Chapter 0.

E2. Arrange the following functions into increasing order; that is, $f(n)$ should come before $g(n)$ in your list if $f(n)$ is $\mathcal{O}(g(n))$.

[参考答案]

$$10000 \le \log\log n \le (\log n)^3 \le n^{0.1} \le n + \log n \le n\log n \\ \le n^2 \le n^3 - 100n^2 \le 2^n$$

E3. Divide the following functions into classes so that two functions $f(n)$ and $g(n)$ are in the same class if and only if $f(n)$ is $\Theta(g(n))$. Arrange the classes from the lowest order of magnitude to the highest. [A function may be in a class by itself, or there may be several functions in the same class.]

[参考答案]

$$\{5000\},\{\log\log n^2\}, \{\log n, \log n^2\}, \{(\log n)^5\}, \{n^{0.3}\}, \\ \{n + \log n, \sqrt{n^2+4}, 4n + \sqrt{n}\}, \{n^2 - 100n, n^2\}, \{n^2 \log n\}, \{2^n\}, \{3^n\}$$

E4. Show that each of the following is correct.

[参考答案]

(a) $3n^2 - 10n\log(n) - 300$ is $\Theta(n^2)$ ✓

(b) $4n\log n + 100n - \sqrt{n+5}$ is $\Omega(n)$ ✓

(c) $4n\log n + 100n - \sqrt{n+5}$ is $o(\sqrt{n^3})$ ✓

(d) $(n-5)(n + \log n + \sqrt{n})$ is $\mathcal{O}(n^2)$ ✓

(e) $\sqrt{n^2+5n+12}$ is $\Theta(n)$ ✓

Chapter 1.

1. 设计一个算法，从顺序表中删除值在给定值s与t之间（$s \leq t$）的所有元素，如果顺序表为空，则显示出错信息并退出运行.

[参考答案]

```
template <class List_Entry>
Error_code List<List_Entry>:: removeElemInRange(
    const List_Entry &s, const List_Entry &t){
    if(count < 0) return range_error;
    for(int i=0; i<count; i++){
        List_Entry temp;
        if(retrieve(i, temp)!=success) continue;
        if(temp.key >= s.key && temp.key <= t.key){
            remove(i, temp);
    } }
    return success; }
```

2. 设计一个算法，将两个有序顺序表合并成一个有序顺序表，并由函数返回结果顺序表.

[参考答案]

```
template <class List_Entry>
List<List_Entry> List<List_Entry>::
    mergeSort(const List<List_Entry> &a, const List<List_Entry> &b){
    List<List_Entry> tempList, remainList;
    int i = 0, j = 0, k = 0, m = 0;
    while(i < a.size() && j < b.size()){
        List_Entry tempListEntry_A, tempListEntry_B;
        a.retrive(i, tempListEntry_A);
        b.retrive(j, tempListEntry_B);
        if(tempListEntry_A.key < tempListEntry_B.key){
            tempList.insert(k, tempListEntry_A);
            k++; i++;
        }else{
            tempList.insert(k, tempListEntry_B);
            k++; j++; }}
    if (i < a.size()) {remainList = a; m = i;}
    if (j < b.size()) {remainList = b; m = j;}
    while(remainList.size() > 0 && m < remainList.size()){
        List_Entry tempListEntry;
        remainList.retrive(m, tempListEntry); m++;
        tempList.insert(k, tempListEntry); k++;}
    return tempList;}
```

3. 设h_a, h_b分别是两个带头结点的非递减有序单链表的表头指针，设计一个算法，将这两个有序表合并成一个非递增有序的单链表。要求结果链表仍使用原来两个链表的存储空间，不需要占用辅助存储空间。表中允许有重复数据.

［参考答案］

```
void ReverseList(LinkList L){
   Node *p, *q;
   p = L->next;
   L->next = NULL;
   while(p != NULL){
       q = p->next;
       p->next = L->next;
       L->next = p;
       p = q;
}}
```

```
LinkList MergeList(LinkList LA,
   LinkList LB){
   ReverseList(LA);
   ReverseList(LB);
   LinkList LC;
   Node *pa, *pb, *r;
   pa = LA->next; pb = LB->next;
   LC = LA;
   LC->next = NULL;
   r = LC;
   while(pa != NULL && pb != NULL){
       if(pa->data <= pb->data){
           r->next = pb; r = pb;
           pb = pb->next;}
       else{
           r->next = pa; r = pa;
           pa = pa->next;}
       if(pa){
           r->next = pa;}
       else{
           r->next = pb; }}
      return LC; }
```

4. 设计一个算法，将一个带头结点的单循环链表中的所有节点的链接方向逆转.

［参考答案］

```
template <class List_entry>
Error_code List<List_entry>::reverse(){
      Node<List_entry> *prev=NULL, *temp=NULL, *rear=NULL;
      rear = head -> next;
      while(rear != head){
            temp = rear->next;
            rear->next = prev;
            prev = rear; rear = temp;}
      head -> next = prev;
      return success; }
```

Chapter 2 ~ Chapter 4.

E2 (P64). Start with the stack methods, and write a function *copy_stack* with the following specifications.

[参考答案]

(a)源码如下--

```
Error_code copy_stack(Stack &dest, Stack &source){
    dest = source;
    return success; }
```

(b)源码如下--

```
Error_code copy_stack(Stack &dest, Stack &source){
    Error_code detected = success;
    Stack temp;
    Stack_entry item;
    while (detected == success && !source.empty( )) {
        detected = source.top(item);
        detected = source.pop( );
        if (detected == success) detected = temp.push(item); }
    while (detected == success && !temp.empty( )) {
        detected = temp.top(item);
        detected = temp.pop( );
        if (detected == success) detected = source.push(item);
        if (detected == success) detected = dest.push(item); }
    return detected;}
```

(c)源码如下--

```
Error_code copy_stack(Stack &dest, Stack &source){
    dest.count = source.count;
    for (int i = 0; i < source.count; i++)
        dest.entry[i] = source.entry[i];
    return success;}
```

(***which of these is easiest to write…***)

• 方法(1)最简单;

• 在栈近满时，方法(1)执行速度最快；在栈近空时，方法(3)执行速度最快，一般而言，方法(2)的执行速度最慢，但是在栈近空时，其执行速度可能优于方法(1)；

• 方法(2)能够更好地包容变化;

• 方法(2)&(3)都能够接受 const 类型的参数传递.

E3 (P65). Write code for the following functions. [Your code must use stack methods, but should not make any assumptions about how stacks or their methods are implemented]…

[参考答案]

(c)源码如下—

```
void clear(Stack &s){
    while (!s.empty( ))
    s.pop( ); }
```

(e)源码如下--

```
void delete_all(Stack &s, Stack_entry x){
    Stack temp;
    Stack_entry item;
    while (!s.empty( )) {
        s.top(item);
        s.pop( );
        if (item != x) temp.push(item);}
    while (!temp.empty( )) {
        temp.top(item);
        temp.pop( );
        s.push(item); }}
```

E4 (P65). Sometimes a program requires two stacks containing the same type of entries. If the two stacks are stored in separate arrays, then one stack might overflow while there was considerable unused space in the other….

[参考答案]

```
const int maxstack = 10; // small value for testing
class Double_stack {
    public:    Double_stack( );
        bool empty_a( ) const;
        bool empty_b( ) const;
        bool full( ) const; // Same method checks both stacks for fullness.
        Error_code pop_a( );
        Error_code pop_b( );
        Error_code stacktop_a(Stack_entry &item) const;
        Error_code stacktop_b(Stack_entry &item) const;
        Error_code push_a(const Stack_entry &item);
        Error_code push_b(const Stack_entry &item);
    private:    int top_a; // index of top of stack a; −1 if empty
        int top_b; // index of top of stack b; maxstack if empty
        Stack_entry entry[maxstack]; }
```

```
Double_stack :: Double_stack( ){
    top_a = −1;
    top_b = maxstack; }

Error_code Double_stack :: push_a(const Stack_entry &item){
    if (top_a >= top_b − 1) return overflow;
    entry[++top_a] = item;
    return success; }

Error_code Double_stack :: push_b(const Stack_entry &item){
    if (top_a >= top_b − 1) return overflow;
    entry[−−top_b] = item;
    return success; }

Error_code Double_stack :: stacktop_a(Stack_entry &item) const{
    if (top_a == −1) return underflow;
    item = entry[top_a];
    return success;}

Error_code Double_stack :: stacktop_b(Stack_entry &item) const{
    if (top_b == maxstack) return underflow;
    item = entry[top_b];
    return success;}

bool Double_stack :: full( ) const{
    return top_a >= top_b − 1;}

bool Double_stack :: empty_a( ) const{
    return top_a <= −1;}

bool Double_stack :: empty_b( ) const{
    return top_b >= maxstack;}

Error_code Double_stack :: pop_a( ){
    if (top_a <= −1) return underflow;
    else −−top_a;
    return success;}

Error_code Double_stack :: pop_b( ){
    if (top_b >= maxstack) return underflow;
    else ++top_b;
    return success;}
```

E2 (P84). Suppose that you are a financier and purchase 100 shares of stock in Company X in each of January, April and September and ….

［参考答案］

买进卖出统计——

January:	purchase	100×$10=$1000
April:	purchase	100×$30=$3000
June:	sell	100×$20=$2000
September:	purchase	100×$50=$5000
November:	sell	100×$30=$3000

Total Profit = $1000, *Total Loss* = $3000.

E3 (P84). Use the methods and queues developed in the text to write functions that will do each of the following tasks…

［参考答案］

(b)源码如下—

```
Error_code queue_to_stack(Stack &s, Queue &q){
    Error_code outcome = success;
    Entry item;
    while (outcome == success && !q.empty( )) {
        q.retrieve(item);
        outcome = s.push(item);
        if (outcome == success)    q.serve( );}
    return (outcome); }
```

(e)源码如下--

```
Error_code reverse_queue(Queue &q){
    Error_code outcome = success;
    Entry item;
    Stack temp;
    while (outcome == success && !q.empty( )) {
        q.retrieve(item);
        outcome = temp.push(item);
        if (outcome == success) q.serve( ); }
    while (!temp.empty( )) {
        temp.top(item);
        q.append(item);
        temp.pop( ); }
    return (outcome); }
```

E5 (P91). Write the methods to implement queues in a linear array with two indices $front$ and $rear$, such that…

［参考答案］

```
const int maxqueue = 10; // small value for testing
class Queue {
public:
    Queue( );
    bool empty( ) const;
    Error_code serve( );
    Error_code append(const Queue_entry &item);
    Error_code retrieve(Queue_entry &item) const;
protected:
    int front, rear;
    Queue_entry entry[maxqueue];
};
```

```
Queue :: Queue( ){
    rear = −1;
    front = 0; }

bool Queue :: empty( ) const{
    return rear < front; }

Error_code Queue :: append(const Queue_entry &item){
    if (rear == maxqueue − 1 || rear == maxqueue − 2)
        for (int i = 0; i <= rear − front; i++) {
            entry[i] = entry[i + front];
            rear = rear − front;
            front = 0; }
    if (rear == maxqueue − 1) return overflow;
    entry[++rear] = item;
    return success; }

Error_code Queue :: serve( ){
    if (rear < front) return underflow;
    front = front + 1;
    return success; }

Error_code Queue :: retrieve(Queue_entry &item) const{
    if (rear < front) return underflow;
    item = entry[front];
    return success; }
```

E7 (P92). Rewrite the methods of queue processing from the text, using a flag to indicate a full queue instead of keeping a count of the entries in the queue.

［参考答案］

```
const int maxqueue = 10; // small value for testing
class Queue {
public:
    Queue( );
    bool empty( ) const;
    Error_code serve( );
    Error_code append(const Queue_entry &item);
    Error_code retrieve(Queue_entry &item) const;
protected:
    int front, rear;
    Queue_entry entry[maxqueue];
    bool is_empty; };
```

```
Queue :: Queue( ){
    rear = − 1;
    front = 0;
    is_empty = true; }

bool Queue :: empty( ) const{
    return is_empty; }

Error_code Queue :: append(const Queue_entry &item){
    if (!empty( ) && (rear + 1)%maxqueue == front)    return overflow;
    is_empty = false;
    rear = ((rear + 1) == maxqueue) ? 0 : (rear + 1);
    entry[rear] = item;
    return success; }

Error_code Queue :: serve( ){
    if (empty( )) return underflow;
    if (rear == front) is_empty = true;
    front = ((front + 1) == maxqueue) ? 0 : (front + 1);
    return success; }

Error_code Queue :: retrieve(Queue_entry &item) const{
    if (empty( )) return underflow;
    item = entry[front];
    return success; }
```

E2 (P137). What is wrong with the following attempt to use the copy construction to implement the overloaded assignment operator for a lined Stack? ...

［参考答案］

```
void Stack :: operator = (const Stack &original){
    Stack new_copy(original);
    Node *temp = top_node;
    top_node = new_copy.top_node;
    new_copy.top_node = temp; // Make sure old stack value is deleted
    // and that new stack value is kept.
}
```

E5 (P140). A circularly linked list is a linked list in which the node at the tail of the list, instead of a NULL a pointer, points back to the node at the head of the list…

［参考答案］

(a) 源码如下—

```
class Queue {
public:
    // standard Queue methods
    Queue( );
    bool empty( ) const;
    Error_code append(const Queue_entry &item);
    Error_code serve( );
    Error_code retrieve(Queue_entry &item) const;
    // safety features for linked structures
    ~Queue( );
    Queue(const Queue &original);
    void operator = (const Queue &original);
protected:
    Node *tail;
};
```

```
Queue :: Queue( ) { tail = NULL; }
Queue :: ~Queue( ) { while (!empty( )) serve( ); }
bool Queue :: empty( ) const { return tail == NULL; }

Error_code Queue :: retrieve(Queue_entry &item) const{
    if (tail == NULL)    return underflow;
    item = (tail->next)->entry;
    return success; }
```

```
Error_code Queue :: append(const Queue_entry &item){
    Node *new_rear = new Node(item);
    if (new_rear == NULL)    return overflow;
    if (tail == NULL) {
          tail = new_rear;
          tail ->next = tail;}
    else {
          new_rear ->next = tail ->next;
          tail->next = new_rear;
          tail = new_rear; }
    return success; }

Error_code Queue :: serve( ){
    if (tail == NULL) return underflow;
    Node *old_front = tail->next;
    if (tail == old_front) tail = NULL;
    else tail->next = old_front->next;
    delete old_front;
    return success; }

Queue :: Queue(const Queue &copy) {
    tail = NULL;
    if (copy.tail == NULL) return;
    Node *copy_node = (copy.tail)->next;
    do {
        append(copy_node->entry);
        copy_node = copy_node->next;
    } while (copy_node != (copy.tail)->next);}

void Queue :: operator = (const Queue &copy){
    while (!empty( )) serve( );
    tail = NULL;
    if (copy.tail == NULL) return;
    Node *copy_node = (copy.tail)->next;
    do {
        append(copy_node->entry);
        copy_node = copy_node->next;}
    while (copy_node != (copy.tail)->next);}
```

(b) 该种方式的缺点是对链表的操作不够直接，例如在删除链表头结点时，需要通过 $tail->next$ 来访问头结点。一般而言，为了节省一个指针的存储空间而增加函数实现的复杂性是不明智的。

5. 回文是指正读与反读一致的字符串，例如 $'abcba'$，写一个算法判断输入的字符串是否为回文，并写出算法的时间复杂度.

［参考答案］

```
bool detectPalindrome(const string &str){
    Stack<char> s;
    int i = 0, strLen = str.length();
    for(i=0; i<strLen/2; i++){
        s.push(str[i]); }
    if(strLen%2 > 0) i++;
    for(; i<strLen; i++){
        if(str[i] != s.pop())
            return false; }
    return true; }
```

算法的时间复杂度为：$\mathcal{O}(n)$

Chapter 10.

E10 (P443). Write an overloaded binary tree assignment operator.

［参考答案］

```
template <class Entry>
Binary_tree<Entry> &Binary_tree<Entry> :: operator = (const
                                        Binary_tree<Entry> &original){
    Binary_tree<Entry> new_copy(original);
    clear( );
    root = new_copy.root;
    new_copy.root = NULL;
    return *this;}
```

2. 写一个递归算法求二叉树中度为 2 的节点数.

［参考答案］

```
template <class Entry>
int Binary_tree<Entry>::D2Nodes(Binary_node<Entry> *sub_root){
    if(!sub_root || !sub_root->left && !sub_root->right) return 0;
    if(sub_root->left && sub_root->right)
        return 1 + D2Nodes(sub_root->left) + D2Nodes(sub_root->right);
    else
        return D2Nodes(sub_root->left) + D2Nodes(sub_root->right);}
```

3. 写一算法判断两棵树是否相似，所谓相似是指两棵树除节点数据外完全相同.

［参考答案］

```
template <class Entry>
bool Binary_tree<Entry>::similar(
                    Binary_node<Entry> *T1, Binary_node<Entry> *T2){
    if(!T1 && !T2) return true;
    if(!T1 || !T2) return false;
    return similar(T1->left, T2->left) && similar(T1->right, T2->right);}
```

4. 设二叉树中各节点数据互不相同，设计算法找到值为k的节点，并将以该节点为根节点的子树从树中分离出来，使之变为两棵二叉树.

［参考答案］

```
template <class Entry>
bool Binary_tree<Entry>::separateTree(
        Binary_node<Entry> **T, Entry k, Binary_node<Entry> **subTree){
        bool find = false;
        if(*T == NULL) return false;
        if((*T)->data == k){
                *SubTree = *T;
                *T = NULL;
                find = true;}
        else {
                find = separateTree(&(*T)->left, subTree) ||
                                separateTree(&(*T)->right, subTree); }
        return find;}
```