid black lines, and the back edges in solid green lines. This graph illustrates the results of the BFS.  Assume for the purpose of the discussion that follows that v = v0, u = v1, w= v7, and x = v4.



Recall that BFS begins with some vertex v.  This vertex is marked as visited, and the search continues with v's adjacency list. Each vertex on v's adjacency list (u0..un-1) is marked as visited, and placed on a queue.  The initial vertex (v) is the root of the tree.  Each of its adjacent vertices must be tree edges unless the graph contains self-loops.  The algorithm removes a vertex (u0) from the queue, and examines each of its adjacent   
vertices. If the adjacent vertex is unvisited, it is marked as visited, and it is  placed on the queue.  Since unvisited vertices lead to new edges, they are tree edges.  However,  assume that at  some point in the traversal we begin examining vertex w's adjacency list, and discover the edge (w,x) where x is a previously visited vertex.  Because x is an ancestor of w, it is a back edge. Since these edges are ignored by the algorithm, the completed   
traversal must include only tree edges.

**Page 292 Exercise 14**

Listing the bridges, rather than the biconnected components of a graph, is a much easier programming task.  We no longer need a stack to hold all of the connected components.  We simply change  bicon() to print out only those edges whose low value is greater than their dfn value.  The code appears below:

|  |
| --- |
| void bridges(int u, int v)  {/\* Find the bridges \*/  node\_pointer ptr;  int w,x,y;  dfn[u] = low[u] = num++;  for (ptr = graph[u]; ptr; ptr = ptr->link) {  w = ptr->vertex;  if (dfn[w] < 0) {  /\*if †the node has not been examined then call bridges with it\*/  bridges(w,u);  low[u] = MIN2(low[u],low[w]);  if (low[w] > dfn[u])  /\*if the low value of node w is greater than the depth  first search value of u, then is a bridge\*/  printf("%d %d\n", w,u);  }  else if (w != v )  /\*determine if the low value of u needs to be adjusted\*/  low[u] = MIN2(low[u],dfn[w]);  }  } |

**Page 292 Exercise 15**

This problem is a nice application of Sterling numbers.  Sterling numbers are discussed quite thoroughly in Concrete Mathematics by Ronald Graham, Donald Knuth, and Oren Patashnik.  Sterling numbers are divided into two varieties, and it is the first kind that is of interest to us.  Sterling numbers of the first kind are used to indicate the number of partitions that can be produced from n objects taken k at a time.  For example, if we   
take three objects and split them into two groups, we will obtain 3 distinct groupings: {1} U {23}, {2} U {13}, and {3} U {12}.

Actually the case of k = 2 is somewhat trickier than k = anything else, and I refer you to Graham, et al.  The answer for k = 2, however, is always 2n - 1.

**Page 298 Exercise 1**

To demonstrate that Prim's algorithm produces a minimum spanning tree, we must verify that  1) a tree is formed, and 2) it has the lowest weight.

We can easily demonstrate that Prim's algorithm creates a directed, acyclic graph, and, hence, a tree.  First, we note that, at each stage, the algorithm adds an edge only if one of its vertices is already in the tree.  Thus, the graph is always connected.  Because the algorithm does not add a vertex if the other vertex incident to the edge has the same root, cycles are avoided.  In addition, since exactly n-1 edges are added to the tree, the graph connect all vertices.

The graph intuitively is of minimum weight because each stage adds the lowest weight  unprocessed edge with a vertex in the tree.  Since the lowest weight edges are always picked, the final result must be minimal.

**Page 298 Exercise 3**

As was true of Prim's and Kruskal's algorithms, we must demonstrate that Sollin's algorithm produces a connected, acyclic graph of minimum weight.  The explanation of the that appears in Page 299, Problem 7 furnishes much of the required proof, and I will only elaborate here.

Sollin's alogorithm, like Kruskal's, creates a spanning forest which eventually forms a tree, if one exists.  Unlike Kruskal's algorithm, several edges are selected for inclusion in each pass.  At each stage, edges are selected only if the two vertices of the edge do not share the same root.  An edge whose vertices share a root is a back edge.  Such  an edge will produce a cycle.   Edges incident to all vertices are selected at each stage.  Thus, the graph must connect all vertices.  Since exactly n-1 edges are selected for inclusion in the final graph,  the graph is also acyclic.

Because the edges are maintained in ascending order, and because an edge is excluded from consideration only if both vertices already have edges represented  at that stage, a minimal graph is produced.  That is, an edge whose vertices have already been selected must have a higher weight, since an edge is selected   
only if its weight is the lowest for each of its vertices.

**Page 299 Exercise 4**

Sollin's algorithm is quite elegant, and has a complexity far better than Prim's algorithm.  Using the example that appears in Figure 6.25 of the text, it is quite obvious that Sollin's algorithm makes only two passes over the graph before the spanning tree is completed.  Each pass adds ⌈ n/2⌉ edges to the tree.  For the example in the text, ⌈7/2 ⌉ = 4 edges are added on the first pass.  The remaining 2 edges ⌈4/2⌉ are added on the second pass.  The complexity of this algorithm is slightly better than log2n.

**Page 299 Exercise 7**

Assume that T is a properly constructed spanning tree of G.  This means that T must contain all of the tree edges of G.  Any new edge introduced in T must be a member of the set of back edges.  As Page 292, Problems 12 demonstrated, a back edge always creates a cycle since it connects two vertices already in the tree.

**Page 313, Exercise 9**

The ShortestPath algorithm produced the following distances and paths for Figure 6.36.

|  |
| --- |
| Vertex Distance Path    0 0  1 2 0 1  2 3 0 2  3 6 0 1 3  4 7 0 1 3 4  5 8 0 1 3 5  6 9 0 1 3 4 6 |

The only path that is correct is the one between 0 and 2; all of the remaining paths are erroneous.  The problem is the negative path between vertices 1 and 2.  Since vertex 1 has the smallest distance it is initially picked, and all vertices descendant from vertex 1 will use its distance from vertex 0 to determine their own paths.  However, there is a smaller path from vertex 0 to vertex 1, which is obtained by going through vertex 2.  Since the initial selection only examines direct paths, smaller indirect paths are ignored.  The correct paths and distances for   
Figure 6.36 should be:

|  |
| --- |
| Vertex Distance Path  0 0  1 1 0 2 1  2 3 0 2  3 5 0 2 1 3  4 6 0 2 1 3 4  5 7 0 2 1 3 5  6 8 0 1 2 3 4 |

**Page 314, Exercise 18**

AllCosts fails miserably with the graph of Figure 6.35.  On my computer, AllCosts produced the following results:

|  |
| --- |
| ä 0 1 2 3 4 5 6  0 -32767 -32765 -32763 -32767 -32767 -32767 -32767  1 0 -32767 -32766 -32768 -32768 -32768 -32768  2 -32767 -32767 -32766 -32768 -32768 -32768 -32768  3 -32767 -32767 -32768 -348 -354 -390 -606  4 -32768 -32767 -32766 -354 -360 -396 -612  5 -32767 -32767 -32766 -390 -396 -432 -648  6 -32767 -32767 -32766 -606 -612 -648 -864 |

As you can see, these results are quite interesting.  The negative numbers arise from the arithmetic overflow formed when a path containing MAX\_INT is added to any other path.  Since AllCosts does not test the two paths before it adds them, if one of the paths is MaxInt and the other path is anything but a zero, a new   
negative path is created.  The graph of Figure 6.35 has many non-existent paths which  corrupt the entire table.  This is in stark contrast to the ShortestPath algorithm which very selectively chooses the path that has the lowest cost. This strategy isolates  a path's ability to corrupt other paths.

**Page 314 Exercise 20**

The equation in the text is a simple application of boolean algebra.  Recall that A refers to the original matrix, and A\* to the matrix that places a 1 in positioni,j, if there is a simple path from i to j, and a 0 otherwise.  A + differs from A\* only in the treatment of the diagonals.  A+ will have a 1 in a diagonal only if there is a simple cycle for the diagonal.

The equation states that a simple cycle exists, if there is a path from i back to itself through any intervening vertices.  The 'and' indicates that there must be a path from i back to itself.  The 'or' indicates that we only need to find one path.

For example, using Figure 6.354 (a) from the text, the following application of the equation determines if there is a simple cycle from <0,0>:

<0,0> ∧ <0,0> ∨ <0,1> ∧ <1,0> ∨ <0,2> ∧ <2,0> ∨ <0,3> ∧ <3,0> &or <0,4> ∧ <4,0>

→ (0 ∧ 1) ∨ (1 ∧ 0) ∨ (1 ∧ 0) ∨ (1 ∧ 0) ∨ (1 ∧ 0)

→  0 ∨ 0 ∨ 0 ∨ 0 ∨ 0

→  0

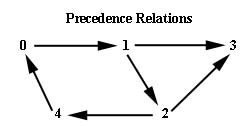
Since the result is zero, there is no simple cycle from 0 to 0.

**Page 314 Exercise 21**

Page 278, Problem 5 demonstrated that the graph in Figure 6.13 was strongly connected.  Since there is a path from each vertex to every other vertex, and vice-versa,  A+ and A\* are identical.

They are also quite dull since every cell contains a 1. **Page 328 Exercise 1**

The precedence relations do not sketch out a partial order, and this is quite apparent if we diagram the relations.  Assuming that vi < vj implies that vi is a predecessor of vj, the graphic protrayal of the relations is as follows.



As the text states, there can be no partial order, if every node has a predecessor.  This is equivalent to saying that a   
partial order exists iff all arrows flow to the right.  As is obvious from the graph above,  all arrows do not flow to the right. Thus, every node has a predecessor.  There is no partial order.

**Page 329 Exercise 2**

Use this graph to refer to the activity numbers:

|  |
| --- |
| aoe Graph |

a.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Activity | Time | e(i) | l(i) | Critical Event |
| a1 | 5 | 0 | 0 |  |
| a2 | 6 | 0 | 0 | √ |
| a3 | 3 | 5 | 6 |  |
| a4 | 6 | 6 | 6 | √ |
| a5 | 3 | 6 | 9 |  |
| a6 | 3 | 12 | 12 | √ |
| a7 | 4 | 12 | 15 |  |
| a8 | 4 | 12 | 12 | √ |
| a9 | 1 | 15 | 15 | √ |
| a10 | 4 | 15 | 15 | √ |
| a11 | 2 | 19 | 19 | √ |
| a12 | 5 | 16 | 16 | √ |
| a13 | 4 | 16 | 16 |  |
| a14 | 2 | 21 | 21 | √ |

b. The earliest time the project can complete is 23 units.

c. The critical activities are indicated on the chart.

d. Speeding up activity 4 would reduce the overall time. However since activity 4 takes only 3 time units, any reduction would be minimal. The major problem is not the activity 4's speed, but the fact that it is a bottleneck. If we could split its function, we might be able to achieve an earlier completion time.

**Page 331, Exercise 1**

BiPartitioning a graph is often called the "two-color" problem.  That is, can we color the vertices of the graph using two colors, say red and blue, so that no edge on the graph connects vertices of the same color.  The solution is to create an array whose indices range from 1 to the maximum number of vertices in the array.  The component type  of the array holds the color code, which ranges from 0 (color red), to 1 (color blue), to 2   
(not yet colored).  The algorithm arbitrarily picks a starting vertex and marks it with a 1.  All vertices adjacent to it are marked with a 0 and placed on a stack.  The algorithm removes a vertex from the stack and processes all of its vertices.  If these vertices are unmarked, they are marked with the opposite color of their parent.  Previously marked vertices must have the opposite color of the current vertex, or the graph cannot be bipartitioned.    
If the algorithm halts with an empty stack, the graph has been bipartitioned.  The array match can be used to print out the two sets.  If it halts with done set to true, the graph cannot be bipartitioned. The time complexity of this algorithm is O(n+e), where n = number of vertices and e = number of edges.

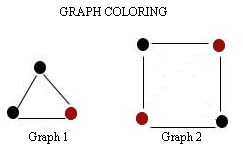
|  |
| --- |
| BiPartitioning Algorithm |
| Algorithm BiPartite(Adj:AdjList; N:integer  var Match:MatchType; var done:boolean)  /\*(determine if the graph can be partitioned into two disjoint sets\*/  var i, Top:integer;  Stack:array[0..MaxSize] of integer;  p:listPointer;  begin  for i←0 to n-1 do  */\*all elements of match are unmarked\*/*  match[0] ←2;  Top ← 0;  */\*mark the first vertex as a one\*/*  p←Adj[0];  match[0]←1;  while (p != NULL) do begin  */\*mark all of the vertices adjacent to one with a zero*  *place these vertices on the stack\*/*  top ←top +1;  stack[top] ← p.vertex;  match[p.vertex] ← 0;  p ←p.link  end;  i←1;  done←false;  while (Top > 0) and not done do begin  */\*remove a vertex from the stack examine each vertex adjacent to it*  *if unmarked, mark with the opposite mark of the vertex*  *if marked, check to see that the signs are opposite*  *if the signs are the same, the graph cannot be bipartitioned \*/*  i←stack[top];  top← top-1;  p←adj[i];  while (p != nil) and !done do begin  */\*examine each vertex adjacent to i\*/*  if match[p.vertex] > 1 then begin  if (match[i] = 0) then  match[p.vertex]←1  else  match[p.vertex] ←0;  top← top+1;  stack[top] ← p.vertex  end  else if (match[i] = match[p.vertex]) then  done ← true;  p ←p.link  end;  end;  end; |

**Page 331 Exercise 2**

Demonstrating that all trees are bipartite is a relatively simple task.  Assume that we mark the root vertex with a one.  Also assume that we mark each of its children with a zero.  Because only one simple path exists between any two vertices in a tree, each vertex will appear on only one adjacency list.  This means that each vertex is either an unmarked child or a previously visited parent.  If it is an unmarked child, it will be given the opposite mark of its parent.  If it is a previously visited parent it will have the opposite sign of the child.  A mismatch can occur   
only if there is more than one path between any two vertices.  This means that the graph is not a tree, which is a contradiction.

**Page 331 Exercise 3**

I asked my daugher when she was 8 to color each of the graphs below.    
I told her that she could use only two colors and that she couldn't put the same color on any circles connected by a line. After giving me a chagrined look, she told me that it couldn't be done with the first graph.  As is obvious this  graph contains an odd cycle.  To alternate colors,  you must exclude odd cycles because the first and last vertices will be of the same color.  Two coloring the second graph, which contains an even cycle, is   
quite easy.



**Page 332, Exercise 9**

|  |
| --- |
| void bridge(int u, int v)  {/\* Modified version of bicon to determine  the bridges.  compute dfn and low, and output the edges of G by their  biconnected components, v is the parent (if any) of the spanning  tree of u. dfn[0] to dfn[MAX\_SIZE] is initialized to -1, and  num is set to 0 before the function begins \*/  NodePointer ptr;  int w,x,y;  dfn[u] = low[u] = num++;  for (ptr = graph[u]; ptr; ptr = ptr->link) {  w = ptr->vertex;  if (dfn[w] == 0)  {/\* if the node hasn't been examined call brige with it \*/  bridge(w,u);  low[u] = MIN2(low[u], low[w]);  if (low[w] > dfn[u])  /\*{if the low value of node w is greater than the depth  first search value of u, then is a bridge\*/  printf("%d, %d\n", w, u);  else if (w!= u)  /\* determine if the low value of u needs to be adjusted \*/  low[u] = MIN2(low[u], dfn[w]);  }  }  } |

# CHAPTER 7

**Page 340, Exercise 1**

INSERTION SORT

Pass Array

1 12

2 2 12

3 2 12 16

4 2 12 16 30

5 2 8 12 16 30

6 2 8 12 16 28 30

7 2 4 8 12 16 28 30

8 2 4 8 12 16 20 28 30

9 2 4 8 10 12 16 20 28 30

10 2 4 6 8 10 12 16 20 28 30

11 2 4 6 8 10 12 16 18 20 28 30

**Page 343, Exercise 1**

NOTE: The results for the quick sort are based on the beginning of the pass rather than the end. If the array position is blank, it indicates that the pass did not use the position.

Pass Left Right 1 2 3 4 5 6 7 8 9 10 11

0 1 11 12 2 16 30 8 28 4 10 20 6 18

1 1 5 4 2 6 10 8

2 1 1 2

3 3 5 6 10 8

4 3 2

5 4 5 8 10

6 6 5

7 7 11 28 30 20 16 18

8 7 9 16 18 20

9 7 6

10 8 9 18 20

11 8 7

12 9 9 20

13 11 11 30

**Page 343, Exercise 4**- Quick Sort

In the worse case, the quick sort splits the file into two sections, the first of which contains only one record.  Under these conditions the stack could equal the number of records in the file.  Processing the smaller subfiles first and stacking the larger ones is actually the opposite of what we want to do.  To limit the stack size to log2n, we want to process the larger subfiles first, and stack the smaller ones.

**Page 343, Exercise 5**- Quick Sort

The following array demonstrates quite vividly the instability of the quicksort.  The arrows indicate the array elements that are exchanged on the first pass.  I have appended an 'a' or 'b' to the duplicate numbers to illustrate the changes.

STAGE 1: Array prior to exchanges, with first exchange highlighted.

90a 10a 80a 20a 50a 50b 80b 20b 90b 10b

STAGE 2: First set of elements has been exchanged.

10b 10a 80a 20a 50a 50b 80b 20b 90b 90a

As the second diagram shows, this exchange produced two examples of instablity.  This pattern will continue throughout the sort.

**Page 352, Exercise 1**

**NOTE:** Because the merge sort is much harder to trace, the results are given at the start of each pass. In addition to the array elements, the left and right boundaries are also shown.

Pass Left Right Array Elements (in order)

1 1 2 2 12

2 3 4 16 30

3 5 6 8 28

4 7 8 4 10

5 9 10 6 20

6 1 3 2 12 16

7 5 7 4 8 10

8 9 11 6 18 20

9 1 5 2 4 8 10 12

10 1 8 2 4 6 8 10 12 16 18

11 9 9 20

Final Pass Merges array positions 1-11.

**Page 352, Exercise 2** - Merge Sort

The stability of the merge sort is found in the following line from  
Program 7.7.

if (initList[i].key <= initList[j].key)

mergedList[k++] = initList[i++]; /\* keep in original order

This line determines the sequence for merging the files.  Since elements from lower array positions are always   
placed in the merged array first, the sort will be stable. **Page 355, Exercise 1**

HEAP SORT

Pass ARRAY

0 30 20 28 16 18 12 4 2 10 6 8

1 28 20 12 16 18 8 4 2 10 6 30

2 20 18 12 16 6 8 4 2 10 28 30

3 18 16 12 10 6 8 4 2 20 28 30

4 16 10 12 2 6 8 4 18 20 28 30

5 12 10 8 2 6 4 16 18 20 28 30

6 10 6 8 2 4 12 16 18 20 28 30

7 8 6 4 2 10 12 16 18 20 28 30

8 6 2 4 8 10 12 16 18 20 28 30

9 4 2 6 8 10 12 16 18 20 28 30

10 2 4 6 8 10 12 16 18 20 28 30

**Page 355, Exercise 2** - Heap Sort

Assume that our initial array contains the values: 90 80 90 70 60. The initial heap would be as follows:

90a

/ \

80 90b

/ \

70 60

The first pass of the sort places the first 90 in the last array position.  The second 90 will appear earlier in the sorted array.  Hence, the sort is unstable.

**Page 359, Exercise 1**

Queues: Created by Pass 1

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

| | | | | | | | |

20 | | | | | | | | 18 |

10 | | 2 | | | | 6 | | 28 |

30 | | 12 | | 4 | | 16 | | 8 |

CHAIN AFTER PASS ONE:

30 → 10 → 20 → 12 → 2 → 4 → 16 → 6 → 8 → 28 → 18

Queues: Created by Pass 2

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

| | | | | | | | |

8 | 18 | | | | | | | |

6 | 16 | | | | | | | |

4 | 12 | 28 | | | | | | |

2 | 10 | 20 | 30 | | | | | |

CHAIN AFTER PASS TWO:

2 → 4 → 6 → 8 → 10 → 12 → 16 → 18 → 20 → 28 → 30

**Page 359, Exercise 2**

An MSD radix sort would be more efficient than the LSD sort when the numbers form some type of "sorting" index. For example, zip codes are arranged by geographic areas such that the first digit typically identifies a particular region. Each additional digit serves to futher identify the location. By sorting on the most significant digit the post office is able to deliver the mail more efficiently. Similar examples can be found in telephone numbers and school rosters. In the case of telephone numbers the area codes identify regions of the country, and the first three digits of the local phone number identify particular areas of a city. Similarly, roster numbers for a schedule of courses are usually arranged so that the roster number roughly coincides with the alphabetized course schedule. For example Computer Science 101 appears before Computer Science 211, and Art 300 appears before English 101. Sorting by the MSD would be more useful in these cases.

**Page 359, Exercise 3**

The radix sort given in the text is indeed stable. The use of queues to determine the placement of the array elements insures that duplicate elements will be ordered relative to their original array positions. The following example illustrates the stablity of the algorithm. The duplicate elements are labeled 'a' and 'b' to distinguish between positions.

ORIGINAL ARRAY:

R 1 R 2 R 3 R 4 R 5 R 6 R 7 R 8 R 9 R 10

179 208a 306 93a 859 93b 208b 9a 271 9b

Queues For Pass One:

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

| | | | | | | | |

| | | | | | | | | 9b

| | | | | | | | | 9a

| | 93b | | | | | | 208b | 859

| 271 | 93a | | | | 306 | | 208a | 179

CHAIN AFTER PASS ONE:

R 1 R 2 R 3 R 4 R 5 R 6 R 7 R 8 R 9 R 10

271 93a 93b 306 208a 208b 179 859 9a 9b

Queues: Created by Pass 2

0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

9b → → → → → → → → →

9a → → → → → → → → →

208b → → → → → → → → →

208a → → → → → → → 179 → → 93b

CHAIN AFTER PASS TWO:

R 1 R 2 R 3 R 4 R 5 R 6 R 7 R 8 R 9 R 10

306 208a 208b 9a 9b 259 271 279 93a 93b

QUEUES FOR PASS THREE:

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

| | | | | | | | |

93b | | | | | | | | |

93a | | 259 | | | | | | |

9b | | 208b | | | | | | |

9a | 179 | 208a | 306 | | | | | 859 |

CHAIN AFTER PASS THREE:

R 1 R 2 R 3 R 4 R 5 R 6 R 7 R 8 R 9 R 10

9a 9b 93a 93b 179 208a 208b 259 306 859

**Page 361, Exercise 5**

Knuth provides the solution to this problem in his third volume of *The Art of Computer Programming*. Assume that we want to represent numbers in the range of 0..9999. Also assume that N=100 and n 2 =10000. We can sort the array using only two passes, if we increase the number of buckets to N. On the first pass we sort the array on the least significant digit (K mod n 2 ); on the second we sort the most significant digit. For example, if our array contained 9998, 452, and 4508, the first pass would place 9998 in the 98 bucket, 452 in the 52 bucket and 08 in the 08 bucket. The second pass would place 9998 in the 99 bucket, 452 in the 4 bucket and 4508 in the 45 bucket. Thus, we can decrease time by increasing the radix.

**Page 361, Exercise 6**

The general schema is just an extension of the process illustrated in the previous problem.  By increasing the radix appropriately we can always reduce the number of passes over the data to two.  Knuth provides a nice chart showing the best radii for a variety of values.  I would highly recommend that you examine Section5.2.5 of *The Art of Computer Programming, Vol. 3* for more details.

**Page 372, Exercise 1**

The count sort uses the following struct.

|  |
| --- |
| typedef struct {  int key;  int count;} element; |

The distribution count works by making N-1 passes over the data. All counts start out with a value of one. The count is incremented only if the value is truly less than the value with which it is compared. This means that the counts will match the relative order of the keys. Hence, a sort which uses these counts will be stable.

An example might illuminate count's operation.  Assume that an array consists of the following: 10, 5, 20, 15.  The function will make n-1 = 3 passes over the array.  On the first pass it will compare positions 0 & 1, 0 & 2, and 0 & 3.  On the second pass, it compares 1 & 2 and 1 & 3; and on the final passes it compares 2 & 3.  The final counts will be: 1, 0, 3, 2. These counts indicate the sorted array position of the keys.

The loop structure is commonly found in computer science.  Its complexity is O(n·(n-1)/2) ≈ O(n2).  This can be verified easily from the nested for loop structure.

The call is:

CountSort(list,n);

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

printf("%d\t%d\n",list[i].key, list[i].count);

The function definition is:

|  |
| --- |
| void CountSort(element list[], int n)  /\* count the number of elements in the array < then those in the ith  or jth position; The element itself is given the number 1 to match  array subscripts\*/  {  int i,j;  for (i = 0; i < n; i++) /\*set counts to 0 \*/  list[i].count = 0;  for (i = 0; i < n-1; i++)  /\* determine the number of keys less than each key \*/  for (j = i+1; j < n; j++)  if (list[i].key > list[j].key)  list[i].count++;  else  list [j].count++;  } |

**Page 372, Exercise 2**

The call is:

table(list,n);

The function definition is:

|  |
| --- |
| void table(element list [], int n)  {/\*modified table to rearrange results of distribution counting\*/  int i, k;  element temp;  for (i = 0; i < n; i++)  if (list[i].count != i)  /\*if records are out of order swap until they're in place  very simple exchange version\*/  do {  k=list[i].count;  temp=list[k];  list[k]=list[i];  list[i]=temp;  }while (list[i].count != i);  } |

**Page 375, Exercise 6**

I assume that the meat of the problem is the placement of permuted words into lists.  That is, suppose we have a list with the following five letter words:  tooth, tough, goose, baker, month, child, geese, brake.  We would like baker and brake grouped together because brake is an anagram of baker.  The easiest approach places the words in a dynamically linked list.  We then compare the first word with each of the other words in the list.  If all the letters match we put the two words into a linked list  and, continue with the next word in the list.  Each word that matches on all of the letters is added to the list.  When we reach the end of the list, we check to see if there are words in it.  If   
there are words left, we proceed with the first word of the remaining list. If this word matches any word on the list, we create another linked list to hold these anagrams.  If  the word has no anagram, we remove it from the list and examine the next word on the remaining list.  The search terminates when the list is empty.

**Page 375, Exercise 7**

A radix sort which assumes that name is the least significant digit, and state the most significant would be the best approach.  Let me illustrate with an example.  Assume that we have the following records:

|  |  |  |  |
| --- | --- | --- | --- |
| Record | Name | County | State |
| R1 R2 R3 R4 R5 R6 R7 R8 | Andrews Freed Edwards Apple Baker Durer Kane Wood | Brown McLean Green Brown McLean Brown Peach Orange | WI IL IN WI IL WI WI IN |

The name and the county are stored as arrays with blanks appended to the end. The state is stored as a two character array. We assume that strcmp() is used to compare the strings.

The first pass sorts the names into twenty-six buckets based on the first character of the last name. It would produce the following results.

Durer

Apple Baker Durer Edwards Freed Kane Wood

Andrews

----------------------------------------------------------------

A B .... D E F ... K ... W

After this pass the records are ordered as follows:

Andrews → Apple→ Baker→Durer→ Edwards→ Freed→ Kane→ Wood

The second pass sorts by county. It also uses twenty-six buckets indexed by the first character of the county name. This pass produces the following bucket structure. The county names are included for reference.

Durer

Apple Freed

Andrews Edwards Baker Kane Wood

----------------------------------------------------------------

(Brown) (Green) (McLean) (Orange) (Peach)

The records would be in the following order after this pass.  (They are referenced by last name only):

Andrews → Apple → Durer → Edwards → Baker→ Freed → Wood → Kane

The last pass sorts by states and uses twenty buckets indexed by the first character in the state's name.  Since the   
previous passes sorted by name and county, this order will bemaintained in the last pass.

Wood Kane

Edwards Durer

Freed Apple

Baker Andrews

---------------------

The records would be in the following order after this pass.

Baker McLean IL

Freed McLean IL

Edwards Green IN

Wood Orange IN

Andrews Brown WI

Apple Brown WI

Durer Brown WI

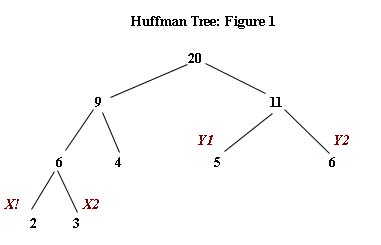
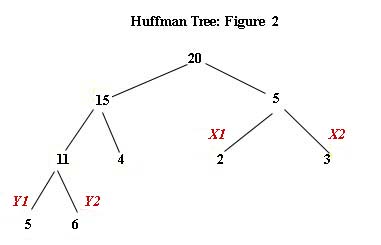
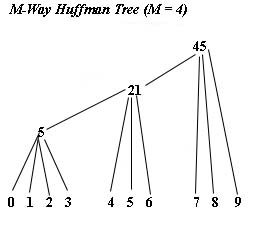
Kane Peach WI

As the list shows, the records are arranged alphabetically by state.  Within each state they are arranged alphabetically by county, and within each county they are arranged alphabetically by name.

**Page 375, Exercise 8**

1. Bubble Sort

|  |
| --- |
| bubble(list,n); |
| void bubble(element list[], int n)  /\* for i = 0, ..., n-2, compare Ri and Ri+1, if they are out of order  exchange the two elements \*/  {  int i,j;  element temp;  for (i = 0; i < n-1; i++) {  for (j = 0; j < n-i-1; j++)  if (list[j].key > list[j+1].key)  SWAP(list[j],list[j+1],temp);  }  } |

1. O(n2)
2. O(n2) - Comparisons don't change.
3. O(n2)
4. **Page 394, Exercise 1**
5. The assumption is that we will developing buffering formulas. This is a fairly complex task since as the text states: the computing time is a function of the "disk parameters and the amount of internal memory." Chapter 4 of Design of Computer Data Files by Owen Hanswer contains a good summary of buffering techniques. Another excellent source is The Art of Computer Programming, Volume 2 Section 5.4.6 by Donald Knuth.
6. **Page 394, Exercise 2** - Huffman Trees
7. The usual proof involves induction on the number of nodes.   Remember that the Huffman algorithm utilizes the greedy method to determine the merging.  At each instance, it picks the optimal solution given current information.  If the solution is optimal then we should be able to show that exchanging two leaf nodes attached to an internal node in the tree with two leaf nodes that have not been added to the tree produces a tree that has the same   
   path lengths and  weights as the original tree.  For example, assume that we have two nodes X1 and X2 that are attached to the tree, and two nodes Y1 and Y2 that are not attached to the same internal node as X1 and X2.  We should be able to swap X1 with Y1 and X2 with Y2 and not change the final outcome.  The following figure shows a sample tree.
8. 
9. The tree, after swapping the two sets of nodes, has not changed in either the path length or the total weight.
10. 
11. b) The rule "First add (1-n) mod (m-1) runs of length 0..." is not quite correct.  For m-way Huffman trees the rule, according to Knuth, assumes that n mod (m-1) = 1.  If this is not correct, we add nodes  containing a zero until the assumption is met.
12. **Example:**   Assume that n = 9 and m = 4.  Since 9 mod 3 = 0, we need to add one dummy node.  The tree is constructed, using the same technique discussed for the binary version: on each pass the m smallest elements are merged.  Assuming our n values are 1 .. 9, we would obtain the following tree.
13. 
14. The proof is analogous to the proof given in a.  We simply demonstrate that exchanging two leaf nodes attached to an internal node, and two leaf nodes not yet attached to the tree, produce a tree with the same depth and weight.

# CHAPTER 8

**Page 408, Exercise 2**

(a) If D is a prime number in the range of [10, 20], then D must be 11, 13, 17, or 19. We can test the hash function with each of these.

(b) if D is of the form 2k where k = 1 to 5, then D must be either 17 or 19 (k = 4 == 16).

**Page 408, Exercise 4**

The technique discussed does not produce a uniform hash   
function.  Let me illustrate:

Assume that we allow two character identifiers of the form: A, A1, A2, B, etc.  Further assume that x=16 bits with 8 bits used to represent each identifier. According to the text the first bit (the leftmost) of x1 must be 1. This leaves us 7 free bits in each identifier which is sufficient to store the characters by their ASCII representation. The hash values for the identifiers A, A1, A2, and A3 are:

Identifier: A        A1               A2              A3

x1  1100 0001     1100 0001         1100 0001       1100 0001  
x2  1000 0000     1011 0001         1011 0010       1011 0011  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_   \_\_\_\_\_\_\_\_\_\_\_\_\_     \_\_\_\_\_\_\_\_\_\_\_    \_\_\_\_\_\_\_\_\_\_\_\_  
xor 0100 0001     0111 0000         0111 0011       0111 0010

As you can see, if we select the middle four bits, identifiers A1, A2 and A3 will all have the same hash value.

# CHAPTER 9

**Page 441, Exercise 4**

**Induction Base:**  The finding is true for a tree of degree 0 since 20=1.  A tree of degree 1 has 20·2= 2 nodes. **Induction Hypothesis**: For all 1 < i < n, the number of nodes = 2n-1·2.  
**Induction Step:**  The number of nodes in a tree of degree n is 2n = 2n-1·n, which is obviously true.

**Page 448, Exercise 1**

(1) **Insertion:** Insertion into a Fibonacci heap differs from insertion into a binomial heap in that there is not attempt to combine tree of the samedegree. Thus, if only insertion is carried out, the Fibonacci heap will consist of a chain of trees of degree 0. All trees of degree zero are binomial heaps.

(2) **Combining Fibonacci Heaps:** Combining Fibonacci Heaps consists of combining the two circular lists into one list. The minimum pointer of the new heap is the pointer to the heap with the lowest key. Since combining heaps does not change the structure of the subtrees, if the subtrees were unordered binomial trees prior to combining, they will be unordered binomial subtrees after combining.

(3) **Deleting the minimum key:** Deleteing the minimum key removes all of the deleted node's children and places them as trees at the root level. Since each of the children is an unordered binomial tree, this operation creates a heap that has only unordered binomial trees. Trees of the same degree are then combined until only one tree of each degree exists. Combining trees of the same degree produces and unordered binomial treee that is one degree larger than the two component trees. Thus, the final result is a heap that consists of unordered binomial trees.

**Page 449, Exercise 7**

a. There is little to prove here because the formula is simply  
the generating function for Fibonacci numbers.

**b.** Observe that:

        N0 = F2 = 1  
        N1 = F3 = N

From the text, we have

A. Ni = N0 + ∑Nk +1 where 0 ≤ k ≤ i-2 for all i ≥ 1

B. Fh = ∑Fk + 1, 0 ≤ k ≤ h-2 for all i > 1

 Examining equation A, we know that N0 = F2.  As is also obvious from A and B, the two equations are identical except for the N0 term in  A, and the stipulation of i ≥ 1.  This means that if we simply shift the Ni formula by two, we'll get the Fibonnaci formula.  Therefore, Ni = Fi+2.

c. Φ, or the golden ratio, fits prominently in several of the fomulas for closed forms of the Fibonnaci generating function.  Φ, and its mirror, (1-√5)/2), can be used to generate the form: Fn = 1√5 (Φn- (1-√5)/2)n).  When n is even, Fn will be slightly   
larger than  Φn/√5.

# CHAPTER 10

**Page 491, Exercise 1**

The complete proofs on path length are quite involved.  They are discussed in depth in The Art of Computer Programming, Volume 1 by Donald Knuth (Section 2.3.4.5).

**Page 491, Exercise 2**

The w, c, and r values are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| w00 = 20 c 00 = 0  r 00 = 0 | w 11= 10 c 11 = 0 r 11 = 0 | w22 = 30 c 22 = 0 r 22 = 0 | w33 = 5  c 33= 0 r 33 = 0 | w44 = 5  c 44 = 0 r 44= 0 |
| w 01 =35 c 01 = 35 r 01 = 1 | w 12 = 50 c 12 = 50 r 12 = 2 | w23 = 35 c 23 =35 r 23=3 | w34 = 45 c 34 =60 r34 =3 |  |
| w 02 = 75 c 02 =110 r 02 =2 | w 13 =65 c 13 =100 r 13 =2 | w 24 = 45 c 24= 60 r 24 = 3 |  |  |

**Page 491, Exercise 3**

|  |
| --- |
| a. Code for Knuth Min |
| /\* find the KnuthMin \*/  min = INT\_MAX;  for (k = root[i][j-1]; k <= root[i+1][j]; k++)  if (cost[i][k-1] + cost[k][j] < min) {  min = cost[i][k-1] + cost[k][j];  minpos = k;  }  k = minpos;  cost[i][j] = weight[i][j] + cost[i][k-1] + cost[k][j];  root[i][j] = k; |

b. The Obst function will take O(n2) because of the structure of the nested for loops.  This loop is commonly found in many types of programs.  In Chapter 1, we showed that this loop requires (N·(N-1))/2 iterations.  This obviously translates into O(n2).

**Page 505, Exercise 1**

There are only three types of RL rotation because there are only three possiblities for insertion.  Case 1 occurs if the new node is the GrandChild.  Case 2 occurs if the new node is the GrandChild's right child, and case 3 occurs if the new node is the GrandChild's left child.  The graphs which follow illustrate all three cases.  Instead of using the "balance" graphs that appear in the text, I have used examples to illustrate the rotations.

|  |
| --- |
| Case 1: |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp10/AVL_RL.jpg |

|  |
| --- |
| Case 3: |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp10/AVL_RL2.jpg |

|  |
| --- |
| Case 3: |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp10/AVL_RL3.jpg |

Notice that these are all mirror images of the LR rotations.

**Page 505, Exercise 3**

   This proof is similar to the one given in problem 7, page 449.  That problem showed the relation between the number of elements in an F-heap and the Fibonacci numbers.  This relation was expressed by the formula: Ni = Fi+2, i ≥ 0.   In the current problem, we want to prove the minimum number of elements for a given level of an AVL tree is: Nh = Fh+2-1, h ≥ 0.

**Induction Base:** The statement is true for h=0 since an AVL tree with a height of zero contains only the root node.

**Induction Hypothesis:** Assume that for k= 2 to m,  Nk= Fk+2-1.

**Induction Step:** Prove that Nk+1 = Fk+1+2-1.  This is equivalent to Nk+1 = Fk+3-1.  This is obviously true since we have only shifted the formula right by two positions.

**Page 505, Exercise 4**

|  |
| --- |
|  |
| void right\_rotation(tree\_pointer \*parent, int \*unbalanced)  {  tree\_pointer child, grand\_child;  child = (\*parent)->right\_child;  if (child->bf == -1) {  /\* single rotation \*/  printf("Single Right Rotation\n");  (\*parent)->right\_child = child->left\_child;  child->left\_child = \*parent;  (\*parent)->bf = 0;  \*parent = child;  }  else {  /\* double rotation \*/  printf("Double Right Rotation \n");  grand\_child = child->left\_child;  child->left\_child = grand\_child->right\_child;  grand\_child->right\_child = child;  (\*parent)->right\_child = grand\_child->left\_child;  grand\_child->left\_child = \*parent;  switch (grand\_child->bf) {  case 1: (\*parent)->bf = 0;  child->bf = -1;  break;  case 0: (\*parent)->bf = child->bf = 0;  break;  case -1: (\*parent)->bf = 1;  child->bf = 0;  }  \*parent = grand\_child;  }  (\*parent)->bf = 0;  \*unbalanced = FALSE;  } |

**Page 505, Exercise 11**

|  |
| --- |
|  |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp10/hashEx.jpg |

# CHAPTER 11

**Page 548 Exercise 1**

Page 536 of the text indicates that B-trees of order 2 full binary trees.  
Chapter 5 of the text defines a full binary tree of depth k as one which contains 2k-1 nodes.  This is illustrated nicely in Figure 5.11.

**Page 548 Exercise 2**

|  |
| --- |
| Insert: Leaf Nodes Not shown |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp11/p548e2.jpg |

**Page 549 Exercise 4**

|  |
| --- |
| Deletion: Leaf Nodes Not Shown |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp11/p549e4a.jpg |

|  |
| --- |
| Deletion: Leaf Nodes Not Shown |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp11/p549e4b.jpg |

|  |
| --- |
| Deletion: Leaf Nodes Not Shown |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp11/p549e4c.jpg |

|  |
| --- |
| Deletion: Leaf Nodes Not Shown |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp11/p549e4d.jpg |

|  |
| --- |
| Deletion: Leaf Nodes Not Shown |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp11/p549e4e.jpg |

|  |
| --- |
| Deletion: Leaf Nodes Not Shown |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp11/p549e4f.jpg |

**Page 549 Exercise 9**

|  |
| --- |
| 23Tree Insert Only |
| #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #define FALSE 0  #define TRUE 1  #define SWAP(x,y,t) ((t) = (x), (x) = (y), (y) = (t))  typedef struct {  int key; } element;  typedef struct TwoThree \*TwoThreePtr;  struct TwoThree {  element dataL, dataR;  TwoThreePtr LeftChild, MiddleChild,  RightChild ; };  typedef struct stack \*StackPtr;  struct stack {  TwoThreePtr data;  StackPtr link; };  StackPtr top = NULL;  int compare(element, TwoThreePtr);  TwoThreePtr search23(TwoThreePtr, element);  void NewRoot(TwoThreePtr \*, element, TwoThreePtr);  TwoThreePtr pop(void);  void push(TwoThreePtr);  TwoThreePtr FindNode(TwoThreePtr, element);  void InsertionError(void);  void PutIn(TwoThreePtr \*, element, TwoThreePtr);  void split(TwoThreePtr, element \*, TwoThreePtr \*);  void order(element[]);  void PrintTree(TwoThreePtr, int);  void EmptyStack(void);  void insert23(TwoThreePtr \*, element);  TwoThreePtr FindDelNode(TwoThreePtr, element x);  void DeletionError(void);  void RotateRoot(TwoThreePtr \*, TwoThreePtr \*);  void Rotate(TwoThreePtr\*, TwoThreePtr\*, TwoThreePtr\*, int);  void combine(TwoThreePtr \*, TwoThreePtr \*, TwoThreePtr, int, int\*);  void delete23(TwoThreePtr \*, element x)  int main()  {  TwoThreePtr root = NULL, node;  int choice;  element x;  top = NULL;  printf("1. Insert, 2. Delete 3. Search, 0. Quit: ");  scanf("%d",&choice);  while (choice) {  switch(choice) {  case 1: printf("Enter your number: ");  scanf("%d",&x.key);  insert23(&root,x);  PrintTree(root,0);  break;  case 2: printf("Enter your number: ");  scanf("%d",&x.key);  delete23(&root,x);  PrintTree(root,0);  break;  case 3: printf("Enter your number: ");  scanf("%d", &x.key);  if (search23(root,x) == NULL)  printf("The key is not in the tree\n");  else  printf("The pointer to the tree was returned\n");  }  if (top) EmptyStack();  printf("1.Insert, 2.Delete, 3. Search, 0. QUit: ");  scanf("%d",&choice);  }  }  int compare(element x, TwoThreePtr node)  {  /\* send back an integer to indicate the correct child pointer  for element x \*/  if (x.key < node->dataL.key)  return 1;  else if ((x.key > node->dataL.key) && (x.key < node->dataR.key))  return 2;  else if (x.key > node->dataR.key)  return 3;  else  return 4;  }  TwoThreePtr search23(TwoThreePtr root, element x)  {  /\* search the 2-3 tree for an element that matches x.key. If  the key is found a pointer to its node is returned, otherwise a  NULL pointer is returned \*/  int done = FALSE;  TwoThreePtr node = root;  while (node && !done)  switch(compare(x,node)) {  case 1: node = node->LeftChild;  break;  case 2: node = node->MiddleChild;  break;  case 3: node = node->RightChild;  break;  case 4: done = TRUE;  }  return node;  }  void NewRoot(TwoThreePtr \*NewKid, element x, TwoThreePtr MiddleKid)  {  /\* create a new root the old root is placed as the left child,  and MiddleKid is placed in the middle position. The root is  returned in NewKid \*/  TwoThreePtr temp;  temp = (TwoThreePtr)malloc(sizeof(struct TwoThree));  temp->dataL = x;  temp->dataR.key = INT\_MAX;  temp->LeftChild = \*NewKid;  temp->MiddleChild = MiddleKid;  temp->RightChild = NULL;  \*NewKid = temp;  }  TwoThreePtr pop(void)  {  /\* pop a node from the global stack \*/  TwoThreePtr tempval;  StackPtr temp;  if (top) {  tempval = top->data;  temp = top;  top = top->link;  free(temp);  return tempval;  }  }  TwoThreePtr FindNode(TwoThreePtr node, element x)  {  /\* modified search procedure to determine the correct leaf node. This  node is returned in FindNode. The stack contains the node's ancestors \*/  StackPtr temp;  int done = FALSE;  top = NULL;  while (node && !done) {  push(node);  switch(compare(x,node)) {  case 1: node = node->LeftChild;  break;  case 2: node = node->MiddleChild;  break;  case 3: node = node->RightChild;  break;  case 4: done = TRUE;  }  }  if (done)  return NULL;  else  return pop();  }  void InsertionError(void)  {  printf("The key is already in the tree\n");  }  void PutIn(TwoThreePtr \*node, element y, TwoThreePtr a)  {  /\* node has room for another element, so add it \*/  if (y.key < (\*node)->dataL.key) {  (\*node)->dataR = (\*node)->dataL;  (\*node)->dataL = y;  (\*node)->RightChild = (\*node)->MiddleChild;  (\*node)->MiddleChild = a;  }  else {  (\*node)->dataR = y;  (\*node)->RightChild = a;  }  }  void order(element child[3])  {  /\* node must be split, order the two elements and the new one to  determine the spliting \*/  int i,j,min;  element temp;  for (i = 0; i <= 1; i++) {  min = i;  for (j = 1; j <= 2; j++)  if (child[j].key < child[min].key)  min = j;  SWAP(child[min],child[i],temp);  }  }  void split(TwoThreePtr node, element \*y, TwoThreePtr \*a)  {  /\* node is a three node, split it into two nodes \*/  element child[3];  TwoThreePtr right;  child[0] = node->dataL;  child[1] = node->dataR;  child[2] = \*y;  order(child);  node->dataL = child[0];  node->dataR.key = INT\_MAX;  right = (TwoThreePtr)malloc(sizeof(struct TwoThree));  right->dataL = child[2];  right->LeftChild = node->RightChild;  node->RightChild = NULL;  right->dataR.key = INT\_MAX;  if (\*a != NULL) {  right->MiddleChild = \*a;  right->RightChild = NULL;  }  else {  right->MiddleChild = right->RightChild = NULL;  }  \*a = right;  \*y = child[1];  }  void insert23(TwoThreePtr \*root, element y)  {/\* insert the element y into the 23 tree \*/  TwoThreePtr a, node, temp;  if (!\*root) /\* tree is empty \*/  NewRoot(root, y, NULL);  else {  /\* insert into a non-empty tree \*/  node = FindNode(\*root,y);  if (!node)  InsertionError();  else {  a = NULL;  while (1)  if (node->dataR.key == INT\_MAX) { /\*2-node \*/  PutIn(&node,y,a);  break;  }  else { /\* 3-node \*/  split(node,&y,&a);  if (node == \*root) { /\* split the root \*/  NewRoot(root,y,a);  break;  }  else  node = pop();  }  }  }  }  void PrintTree(TwoThreePtr node, int level)  {  int i;  if (node) {  for (i =1; i <= level; i++)  printf(" ");  printf("%5d",node->dataL.key);  if (node->dataR.key != INT\_MAX)  printf("%5d",node->dataR.key);  printf("\n");  PrintTree(node->LeftChild, level+1);  PrintTree(node->MiddleChild, level+1);  PrintTree(node->RightChild, level+1);  }  }  void EmptyStack(void)  {  StackPtr temp;  while(top) {  temp = top;  top = top->link;  free(temp);  }  top = NULL;  }  void push(TwoThreePtr node)  {  StackPtr temp;  temp = (StackPtr)malloc(sizeof(struct stack));  temp->data = node;  temp->link = NULL;  if (top)  temp->link = top;  top = temp;  }  TwoThreePtr FindDelNode(TwoThreePtr node, element x)  {/\* modified search procedure to determine the correct leaf node. This  node is returned in FindNode. The stack contains the node's ancestors \*/  StackPtr temp;  int done = FALSE;  top = NULL;  while (node && !done) {  push(node);  switch(compare(x,node)) {  case 1: node = node->LeftChild;  break;  case 2: node = node->MiddleChild;  break;  case 3: node = node->RightChild;  break;  case 4: done = TRUE;  }  }  if (done)  return pop();  else  return NULL;  }  void DeletionError(void)  { printf("The key is not in the tree \n");}  void rotate(TwoThreePtr \*node, TwoThreePtr \*sibling,  TwoThreePtr \*root, int num)  {  if (\*sibling)  switch(num) {  case 1: /\* rotation when node is left child of root \*/  printf("Rotate node is left \n");  (\*node)->dataL = (\*root)->dataL;  (\*root)->dataL = (\*sibling)->dataL;  (\*sibling)->dataL = (\*sibling)->dataR;  (\*sibling)->dataR.key = INT\_MAX;  (\*node)->MiddleChild = (\*sibling)->LeftChild;  (\*sibling)->LeftChild = (\*sibling)->MiddleChild;  (\*sibling)->MiddleChild = (\*sibling)->RightChild;  (\*sibling)->RightChild = NULL;  break;  case 2: /\* rotation when node is the middle child of root \*/  printf("Middle rotate \n");  (\*node)->dataL = (\*root)->dataL;  (\*root)->dataL = (\*sibling)->dataR;  (\*sibling)->dataR.key = INT\_MAX;  (\*node)->MiddleChild = (\*node)->LeftChild;  (\*node)->LeftChild = (\*sibling)->RightChild;  (\*sibling)->RightChild = NULL;  break;  case 3: /\* rotation when node is the right child of root \*/  printf("Right rotate \n");  (\*node)->dataL = (\*root)->dataR;  (\*root)->dataR = (\*sibling)->dataR;  (\*sibling)->dataR.key = INT\_MAX;  (\*node)->MiddleChild = (\*node)->LeftChild;  (\*node)->LeftChild = (\*sibling)->RightChild;  (\*sibling)->RightChild = NULL;  }  }  void combine(TwoThreePtr \*node, TwoThreePtr \*sibling,  TwoThreePtr root, int num, int \*done)  {  if (\*sibling)  switch(num) {  case 1: /\* node is the left child \*/  printf("Left combine\n");  (\*node)->dataL = (root)->dataL;  (\*node)->dataR = (\*sibling)->dataL;  (\*node)->MiddleChild = (\*sibling)->LeftChild;  (\*node)->RightChild = (\*sibling)->LeftChild;  if ((root)->dataR.key == INT\_MAX)  /\* root was a two node \*/  (root)->dataL.key = INT\_MAX;  else {  (root)->dataL = (root)->dataR;  (root)->dataR.key = INT\_MAX;  (root)->MiddleChild = (root)->RightChild;  (root)->RightChild = NULL;  \*done = TRUE;  }  break;  case 2: /\* node is the middle child \*/  printf("middle combine\n");  if ((root)->dataR.key == INT\_MAX) {  (root)->dataR = (root)->dataL;  (root)->RightChild = (\*node)->LeftChild;  (root)->MiddleChild = (\*sibling)->MiddleChild;  (root)->LeftChild = (\*sibling)->LeftChild;  (root)->dataL = (\*sibling)->dataL;  free(\*sibling);  }  else {  (\*sibling)->dataR = (root)->dataL;  (\*sibling)->RightChild = (\*node)->LeftChild;  (root)->dataL = (root)->dataR;  (root)->dataR.key = INT\_MAX;  (root)->MiddleChild = (root)->RightChild;  (root)->RightChild = NULL;  }  free(\*node);  \*done = TRUE;  break;  case 3: /\* node is the right child, root must have 2 kids \*/  printf("Right combine\n");  (\*sibling)->dataR = (root)->dataR;  (\*sibling)->RightChild = (\*node)->LeftChild;  (root)->dataR.key = INT\_MAX;  (root)->RightChild = NULL;  free(\*node);  \*done = TRUE;  }  }  void delete23(TwoThreePtr \*tree, element x)  {  TwoThreePtr node, root, sibling;  int done;  node = FindDelNode(\*tree, x);  if (!node)  DeletionError();  else {  printf("IN deletion, left = %d, right = %d\n",node->dataL.key, node->dataR.key);  if (x.key == node->dataL.key)  if (node->dataR.key != INT\_MAX) {  node->dataL = node->dataR;  node->dataR.key = INT\_MAX;  }  else  node->dataL.key = INT\_MIN;  else  node->dataR.key = INT\_MAX;  done = !(node->dataL.key == INT\_MIN);  while (!done) {  root = pop();  if (node == root->LeftChild) {  sibling = root->MiddleChild;  if (sibling->dataR.key != INT\_MAX) {  rotate(&node,&sibling,&root,1);  done = TRUE;  }  else {  if (root == \*tree) {  combine(&node,&sibling,root,1,&done);  \*tree = (\*tree)->LeftChild;  }  else  combine(&node, &sibling, root,1, &done);  }  } /\* left child \*/  else if (node == root->MiddleChild) {  sibling = root->LeftChild;  if (sibling->dataR.key != INT\_MAX) {  rotate(&node,&sibling,&root,2);  done = TRUE;  }  else  combine(&node,&sibling,root,2,&done);  }  else { /\* node is the right child \*/  sibling = node->MiddleChild;  if (sibling->dataR.key != INT\_MAX) {  rotate(&node, &sibling, &root, 3);  done = TRUE;  }  else  combine(&node, &sibling, root, 3, &done);  }  }  }  } |

**Page 549 Exercise 10**

|  |
| --- |
| 234Tree -- Insert Only |
| #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #define FALSE 0  #define TRUE 1  typedef struct {  int key; } element;  typedef struct two34 \*two34pointer;  struct two34 {  element dataL, dataM, dataR;  two34pointer LeftChild, LeftMidChild,  RightMidChild, RightChild; } ;  typedef enum {equal,leaf,lchild,lmchild,rmchild,rchild} CompareResult;  typedef enum {TwoNode,ThreeNode} NodeResult;  CompareResult compare(element,two34pointer);  void NewRoot(two34pointer \*, element);  int FourNode(two34pointer);  NodeResult NodeType(two34pointer);  void InsertionError(void);  void PutIn(element, two34pointer \*);  void SplitRoot(two34pointer \*);  void SplitChildOf2(two34pointer \*, two34pointer \*);  void SplitChildOf3(two34pointer \*, two34pointer \*);  void insert234(two34pointer \*, element);  void PrintTree(two34pointer, int);  int main()  {  two34pointer root = NULL;  element num;  printf("Enter a number, <0> to quit: ");  scanf("%d",&num.key);  while (num.key) {  insert234(&root,num);  printf("Your tree contains: \n");  PrintTree(root,0);  printf("\n\n Enter a number, <0> to quit: ");  scanf("%d",&num.key);  }  }  CompareResult compare(element x, two34pointer node)  {  if (x.key == node->dataL.key || x.key == node->dataM.key ||  x.key == node->dataR.key)  return equal;  else if (node->LeftChild == NULL && node->RightChild == NULL &&  node->LeftMidChild == NULL && node->RightMidChild == NULL)  return leaf;  else if (x.key < node->dataL.key)  return lchild;  else if (x.key < node->dataM.key)  return lmchild;  else if (x.key < node->dataR.key)  return rmchild;  else  return rchild;  }  void NewRoot(two34pointer \*root, element x)  {  \*root = (two34pointer)malloc(sizeof(struct two34));  (\*root)->dataL = x;  (\*root)->dataM.key = (\*root)->dataR.key = INT\_MAX;  (\*root)->LeftChild = (\*root)->LeftMidChild = (\*root)->RightMidChild =  (\*root)->RightChild = NULL;  }  int FourNode(two34pointer node)  { return (node->dataR.key != INT\_MAX);}  NodeResult NodeType(two34pointer node)  { (node->dataM.key == INT\_MAX) ? TwoNode : ThreeNode;}  void InsertionError()  { printf("The key is already in the tree\n");}  void PutIn(element x, two34pointer \*node)  {/\* node is a two node, and has room for another key \*/  printf("IN PUT IN: ");  if (x.key < (\*node)->dataL.key) {  (\*node)->dataR = (\*node)->dataM;  (\*node)->dataM = (\*node)->dataL;  (\*node)->dataL = x;  }  else if (x.key < (\*node)->dataM.key) {  (\*node)->dataR = (\*node)->dataM;  (\*node)->dataM = x;  }  else  (\*node)->dataR = x;  }  void SplitRoot(two34pointer \*root)  {  two34pointer left,right;  printf("IN split ROOT\n");  left = (two34pointer)malloc(sizeof(struct two34));  left->dataL = (\*root)->dataL;  left->dataM.key = left->dataR.key = INT\_MAX;  left->LeftChild = (\*root)->LeftChild;  left->LeftMidChild = (\*root)->LeftMidChild;  left->RightMidChild = left->RightChild = NULL;  right = (two34pointer)malloc(sizeof(struct two34));  right->dataL = (\*root)->dataR;  right->dataM.key = right->dataR.key = INT\_MAX;  right->LeftChild = (\*root)->RightMidChild;  right->LeftMidChild = (\*root)->RightChild;  right->RightMidChild = right->RightChild = NULL;  (\*root)->dataL = (\*root)->dataM;  (\*root)->dataM.key = (\*root)->dataR.key = INT\_MAX;  (\*root)->LeftChild = left;  (\*root)->LeftMidChild = right;  (\*root)->RightMidChild = (\*root)->RightChild = NULL;  }  void SplitChildOf2(two34pointer \*node, two34pointer \*parent)  {  two34pointer child;  printf("IN SplitCHildof2\n");  if (\*node == (\*parent)->LeftChild) {  (\*parent)->dataM = (\*parent)->dataL;  (\*parent)->dataL = (\*node)->dataM;  (\*parent)->RightMidChild = (\*parent)->RightChild;  }  else  (\*parent)->dataM = (\*node)->dataM;  (\*node)->dataM.key = INT\_MAX;  child = (two34pointer)malloc(sizeof(struct two34));  child->dataL = (\*node)->dataR;  child->dataM.key = child->dataR.key = (\*node)->dataR.key = INT\_MAX;  child->LeftChild = (\*node)->RightMidChild;  child->LeftMidChild = (\*node)->RightChild;  child->RightMidChild = child->RightChild = NULL;  (\*node)->RightMidChild = (\*node)->RightChild = NULL;  if (\*node == (\*parent)->LeftChild)  (\*parent)->LeftMidChild = child;  else  (\*parent)->RightMidChild = child;  }  void SplitChildOf3(two34pointer \*node, two34pointer \*parent)  {  two34pointer child;  printf("IN SplitChildOf3\n");  if (\*node == (\*parent)->LeftChild) {  (\*parent)->dataR = (\*parent)->dataM;  (\*parent)->dataM = (\*parent)->dataL;  (\*parent)->dataL = (\*node)->dataM;  (\*parent)->RightChild = (\*parent)->RightMidChild;  (\*parent)->RightMidChild = (\*parent)->LeftMidChild;  }  else if (\*node == (\*parent)->LeftMidChild) {  (\*parent)->dataR = (\*parent)->dataM;  (\*parent)->dataM = (\*node)->dataM;  }  else  (\*parent)->dataR = (\*node)->dataM;  (\*node)->dataM.key = INT\_MAX;  child = (two34pointer)malloc(sizeof(struct two34));  child->dataL = (\*node)->dataR;  child->dataM.key = child->dataR.key = (\*node)->dataR.key = INT\_MAX;  child->LeftChild = (\*node)->RightMidChild;  child->LeftMidChild = (\*node)->RightChild;  child->RightMidChild = child->RightChild = NULL;  (\*node)->RightMidChild = (\*node)->RightChild = NULL;  if (\*node == (\*parent)->LeftChild)  (\*parent)->LeftMidChild = child;  else if (\*node == (\*parent)->LeftMidChild)  (\*parent)->RightMidChild = child;  else  (\*parent)->RightChild = child;  }  void insert234(two34pointer \*root, element x)  {/\* insert the key into the 234 tree \*/  two34pointer node, parent;  int done;  if (!\*root)  NewRoot(root, x);  else {  if (FourNode(\*root))  SplitRoot(root);  node = \*root; parent = NULL;  done = FALSE;  while (!done) {  if (FourNode(node)) {  if (NodeType(parent) == TwoNode)  SplitChildOf2(&node,&parent);  else  SplitChildOf3(&node,&parent);  node = parent;  }  parent = node;  switch (compare(x,node)) {  case equal: InsertionError();  done = TRUE;  break;  case leaf: PutIn(x, &node);  done = TRUE;  break;  case lchild: node = node->LeftChild;  break;  case lmchild: node = node->LeftMidChild;  break;  case rmchild: node = node->RightMidChild;  break;  case rchild: node = node->RightChild;  }  }  }  }  void PrintTree(two34pointer node, int level)  {  int i;  if (node) {  for (i = 0; i <= level; i++)  printf(" ");  if (node->dataL.key != INT\_MAX)  printf("%5d",node->dataL.key);  if (node->dataM.key != INT\_MAX)  printf("%5d", node->dataM.key);  if (node->dataR.key != INT\_MAX)  printf("%5d",node->dataR.key);  printf("\n");  PrintTree(node->LeftChild, level+1);  PrintTree(node->LeftMidChild, level+1);  PrintTree(node->RightMidChild, level+1);  PrintTree(node->RightChild, level+1);  }  } |

**Page 550 Exercise 15**

This problem is virtually identical to the one described in   
Exercise 17.  Therefore, I'll describe and illustrate the splitting technique, and develop the splitting equations in this problem.  In problem 16, we develop the algorithm.

Example : Assume that we have a Btree of order 4.  The splitting   
values would be:

P = ⌊ (2·2)/3⌋ = 2

Q = ⌊ (2·4-1)/3⌋   = 2

R = ⌊ (2·4)/3⌋   =   2

We want to insert 25 into the Btree which appears below.

|  |
| --- |
| Leaf Nodes Not Shown |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp11/p550e15a.jpg |

  The easiest, but  not  necessarily the most efficient,  approach would "stuff" the values of P and Q, the Parent's i'th key, and the new value into an array in sorted order.  This would give the following array of keys:  10, 15, 20, 25, 30, 35, 40, 45.  The array's indices range from 1..2m.  We now need to develop formulas to redistribute these values among P, Q, R, Parent's i'th key, and the Parent's i'th +1 key.  These formulas are as follows:

(1)  Place into P the values from Key 1 .. ⌊ (2m-2)/3⌋

(2)  Place into Parenti the value in Key ⌊ (2m+1)/3 ⌋

(3) Place into Q the values from key m .. (m + ⌊ (2m-2)/3 ⌋ -1)

(4) Place into Parenti+1 the value in key (2m - ⌊ 2m/3 ⌋ )

(5) Place into R the values from Key (2m - ⌊ 2m/3⌋ +1) .. 2m  
  
  
The results of splitting our original Btree appear below:

|  |
| --- |
| Leaf Nodes Not Shown |
| http://www.cise.ufl.edu/~sahni/fdsc2ed/exerciseSolutions/2edfds1_8_8/chp11/p550e15b.jpg |

**Page 551 Exercise 1**6

Since x is simply the minimum number of children within a node, the formula for the number of nodes states that the number of nodes in the tree equals 1 + the number of keys divided by the minimum number of keys/ node.  This is fairly obvious.  
   
The depth formula states that: depth ≤ 1 + logmininimum number of keys{ (n+1)/2}.

Since the N nodes are split three ways rather than 2, this  formula actually overestimates the depth. We can also say that depth ≤ 1 + logminimum number of keys{(n+1)/3}.

**Page 551 Exercise 17**

The modification of the insertb algorithm consists of these lines from Program 11. 2:

/\* node p has to be split\*/

Let p and q be defined as in Eq. (11.5);

With the following lines:

Let Parent = p's parent  
Let Ki = the key in Parent which points to P  
Let q = p's right sibling  
If q→n ≤ m-1 then begin  
/\*the right child has space so shift entries\*/  
      Insert Ki into the appropriate place in q  
      Insert x into Ki  
      Output(p,q,parent) to disk  
      Exit  
end  
else begin  
/\*use the splitting technique of Bayer &McCreight\*/  
    Insert p's keys into the array Key in sorted order  
   Insert Ki into the array Key in sorted order  
   Insert x into the array Key in sorted order  
   Insert q's keys into the array Key in sorted order  
   Insert Key(s) 1 .. ⌊ (2m-2)/3) ⌋ into p  
  
   Insert Key ⌊ (2m+1)/3 ⌋ into Ki  
  
   Insert Key(s)  m .. (m + ⌊ (2m-2)/3 ⌋ -1) into q  
      
   Insert Key (2m -⌊ 2m/2 ⌋) into Ki+1  
  
   Insert Key(s) (2m - ⌊ 2m/3 ⌋+1) .. 2m into R  
  
   Set p→n to ⌊ (2m-2)/3 ⌋  
  
   Set q→n to ⌊ (2m-1)/3⌋  
  
   Set r→n to⌊ 2m/3⌋  
             
   Set Parent→n to Parent→n+1

   Set ⌊ (2m-2)/3 ⌋+1 .. m-1 key values in p to void and pointers to NULL  
   Set q's ⌊ (2m-1)/3⌋ +1 .. m-1 key values to void and pointers to NULL  
   Set r's ⌊ 2m/3 ⌋ +1 ... m-1 key values and pointers to nil  
   Output(Parent,p,q,r) to disk  
   exit

end

**Page 551 Exercise 18**

/\* delete element with key x \*/  
Search the B-tree for the node p that contains the element whose key is x;  
if there is no such p return; /\* no element to delete \*/  
Let p be of the form n, A0 , (E1,A1), . . . , (En,An) and Ei.K = x;  
if p is not a leaf {  
Replace Ei with the element with the smallest key in subtree Ai;  
Let p be the leaf of Ai from which this smallest element was taken;  
Let p be of the form n, A0, (E1,A1), . . . , (En ,An);  
i = 1;  
}

Remove x from p;  
p→n = p→n-1;  
q← p's left sibling;  
r← p's right sibling;  
if (q→n + p→n + r→n + 2 <= 2m-1) then begin  
/\*collapse the nodes\*/  
    Insert Ki into the array Key;  
    Insert p's remaining keys into the array Key ;  
    Insert Ki+1 into the array Key;  
    j =1;  
    Diff = MaxSize - q→n + 1;  
    For l = q→n to MaxSize do begin  
    /\*fill q with the first set of keys\*/  
         Insert Key[j] into q[k];  
         j++;  
    end;  
    q→n = MaxSize;  
    Ki = Key[j];  
    j++;  
   o = r→n;  
    for l ← j to Size of Key do begin  
    /\*transfer remaining keys to r\*/  
       insert Key[l] into r[o];  
    r→n = r→n + (Size of Key - j + 1);  
    dispose(p);

if ( n) write p: (n, A0, . . . , (En,An ))  
else root = A0 ; /\* new root \*/

# CHAPTER 12

**Page 551 Exercise 18**

/\* delete element with key x \*/  
Search the B-tree for the node p that contains the element whose key is x;  
if there is no such p return; /\* no element to delete \*/  
Let p be of the form n, A0 , (E1,A1), . . . , (En,An) and Ei.K = x;  
if p is not a leaf {  
Replace Ei with the element with the smallest key in subtree Ai;  
Let p be the leaf of Ai from which this smallest element was taken;  
Let p be of the form n, A0, (E1,A1), . . . , (En ,An);  
i = 1;  
}

Remove x from p;  
p→n = p→n-1;  
q← p's left sibling;  
r← p's right sibling;  
if (q→n + p→n + r→n + 2 <= 2m-1) then begin  
/\*collapse the nodes\*/  
    Insert Ki into the array Key;  
    Insert p's remaining keys into the array Key ;  
    Insert Ki+1 into the array Key;  
    j =1;  
    Diff = MaxSize - q→n + 1;  
    For l = q→n to MaxSize do begin  
    /\*fill q with the first set of keys\*/  
         Insert Key[j] into q[k];  
         j++;  
    end;  
    q→n = MaxSize;  
    Ki = Key[j];  
    j++;  
   o = r→n;  
    for l ← j to Size of Key do begin  
    /\*transfer remaining keys to r\*/  
       insert Key[l] into r[o];  
    r→n = r→n + (Size of Key - j + 1);  
    dispose(p);

if ( n) write p: (n, A0, . . . , (En,An ))  
else root = A0 ; /\* new root \*/