You may take this test with you afterwards, but you must turn in your answer sheet.

This test has the following sections:
   I. True/False ....................... 80 points; (40 questions, 2 points each)
   II. Multiple Choice............... 20 points; (10 questions, 2 points each)

This test is worth 10% of your final grade. You must put your answers on the bubble form. You are allowed to have resources with you printed on paper, but no computers. For the multiple choice problems, select the best answer for each one and select the appropriate letter on your answer sheet. Be careful - more than one answer may seem to be correct. Some questions are tricky.

**True/False: (2 points each)** On your bubble form fill out A for true and B for false.

1. By definition a queue must be implemented using a linked-list.  
   T  F

2. Both arrays and linked lists can be dynamically allocated.  
   T  F

3. The order of an algorithm can be determined by repeatedly running a program.  
   T  F

4. The order of an algorithm can be determined by looking at the code.  
   T  F

5. If for some reason you only had space to store either the circularly linked list tail pointer or the head pointer, it would be best to store the head. Assume the list is singly-linked.  
   T  F

6. Any code that can be written recursively can be rewritten without recursion.  
   T  F

7. An algorithm that is \( O(n \log n) \) is faster than a second algorithm which is \( O(n) \) because the value \( n \log n \) is smaller than \( n \) when \( n \) is 2,3,4,5,…  
   T  F

8. Typically there is a tradeoff between computation and storage.  
   T  F

9. The following shows correct dominance relationships with regards to Big-Oh:
   \( O(n!) \gg O(2^n) \gg O(n^2) \gg O(\log n) \gg O(n \log n) \)  
   T  F

10. A business holds a competition for an algorithm that will work with groups of no more than 1,000 items at a time. For this data your \( O(n^3) \) algorithm is faster than your friend’s is \( O(2^n) \) algorithm.  
    T  F

11. When it comes to comparing programs that are in different Big-Oh dominance classes, constants can be ignored, regardless of whether or not we know the maximum size of the input.  
    T  F

12. An algorithm that runs in \( O(2n) \) runs faster than another version that runs in \( O(4n) \).  
    T  F
13. $n$ unique consecutive values could be sorted in $O(n)$ time by using an additional array of size $n$.

14. If inserting a value into a Binary Search Tree (BST) is $O(\log n)$, then inserting $n$ values into a BST will always be $O(n \log n)$.

15. Given a proper implementation of a stack, along with properly defined operators `push(...)` and `pop(...)`, the code shown below would allow us to use a manually implemented stack to reverse input, so that input of: 251$ would give as output: 152.

```c
void f1()
{
    Node *pTop = NULL;
    char value;

    do {
        cin >> value;
        if( value!='$') {
            push( pTop, value);
        }
    } while (value!='$');

    while( pTop!=NULL) {
        cout << pop(pTop);
        pTop->pNext;
    }
}
```

16. A node is another name for a vertex in a tree.

17. A tree is a set of nodes connected by edges where there is precisely one path connecting any two nodes.

18. In an ordered tree the two children of a node can be exchanged without affecting the ability to traverse the tree and display the nodes in order.

19. Nodes with no children are leaf nodes only when they are at the bottom-most level in the tree.

20. Every full binary tree is also a complete binary tree.

21. A binary tree is a tree where each internal node always has two children.

22. A pre-order binary tree traversal can be written without using recursion, but only if you also implement a parent pointer for each node.

23. Given an unsorted binary tree of $n$ randomly distributed integer values, it is possible to use only the `insert(...)` and the `preOrderTraversal(...)` functions of a binary search tree along with a stack to print out those $n$ values in descending sorted order.

24. Given an unsorted binary tree of $n$ randomly distributed integer values, it is possible to use
only the \textit{insert(...)} and the \textit{inOrderTraversal(...)} functions of a binary search tree to print out those \(n\) values in sorted order.

\begin{itemize}
\item $T$  $F$  25. Given an unsorted binary tree of \(n\) randomly distributed integer values, it is possible to use only the \textit{minimum(...), successor(...) and the insert(...)} functions of a binary search tree to print out those \(n\) values in sorted order in \(O(n \log n)\). \(\text{Traverse original tree, building a BST using insert(). Call minimum() & successively call successor().}\)
\item $T$  $F$  26. Given a well-balanced Binary Search Tree (BST) of \(n\) integer values, it is possible to use only the \textit{minimum(...)} and the \textit{delete(...)} functions of a binary search tree to print out those \(n\) values in sorted order in \(O(n \log n)\).
\item $T$  $F$  27. Consider if the values inserted into a Binary Search Tree (BST) were inserted in the following order: 50, 49, 51, 48, 52, 47, 53, … and so on, starting at 50 and alternating each time to the next smallest and the next largest. The resulting BST will be a complete tree.
\item $T$  $F$  28. Assume you have a binary search tree with \(O(\log n)\) for search, where the Node declaration contains a field for \textit{data}, and a field each for \textit{leftChild} and \textit{rightChild}, there is a tree \textit{root} pointer and there are no other pointers into parts of the tree. Assuming some node \(A\) is already found, implementing Node * \textit{predecessor(A)} in \(O(\log n)\) requires modifying the Node declaration.
\item $T$  $F$  29. Assume a full Binary Search Tree (BST) has been built using \(n\) integer values. Regardless of the distribution of the input values used to build the tree, those values can be printed out in order in \(O(n)\) time.
\item $T$  $F$  30. Assume a binary search tree has been built using \(n\) integer values. Regardless of the distribution of the input values used to build the tree, any value can be found in \(O(n)\) time or better.
\item $T$  $F$  31. Assume a binary search tree has been built using \(n\) integer values. Regardless of the distribution of the input values used to build the tree, any new value can be added in \(O(\log n)\) time.
\item $T$  $F$  32. Assuming you are writing a program where a very large number of values are read in at the beginning of the day where time is not at all an issue. Then throughout the day the smallest value is displayed and then deleted, where doing this as quickly as possible is extremely important. Implementing this as a linked list of sorted values will give the same performance as implementing it using a binary search tree. \textit{Linked list will be faster}
\item $T$  $F$  33. In a Binary Search Tree (BST), for any node \(p\text{Temp}\) we can find the next greatest value using:
\[
\text{getMin( pTemp->pRight)}
\]
34. Adding a dummy head node to an ordered linked list simplifies the code to insert a node. **T**

35. Adding a dummy head node to a circularly-linked list simplifies the code to do a list traversal. **F**

36. **Locality of reference** and **multi-level caching** explain why doing many inserts of random numbers into an ordered linked list is faster than doing the same thing using an array. **T**

37. The following code using an explicit stack will display the `theNumber` equivalent in binary, but with the digits in reverse order. **T**

```c
void decimalToBinaryStack( int theNumber) {
    Node *pHead = NULL;
    while( theNumber > 0) {
        push( pHead, theNumber%2);
        theNumber = theNumber / 2;
    }
    while( pHead!=NULL) {
        printf("%d", pop(pHead));
    }
}
```

38. The following code to do an in-order traversal of a binary search tree could be rewritten without using recursion: **T**

```c
void inOrderTraversal( Node * pRoot) {
    if( pRoot != NULL) {
        inOrderTraversal( pRoot->pLeft);  // recurse down to the left
        cout << pRoot->data << " ";   // display contents
        inOrderTraversal( pRoot->pRight);  // recurse down to the right
    }
} // end inOrderTraversal(...)
```

39. The following code works properly to search for a value in a non-empty binary search tree: **F**

```c
Node * searchTree( Node *pRoot, int searchValue) {
    if( pRoot->data == searchValue) {
        return pRoot;
    } else if( searchValue < pRoot->data) {
        return searchTree( pRoot->pLeft, searchValue);}
    else {
        return searchTree( pRoot->pRight, searchValue);
    }
    return NULL;
}
```
The following code works properly to allow inserting duplicate values into a binary search tree:

```c
void insertIntoTree( Node *pRoot, int value)
{
    if (pRoot == NULL) {
        pRoot = newNode(value, NULL, NULL);
    } else if (value < pRoot->data) {
        insertIntoTree(pRoot->pLeft, value);
    } else {
        insertIntoTree(pRoot->pRight, value);
    }
}
```

Multiple Choice (2 points each)

41) The expression \( A+B*C+D \) converted to postfix is:

- a) \( ABC*+D+ \)
- b) \( A+BC*+D \)
- c) \( AB+C*D+ \)
- d) \( ABC*D++ \)
- e) None of the above

42) The equivalent of the expression \( ABC+/D* \) in infix is:

- a) \( (A+B)/C*D \)
- b) \( (A/B)+(C*D) \)
- c) \( A/B+C*D \)
- d) \( A/(B+C)*D \)
- e) None of the above

43) The expression \( (A \ast (B + (D \ast E))/F) + (G + H) \ast I \) converted to postfix is:

- a) \( ABDE**F/GH*I++ \)
- b) \( ABDE**F/GH++I* \)
- c) \( AB+D*E*F/G+HI++ \)
- d) \( AB+DE**F/GH++I* \)
- e) None of the above
44) Consider the following code used to convert a properly formatted infix expression to postfix:

```c
void convertInfixToPostfix( char inputLine[]) {
    Node *pTop = NULL; // create the stack top pointer
    char c; // current input character
    for(int i=0; i<strlen(inputLine); i++) {
        // get next input character
        c = inputLine[i];
        // Handle operators
        else if(c == '+' || c == '-' || c == '*' || c == '/') {
            while( pTop!=NULL && (operatorWeight(pTop->data) >= operatorWeight(c)) )
                cout << pop( pTop);
            push( pTop, c);
        } else if( c>='0' && c<='9')
            cout << c;
    } // end for( int i...
    while(pTop != NULL) {
        cout << pop( pTop);
    }
}
```

For the sample input: 2*3*4+5 what is the effect if the >= in the inner while loop (pointed to by the arrow) is converted to ==, assuming the rest of the code is correct, as seen in class?

a) Two of the options below are true
b) Multiplication and addition have the same order of precedence
c) Addition always gets precedence over multiplication
d) Multiplication always gets precedence over addition
e) Precedence within multiplication ends up right-to-left instead of left-to-right

45) What is the order of complexity (Big-Oh) for the section of code shown at right below?

```c
for(int i=0; i<n; i++) {
    for(int j=0; j<i; j++) {
        cout << i+j << " ";
    }
    cout << endl;
}
```
Consider the following code:

```c
void sort1(int theArray[], int arraySize)
{
    for (int pass=1; pass < arraySize; pass++) {
        for (int current=0; current < arraySize-pass; current++) {
            if (theArray[current] > theArray[current+1]) {
                swap(theArray, current, current+1);
            }
        }
    }
}
```

The run-time complexity (Big-Oh) for the above code is:

a) Constant: O(1)
b) Linear: O(n)
c) Logarithmic: O(log n)
d) Loglinear: O(n log n)
e) Quadratic: O(n²)

47) Assume you have a stream of n integer values in ascending order that you want to insert into a binary search tree. What is the run-time complexity (Big-Oh) of running this program on this data?

a) O(n) for each of n values, so O(n²)
b) O(n²)
c) O(n log n)
d) O(log n)
e) O(n)

48) Assume you have a stream of n integer values in random order that you want to insert into a binary search tree. What is the run-time complexity (Big-Oh) of running this program on this data?

a) O(n)
b) O(n²)
c) O(n log n)
d) O(log n)
e) O(n)
49) Consider a full binary tree $T$ stored in an array, where the nodes are numbered in a breadth-first fashion, where the root is 1, its immediate left and right children are 2 and 3, and their children respectively are 4, 5 and 6, 7, and so on. Which of the following are true about $T$?

- A. It could be implemented using a static (not dynamic) array, but the total number of Nodes must be known.
- B. Implementation using an array would require less storage than using a tree of Node structures.
- C. In an array implementation, which level a node is in can be determined by its node number.
- D. Accessing any node can be done in constant time $O(c)$.

a) A, B  

b) A, C, D  

c) A, B, D  

d) B, C, D  

e) A, B, C, D

50) Consider the following code to find the parent node of some given node $pTemp$.

The parent should be assigned to parameter $pParent$.

```c
void findParent2(Node *pRoot, Node *pTemp, Node *&pParent)
{
    if (pRoot == NULL) {
        return;
    }
    else if (pRoot->pLeft == pTemp || pRoot->pRight == pTemp) {
        pParent = pRoot;
        printf("Found"); // parent was found
    }
    else {
        findParent2(pRoot->pLeft, pTemp, pParent);
        if (pParent == NULL) {
            findParent2(pRoot->pRight, pTemp, pParent);
            printf("*"); // print an asterisk
        }
    }
}
```

Which choice best describes the behavior of this code?

a) It never prints Found  

b) Even when the node is in the tree, it only prints Found in some circumstances  

c) After printing Found it prints exactly one asterisk  

d) After printing Found it may print more than one asterisk  

e) None of the above