Extra Review Answers

1.

a) 0
2
6
14
30
62
126

b) no difference

c) **17**

2.

a)

```
R1    R1 + R2
R2    R1 - R2
R1    R1 - R2
```

It exchanges the values stored in registers 1 & 2. (Try it with some sample numbers if you don't believe that.)

b)

```
R1    9110
R2    9211
R1    R2 - R1  (= 0101)
R3    2121
R1    R1 + R3  (= 2222)
```

R1: 2222

c) **B** is the exception (all other instructions put the value 0 in R1 no matter what it contains prior to executing the instruction.)

3.

a) **22**

b) **19**

c) **1**

d) The function strange returns x.

e) **20**

4.

a) **1 -1 1 -1 1 -1**

b) **1 7 21 35 35 21 7 1**

c)

```
     #include <stdio.h>
     #define MAX 100
     int A[MAX];

     void main()
     {
            int closest, i, n;
            float avg, sum;

 (1)        for (n = 0; n < MAX && scanf("%d", &A[n]) != EOF; n++)
 (2)                ;
 (3)        sum = 0;
 (4)        for (i = 0; i < n; i++)
 (5)                sum += A[i];

 (6)        avg = sum/n;
 (7)        closest = 0;
 (8)        i = 1;

 (9)        while (i < n) {
                    /* squaring elements in the test below eliminats the need
       to distinguish positive and negative differences */

 (10)               if ((A[i]-avg)*(A[i]-avg) < (A[closest]-avg)*(A[closest]-avg))
 (11)                       closest = i;
 (12)               i++;
            }

 (13)       printf("%d\n", closest);
      }
```

d) The assignment statements at (2), (4), (5), (6), (7), (10), (11) each take O(1) time. The for-statement, (3)-(4), takes O(n) time. The if-statement, (9)-(10), takes O(1) time and the while statement, (8)-(11), takes O(n) time. The running time of the entire program is O(n).

e)

```
int i, max = 0, count[51];
for (i = 0; i <= 50; i++) count[i] = 0;

for (i = 0; i < N; i++) count[a[i]]++;

for (i = 0; i <= 50; i++)
     if (count[i] > max) max = count[i]);
for (i = 0; i <= 50; i++)
     if (count[i] == max) printf("%d\n", i);
```

5.

a) `struct card { int rank; char suit; };`
Assume that rank=1 -> Ace, rank=11 -> Jack, rank=12 -> Queen, rank=13 -> King

b) A helper function 'suit_value' was written to simplify the 'less' function (Spade = s, Heart = h, Diamond = d, Club = c)

```
int suit_value (char s)
{
        switch (s) {
                case 's': return 3; break;
                case 'h': return 2; break;
                case 'd': return 1; break;
                case 'c': return 0; break;
                default: return -1; break;
        }
        return -1;
}

int less (struct card a, struct card b)
{
        // case 1: 2 different suits
        if (suit_value(a.suit) < suit_value(b.suit))
                return 1;
        if (suit_value(a.suit) > suit_value(b.suit))
                return 0;

        // case 2: same suit
        if (a.rank == 1)     // aces high
                return 1;
        if (b.rank == 1)
                return 0;
        if (a.rank < b.rank)// all other ranks
                return 1;
        return 0;
}
```

c) Function 'less' defined in part b is used to simplify this function.

```
struct card highest (struct card c[], int n)
{
        int i = 1;
        struct card best = c[0];
        while (i < n)
        {
                if (less(best, card[i]))
                        best = card[i];
                +;
        }
        return best;
}
```

d) Here's a solution using the elementary insertion sort algorithm. (Sedgewick chapter 6)

```
void sort (struct card c[], int n)
{
        int i, j;
        struct card temp;
        for (i = 1; i < n; i++)
        {
                for (j = i; j > 0; j--)
                {
                        if (less(a[j], a[j-1]))
                        {
                                temp = a[j];
                                = a[j-1];
                                = temp;
                        }
                }
        }
}
```

6.

a)

```
link t = malloc(sizeof *t);
t->next = list->next;
list->next = t;
t->key = 999;
```

b)

```
link findZero (link x)
{
    if (x == NULL) return x;
    if (x->key == 0 return x;
    return findZero(x->next);
)
```

c)

```
link remove0 (link x)
{
    if (x == NULL) return x;
    x->next = remove0(x->next);
    if (x->key == 0) return x->next;
    else return x;
}
```

d)

```
int add (link x)
{
    if (x == NULL) return 0;
    return (x->key) + add(x->next);
}
```

e)

```
link deleteEven (link x)
       {
          link t;
          if (x != NULL)
          {
             t = x->next;
             if (t != NULL)
             {
                x->next = deleteEven (t->next);
                free (t);
             }
          }
          return x;
       }
```

f)

```
int lookup (int value, link list)
{
       if (list == NULL) return 0;
       if (value > list->element) return lookup(value, list->next);
       if (value == list->element) return 1;
       return 0;
}
```

g)

```
link insert ( int value, link list)
{
        link t;
        if (list == NULL) {
                t = malloc(sizeof *t);
                t->key = value;
                t->next = NULL;
                return t;
        }
        if (value > list->key)
                list->next = insert(value, list->next);
        else {
                t = malloc(sizeof *t);
                t->key = value;
                t->next = list;
                return t;
        }
        return list;
}
```

7. See Sedgewick section 3.3

a)

```
int count (link list)
{
        int count = 0;
        link t = list;
        if (t == NULL) return count;
        do {
                count++;
                t = t->next;
        } while (t != list);
        return count;
}
```

b)

```
int count_between (link x, link t)
{
       int count = 0;
        link b = x;
        if (x == t) return 0;       // both pointing to same node
        while (b->next != t)
        {
                count++;
                b = b->next;
        }
        return count;
}
```

c)

```
        link end_of_t = t;
        while (end_of_t->next != t) end_of_t = end_of_t->next;

        end_of_t->next = x->next;
        x->next = t;
```

d)

```
        link moving_node = t->next;
        t->next = moving_node->next;
        moving_node->next = x->next;
        x->next = moving_node;
```

8.

a)

```
int countLeaves (link x)
{
        if (x == NULL) return 0;
        if (x->left == NULL && x->right == NULL) return 1;
        return countLeaves(x->left) + countLeaves(x->right);
}
```

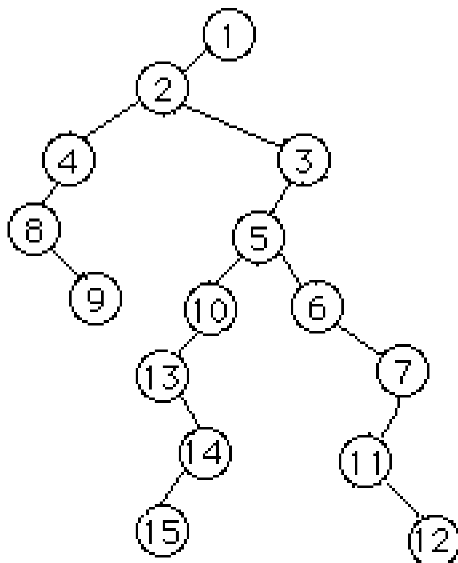
b)

```
link deleteKeys (link x, int key)
{
    if (x == NULL) return NULL;
    if (x->left == NULL && x->right == NULL)
    {
           if (x->key == key)
           {
                  free(x);
                  return NULL;
           }
           return x;
    }
    x->left = deleteKeys(x->left, key);
    x->right = deleteKeys(x->right, key);
    return x;
}
```

c)

```
int sum (link x)
{
    if (x == NULL)
           return 0;
    return ( x->key + sum(x->left) + sum(x->right);
}
```

9.



a)

b) Preorder: 1, 2, 4, 8, 9, 3, 5, 10, 13, 14, 15, 6, 7, 11, 12

c) To change the preorder traversal to a postorder traversal, move the printf() statement so that it comes after the while loop instead of before it.
Postorder: 8, 9, 4, 2, 13, 15, 14, 10, 5, 6, 11, 12, 7, 3, 1

10.

a) 104

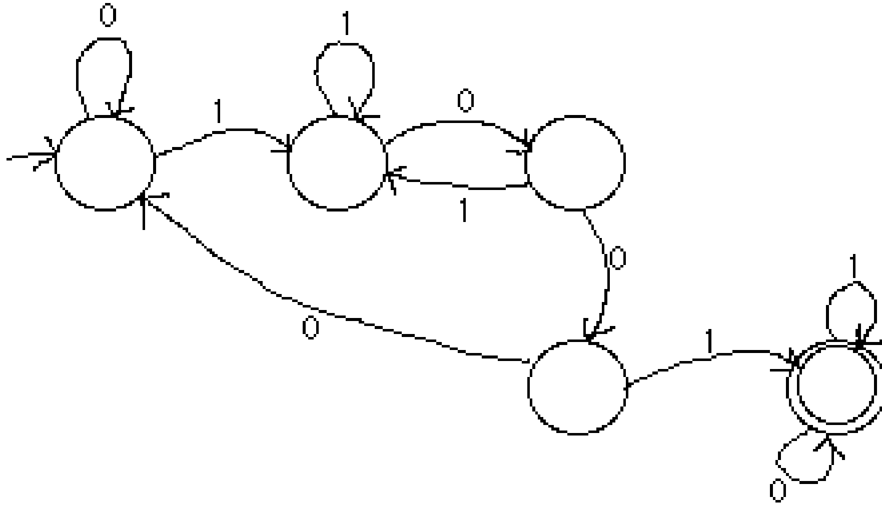
b) infix: ((a + b) * c) / (d - e) + f
   prefix: + / * + a b c - d e f

c) value: 12
   postfix: 2 2 3 + * 6 * 5 /

11. This function removes all occurrences of character **c** from the string **s**.

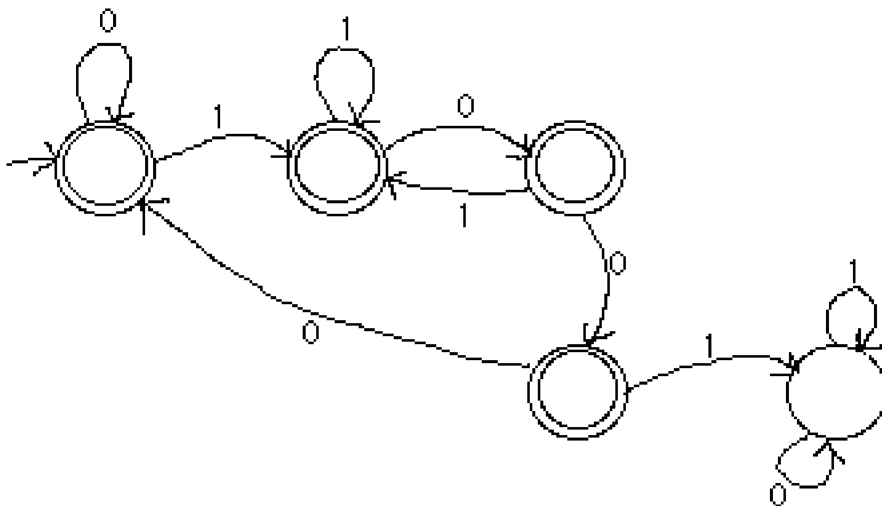
12.

a)

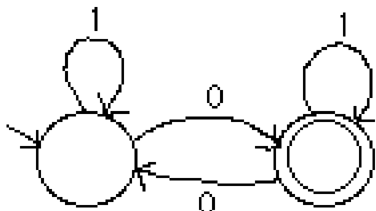


b) Here's one correct answer: (0*11*(01)*000)*(0*11*(01)*001)(0+1)*

c)

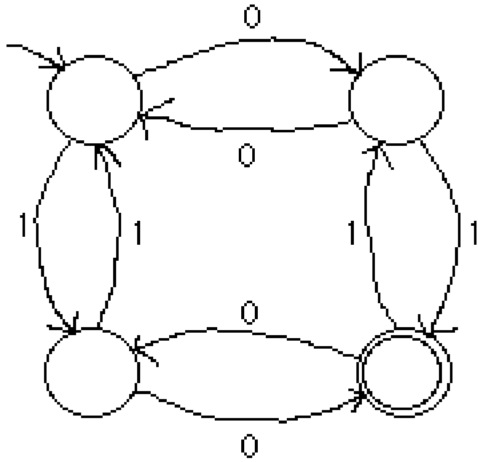


d)



e) Here's one correct answer: 1*01*(01*01*)*

f)



13. There is more than one answer to these types of questions.

a) (ac + abc + abbc) and a(+ b(c + bc)) are two other regular expressions for this language.

b) b*((aa)*b*)*

c) (0+1+2+3+4+5+6+7+8+9) (0+1+2+3+4+5+6+7+8+9)*. (0+1+2+3+4+5+6+7+8+9)

d) 0*(10*10*)*

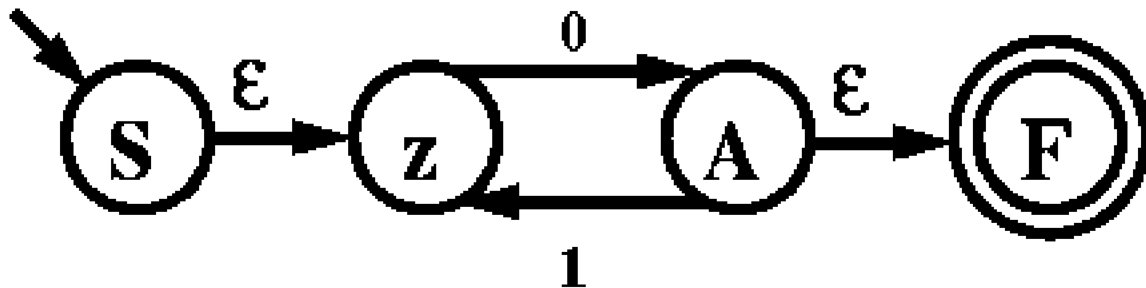
e) the set of all strings of a's and b's (including the empty string)

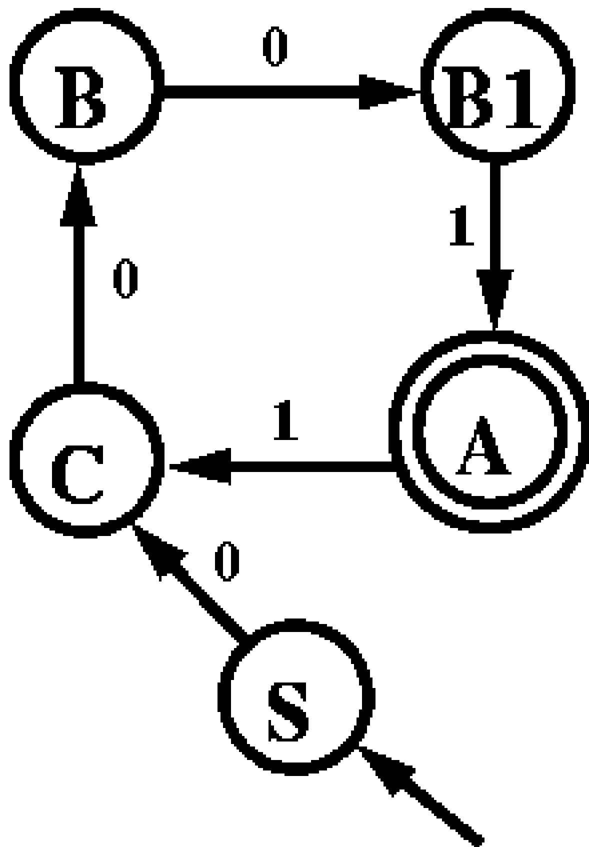
f) the set of all strings of a's and b's containing an even number of b's.

g) C

14. Note that answer (a) is a non-deterministic FSA (can you convert it to a deterministic FSA?) and answer (b) is a deterministic one. In these two pictures, I'm using a slightly cleaner notation that omits the rejected edges coming out of each state into a dead sink state.

a)

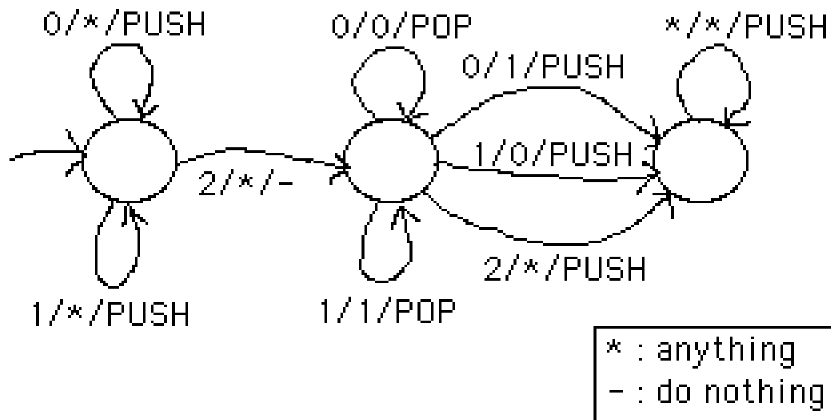




b)

15.

a) This grammar describes the language of all strings that have a 2 surrounded by 0 or more pairs of 0's and/or 1's. (The strings are a subset of odd-length palindromes.)



b)

c) ?

d) All strings of 0's and 1's that contain the substring 1001

e) Yes. That language is regular. In question 4, part b, we gave the corresponding regular expression.

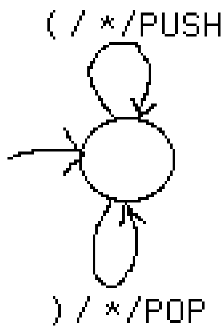
16.

```
<Letter> -> A | B | ... | Z | a | b | ... | z | _
<Digit> -> 0 | 1 | ... | 9
<Identifier> -> <Identifier><Letter>
<Identifier> -> <Identifier><Digit>
<Identifier> -> <Letter>
```

17.

a) No. To create such an FSA, you would need to be able to count how many left-hand parentheses you were reading in order to match them up with right-hand parentheses. With only a finite number of states, you can't keep track of how many left-hand parentheses you've read, so you won't be able to create an FSA that determines whether parentheses in an expression are balanced.

b) Yes. Here's a simple PDA that does this. If there are more right-hand parentheses than left at any time, the machine will crash (b/c it will attempt to 'pop' from an empty stack.) If there are more left-hand parentheses than right at the end, the stack will not be empty, and the string will be rejected from the language. A transition could also be added to allow other characters to be read & ignored.



c) Yes. Review Sedgewick section 4.2 for the stack implementation. This function takes a string as its input & outputs 1 if the string contains a balanced parentheses expression, 0 otherwise. The operations empty(), pop(), and push() execute on an initially empty stack of characters.

```
int balanced (char * exp)
{
        int i = 0;
        char c = exp[i];
        while (c != '\0')
        {
                if (c == '(')
                        push(c);
                if (c == ')')
                {
                        if (empty())
                                return 0;
                        pop();
                }
                i++;
                c = exp[i];
        }
        if (empty()) return 1;
        return 0;
}
```

18.

1. Construct a non-deterministic Turing machine that solves the WIZBAN problem.

2. Turn this Turing machine and the instance of the WIZBAN problem into a number of logical clauses, so that the resulting set of clauses has a satisfying assignment if and only if the WIZBAN problem was a yes instance. This can be done in polynomial time.

3. Apply the alien algorithm which is able to detect yes instances of SAT in polynomial time. We have now solved the WIZBAN problem in polynomial time.

19.

a) The string **str** is not long enough to hold the string "this is a test".

b)

```
#include <string.h>
#include <stdio.h>

void main(int argc, char *argv[])
{
        int i, j;
        char *s;

        for (i = argc - 1; i > 0; i--)
        {
                s = argv[i];
                for (j = strlen(s) - 1; j >= 0; j--)
                        printf("%c", s[j]);
                printf(" ");
        }
        printf("\n");
}
```

c) See the function given in question # 8 as a starting point.

```
void squeeze2 (char s1[], char s2[])
{
        int i, j, k;

        for (i = j = 0; s1[i] != '\0'; i++)
        {
                for (k = 0; s2[k] != '\0'; k++)
                        if (s2[k] == s1[i])
                                break;
                if (s2[k] == '\0')
                        s1[j++] = s1[i];
        }

        s1[new_s1_pos] = '\0';
}
```

d)

```
int squeeze3 (char s1[], char s2[])
{
        int i, k;

        for (i = 0; s1[i] != '\0'; i++)
        {
                for (k = 0; s2[k] != '\0'; k++)
                        if (s2[k] == s1[i])
                                return i;
        }

        return -1;
}
```

20. If you have trouble figuring out what a function does just by reading the code, try tracing through it with a small sample array. For example, try a: 4, 2, 6, 1, 7

a) The function bs sorts the array of integers in increasing order

b) The function is executed (asize - 1) + (asize - 2) + ... + 3 + 2 + 1, which is equal to (asize)*(asize - 1) / 2

c) $O(n^2)$

d)

```
void swap (int * x, int * y)
{
        int t = *x;
        *x = *y;
        *y = t;
}
```

21. Lines (1) - (3) each time one time unit. For line (4), the test takes one unit, and it is executed *n* times. Lines (5) and (6) each take one unit and they are executed *n* times. Line (7) takes one unit. Thus, the total time taken by the program is $3n + 4$ units.

22.

a) With 5 elements in the array, selection sort makes 4 iterations with the loop index i = 0, 1, 2, 3. The first iteration makes 4 comparisons, the second 3, the third 2, the fourth 1, for a total of 10 comparisons. With the array 6, 8, 14, 17, 23, there are no swaps (exchanges of elements in any iteration

b) On the array 17, 23, 14, 6, 8, selection sort makes 4 iterations. The numbers of comparisons and swaps made during each iteration are summarized in the following table.

ITERATION	ARRAY AFTER ITERATION	NO OF COMPARISONS	NO OF SWAPS
Start	17, 23, 14, 6, 8	-	-
1	6, 23, 14, 17, 8	4	1
2	6, 8, 14, 17, 23	3	1
3	6, 8, 14, 17, 23	2	0
4	6, 8, 14, 17, 23	1	0

c) Again, 4 iterations will be made. The numbers of comparisons and swaps made during each iteration are summarized in the following table.

ITERATION	ARRAY AFTER ITERATION	NO OF COMPARISONS	NO OF SWAPS
Start	23, 17, 14, 8, 6	-	-
1	6, 17, 14, 8, 23	4	1
2	6, 8, 14, 17, 23	3	1
3	6, 8, 14, 17, 23	2	0
4	6, 8, 14, 17, 23	1	0

23. See Sedgewick section 6.3

a)

```
void sort (int a[], int asize)
{
   int i, j, v;
   for (i = asize - 1; i > 0; i--)
          if (a[i-1] > a[i])
                 swap(a[i-1], a[i]);
   for (i = 2; i < isize; i++)
   {
          j = i;
          v = a[i]
          while (v < a[j-1])
          {
                 a[j] = a[j-1];
                 j--;
          }
          a[j] = v;
   }
}
```

b) The best case number of comparisons in this efficient version of the insert sort algorithm is (asize - 1) comparisons in the first `for` loop and (asize - 2) comparisons in the second `for` loop. The best case order is already-sorted.

The worst case number of comparisons is (asize - 1) comparisons in the first `for` loop and [(asize - 2) + (asize - 3) + ... + 2 + 1] comparisons in the second `for` loop. The total number of comparisons is (asize)*(asize - 1) / 2. The worst case order is an array with all items initially in reversed order.

c) first for loop:

```
8 6 7 5 3 0 9 (initially)
8 6 7 5 3 0 9
8 6 7 5 0 3 9
8 6 7 0 5 3 9
8 6 0 7 5 3 9
8 0 6 7 5 3 9
0 8 6 7 5 3 9
```

second for loop:

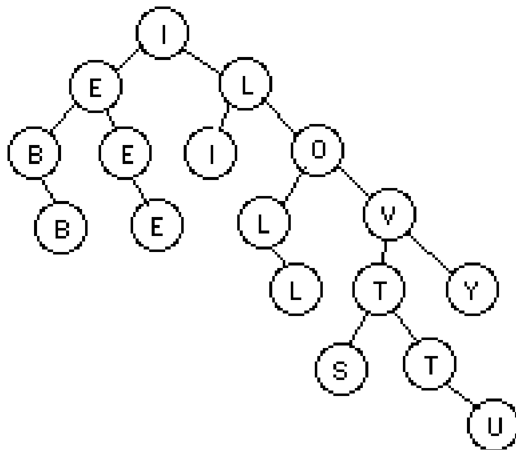
```
0 8 6 7 5 3 9 (initially)
0 6 8 7 5 3 9
0 6 7 8 5 3 9
0 5 6 7 8 3 9
0 3 5 6 7 8 9
0 3 5 6 7 8 9 (final)
```

24.

a) best case: (2 N ln N) comparisons
assuming that the last element is used as the pivot, here's an example of a best case order for an array with seven items: 1 3 2 6 5 7 4
worst case: about (N^2 / 2) comparisons
a worst case order is all elements in already sorted or reverse order.

b) no, see Sedgewick section 7.2 for a discussion of the performance characteristics of quick sort.

25.



a)

b) B B E E E I I L L L O S T T U V Y

c) B B E E E I I L L L O S T T U V Y

d) 15 + 14 + 13 + ... + 2 + 1
total: 15*16/2 = 120 comparisons

e)

```
int sum (link t)
{
        if (t == NULL) return 0;
        return t->value + sum(t->left) + sum(t->right);
}
```

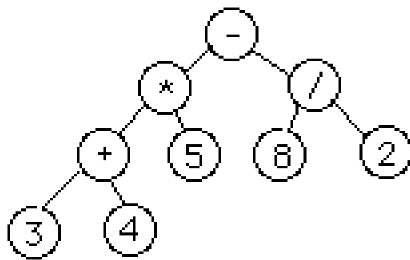
f) The easiest way to do this is to write a 'helper' function which computes the maximum of two integer values.

```
int max( int a, int b)
{
        if (a > b) return a;
        return b;
}

int depth (link t)
{
        if (t == NULL) return 0;
        return 1 + max( depth(t->left), depth(t->right));
}
```

26.

a)



b)

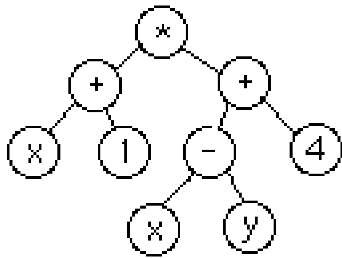
```
void inorder_print (link t)
{
        if (t == NULL) return;
        if (t->l != NULL) printf("(");
        inorder_print(t->l);
        printf("%c", t->token);
        inorder_print(t->r);
        if (t->r != NULL) printf(")");
}
```

c) A trick is used here to convert from a character to its corresponding integer value.
(See Kernighan & Ritchie, section 1.6)

```
int calculate (link t)
{
        if (t->l ==  NULL)                    // if node is a leaf node,
                return (t->token - '0');              // return integer value
        if (t->token == '+') return calculate(t->l) + calculate(t->r);
        if(t->token == '-') return calculate(t->l) - calculate(t->r);
        if(t->token == '*') return calculate(t->l) * calculate(t->r);
        if(t->token == '/') return calculate(t->l) / calculate(t->r);
        return 0;                             // should never happen
}
```

27.

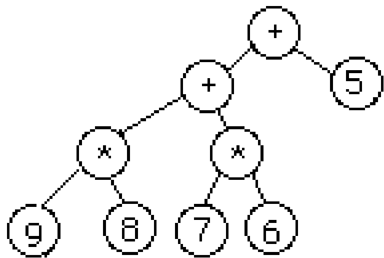
a) reverse polish notation: x 1 + x y - 4 + *
abstract syntax tree:



TOY code:

```
00:  0001
01:  0002
1A:  9100        R1 <- M[R0+0]
1B:  B201        R2 <- 1
1C:  1112        R1 <- R1 + R2
1D:  9200        R2 <- M[R0+0]
1E:  9301        R3 <- M[R0+1]
1F:  2223        R2 <- R2 - R3
20:  B304        R3 <- 4
21:  1223        R2 <- R2 + R3
22:  3112        R1 <- R1 * R2
23:  4102        print R1
24:  0000        halt
1A
```

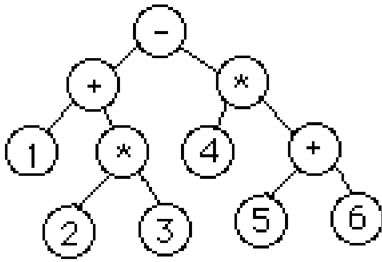
b) reverse polish notation: 9 8 * 7 6 * + 5 +
abstract syntax tree:



TOY code:

```
1A:  B109        R1 <- 9
1B:  B208        R2 <- 8
1C:  3112        R1 <- R1 * R2
1D:  B207        R2 <- 7
1E:  B306        R3 <- 6
1F:  3223        R2 <- R2 * R3
20:  B305        R3 <- 5
21:  1223        R2 <- R2 + R3
22:  1112        R1 <- R1 + R2
23:  4102        print R1
24:  0000        halt
1A
```

c) reverse polish notation: 1 2 3 * + 4 5 6 + * -
abstract syntax tree:



TOY code:

```
1A:  B101        R1 <- 1
1B:  B202        R2 <- 2
1C:  B303        R3 <- 3
1D:  3223        R2 <- R2 * R3
1E:  B304        R3 <- 4
1F:  B405        R4 <- 5
20:  B506        R5 <- 6
21:  1445        R4 <- R4 + R5
22:  3334        R3 <- R3 * R4
23:  2223        R2 <- R2 - R3
24:  1112        R1 <- R1 + R2
25:  4102        print R1
26:  0000        halt
1A
```

28.

```
int getToken (FILE *in, char s[])
{
        int pos = 0;
        char c;

        /* eat up all leading whitespace */
        while ((c = getc(in)) != EOF)
                if (isspace(c) == 0)
                        break;

        /* single-character operators */
        if ((c=='(') || (c==')') || (c=='+') || (c=='-') || (c=='*') || (c=='/'))
        {
                s[pos++] = c;
                s[pos] = '\0';
                return 2;
        }

        /* integer constants */
        if (isdigit(c))
        {
                s[pos++] = c;
                while ((c = getc(in)) != EOF)
                {
                        if (isdigit(c))
                                s[pos++] = c;
                        else
                                break;
                }
                ungetc(c, in);
                s[pos] = '\0';
                return 1;
        }

        /* identifiers */
        if (isalpha(c))
        {
                s[pos++] = c;
                while ((c = getc(in)) != EOF)
                {
                        if ((isdigit(c)) || (isalpha(c)))
                                s[pos++] = c;
                        else
                                break;
                }
                ungetc(c,in);
                s[pos] = '\0';
                return 0;
        }

        /* unrecognized token */
        return -1;
}
```

29.

a) It saves the execution context (registers and PC) of the currently running program and restores the the execution context of some other program which TOS saved earlier. Then execution control is transferred to this other program.

b) TOY instructions use physical memory addresses, so unwanted sharing of the same memory data can occur among multiple instances of the same program, resulting in interference of each other's operation.

c) Virtual memory. Instructions that access memory use virtual memory addresses that must be translated to physical memory addresses via a translation table. Multiple instances of the same program have different translations for the same virtual addresses, avoiding unwanted sharing.

30.

a) CPU utilization:

process:	1	2	3	4	5
time:	0	16	19	26	106

avg = (0 + 16 + 19 + 26 + 106) / 5 = 33.4

b) CPU utilization:

process:	2	3	5	1	4
time:	0	3	10	21	37

avg = (0 + 3 + 10 + 21 + 37) / 5 = 14.2

c) CPU utilization:

process:	1	2	3	4	5	1	4	5	4
time:	0	10	13	20	30	40	46	56	57

wait times: P1= 0 + 30 = 30
P2 = 10
P3 = 13
P4 = 20 + 16 + 1= 37
P5 = 30 + 16 = 46
avg = (30 + 10 + 13 + 37 + 46) / 5 = 27.2

d) CPU utilization:

process:	1	2	3	4	5	1	4	5	4
time:	0	8	11	18	26	34	42	50	53

wait times: P1= 0 + 26 = 26
P2 = 8
P3 = 11
P4 = 18 + 16 + 3 = 37
P5 = 26 + 16 = 42
avg = (26 + 8 + 11 + 37 + 42) / 5 = 24.8

- The shortest job first strategy gives the best results. This strategy is always optimal. However, in general we can't 'predict the future.' That is, the operating system doesn't usually know how long a process will take until after it's finished. First-come-first-serve is the strategy that gave the least efficient average wait time. The cause of the inefficiency in this example was the size of process 4. Process 5 had to wait much longer than any other process for access to the CPU. In general, processes are of vastly different sizes, so any process scheduling strategy needs to take this into account. The advantages of the first-come-first-serve strategy are that it's simple to implement *and* it's fair: processes are granted access to the CPU in the order in which they request it. The round-robin strategy is a good compromise between first-come-first-serve and shortest job first. It limits the amount of time any process will have to wait for the CPU. The shorter the time quantum, the shorter the wait will be for initial access. However, if the time quantum is too small, too much time will be wasted by the CPU in switching between different processes.

31.

a) FIFO:

in memory	page request	page fault?
-	0	yes
0	1	yes
0, 1	2	yes
0, 1, 2	3	yes
1, 2, 3	0	yes
2, 3, 0	1	yes
3, 0, 1	4	yes
0, 1, 4	0	no
0, 1, 4	1	no
0, 1, 4	2	yes
1, 4, 2	3	yes
4, 2, 3	4	no

at end: pages 4,2,3

LRU:

in memory	page request	page fault?
-	0	yes
0	1	yes
0, 1	2	yes
0, 1, 2	3	yes
1, 2, 3	0	yes
2, 3, 0	1	yes
3, 0, 1	4	yes
0, 1, 4	0	no
1, 4, 0	1	no
4, 0, 1	2	yes
0, 1, 2	3	yes
1, 2, 3	4	yes

at end: pages 2,3,4

b) FIFO:

in memory	page request	page fault?
-	0	yes
0	1	yes
0, 1	2	yes
0, 1, 2	3	yes
0, 1, 2, 3	0	no
0, 1, 2, 3	1	no
0, 1, 2, 3	4	yes
1, 2, 3, 4	0	yes
2, 3, 4, 0	1	yes
3, 4, 0, 1	2	yes
4, 0, 1, 2	3	yes
0, 1, 2, 3	4	yes

at end: pages 1,2,3,4

LRU:

in memory	page request	page fault?
-	0	yes
0	1	yes
0, 1	2	yes
0, 1, 2	3	yes
0, 1, 2, 3	0	no
1, 2, 3, 0	1	no
2, 3, 0, 1	4	yes
3, 0, 1, 4	0	no
3, 1, 4, 0	1	no
3, 4, 0, 1	2	yes
4, 0, 1, 2	3	yes
0, 1, 2, 3	4	yes

at end: pages 1,2,3,4

c) When there was room for just 3 pages, FIFO worked slightly better. However, when the number of pages in memory was increased to 4, LRU performed better than FIFO. FIFO would be easier to implement (with just a simple queue). An implementation of LRU involves a data structure in which you can extract an element from the middle of the list and insert it on the end efficiently, to maintain the 'recent-use' order. There is no 'right' answer. Rather, as an operating systems designer, you need to be aware of the relative trade-offs among page replacemnt policies.

32. Java questions

a) When a variable is of object type (that is, declared with a class as its type rather than one of Java's primitive types), the value stored in the variable is not an object. Objects exists in a part of memory called the heap, and the variable holds a *pointer* or *reference* to the object. Null is a special value that can be stored in a variable to indicate that it does not actually point to any object.

b) A constructor is a kind of subroutine that is called when the new operator is used to create a new object. Its purpose is to perform any necessary initialization of the object -- and to perform any other actions that the programmer wants to have happen automatically when an object is created.?

c) A class is used as a template for making objects. That is, a class contains information about what sort of data an object should contain and what sort of behaviors an object should have. One class can be used to make many objects. All the objects have the same behavior, as specified by the class, and they all hold the same kind of instance variables (although the data stored in those instance variables can be different from one object to another).

   Classes and objects are totally different sorts of things! Objects do not exist until they are created, after a program is running, and they are destroyed before the program ends. A class is actually part of the program, so of course it exists the whole time a program is running. If you have a class, you can create a subclass based on that class, but you can't do anything similar with an object.

d) An instance variable is part of an object. An object can contain data, and that data is stored in instance variables. The instance variables of an object are determined by the class of that object, but the actual variables belong to individual objects, not to the class. Each object of the class has its own set of instance variables.

e) If a variable or method is declared to be `static`, than that variable or method really belongs to the class as a whole, rather than to any of the objects that might be created from that class. This means, for example, that the variable or method is shared by all such objects and that there is only one copy of the variable or method that exists the whole time a program is running.

f) Math.sqrt is a static method. Math is the name of the class in which the sqrt method is defined. sqrt is the name of the method itself. "Math.sqrt" is the name for the method when it is called from outside the Math class.

g) In object oriented programming, one class can inherit all the properties and behaviors from another class. It can then add to and modify what it inherits. The class that inherits is called a subclass, and the class that it inherits from is said to be its superclass. In Java, the fact that `ClassA` is a subclass of `ClassB` is indicated in the definition of `ClassA` as follows:

```
class ClassA extends ClassB {...}
```

h) Applets requuire a Web browser or a tool such as the applet viewer to provide an execution environment. Applications are directly interpreted by a Java interpreter.

i) Bytecodes are a platform-independent representation of a program. They are generated by the Java compiler.

j) The purpose of garbage collection is to identify objects that can no longer be used, and to dispose of such objects and reclaim the memory space that they occupy. If garbage collection is not used, then programmer must be responsible for keeping track of which objects are still in use and disposing of objects when they are no longer needed. If the programmer makes a mistake, then there is a "memory leak," which might gradually fill up memory with useless objects until the program crashes for lack of memory.

33.

a) Here is a possible answer. (Note that the initialization of the instance variable, value, to zero is not really necessary, since it would be initialized to zero anyway if no explicit initialization were provided.)

```
public class Counter {
    //An object of this class represents a counter that counts up from zero.

    private int value = 0;        // current value of the counter

    public void increment() {     // add one to the value of the counter
            value++;
    }

    public int getValue() {       // get the current value of the counter
            return value;
    }

} // end of class Counter
```

b) The variable headCount is a variable of type Counter, so the only thing that you can do with it is call the instance methods headCount.increment() and headCount.getValue(). Similarly for tailCount. Here is the program with calls to these instance methods filled in:

```
Counter headCount = new Counter();
Counter tailCount = new Counter();

for (int flip = 0; flip < 100; flip++) {
    if (Math.random() < 0.5)
            headCount.increment();    // count a "head"
    else
            tailCount.increment();    // count a "tail"
}

System.out.println("There were " + headCount.getValue() + " heads.");
System.out.println ("There were " + tailCount.getValue() + " tails.");
```

c) Note that the data for employee number i is stored in emp[i]. In particular, the hourly wage for employee i is emp[i].hourlyWage. The number of employees with hourly wage greater than $25.00 can be determined as follows:

```
int count = 0;
for (int i = 0; i < 1000; i++) {
    if (emp[i].hourlyWage > 25)
    count++;
}
```

d)

```
public class BankAccount {

    private double balance;    // amount of money in the account

    public BankAccount(double initialDeposit) {   // constructor
        balance = initialDeposit;
    }

    public void deposit(double amount) {
        balance = balance + amount;
    }

    public void withdraw(double amount) {
        balance = balance - amount;
    }

    public double checkBalance() {
        return balance;
    }

}
```

e) i = 3; s = New string; d = 6.02E23
i = 3; s = New string; d = 6.02E23

34.

```
% java Test
5
6
1
4
```