**Questions on the exam will be of the same type and style as the following questions. Specific details of most questions will differ from the questions below, but there will be no "surprise" or "trick" questions on the exam.**

## Asymptotic Analysis

1. Define "Big Oh," "Big Omega," and "Big Theta." Use formal, mathematical definitions.
2. Number these functions in ascending "Big Oh" order:

| Number | Big Oh |
|---|---|
| | $O(n^3)$ |
| | $O(n^2 \log n)$ |
| | $O(1)$ |
| | $O(\log^{0.1} n)$ |
| | $O(n^{1.01})$ |
| | $O(n^{2.01})$ |
| | $O(2^n)$ |
| | $O(\log n)$ |
| | $O(n)$ |
| | $O(n \log n)$ |
| | $O(n \log^5 n)$ |

3. Prove:

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

4. Let $T_1(x) = O(f(x))$ and $T_2(x) = O(g(x))$. Prove $T_1(x) + T_2(x) = O(max(f(x), g(x)))$
5. Prove: $O(cf(x)) = O(f(x))$, where $c$ is a positive constant.
6. Let $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$. Prove $T_1(n)T_2(n) = O(f(n)g(n))$
7. Prove $2^{n+1} = O(2^n)$
8. Show the following using one or more of the theorems in questions 4, 5, 6 and 7. Be explicit and mention *each* of the theorems that applies.
   (a) $n^2 + 2n = O(n^2)$
   (b) $n^2 - 3n = O(n^2)$
   (c) $n^2 + 98 = O(n^2)$

9. The following C++ function computes the maximum contiguous subsequence sum of elements in the given vector.

```
int maxSubSum2 (const vector<int> & a)
{
  int maxSum = 0;

  for (int i = 0; i < a.size(); i++)
  {
    int thisSum = 0;
    for (int j = i; j < a.size(); j++)
    {
      thisSum += a[j];
      if (thisSum > maxSum)
        maxSum = thisSum;
    }
  }
  return maxSum;
}
```

What is the Big-Oh asymptotic time performance of this function, worst-case, as a function of the vector size $N$?

10. The following C++ function computes $\sum_{i=1}^{N} i^3$

```
int sum (int N)
{
  int partialSum;

  partialSum = 0;
  for (int i = 1; i <= N; i++)
    partialSum += i * i * i;
  return partialSum;
}
```

What is the Big-Oh asymptotic time performance of this function, worst-case, as a function of the argument $N$?

## C++, ADTs, OOP

11. What does it mean when a C++ member function is declared to be const? For example, `int foo(float) const;`

12. The following C++ function returns a reference, but it's not a good idea. Why not?

```
int & Foo ()
{
  int temp = 5;
  return  temp;
}
```

13. Rewrite the following C++ function as a template function which can take any (identical) type reference for x and y and return the appropriate type.

```
int Foo (int & x, int & y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
  return x + y;
}
```

Show how the function would be called with `f` and `g` as arguments in:

```
{
  float f = 1.2;
  float g = 3.7;

  // call to Foo with f and g as arguments is:
}
```

14. The text uses template parameters `Object` and `Comparable`. Explain what is meant by each.
15. Consider the following ADT for `Set`:

    - A `Set` is a homogeneous collection of elements with no duplicates.
    - The operations are:
      - `bool isEmpty()` - returns `true` if this `Set` contains no elements.
      - `bool isFull()` - returns `true` if this `Set` can accept no more elements.
      - `bool insert(Element e)` - inserts `e` into this `Set` given that the no-duplicate property will not be violated and this `Set` is not full. Returns `true` if the operation succeeds.

    Define the `C++` class `Set` such that the elements are stored in a normal C array. Do not implement `Set`, just define it. Show relevant data members and describe their meaning. Assume the elements are of type `Object`.

16. Define two classes in enough detail to show an example of composition (also known as aggregation or the HAS-A relationship). Do not implement the classes.

## Stacks and Queues

Please see the definition of `Stack` on page 9 and of `Queue` on page 10. These are the definitions from the text with minor modifications noted.

17. Suppose that `q` is an object of the `Queue` class and was constructed as

```
Queue<char> q; // Queue of size 5, initially empty
```

For the following problems, assume that `q` is initially empty. Show the contents of `element` and the values of data members `front` and `back` initially and after each statement has executed. Indicate any errors that occur.

```
(a)               initial:   _ _ _ _ _    front=     back=

   q.enqueue('A');           _ _ _ _ _    front=     back=

   q.enqueue('B');           _ _ _ _ _    front=     back=

   q.enqueue('C');           _ _ _ _ _    front=     back=

   char ch = q.Dequeue();    _ _ _ _ _    front=     back=

   q.enqueue(ch);            _ _ _ _ _    front=     back=
(b)               initial:   _ _ _ _ _    front=     back=

   q.enqueue('X');           _ _ _ _ _    front=     back=

   q.enqueue('Y');           _ _ _ _ _    front=     back=

   q.enqueue('Z');           _ _ _ _ _    front=     back=
   while (!q.Empty())
     {
       char ch = q.Dequeue();  _ _ _ _ _    front=     back=  // each time
     }
(c)               initial:   _ _ _ _ _    front=     back=

   char ch = 'q';
   for (int i = 1; i <= 3; ++i)      // show result for each iteration
     {
       q.enqueue(ch);       _ _ _ _ _    front=     back=

       ch++;                _ _ _ _ _    front=     back=

       q.enqueue(ch);       _ _ _ _ _    front=     back=

       q.Dequeue();         _ _ _ _ _    front=     back=
     }
```

18. Use the given operations for `Stack` for this problem. Write a `C++` function

```
template <class Object>
Stack<Object> copyStack(Stack<Object> & stk)
```

that returns a `Stack` containing the elements of `stk`, in the same order as in `stk`.

19. Use the given operations for `Queue` and `Stack` for this problem.
Write a `C++` program that determines if a given string is a palindrome (*i.e.* reads the same forward and backward). Your program should print (to `cout`) "palindrome" if the string is a palindrome and "not-palindrome" if it is not. You may assume that the string to be tested is initially stored in a `null`-terminated array of `char`. You are to `push` each character onto a stack and `enqueue` it onto a queue. The test is to use only operations on the queue and the stack.

20. Describe (pseudo-code is fine) how the operations `isEmpty`, `pop`, and `push` are implemented in the text's array implementation of `Stack` (shown on page 9).

21. Describe (pseudo-code is fine) how the operations `isEmpty`, `pop`, and `push` are implemented in the `List`-based implementation of `Stack` given in the lecture notes.

22. Describe the advantages and disadvantages of the text's array and the lecture note's list implementations of the *stack* ADT. Consider the asymptotic behavior for each of the operations `isEmpty`, `pop`, and `push`. Also consider the storage requirements for each implementation.

## Lists

Please refer to the definition of `ListNode` on page 10, `List` on page 11, and `ListItr` on page 11. Assume that all the member functions of the classes have been defined and are available for your use. You do not have to define them.

23. Suppose you have constructed a `List` named `lst` that may or may not have elements in it. Write valid `C++` code to construct an iterator for `lst` that is positioned at the last element. If `lst` is empty, the iterator is to be past end.

24. (a) Write a member function for `List`

```
template <class Object>
void List<Object>::reversePrint(ostream & out)
```

that uses `ListItrs` to print the elements of the `List` to `out` in reverse order (from tail to head). The elements are to be comma-separated and enclosed in "<>" delimiters.

(b) What is the "Big-Oh" asymptotic time performance of this method in best, worst, and average cases. Please describe each of the cases.

25. Write a definition for `splice`, a new member function of the `List` class:

```
template <class Object>
bool List<Object>::splice (const List<Object>& lst,
                           ListItr<Object> & pos)
```

This function "splices" the specified list `lst` into this `List` object at the specified position (`pos` is a `ListItr` over "this" `List`. `splice` returns false if `pos` is past end.

What is the worst-case Big-Oh asymptotic time performance of your method?

Example: Suppose `lst1 is (1,2,3,4,5)` and `lst2 is (10,20,30)`. Suppose `itr` is constructed as `lst1.first()`, and advanced twice so it is positioned at the "3" element. Then the expression `lst1.splice(lst2,itr)` returns `true` and causes `lst1` to become `(1,2,3,10,20,30,4,5)`

26. Given two lists of lengths $n1$ and $n2$:

$$L_1 = (x_1, x_2, \cdots, x_{n1})$$
$$L_2 = (y_1, y_2, \cdots, y_{n2})$$

the *shuffleMerge* of $L_1$ and $L_2$ is the new list

$$L_3 = (x_1, y_1, x_2, y_2, \cdots)$$

Examples:

```
L1 = <1,2,3>; L2 = <4,5,6>;  shuffleMerge ==> <1,4,2,5,3,6>

L1 = <1,2,3>; L2 = <4>;      shuffleMerge ==> <1,4,2,3>

L1 = <1,2,3>; L2 = <>;       shuffleMerge ==> <1,2,3>
```

Write a definition of:

```
template <class Object>
List<Object>
List<Object>::shuffleMerge(const List<Object> lst)
```

that returns a new List produced by the shuffle merge of this list with `lst`. An empty list is returned if the given lists are both empty.

27. Fill in the following table of Big-Oh asymptotic time performance for the given operations on `List` (page 11). Give your answers in terms of $n$, the number of elements in the list.

| Operation | Best Case | Average Case | Worst Case |
|---|---|---|---|
| `insert` | | | |
| `find` | | | |
| `findPrevious` | | | |
| `remove` | | | |
| `makeEmpty` | | | |
| `isEmpty` | | | |
| `first` | | | |

## General Trees

28. Define *tree*.

29. For any tree $T$, define *path* in $T$, *length* of a path in $T$, *height* of a node in $T$, *depth* of a node in $T$, *height* of $T$, and *depth* of $T$.

30. Know how to draw the binary tree (first child, next sibling) representation of a given general tree.

31. Know how to draw the general tree represented by a given binary tree (first-child, next-sibling) representation.

32. Prove: there are $n - 1$ edges in any tree with $n$ nodes.

## Binary Trees

33. Define *binary tree*, *full* binary tree, *complete* binary tree, and *perfect* binary tree.

34. Define *internal node* and *external node* of a rooted tree.

35. Define *internal path length* and *external path length* of a rooted tree.

36. Define *augmented* binary tree.

37. Prove: A perfect binary tree of height $h$ has $2^h$ leaf nodes.

38. Prove: A perfect binary tree of height $h$ has $2^{h+1} - 1$ vertices.

39. Prove: A full binary tree with $n_i$ internal nodes has $n_i + 1$ leaf nodes.

40. Prove: In any binary tree of $n$ nodes, there are $n + 1$ "null pointers."

41. Consider a binary tree that has $n_i$ *internal* nodes. Let $E(n_i)$ and $I(n_i)$ denote the external and internal path lengths of the tree, respectively. Show that for a full binary tree, $E(n_i) = I(n_i) + 2n_i$.

42. Prove: the internal path length of an augmented binary tree T with $n$ internal nodes is

$$I(n) = I(n_L) + I(n - n_L - 1) + n - 1$$

where $n_L$ is the number of internal nodes in the left subtree.

43. Suppose you have the following two traversal sequences from the same binary tree:

*pre-order*: `A D F G H K L P Q R W Z`

*in-order*: `G F H K D L A W R Q P Z`

Draw the tree.

44. Suppose you have the following two traversal sequences from the same binary tree:

*post-order*: `F G H D A L P Q R Z W K`

*in-order*: `G F H K D L A W R Q P Z`

Draw the tree.

## Binary Search Trees

Please see the definition of `BinaryNode` on page 13 and of `BinarySearchTree` on page 12.

45. Define *binary search tree.*

46. Write valid `C++` code to complete the definition of the `find` member function of the `BinarySearchTree` class. The function takes two arguments:

`const Comparable & x` the element to search for

`BinaryNode<Comparable> * t` pointer to the node at which to start the search.

If `x` is found in the tree, a pointer to its node is returned; otherwise return `NULL` (see definition of the `BinarySearchTree` class on page 12).

```
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::
find( const Comparable & x, BinaryNode<Comparable> *t ) const
{
   // write the body of the method
}
```

47. Describe how deletion is performed in a binary search tree. Mention all relevant cases.
48. Explain: the number of comparisons needed to build a binary search tree of $n$ elements is $O(n^2)$ worst case and $O(n \lg n)$ best case. Describe the best and worst cases.
49. For the following list of `C++` keyword strings:
    `break, operator, if, typedef, else, case, while, do, return, unsigned, for, true, double, void`
    (a) Draw the binary search tree that results when the words are inserted into an initially empty tree in the order given. Show the tree after each insertion. Assume that `operator<` for strings means "alphabetical order."
    (b) Show the sequences of keywords under pre-order, post-order, and in-order traversals of the tree.
50. Given the following binary search tree:

```
                50
                /\
               /  \
              /    \
             /      \
           15        62
           /\        /\
          /  \      /  \
         5    20  58    91
        / \    \    \
       /   \    \    \
      3     8   37   60
                /
               /
             24
```

    Draw the binary search trees that result after each of the following operations or sequences of operations are performed. Each problem below is separate and begins with the tree shown above.
    (a) Insert 7
    (b) Insert 7, 1, 55, 29, and 19, in that order
    (c) Delete 8
    (d) Delete 8, 37, and 62, in that order
    (e) Insert 7, Delete 8, Insert 59, Delete 60, Insert 92, Delete 50, in that order
51. Suppose you have the following traversal sequence from a binary search tree:
    *pre-order*: `L H D A F G K W R Q P Z`
    Draw the tree.

52. Give an example of a binary search tree situation in which the immediate predecessor of some node is not in that node's left subtree.

53. Give an example of a binary search tree situation in which the immediate successor of some node is not in that node's right subtree.

54. Write pseudocode for finding the immediate predecessor of a node in a binary search tree.

55. Write pseudocode for finding the immediate successor of a node in a binary search tree.

56. Prove that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

---

## Definition of `Stack` Class

Note: this is directly out of the text. The meanings of the operations are the same as in the text.

```
template <class Object>
class Stack
{
  public:
    explicit Stack( int capacity = 10 );

    bool isEmpty( ) const;
    bool isFull( ) const;
    const Object & top( ) const;

    void makeEmpty( );
    void pop( );
    void push( const Object & x );
    Object topAndPop( );

  private:
    vector<Object> theArray;
    int            topOfStack;
};
```

## Definition of `Queue` Class

Note: this differs from the text only in that the initial size is 5, not 10. The meanings of the operations are the same as in the text.

```
template <class Object>
class Queue
{
  public:
    explicit Queue( int capacity = 5 );

    bool isEmpty( ) const;
    bool isFull( ) const;
    const Object & getFront( ) const;

    void makeEmpty( );
    Object dequeue( );
    void enqueue( const Object & x );

  private:
    vector<Object> theArray;
    int            currentSize;
    int            front;
    int            back;

    void increment( int & x );
};
```

## Definition of `ListNode` Class

This definition is directly from the text.

```
template <class Object>
class ListNode
{
  ListNode( const Object & theElement = Object( ),
            ListNode * n = NULL )
    : element( theElement ), next( n ) { }

  Object   element;
  ListNode *next;

  friend class List<Object>;
  friend class ListItr<Object>;
};
```

## Definition of `List` Class

This definition is directly from the text.

```
template <class Object>
class List
{
  public:
    List( );
    List( const List & rhs );
    ~List( );
    bool isEmpty( ) const;
    void makeEmpty( );
    ListItr<Object> zeroth( ) const;
    ListItr<Object> first( ) const;
    void insert( const Object & x, const ListItr<Object> & p );
    ListItr<Object> find( const Object & x ) const;
    ListItr<Object> findPrevious( const Object & x ) const;
    void remove( const Object & x );
    const List & operator=( const List & rhs );
  private:
    ListNode<Object> *header;
};
```

## Definition of `ListItr` Class

This definition is directly from the text.

```
template <class Object>
class ListItr
{
  public:
    ListItr( ) : current( NULL ) { }
    bool isPastEnd( ) const
      { return current == NULL; }
    void advance( )
      { if( !isPastEnd( ) ) current = current->next; }
    const Object & retrieve( ) const
      { if( isPastEnd( ) ) throw BadIterator( );
        return current->element; }
  private:
    ListNode<Object> *current;    // Current position
    ListItr( ListNode<Object> *theNode )
      : current( theNode ) { }

    friend class List<Object>; // Grant access to constructor
};
```

## Definition of BinarySearchTree Class

This definition is directly from the text.

```
template <class Comparable>
class BinarySearchTree
{
public:
  explicit BinarySearchTree( const Comparable & notFound );
  BinarySearchTree( const BinarySearchTree & rhs );
  ~BinarySearchTree( );

  const Comparable & findMin( ) const;
  const Comparable & findMax( ) const;
  const Comparable & find( const Comparable & x ) const;
  bool isEmpty( ) const;
  void printTree( ) const;

  void makeEmpty( );
  void insert( const Comparable & x );
  void remove( const Comparable & x );

  const BinarySearchTree & operator=( const BinarySearchTree & rhs );

private:
  BinaryNode<Comparable> *root;
  const Comparable ITEM_NOT_FOUND;

  const Comparable & elementAt( BinaryNode<Comparable> *t ) const;

  void insert( const Comparable & x, BinaryNode<Comparable> * & t ) const;
  //            void remove( const Comparable & x, BinaryNode<Comparable> * & t ) const;
  void remove( const Comparable & x, BinaryNode<Comparable> * & t );
  BinaryNode<Comparable> * findMin( BinaryNode<Comparable> *t ) const;
  BinaryNode<Comparable> * findMax( BinaryNode<Comparable> *t ) const;
  BinaryNode<Comparable> * find( const Comparable & x, BinaryNode<Comparable> *t ) const;
  void makeEmpty( BinaryNode<Comparable> * & t ) const;
  void printTree( BinaryNode<Comparable> *t ) const;
  BinaryNode<Comparable> * clone( BinaryNode<Comparable> *t ) const;
};
```

## Definition of `Binarynode` Class

This definition is directly from the text.

```
template <class Comparable>
class BinaryNode
{
  Comparable element;
  BinaryNode *left;
  BinaryNode *right;

  BinaryNode( const Comparable & theElement,
              BinaryNode *lt, BinaryNode *rt )
    : element( theElement ), left( lt ), right( rt ) { }
  friend class BinarySearchTree<Comparable>;
};
```