The exam will be drawn entirely from the following questions. There will be no "surprise" or "trick" questions on the exam. Every question on the exam is in this handout except that some questions may be phrased differently or involve different data.

Asymptotic Analysis

1. Define "Big Oh," "Big Omega," and "Big Theta."
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: $O(c(f(x))) = O(f(x))$, where c is a positive constant.
6. Prove $2^{n+1} = O(2^n)$

7. Show the following using one or more of the Theorems in questions 4, 5, and 6. 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) $n^2 + 98 = O(n^2)$

C++, ADTs, OOP

8. Explain why C++ programmers should provide a copy constructor for a class that has a pointer data member.

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

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

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

11. 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:
}
```

12. Under what circumstances is a method of a class "dynamically bound" in C++?

13. Consider the following two class definitions:

```
class Base                 |   class Derived  : public Base
{                          |   {
private:                   |    private:
  int _x;                  |      int _x;
public:                    |    public:
  Base() : _x(1) {}        |      Derived() : _x(2) {}
  virtual int GetX()       |      virtual int GetX()
    { return _x;}          |        { return _x; }
};                         |    };
```

What will be printed by the following program?

```
main()
{
  Base b1, *b2, *b3;
  Derived d1, *d2;
  b2 = new Base();
  b3 = new Derived();
  d2 = new Derived();

  cout << b1.GetX() << endl;
  cout << b2->GetX() << endl;
  cout << b3->GetX() << endl;
  cout << d1.GetX() << endl;
  cout << d2->GetX() << endl;
}
```

14. In the following C++ class definition, what does the line

```
class Foo : private Bar
```

mean? Why would a programmer do something like this?

```
class Foo : private Bar
{
    .
    .
};
```

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 homogeneity and duplicate properties will not be violated and this `Set` is not full. Returns `true` if the operation succeeds.

   (a) Define an abstract class `Set` for the above `Set` ADT. Define only the class, do not implement it. The intent of this class is to be used as the base class for various implementing classes. Make it a template class.

   (b) Define a class `ArraySet` that IS-A `Set`. `ArraySet` stores the `Set` elements in an `Array`. Do not implement `ArraySet`, just define it. Show relevant data members and describe their meaning. Assume there is a template class named `Array`.

   (c) Define a class `ListSet` that IS-A `Set`. `ListSet` stores the `Set` elements in a `List`. Do not implement `ListSet`, just define it. Show relevant data members and describe their meaning. Assume there is a template class named `List`.

   (d) Show the relationships among `Set`, `ArraySet`, `ListSet`, `List`, and `Array` in a class diagram.

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.

17. Define two classes in enough detail to show an example of delegation (also known as the KNOWS-A or USES-A relationship). Do not implement the classes.

Stacks and Queues

Please see the definition of `SmQueue` on page 11 and of `SlStack` on page 10. These are the definitions from the textbook.

18. Suppose that `q` is an object of the `SmQueue` class and was constructed as

```
SmQueue<char> q(5); // 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 `head` and `tail` after the code segment has executed. Indicate any errors that occur.

(a)
```
q.Enqueue('A');
q.Enqueue('B');
q.Enqueue('C');
char ch = q.Dequeue();
q.Enqueue(ch);
```

(b)
```
q.Enqueue('X');
q.Enqueue('Y');
q.Enqueue('Z');
while (!q.Empty())
  {
    char ch = q.Dequeue();
  }
```

(c)
```
char ch = 'q';
for (int i = 1; i <= 3; ++i)
  {
    q.Enqueue(ch);
    ch++;
    q.Enqueue(ch);
    q.Dequeue();
  }
```

(d)
```
char ch = 'A';
for (int i = 1; i <= 4; ++i)
  {
    q.Enqueue(ch);
    ch++;
    q.Enqueue(ch);
    ch = q.Dequeue();
  }
```

19. Use the given operations for `SmQueue` and `SlList` 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. Define an ADT for *stack*.

21. Describe an array implementation for the *stack* ADT (describe the data

structure, cursors, etc.). Describe (pseudo-code is fine) how the operations `Empty`, `Pop`, and `Push` are implemented.

22. Describe a linked-list implementation for the *stack* ADT (describe the data structure, pointers, etc.). Describe (pseudo-code is fine) how the operations `Empty`, `Pop`, and `Push` are implemented.

23. Describe the advantages and disadvantages of your array and linked-list implementations of the *stack* ADT. Consider the asymptotic behavior for each of the operations `Empty`, `Pop`, and `Push`. Also consider the storage requirements for each implementation.

Lists

Please refer to the definitions of the `SlNode` and `SlList` classes on page 12. Assume that all the member functions of `SlNode` and `SlList` have been defined and are available for your use. You do not have to define them.

24. Write the definition for `Splice`, a new member function of the `SlList` class:
    `bool SlList<T>::Splice (const SlList<T>& lst, int pos)`
    This function "splices" the specified list `lst` into the `SlList` object at the specified position. `Splice` returns false if `pos` is an invalid position.

    Example: Suppose `lst1 is (1,2,3,4,5)` and `lst2 is (10,20,30)`. Then the expression `lst1.Splice(lst2,3)` returns `true` and causes `lst1` to become `(1,2,10,20,30,3,4,5)`

25. Given two lists of length n:

$$L_1 = (x_1, x_2, \cdots, x_n)$$

$$L_2 = (y_1, y_2, \cdots, y_n)$$

the *shuffle-merge* of L_1 and L_2 is the new list

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

Write the definition for a friend template function of the `SlList` class

```
SlList<T> ShuffleMerge(const SlList<T>& L1, const SlList<T>& L2)
```

The function creates and returns a new list that is the shuffle-merge of the two specified lists. An `assertion` is raised if $L1$ and $L2$ are not of the same length. An empty list is returned if the given lists are both empty.

26. Fill in the following table of Big-Oh asymptotic time performance for the given operations on `DynList` and `SlList` as implemented in the text. Give your answers in terms of n, the number of elements in the list. Assume average cases.

Operation	`DynList`	`SlList`
`Length`		
`Insert`		
`Retrieve`		
`Delete`		
`MakeEmpty`		
`Empty`		

General Trees

27. Define *tree*.

28. 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.

Binary Trees

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

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

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

32. Define *augmented* binary tree.

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

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

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

36. 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, $E(n_i) = I(n_i) + 2n_i$.

37. 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.

38. 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.

39. 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 `BNode` on page 13 and of `BST` on page 14.

40. Define *binary search tree.*

41. Complete the definition of the `Search` member function of the `BST` class. The function takes three arguments:

    `Bnode * u` the root of the sub-tree in which to begin the search,

    `const Etype & value` the value to search for,

    `bool & success` returned `true` if value found in the subtree, `false` otherwise

    If `value` is found in the subtree, its node is returned. Otherwise, return the last node visited prior to the null pointer. (see definition of the `BST` class on page 14).

```
template <class Etype>
Bnode<Etype> *
BST::Search(Bnode<Etype> * u, const Etype & value, bool & success)const
```

42. Describe how deletion is performed in a binary search tree. Mention all relevant cases.

43. 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.

44. For the following list of C++ keywords:
`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.

   (b) Show the sequences of keywords under pre-order, post-order, and in-order traversals of the tree.

45. 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

   (c) Delete 8

   (d) Delete 8, 37, and 62

   (e) Insert 7, Delete 8, Insert 59, Delete 60, Insert 92, Delete 50

46. 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.

47. 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.

48. 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.

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

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

51. 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 `SlStack` class

```
template <class T>
class SlStack : private SlList<T>
{
public:
  SlStack(const T& elm) : SlList<T>(elm) {}
  SlStack() : SlList<T>() {}
  inline void Push(const T& elm) {Append(elm);}
  T & pop();
  inline Boolean Empty() const {return SlList<T>::Empty();}
  friend ostream & operator<<(ostream & os, const SlStack<T>& stk)
    {return operator<<(os, (SlList<T>)stk);}
};
```

Definition of SmQueue class

```
template<class T>
class SmQueue {
  protected:
    int head, tail, size;
    op_type last_op;
    T* element;
  public:
    SmQueue(int sz, const T& elm);
    SmQueue(int sz);
    SmQueue(const SmQueue<T>& que);
    SmQueue() {}
    inline ~SmQueue();
    SmQueue<T>& operator=(const SmQueue<T>&);
    void Enqueue(const T& elm);
    T& Dequeue();
    Boolean Empty() const;
    inline void MakeEmpty();
    friend ostream& operator<<(ostream& os, const SmQueue<T>& que);
};

template<class T>
inline SmQueue<T>::~SmQueue() {delete [] element;}

template<class T>
inline void SmQueue<T>::MakeEmpty() {head=0; tail=size-1; last_op=deque;}
```

Definitions of SlNode and SlList

```
template <class T>
class SlNode
{
  friend class SlList<T>;
private:
  T data;
  SlNode<T> * next;
};


template <class T>
class SlList
{
  friend class SlNode<T>;
protected:
  int _length;
  SlNode<T> * _tail; // tail->next is head of list
public:
  SlList(); // create empty list
  bool Insert(const T& elm, int pos);
  bool Append(const T& elm);
  T& Retrieve(int);
  bool Delete(int);
  int Length();
  bool Empty();
};
```

Definition of Bnode class

```
template<class T>
class Bnode
{
protected:
  T data;
  Bnode<T>* parent;
  Bnode<T>* lchild;
  Bnode<T>* rchild;
public:
  Bnode() : data(*(new T)), parent(0), lchild(0), rchild(0) {}
  Bnode(const T& elm) : data(elm), parent(0), lchild(0), rchild(0) {}

  friend class BinaryTree<T>;
  friend class BST<T>;
  friend ostream& operator<<(ostream&, const BinaryTree<T>&);
  friend ostream& operator<<(ostream& os, const Bnode<T>& v);
};
```

Definition of BinaryTree class

```
template<class T>
class BinaryTree
{
protected:
  Bnode<T>* root;
public:
  BinaryTree() : root(0) {}
  BinaryTree(const T& elm);
  inline ~BinaryTree();
  inline Bnode<T>* Root();
  inline Bnode<T>* LeftChild(Bnode<T>* nd);
  inline Bnode<T>* RightChild(Bnode<T>* nd);
  bool InsertLeaf(Bnode<T>* leaf, Bnode<T>* prnt, Child lr);
  inline bool Empty();
  void DeleteSubtree(Bnode<T>*);
  void MakeEmpty();
  void InorderSubtreePrint(ostream& os, Bnode<T>*) const;

  friend ostream& operator<<(ostream& os, const BinaryTree<T>& tree);
};
```

Definition of BST class

```
template<class T>
class BST : private virtual BinaryTree<T>
{
protected:
  Bnode<T>* SubtreeMin(Bnode<T>*) const;
  Bnode<T>* SubtreeMax(Bnode<T>*) const;
  Bnode<T>* Search(Bnode<T>*, T& elm, bool& success) const;
public:
  BST() : BinaryTree<T>() {}
  BST(const T& elm) : BinaryTree<T>(elm) {}
  T& Search(T & elm, bool& success) const;
  bool Insert(T item);
  bool Delete(T& elm);
  T& Minimum(bool& success) const;
  T& Maximum(bool& success) const;
  T& Predecessor(T& elm, bool& success) const;
  T& Successor(T&elm, bool& success) const;
  BST<T>& operator=(const BST<T>&);

  friend ostream& operator<<(ostream& os, const BST<T>& tree)
    {return operator<<(os, (BinaryTree<T>)tree);}
};
```