CSC221 Spring 2012
Data Structures & Algorithms I
Exam # 2 Practice Problems
Scheduled date: Mon, April. 16, 2012

1. Describe how two data elements can be exchanged without using a temporary to store one of them. *Hint: Use the exclusive or operator.*

   Solution:

```
#include <iostream>
using namespace std ;

int main()
{
   int x, y ;

   x = 1 ; y = 2 ;
   x ^= y ;     // Exchange x and y using exclusive OR.
   y ^= x ;
   x ^= y ;
   cout << x << " " << y << endl ;  // Output is:  2 1
}
```

2. For this problem, refer to the attached copy of `token.h`, and `token.cc`. In our project on syntax trees, we only allowed variables to be a single letter. Modify class `token` and function `get_next()` to allow multiple-character variable names up to 31 characters in length. *Hint: use a fixed-length array of type* `char` *; be sure to remember to null-terminate C-style character strings.*

   Solution: follows:

```
#define MAX_VAR 32

union extra_info {
   char variable[MAX_VAR] ;  //  Make 'variable' a null-terminated
   int number   ;            //  C-style string.
} ;
```

   We also change the in-line function `get_variable()` to:

```
char * get_variable() { return &(details.variable[0]) ; }
```

   In the file "token.cc", we change the following section of `get_next()`

   Old code:

```
if ( isalpha(ch) ) {
      what_type = vari ;
      details.variable = ch ;
}
```

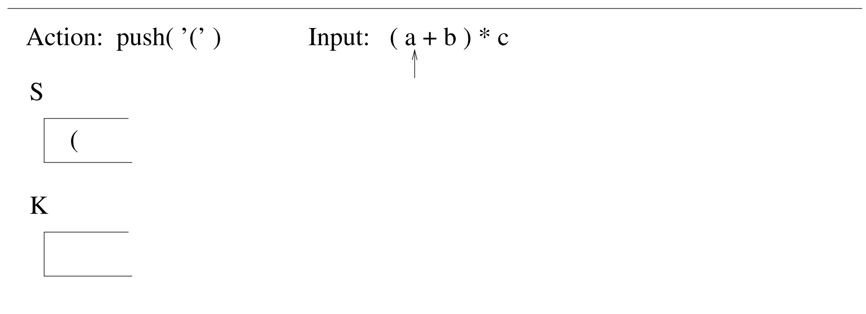
New code:

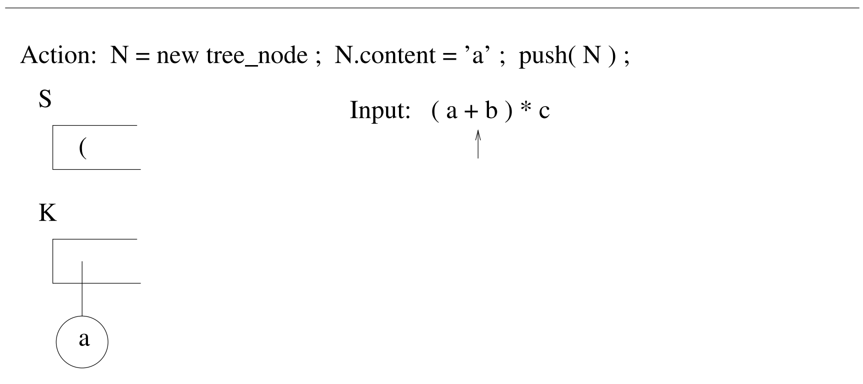
```
if ( isalpha(ch) ) {
      what_type = vari ;
      details.variable[0] = ch ;
      int ch_count = 1 ;
      do {
         ch = buffer[pos] ; pos++ ;
         if ( isalpha(ch) ) {
            details.variable[ch_count] = ch ;
            ch_count++ ;
         }
      } while ( isalpha(ch) ) ;
      pos-- ; // Put this character back.
      details.variable[ch_count] = '\0' ;  // Null terminate the string.
}
```

3. Describe the steps taken by an operator precedence parser to process the expression `(a + b) * c` and build a syntax tree. Draw a sequence of diagrams illustrating the contents of the operator stack, the tree stack, the current character, and the actions taken by the parser.

Solution: See the sequence of diagrams below for the actions taken by an operator precedence parser on input `(a + b) * c`.

Input: (a + b) * c   Initially, input character is '('

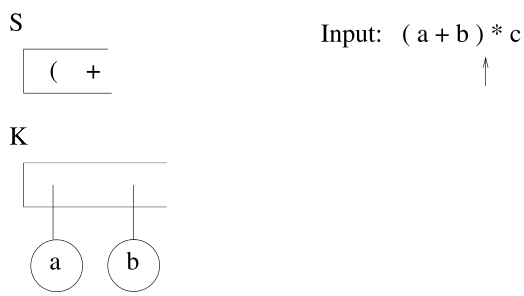




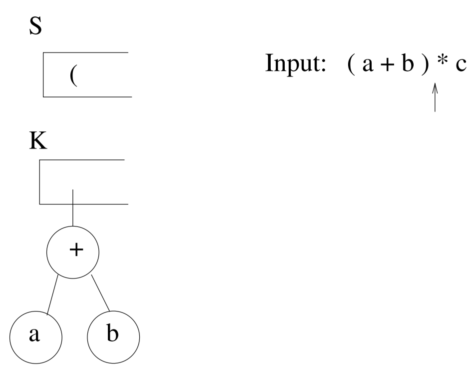
Action: push(+)

Input: (a + b) * c

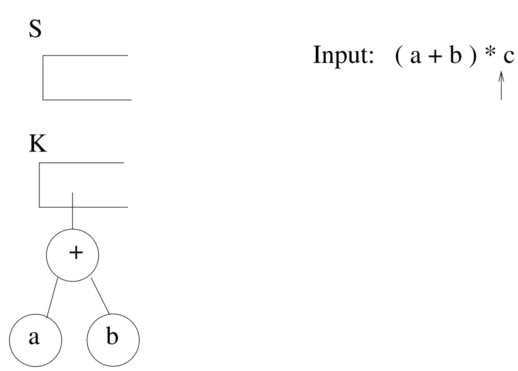
Action: N = new tree_node ; N.set_content(’b’) ; push(N) ;



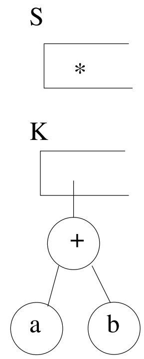
Action: Pop all operations to matching ’(’ and build tree.



Action: Pop the ’(’ and discard the matching ’)’.

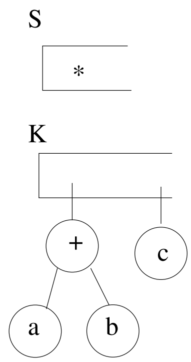


Action: push(*)



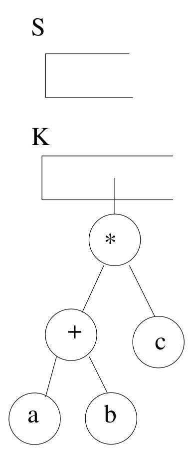
Input: (a + b) * c

Action: N = new treenode ; N.set_content('c') ; push (N) ;



Input: (a + b) * c

Action: Pop remaining ops on S and build tree



Input: (a + b) * c

Parse complete.

4. Consider the following class `two_d_point`

```
class two_d_point {
   private:
      double x ;
      double y ;
   public:
      two_d_point() { x = 0.0 ; y = 0.0 ; }
      two_d_point(double a, double b) { x = a ; y = b ; }
      ~two_d_point() { }
      // Add operator "<" here.
} ;
```

Add the operator "less than" (symbol $<$) to the class `two_d_point`. Add the appropriate declaration to the class declaration (above). Give an implementation, written outside the class using the scope resolution operator. **Note:** A point a is "less than" point b if it is closer to the origin $(0, 0)$.

Solution:

```
class two_d_point {
   private:
      double x ;
      double y ;
   public:
      two_d_point() { x = 0.0 ; y = 0.0 ; }
      two_d_point(double a, double b) { x = a ; y = b ; }
      ~two_d_point() { }
      // Add operator "<" here.
      bool operator<( two_d_point b ) ;
} ;

bool two_d_point::operator<( two_d_point b )
{
   double d0, d1 ;

   d0 = sqrt( x*x + y*y ) ;  // Distance for left operand.
   d1 = sqrt( b.x*b.x + b.y*b.y ) ;  // Distance for right operand.
   return (d0 < d1) ;
}
```

5. Given the following declaration for a circular-array implementation of a queue of integers:

```
#define MAXQ 100
class queue {
  private:
    int head ;
    int tail ;
    int q[MAXQ] ;
  public:
    queue() { head = 0 ; tail = 0 ; }
    ~queue() { }
    void enqueue( int x ) ;
    int dequeue( ) ;
    bool is_full( ) ;
    bool is_empty( ) ;
} ;
```

(a) Write implementations for the functions `enqueue()`, `dequeue()`, `is_full()`, and `is_empty()`.

Solution:

```
#include <iostream>
#include <cstdlib>
using namespace std ;

  #define MAXQ 100
   class queue {
     private:
       int head ;
       int tail ;
       int q[MAXQ] ;
     public:
       queue() { head = 0 ; tail = 0 ; }
       ~queue() { }
       void enqueue( int x ) ;
       int dequeue( ) ;
       bool is_full( ) ;
       bool is_empty( ) ;
   } ;

  void queue::enqueue( int x )
  {
     if ( is_full() ) {
        cerr << "enqueue():  Error: queue is full." << endl ; exit(1) ;
     }
     else {
       q[tail] = x ;   tail = (tail + 1) % MAXQ  ;
     }
  }
```

```
int queue::dequeue( )
{
   int r ;
   if ( is_empty() ) {
      cerr << "dequeue():  Error: queue is empty." << endl ; exit(2) ;
   }
   else {
     r = q[head] ;   head = (head + 1) % MAXQ  ;
     return(r) ;
   }
}

// When tail is advanced, and wraps around to hit the head,
// then the queue is considered full.
bool queue::is_full( )
{
   return ((( tail + 1 ) % MAXQ ) == head )  ;
}

// When head and tail are equal, we consider the queue to be empty.
bool queue::is_empty( )
{
   return ( tail == head )  ;
}
```

6. Illustrate (using a sequence of diagrams) the results of inserting the following numbers into a 2-3 tree (of integers):

   `22 12 30 18 17 8 24 32 19`

   Solution: See a solution as a separate document, **http://menehune.opt.wfu.edu/CSC221**

7. A 2-3 tree with two levels can have a maximum of 8 data items. Consider an arbitrary sequence of eight input numbers used to build a 2-3 tree. Which of the following is true ?

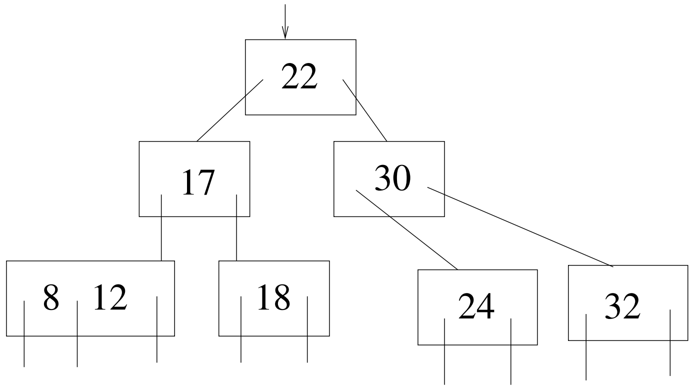
   (a) Eight numbers will always result in a 2-3 tree with two levels.

   (b) The number of levels will depend on the order in which the numbers are inserted.

   Solution: Alternative 7a is false; alternative 7b is true. The number of levels depends on the numbers involved and the order in which they are inserted. Consider the first eight numbers in problem 6 :

   `22 12 30 18 17 8 24 32`

   A 2-3 tree built with these numbers (in this order) is given below:



8. Illustrate (using a sequence of diagrams) the result of inserting the following words in a trie:

```
carrot car cat catch chill check apple dog dock deck
```

Solution: See the solution given in class.

9. Intro to complexity analysis:

   (a) Give the definition of Big $\mathcal{O}$ notation. I.e., what do we mean (precisely) when we write:

   $$f(n) \in \mathcal{O}(g(n))$$

   Solution:
   $f(n) \in \mathcal{O}(g(n))$ means there exists $c > 0$ and an $N_0 > 0$ such that $|f(n)| \leq c|g(n)|$ for all $n \geq N_0$.

   (b) Give the definition of little o notation: I.e., what do we mean (precisely) when we write:

   $$f(n) \in \mathrm{o}(g(n))$$

   Solution: $f(n) \in \mathrm{o}(g(n))$ means for all $c > 0$, there exists an $N_0 > 0$ such that $|f(n)| \leq c|g(n)|$ for all $n \geq N_0$. An equivalent and more compact definition is:

   $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

   (c) Derive an asymptotic upper bound (Big $\mathcal{O}$) for the following iterative code segment. Assume that n is a power of two; i.e., $n = 2^k$ for some $k \geq 0$.

```
t = 0 ;
while ( n > 1 ) {
   m = n ;
   while ( m > 1 ) {
      t ++ ;
      m = m / 2 ; // Integer divide.
   }
   n = n / 2 ; // Integer divide.
}
```

   Solution: Given in class.

(d) Derive an asymptotic upper bound (Big $\mathcal{O}$) for the following recursive code segment. Assume that n is a power of two; i.e., $n = 2^k$ for some $k \geq 0$.

```
int f( int n )
{
   if ( n == 1 ) return 1 ;
   else {
      int t = 0 ;
      for ( int i = 0 ; i < n ; i++ ) t += i ;
      return t + f( n/2 ) ;
  }
}
```

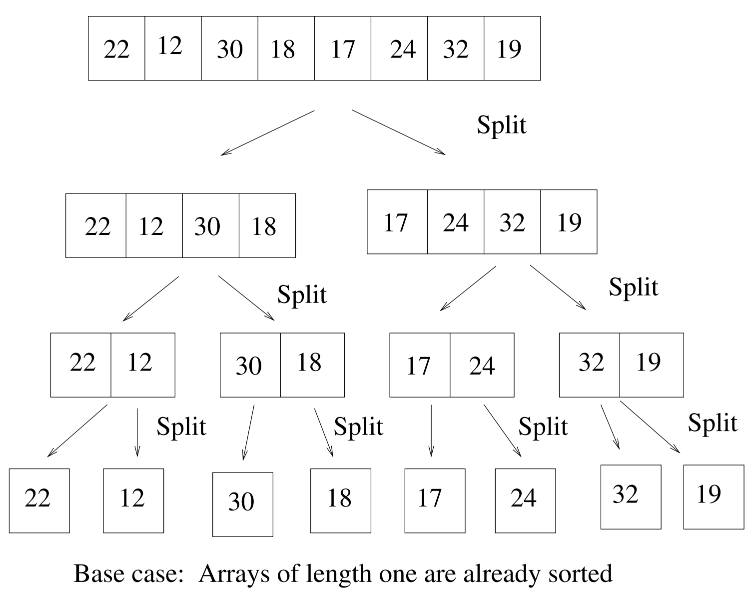
Solution: Given in class.

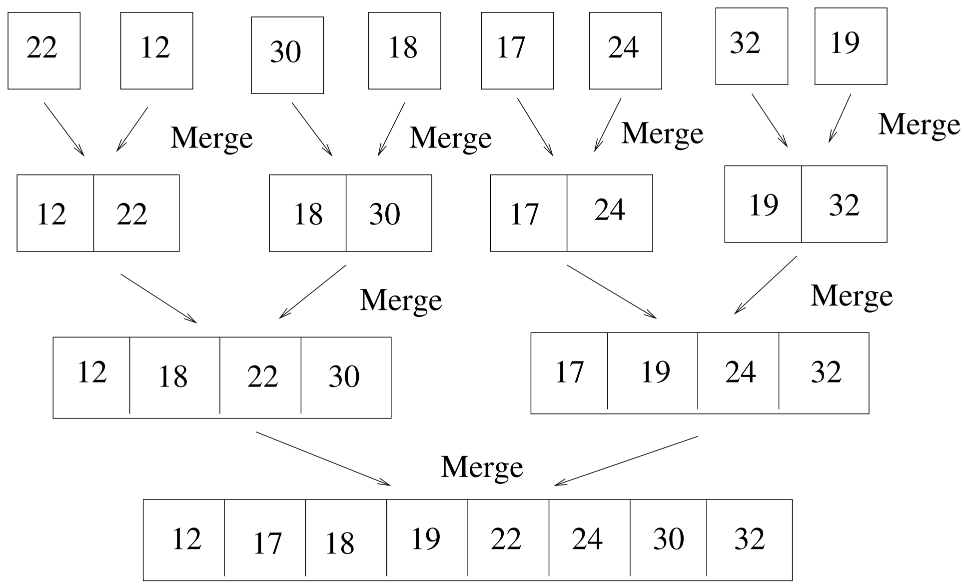
10. Mergesort

Draw a tree illustrating the sequence of function calls (and the returned results) when mergesort is applied to:

22	12	30	18	17	24	32	19

Solution: On the way down the recursive levels, `mergesort()` splits the array. In the base case, each array of length one are already sorted. On the way back up from the recursive levels, the intermediate results are merged.




22
12
30
18
17
24
32
19
Merge
Merge
Merge
Merge
12
22
18
30
17
24
19
32
Merge
Merge
12
18
22
30
17
19
24
32
Merge
12
17
18
19
22
24
30
32